CHAPTER

4

Building Better Web Forms

HTML forms are simple HTML controls you use to collect information from website visitors. They include text boxes people can type into, list boxes they can pick from, checkboxes they can switch on or off, and so on. On the Web, forms let people do everything from getting stock quotes to buying concert tickets.

HTML forms have existed almost since the dawn of HTML, and they haven't changed a wink since last century, despite some serious efforts. Web standards-makers spent years cooking up a successor called XForms, which fell as flat as XHTML 2 (see page 4). Although XForms solved some problems easily and elegantly, it also had its own headaches—for example, XForms code was verbose and assumed that web designers were intimately familiar with XML. But the biggest hurdle was the fact that XForms wasn't compatible with HTML forms in any way, meaning that developers would need to close their eyes and jump to a new model with nothing but a whole lot of nerve and hope. But because mainstream web browsers never bothered to implement XForms—it was too complex and little used—web developers never ended up taking that leap.

HTML5 takes a different approach. It adds refinements to the existing HTML forms model, which means HTML5-enhanced forms can keep working on older browsers, just without all the bells and whistles. (This is a good thing, because Internet Explorer doesn't support any new form features in versions before IE 10.) HTML5 also adds practical form features that developers were already using but that previously required a pile of JavaScript code or a JavaScript toolkit. Now, HTML5 makes these features easily accessible.

In this chapter, you'll tour all the new features of HTML5 forms. You'll see which ones are supported, which ones aren't, and which workarounds can help you smooth

over the differences. You'll also consider a feature that isn't technically part of the HTML5 forms standard but is all about interactivity—putting a rich HTML editor in an ordinary web page.

Understanding Forms

Odds are that you've worked with forms before. But if you're a bit sketchy on the details, the following recap will refresh your memory.

A *web form* is a collection of text boxes, lists, buttons, and other clickable widgets that a web surfer uses to supply some sort of information to a website. Forms are all over the Web—they allow you to sign up for email accounts, review products, and make bank transactions. The simplest possible form is the single text box that adorns search engines like Google (see Figure 4-1).

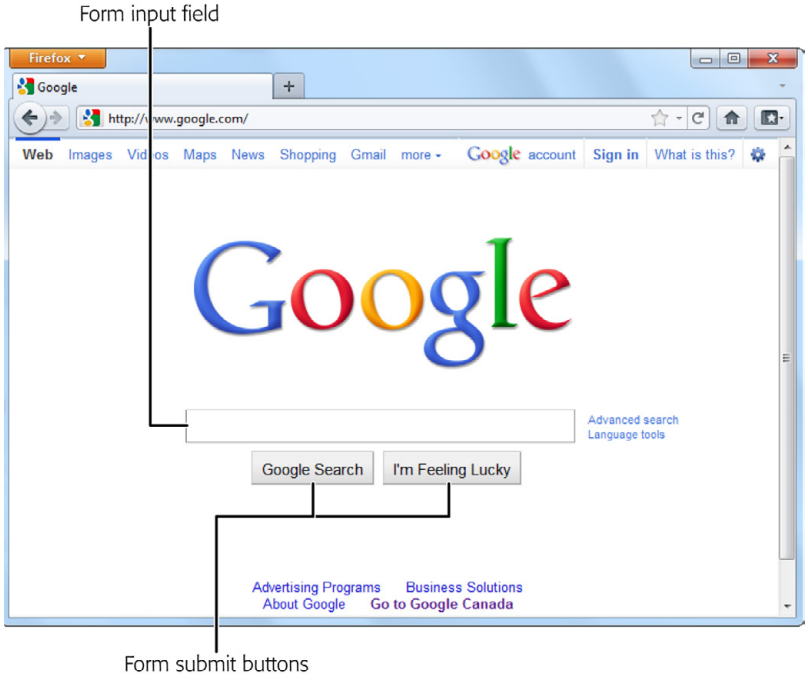


FIGURE 4-1

Google's Spartan search page holds a basic HTML form. You use it in the same way you use any form: You enter some information (in this case, search keywords), and click a button to submit that information.

All basic web forms work in the same way. The user fills in some information and then clicks a button. At that point, the server collects all the data that the user has entered and sends it back to the web server. On the web server, some sort of application digests the information and takes the appropriate next step. The server-side program might consult a database, either to read or to store some information, before sending a new page back to the web browser.

The tricky part of this discussion is that there are hundreds of ways to build the server-side part of the equation—that's the application that processes the information that's submitted from the form. Some developers may use stripped-down scripts that let them manipulate the raw form data, while others may work with higher-level models that package the form details in neat programming objects. But either way, the task is basically the same. You examine the form data, do something with it, and then send back a new page.

NOTE This book doesn't make any assumptions about your choice of server-side programming tool. In fact, it doesn't really matter, because you still need to use the same set of form elements, and these elements are still bound by the same HTML5 rules.

UP TO SPEED

Bypassing Form Submission with JavaScript

It's worth noting that forms aren't the only way to send user-entered information to a web server (although they were, once upon a time). Today, crafty developers can use the `XMLHttpRequest` object (page 377) in JavaScript code to quietly communicate with a web server. For example, the Google search page uses this approach in two different ways—first, to get search suggestions, which it displays in a drop-down list; and second, to get a search results page as you type, if you've enabled the Google Instant feature (*www.google.com/instant*).

It might occur to you that JavaScript can completely bypass the form submission step, as it does in Google Instant. But while it's possible to offer this technique as a *feature*, it's not acceptable to include it as a *requirement*. That's because the JavaScript approach isn't bulletproof (for example, it may exhibit the occasional quirk on a slow connection) and there's still a small fraction of people with no JavaScript support or with JavaScript turned off in their browsers.

Finally, it's worth noting that it's perfectly acceptable to have a page that includes a form but never *submits* that form. You've probably seen pages that perform simple calculations (for example, a mortgage interest rate calculator). These forms don't need any help from the server, because they can perform their calculations entirely in JavaScript and display the result on the current page. For that reason, these forms never need to submit their data.

From the HTML5 point of view, it really doesn't matter whether you submit your form to a server, use the data in an ordinary JavaScript routine, or pass it back to the server through `XMLHttpRequest`. In all cases, you'll still build your form using the standard HTML forms controls.

Revamping a Traditional HTML Form

The best way to learn about HTML5 forms is to take a typical example from today and enhance it. Figure 4-2 shows the example you'll start out with.

The markup is dishwater-dull. If you've worked with forms before, you won't see anything new here. First, the entire form is wrapped in a `<form>` element:

```
<form id="zooKeeperForm" action="processApplication.cgi">
  <p><i>Please complete the form. Mandatory fields are marked with
  a </i><em>*</em></p>
  ...
```

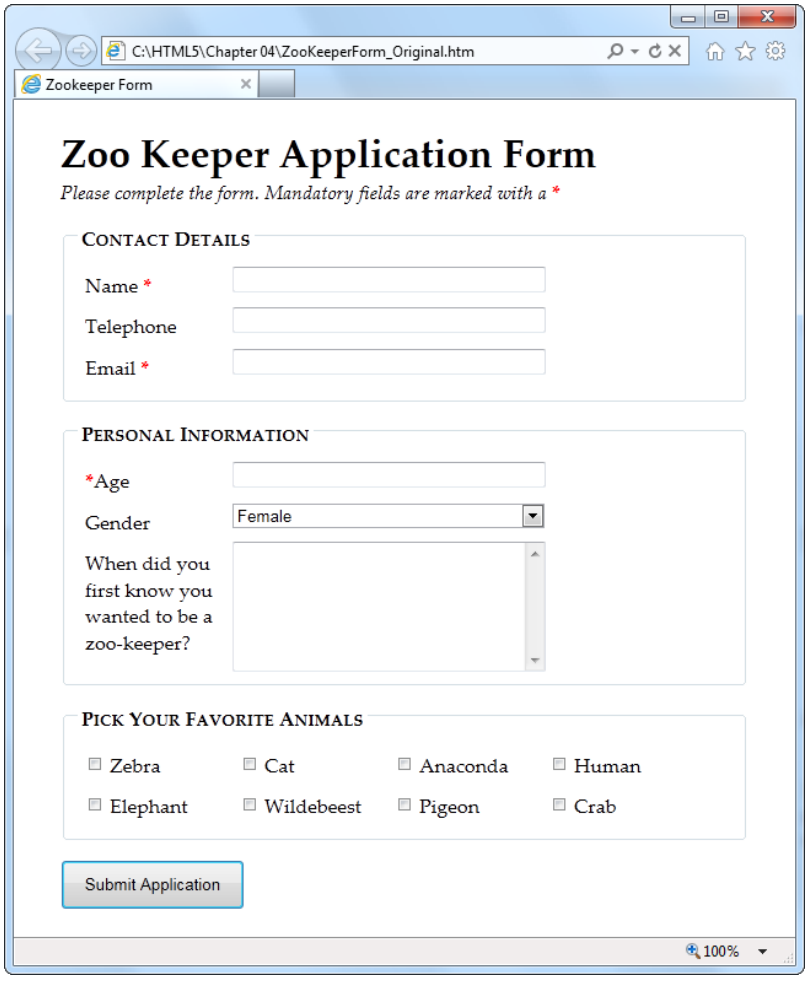


FIGURE 4-2

If you've traveled the Web, you've seen your share of forms like this one, which collects basic information from a web page visitor.

The `<form>` element bundles together all the form widgets (also known as *controls* or *fields*). It also tells the browser where to post the page when it's submitted, by providing a URL in the `action` attribute. If you plan to do all the work in client-side JavaScript code, you can simply use a number sign (#) for the `action` attribute.

NOTE HTML5 adds a mechanism for placing form controls outside of the form to which they belong. The trick is to use the new `form` attribute to refer to the form by its `id` value (as in `form="zooForm"`). However, browsers that don't support this feature will completely overlook your data when the form is submitted, which means this minor feature is still too risky to use in a real web page.

A well-designed form, like the zookeeper application, divides itself into logical chunks using the `<fieldset>` element. Each chunk gets a title, courtesy of the `<legend>` element. Here's the `<fieldset>` for the Contact Details section (which is dissected in Figure 4-3):

```
...
<fieldset>
  <legend>Contact Details</legend>
  <label for="name">Name <em>*</em></label>
  <input id="name"><br>
  <label for="telephone">Telephone</label>
  <input id="telephone"><br>
  <label for="email">Email <em>*</em></label>
  <input id="email"><br>
</fieldset>
...
```

FIGURE 4-3

This `<fieldset>` collects three pieces of information: a name, telephone number, and email address. For each piece of information, you supply a caption (using the `<label>` element) and a control to collect the data (using an `<input>`, `<textarea>`, or `<select>` element).

As in all forms, the bulk of the work is handled by the all-purpose `<input>` element, which collects text and creates checkboxes, radio buttons, and list buttons. Along with `<input>`, the `<textarea>` element gives people a way to enter multiple lines of text, and the `<select>` element creates a list. If you need a refresher, Table 4-1 will fill you in.

TABLE 4-1 *Form controls*

CONTROL	HTML ELEMENT	DESCRIPTION
Single-line textbox	`<input type="text">` `<input type="password">`	Shows a text box where visitors can type in text. If you use the `password` type, the browser won't display the text. Instead, visitors see an asterisk (*) or a bullet (•) in place of each letter as they type in their password.
Multiline textbox	`<textarea>...</textarea>`	Shows a large text box that can fit multiple lines of text.
Checkbox	`<input type="checkbox">`	Shows a checkbox that can be switched on or off.
Radio button	`<input type="radio">`	Shows a radio button (a circle you can turn on or off). Usually, you have a group of radio buttons with the same value for the `name` attribute, in which case the visitor can select only one.

Button	`<input type="submit">`
	`<input type="image">`
	`<input type="reset">`
	`<input type="button">`
		Shows the standard clickable button. A `submit` button always gathers up the form data and sends it to its destination. An `image` button does the same thing, but it lets you display a clickable picture instead of the standard text-on-a-button. A `reset` button clears the visitor's selections and text from all the input controls. A `button` button doesn't do anything unless you add some JavaScript code.

List	`<select>...</select>`	Shows a list where your visitor can select one or more items. You add an `<option>` element for each item in the list.

Here's the rest of the zookeeper form markup, with a few new details (a `<select>` list, checkboxes, and the button that submits the form):

```
...
<fieldset>
  <legend>Personal Information</legend>
    <label for="age"><em>*</em>Age</label>
    <input id="age"><br>
    <label for="gender">Gender</label>
    <select id="gender">
      <option value="female">Female</option>
      <option value="male">Male</option>
    </select><br>
```

```
      <label for="comments">When did you first know you wanted to be a
       zoo-keeper?</label>
      <textarea id="comments"></textarea>
    </fieldset>

    <fieldset>
      <legend>Pick Your Favorite Animals</legend>
      <label for="zebra"><input id="zebra" type="checkbox"> Zebra</label>
      <label for="cat"><input id="cat" type="checkbox"> Cat</label>
      <label for="anaconda"><input id="anaconda" type="checkbox"> Anaconda
      </label>
      <label for="human"><input id="human" type="checkbox"> Human</label>
      <label for="elephant"><input id="elephant" type="checkbox"> Elephant
      </label>
      <label for="wildebeest"><input id="wildebeest" type="checkbox">
       Wildebeest</label>
      <label for="pigeon"><input id="pigeon" type="checkbox"> Pigeon</label>
      <label for="crab"><input id="crab" type="checkbox"> Crab</label>
    </fieldset>
    <p><input type="submit" value="Submit Application"></p>
  </form>
```

You can find the full example, along with the relatively simple style sheet that formats it, on the try-out site (*http://prosetech.com/html5*). Look for the *ZookeeperForm_Original.html* file to play around with a traditional, unenhanced version of the form, and *ZookeeperForm_Revised.html* to get all the HTML5 goodies.

NOTE One limit of HTML forms is that you can't change how the browser draws controls. For example, if you want to replace the standard dull gray checkbox with a big black-and-white box with a fat red checkmark image, you can't. (The alternative is to create a normal element that has checkbox-like behavior using JavaScript—in other words, it changes its appearance back and forth when someone clicks it.)

HTML5 keeps this no-customization limit in place and extends it to the new controls you'll learn about in this chapter. That means ordinary HTML5 forms aren't enough for web developers who want complete control over the look of their pages. Instead, they'll need a JavaScript toolkit like jQuery UI.

Now that you've got a form to work with, it's time to start improving it with HTML5. In the following sections, you'll start small, with placeholder text and an autofocus field.

Adding Hints with Placeholders

Forms usually start out empty. But a column of blank text boxes can be a bit intimidating, especially if it's not absolutely clear what belongs inside each text box. That's why you commonly see some sort of sample text inside otherwise-empty text boxes. This placeholder text is also called a *watermark*, because it's often given a light-gray color to distinguish it from real, typed-in content. Figure 4-4 shows a placeholder in action.

To create a placeholder, simply use the `placeholder` attribute:

```
<label for="name">Name <em>*</em></label>
<input id="name" placeholder="Jane Smith"><br>
<label for="telephone">Telephone</label>
<input id="telephone" placeholder="(xxx) xxx-xxxx"><br>
```

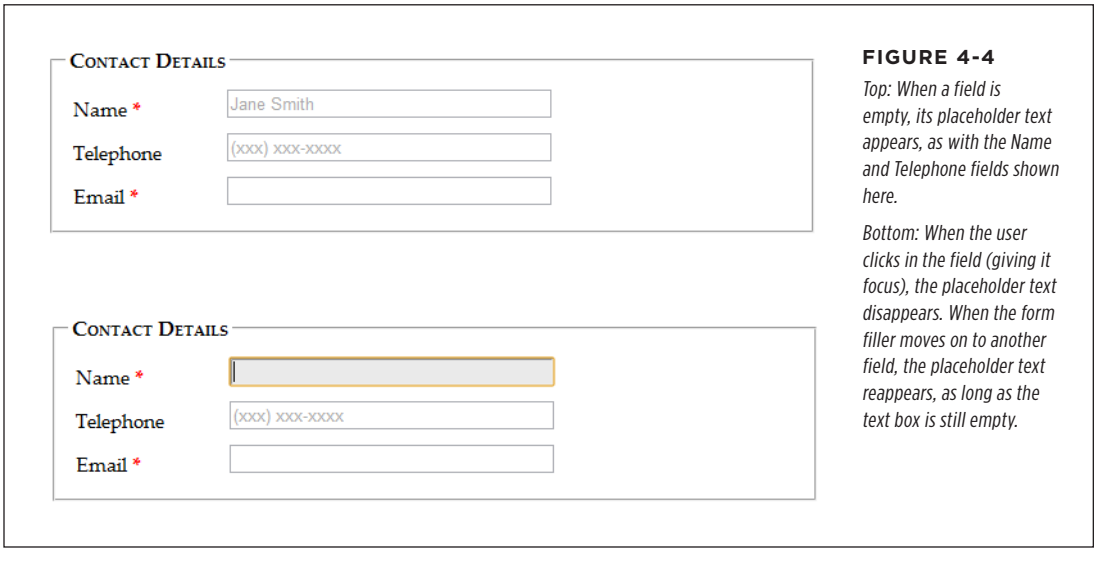


FIGURE 4-4

Top: When a field is empty, its placeholder text appears, as with the Name and Telephone fields shown here.

Bottom: When the user clicks in the field (giving it focus), the placeholder text disappears. When the form filler moves on to another field, the placeholder text reappears, as long as the text box is still empty.

Browsers that don't support placeholder text just ignore the `placeholder` attribute; Internet Explorer (before IE 10) is the main culprit. Fortunately, it's not a big deal, since placeholders are just nice form frills, not essential to your form's functioning. If it really bothers you, there are plenty of JavaScript patches that can bring IE up to speed, painlessly, at *http://tinyurl.com/polyfills*.

Right now, there's no standard, consistent way to change the appearance of placeholder text (for example, to italicize it or to change the text color). Eventually, browser makers will create the CSS styling hooks that you need—in fact, they're hashing out the details even as you read this. But to get it to work right now, you need to fiddle with browser-specific pseudo-classes (namely, `-webkit-input-placeholder` for Chrome, `-ms-input-placeholder` for Internet Explorer, and `-moz-placeholder` for Firefox). Page 443 has the full details about pseudo-classes, and page 183 explains the awkward world of browser-specific styles.

However, you can use the better-supported `focus` pseudo-class without any headaches. You use it to change the way a text box looks when it gets the focus. For example, you might want to assign a darker background color to make it stand out:

```
input:focus {
  background: #eaeaea;
}
```

UP TO SPEED

Writing Good Placeholders

You don't need placeholders for every text box. Instead, you should use them to clear up potential ambiguity. For example, no one needs an explanation about what goes in a First Name box, but the Name box in Figure 4-4 isn't quite as obvious. The placeholder text makes it clear that there's room for a first and a last name.

Sometimes placeholders include a sample value—in other words, something you might actually type into the box. For example, Microsoft's Hotmail login page (*www.hotmail.com*) uses the text *someone@example.com* for a placeholder, making it obvious that you should enter your email address in the box—not your name or any other information.

Other times, placeholders indicate the way a value should be formatted. The telephone box in Figure 4-4 is an example—it shows the value *(xxx) xxx-xxxx* to concisely indicate that telephone numbers should consist of a three-digit area code, followed by a sequence of three, then four digits. This placeholder doesn't necessarily mean that differently formatted input isn't allowed, but it does offer a suggestion that uncertain people can follow.

There are two things you shouldn't try with a placeholder. First, don't try to cram in a description of the field or instructions. For example, imagine you have a box for a credit card's security code. The text "The three digits listed on the back of your card" is *not* a good placeholder. Instead, consider adding a text note under the input box, or using the `title` attribute to make a pop-up window appear when someone hovers over the field:

```
<label for="promoCode">Promotion Code
</label>
<input id="promoCode" placeholder="QRBOO1"
title="Your promotion code is three
letters followed by three numbers">
```

Second, you shouldn't add special characters to your placeholder in an attempt to make it obvious that your placeholder is not real, typed-in text. For example, some websites use placeholders like *[John Smith]* instead of *John Smith*, with the square brackets there to emphasize that the placeholder is just an example. This convention can be confusing.

Focus: Starting in the Right Spot

After loading up your form, the first thing your visitors want to do is start typing. Unfortunately, they can't—at least not until they tab over to the first control, or click it with the mouse, thereby giving it *focus*.

You can make this happen with JavaScript by calling the `focus()` method of the appropriate `<input>` element. But this involves an extra line of code and can sometimes cause annoying quirks. For example, it's sometimes possible for the user to click somewhere else and start typing before the `focus()` method gets called, at which point focus is rudely transferred back to the first control. But if the browser were able to control the focus, it could be a bit smarter, and transfer focus only if the user hasn't already dived into another control.

That's the idea behind HTML5's `autofocus` attribute, which you can add to a single `<input>` or `<textarea>` element, like this:

```
<label for="name">Name <em>*</em></label>
<input id="name" placeholder="Jane Smith" autofocus><br>
```

The autofocus attribute has similar support as the placeholder attribute, which means basically every browser recognizes it except IE 9 and older. Once again, it's easy enough to plug the hole. You can check for autofocus support using Modernizr (page 31) and then run your own autofocus code if needed. Or, you can use a ready-made JavaScript polyfill that adds autofocus support (*http://tinyurl.com/polyfills*). However, it hardly seems worth it for such a minor frill, unless you're also aiming to give IE support for other form features, like the validation system discussed next.

Validation: Stopping Errors

The fields in a form are there to gather information from web page visitors. But no matter how politely you ask, you might not get what you want. Impatient or confused visitors can skip over important sections, enter partial information, or just hit the wrong keys. The end result? They click Submit, and your website gets a whackload of scrambled data.

What a respectable web page needs is *validation*—a way to catch mistakes when they happen (or even better, to prevent them from happening at all). For years, developers have done that by writing their own JavaScript routines or using professional JavaScript libraries. And, truthfully, these approaches work perfectly well. But seeing as validation is so common (just about everyone needs to do error-checking), and seeing as validation generally revolves around a few key themes (for example, spotting invalid email addresses or dates), and seeing as validation is boring (no one really wants to write the same code for every form, not to mention *test* it), there's clearly room for a better way.

The creators of HTML5 spotted this low-hanging fruit and invented a way for browsers to help out, by getting them to do the validation work instead of web developers. They devised a *client-side* validation system (see the box on page 113) that lets you embed common error-checking rules into any <input> field. Best of all, this system is easy—all you need to do is insert the right attribute.

How HTML5 Validation Works

The basic idea behind HTML5 form validation is that you indicate where validation should happen, but you don't actually *implement* the tedious details. It's a bit like being promoted into a management job, just without the pay raise.

For example, imagine you decide a certain field cannot be left blank—the form filler needs to supply some sort of information. In HTML5, you can make this demand by adding the required attribute:

```
<label for="name">Name <em>*</em></label>
<input id="name" placeholder="Jane Smith" autofocus required><br>
```

UP TO SPEED

Validating in Two Places

Throughout the years, crafty developers have approached the validation problem in different ways. Today, the consensus is clear. To make a bulletproof form, you need two types of error-checking:

- **Client-side validation.** These are the checks that happen in the browser, *before* a form is submitted. The goal here is to make life easier for the people filling out the form. Instead of waiting for them to complete three dozen text boxes and click a submit button, you want to catch problems in the making. That way you can pop up a helpful error message right away and in the right spot, allowing the form filler to correct the mistake before submitting the form to the server.
- **Server-side validation.** These are the checks that happen *after* a form is sent back to the web server. At this point, it's up to your server-side code to review the details and make sure everything is kosher before continuing. No matter what the browser does, server-side validation is essential. It's the only way to defend yourself from malicious people who are deliberately trying to tamper with form data. If your server-side validation detects a problem, you send back a page with an error message.

So client-side validation (of which HTML5 validation is an example) is there to make life easier for your web page visitors, while server-side validation ensures correctness. The key thing to understand is that you need both types of validation—unless you have an exceedingly simple form where mistakes aren't likely or aren't a big deal.

Initially, there's no visual detail to indicate that a field is required. For that reason, you might want to use some other visual clue, such as giving the text box a different border color or placing an asterisk next to the field (as in the zookeeper form).

Validation kicks in only when the form filler clicks a button to submit the form. If the browser implements HTML5 forms, then it will notice that a required field is blank, intercept the form submission attempt, and show a pop-up message that flags the invalid field (Figure 4-5).

As you'll see in the following sections, different attributes let you apply different error-checking rules. You can apply more than one rule to the same input box, and you can apply the same rule to as many `<input>` elements as you want (and to the `<textarea>` element). All the validation conditions must be met before the form can be submitted.

This raises a good question: What happens if form data breaks more than one rule—for example, it has multiple required fields that aren't filled in?

Once again, nothing happens until the person filling out the form clicks the submit button. Then, the browser begins examining the fields from top to bottom. When it finds the first invalid value, it stops checking any further. It cancels the form submission and pops up an error message next to this value. (Additionally, if the offending text box isn't currently visible, the browser scrolls up just enough that it appears at the top of the page.) If the visitor corrects the problem and clicks the submit button again, the browser will stop and highlight the next invalid value.

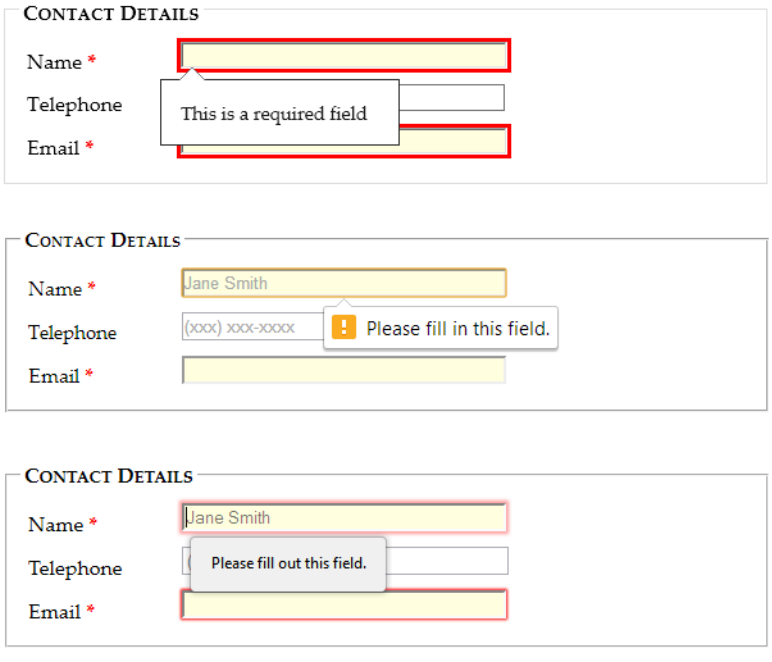


FIGURE 4-5

Here's the same required field in Chrome (top), Internet Explorer (middle), and Firefox (bottom). Browsers are free to choose the exact way they notify people about validation problems, but they all use a pop-up box that looks like a stylized tooltip. Unfortunately, you can't customize the formatting of this box or change the wording of the validation message—at least not yet.

NOTE Web browsers hold off on validation until a submit button is clicked. This ensures that the validation system is efficient and restrained, so it works for everyone.

Some web developers prefer to alert people as soon as they leave an invalid field (when they tab away or click somewhere else with the mouse). This sort of validation is handy in long forms, especially if there's a chance that someone may make a similar mistake in several different fields. Unfortunately, HTML5 doesn't have a way for you to dictate when the web browser does its validation, although it's possible that it might add one in the future. For now, if you want immediate validation messages, it's best to write the JavaScript yourself or to use a good JavaScript library.

Turning Validation Off

In some cases, you may need to disable the validation feature. For example, you might need to turn it off for testing to verify that your server-side code deals appropriately with invalid data. To turn validation off for an entire form, you add the `novalidate` attribute to the containing `<form>` element:

```
<form id="zooKeeperForm" action="processApplication.cgi" novalidate>
```

The other option is to provide a submit button that bypasses validation. This technique is sometimes useful in a web page. For example, you may want to enforce strict validation for the official submit button but provide another button that does something else (like storing half-completed data for later use). To allow this, add the `formnovalidate` attribute to the `<input>` element that represents your button:

```
<input type="submit" value="Save for Later" formnovalidate>
```

You've now seen how to use validation to catch missing information. Next, you'll learn to search for errors in different types of data.

NOTE Planning to validate numbers? There's no validation rule that forces text to contain digits, but there is a new `number` data type, which you'll examine on page 126. Unfortunately, its support is still sketchy.

Validation Styling Hooks

Although you can't style validation messages, you can change the appearance of the input fields based on whether or not they're validated. For example, you can give invalid values a different background color, which will appear in the text box as soon as the browser detects the problem.

To use this technique, you simply need to add a few new pseudo-classes (page 443). Your options include the following:

- **`required`** and **`optional`**, which apply styles to fields based on whether they use the `required` attribute.
- **`valid`** and **`invalid`**, which apply styles to controls based on whether they contain mistakes. But remember that most browsers won't actually discover invalid values until the visitor tries to submit the form, so you won't see the invalid formatting right away.
- **`in-range`** and **`out-of-range`**, which apply formatting to controls that use the `min` and `max` attributes to limit numbers to a range (page 127).

For example, if you want to give required `<input>` fields a light-yellow background, you could use a style rule with the `required` pseudo-class:

```
input:required {
  background-color: lightyellow;
}
```

Or, you might want to highlight only those fields that are required and currently hold invalid values by combining the `required` and `invalid` pseudo-classes like this:

```
input:required:invalid {
  background-color: lightyellow;
}
```

With this setting, blank fields are automatically highlighted, because they break the required-field rule.

You can use all sorts of other tricks, like combining the validation pseudo-classes with the `focus` pseudo-class, or using an offset background that includes an error icon to flag invalid values. Of course, a hefty disclaimer applies: You can use these pseudo-classes to improve your pages, but make sure your form still looks good without them, because support lags in older browsers.

Validating with Regular Expressions

The most powerful (and complex) type of validation that HTML5 supports is based on regular expressions. Seeing as JavaScript already supports regular expression, adding this feature to HTML forms makes perfect sense.

A *regular expression* is a pattern written using the regular expression language. Regular expressions are designed to match patterned text—for example, a regular expression can make sure that a postal code has the right sequence of letters and digits, or that an email address has an @ symbol and a domain extension that's at least two characters long. For example, consider this expression:

```
[A-Z]{3}-[0-9]{3}
```

The square brackets at the beginning define a range of allowed characters. In other words, [A-Z] allows any uppercase letter from A to Z. The curly brackets that follow multiply this effect, so {3} means you need three uppercase letters. The dash that follows doesn't have a special meaning, so it indicates that a dash must follow the three-letter sequence. Finally, [0-9] allows a digit from 0 to 9, and {3} requires three of them.

Regular expression matching is useful for searching (finding pattern matches in a long document) and validation (verifying that a value matches a pattern). HTML5 forms use regular expressions for validation.

NOTE Regular expression geeks take note: You don't need the magic ^ and $ characters to match the beginning or end of a value in a field. HTML5 assumes both details automatically, which means a regular expression must match the *entire* value in a field in order to be deemed valid.

These values are valid, because they match the pattern shown above:

```
QRB-001

TTT-952

LAA-000
```

But these values are not:

```
qrb-001

TTT-0952

LA5-000
```

Regular expressions quickly get much more complex than this example. Writing a regular expression can be quite a chore, which is why most developers simply search for a ready-made regular expression that validates the type of data they want to check. Or they get help.

TIP To learn just enough about the regular expression language to make your own super-simple expressions, check out the concise tutorials at *http://tinyurl.com/regexp-object* or *http://tinyurl.com/jsregex*. To find ready-made regular expressions that you can use with your forms, visit *http://regexlib.com*. And to become a regular expression guru, read *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly).

Once you have a regular expression, you can enforce it in any `<input>` or `<textarea>` element by adding the `pattern` attribute:

```
<label for="promoCode">Promotion Code</label>
<input id="promoCode" placeholder="QRB-001" title=
 "Your promotion code is three uppercase letters, a dash, then three numbers"
 pattern="[A-Z]{3}-[0-9]{3}">
```

Figure 4-6 shows what happens if you break the regular expression rule.

TIP Browsers don't validate blank values. In this example, a blank promotion code passes muster. If this isn't what you want, then combine the `pattern` attribute with the `required` attribute.

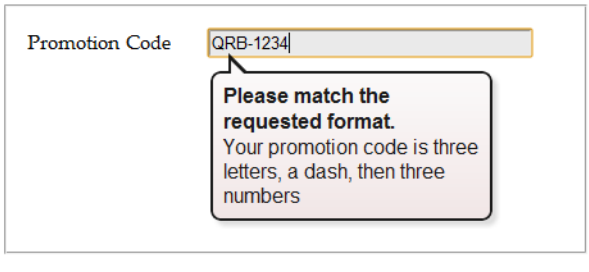


FIGURE 4-6

Smart browsers (like Google Chrome, shown here) don't just catch the mistake, they also grab the text from the `title` attribute and display that as well, to help guide the person who's filling out the form.

NOTE Regular expressions seem like a perfect match for email addresses (and they are). However, hold off on using them this way, because HTML5 already has a dedicated input type for email addresses that has the correct regular expression baked in (page 125).

Custom Validation

The HTML5 specification also outlines a set of JavaScript properties that let you find out if fields are valid (or force the browser to validate them). The most useful of these is the `setCustomValidity()` method, which lets you write custom validation logic for specific fields and have it work with the HTML5 validation system.

Here's how it works. First, you need to check the appropriate field for errors. You do this by handling the onInput event, which is nothing new:

```
<label for="comments">When did you first know you wanted to be a
 zookeeper?</label>
<textarea id="comments" oninput="validateComments(this)"></textarea>
```

In this example, the onInput event triggers a function named validateComments(). It's up to you to write this function, check the current value of the <input> element, and then call setCustomValidity().

If the current value has problems, you need to supply an error message when you call setCustomValidity(). Or, if the current value checks out, you need to call setCustomValidity() with an empty string. This clears any error custom messages that you may have set earlier.

Here's an example that forces the text in the comment box to be at least 20 characters long:

```
function validateComments(input) {
  if (input.value.length < 20) {
    input.setCustomValidity("You need to comment in more detail.");
  }
  else {
    // There's no error. Clear any error message.
    input.setCustomValidity("");
  }
}
```

Figure 4-7 shows what happens if someone breaks this rule and then tries to submit the form.

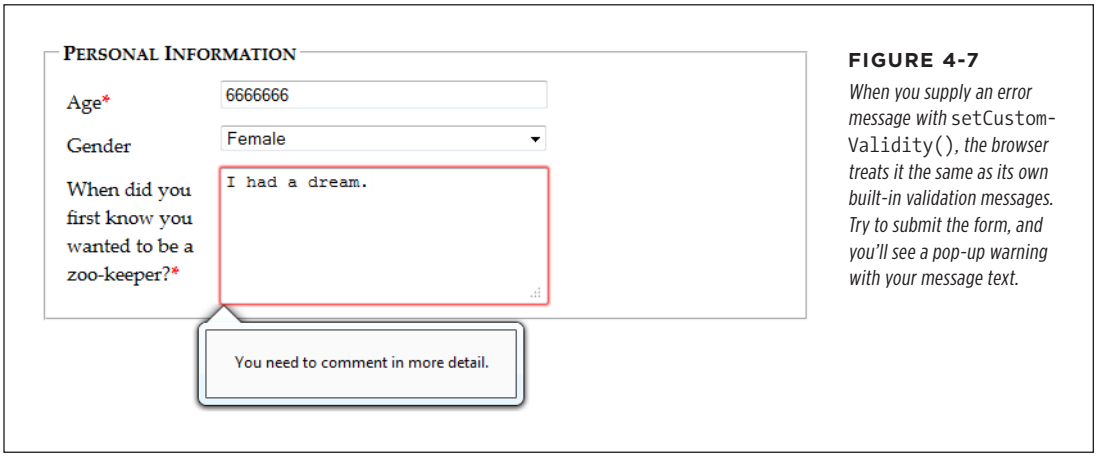


FIGURE 4-7

When you supply an error message with setCustomValidity(), *the browser treats it the same as its own built-in validation messages. Try to submit the form, and you'll see a pop-up warning with your message text.*

Of course, you could solve this problem more neatly with a regular expression that requires long strings. But while regular expressions are great for validating some data types, custom validation logic can do *anything*, from complex algebra to contacting the web server.

NOTE Remember, your web page visitors can see anything you put in JavaScript, so it's no place for secret algorithms. For example, you might know that in a valid promotional code, the digits always add up to 12. But you probably don't want to reveal that detail in a custom validation routine, because it will help shifty people cook up fake codes. Instead, keep this sort of validation in the web server.

Browser Support for Web Forms and Validation

Browser makers added support for HTML5 forms in pieces. That means some browser builds support some validation features while ignoring others. Table 4-2 indicates the minimum browser versions you need to use to get solid support for all the validation tricks you've learned so far. As this table indicates, there are two potential support headaches: old versions of IE, and mobile browsers that run on smartphones and tablets.

TABLE 4-2 *Browser support for validation*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	CHROME FOR ANDROID
Minimum version	10	4	10	5*	10	-	28

* Safari doesn't support the `required` attribute.

Because HTML5 validation doesn't replace the validation you do on your web server, you may see it as a frill, in which case you can accept this uneven support. Browsers that don't implement validation, like IE 9, let people submit forms with invalid dates, but you can then catch these problems on the web server and return the same page with error details.

On the other hand, your website might include complex forms with lots of potential for confusion, and you might not be ready to accept a world of frustrated IE users. In this case, you have two basic choices: Fall back on your own validation system or use a JavaScript library that adds the missing smarts. Your choice depends on the extent and complexity of your validation.

Testing for Support with Modernizr

If all your form needs is a smattering of simple validation, it's probably worth adding your own checks. Using Modernizr (page 31), you can check for a variety of HTML5 web forms features. For example, use the `Modernizr.input.pattern` property to check whether the current browser recognizes the `pattern` attribute:

```
if (!Modernizr.input.pattern) {
  // The current browser doesn't perform regular expression validation.
  // You can use the regular expression features in JavaScript instead.
  ...
}
```

NOTE The `pattern` property is just one of the properties exposed by `Modernizr.input` object. Other properties that are useful for testing web form support include `placeholder`, `autofocus`, `required`, `max`, `min`, and `step`.

Of course, this example doesn't tell you *when* to perform this check or *how* to react. If you want your validation to mimic the HTML5 validation system, it makes sense to perform your validation when the person viewing the form attempts to submit it. You do this by handling the form's `onSubmit` event and then returning either `true` (which means the form is valid and the browser can submit it) or `false` (which means the form has a problem and the browser should cancel the operation):

```
<form id="zooKeeperForm" action="processApplication.cgi"
 onsubmit="return validateForm()">
```

Here's an example of a very simple custom validation routine that enforces required fields:

```
function validateForm() {
  if (!Modernizr.input.required) {
    // The required attribute is not supported, so you need to check the
    // required fields yourself.

    // First, get an array that holds all the elements.
    var inputElements = document.getElementById("zooKeeperForm").elements;

    // Next, move through that array, checking each element.
    for(var i = 0; i < inputElements.length; i++) {

      // Check if this element is required.
      if (inputElements[i].hasAttribute("required")) {
        // If this element is required, check if it has a value.
        // If not, the form fails validation, and this function returns false.
        if (inputElements[i].value == "") return false;
      }
    }
```

```
        // If you reach this point, everything worked out and the browser
        // can submit the form.
        return true;
      }
    }
```

TIP This block of code relies on a number of basic JavaScript techniques, including element lookup, a loop, and conditional logic. To learn more about all these details, check out Appendix B, "JavaScript: The Brains of Your Page."

Polyfilling with HTML5Forms

If you have a complex form and you want to save some effort (while at the same time preparing for the future), you may prefer to use a JavaScript patch to solve all your problems. Technically, the approach is the same—your page will check for validation support and then perform validation manually if necessary. The difference is that the JavaScript library already has all the tedious code you need.

At *http://tinyurl.com/polyfills*, you can find a long, intimidating list of JavaScript libraries that all attempt to do more or less the same thing. One oldie but goody is the HTML5Forms library, which is available from *http://tinyurl.com/html5forms*. To get a copy, click the Download ZIP button. You'll be rewarded with a Zip folder stuffed full of files. Unzip it, and you'll find a pile of useful scripts (in the *shared/js* folder) and a long list of example pages (in the *tests/html5forms* folder).

To get started with HTML5Forms, copy the *shared* folder (with all its subfolders) to your website folder. You can rename it to something else (for example, *html5forms* instead of *shared*), as long as you tweak the name in your script references. Once you've copied the files, you need to add two references to your web page, like this:

```
<head>
  <title>...</title>
  <script src="shared/js/modernizr.com/Modernizr-2.5.3.forms.js"></script>
  <script src="shared/js/html5Forms.js" data-webforms2-support="all"
   data-webforms2-force-js-validation="true">
  </script>
  ...
<head>
```

The first reference points to a small build of Modernizr (here, it's named *Modernizr-2.5.3.forms.js*) that's included with HTML5Forms, and provides feature detection, ensuring that the validation workarounds are loaded only if the browser needs them. If you're already using Modernizr, you should omit this reference. Just make sure that your Modernizr build includes the form detection options. These are the options that begin with `forms-` (for example, `forms-validation`) on the "Non-core detects" section of the Modernizr download page.

The second reference is to the HTML5Forms library. After the familiar `src` attribute, you'll see one or more attributes that specify the features you need. In the example above, the script loads all the `webforms` features. HTML5Forms is a modular library, which means you can opt to use only some of its features. This strategy ensures that your pages don't perform any extra work to load up features you don't need. Here's an example that turns on basic support for validation, required fields, and placeholders:

```
<script src="shared/js/html5Forms.js"
  data-webforms2-support="validation,placeholder"
 data-webforms2-force-js-validation="true">
```

TIP If you want to pick a different combination of features, look for a corresponding example page file in the *tests/html5forms* folder that's included with the HTML5Forms download.

HTML5Forms library also adds surprisingly good support for the form features you'll learn about next, like the slider, date picker, and color chooser. Still, there are inevitable gaps and minor bugs buried in the code. If you plan to use these newer controls, you should test your site with old browser versions (like IE 9) before you go live.

GEM IN THE ROUGH

A Few Rogue Input Attributes

HTML5 recognizes a few more attributes that can control browser behavior when editing forms, but aren't used for validation. Not all of these attributes apply to all browsers. Still, they make for good experimenting:

- **`spellcheck`**. Some browsers try to help you avoid embarrassment by spell-checking the words you type in an input box. The obvious problem is that not all text is meant to be real words, and there are only so many red squiggles a web surfer can take before getting just a bit annoyed. Set `spellcheck` to `false` to recommend that the browser not spellcheck a field, or `true` to recommend that it does. (Browsers differ on their default spell-checking behavior, which is what you get if you don't set the `spellcheck` attribute at all.)
- **`autocomplete`**. Some browsers try to save you time by offering you recently typed-in values when you enter information in a field. This behavior isn't always appropriate—as the HTML5 specification points out, some types of information may be sensitive (like nuclear attack codes) or may be relevant for only a short amount of time (like a one-time bank login code). In these cases, set `autocomplete` to `off` to recommend that the browser not offer autocomplete suggestions. You can also set `autocomplete` to on to recommend it for a particular field.
- **`autocorrect`** and **`autocapitalize`**. Use these attributes to control automatic correction and capitalization features on some mobile devices—namely, the version of Safari that runs on iPads and iPhones.
- **`multiple`**. Web designers have been adding the `multiple` attribute to the `<select>` element to create multiple-selection lists since time immemorial. But now you can add it to certain types of `<input>` elements, including ones that use the `file` type (for uploading files) and ones that use the `email` type (page 125). On a supporting browser, the user can then pick several files to upload at once, or stick multiple email addresses in one box.

New Types of Input

One of the quirks of HTML forms is that one ingredient—the vaguely named `<input>` element—is used to create a variety of controls, from checkboxes to text boxes to buttons. The `type` attribute is the master switch that determines what each `<input>` element really is.

If a browser runs into an `<input>` element with a type that it doesn't recognize, the browser treats it like an ordinary text box. That means these three elements get exactly the same treatment in every browser:

```
<input type="text">
<input type="super-strange-wonky-input-type">
<input>
```

HTML5 uses this default to its benefit. It adds a few new data types to the `<input>` element, secure in the knowledge that browsers will treat them as ordinary text boxes if they don't recognize them. For example, if you need to create a text box that holds an email address, you can use the new input type `email`:

```
<label for="email">Email <em>*</em></label>
<input id="email" type="email"><br>
```

If you view this page in a browser that doesn't directly support the `email` input type (like Internet Explorer 9), you'll get an ordinary text box, which is perfectly acceptable. But browsers that support HTML5 forms are a bit smarter. Here's what they can do:

- **Offer editing conveniences.** For example, an intelligent browser or a handy JavaScript widget might give you a way to get an email from your address book and pop it into an email field.
- **Restrict potential errors.** For example, browsers can ignore letters when you type in a number text box. Or, they can reject invalid dates (or just force you to pick one from a mini calendar, which is easier *and* safer).
- **Perform validation.** Browsers can perform more sophisticated checks when you click a submit button. For example, an intelligent browser will spot an obviously incorrect email address in an email box and refuse to continue.

The HTML5 specification doesn't give browser makers any guidance on the first point. Browsers are free to manage the display and editing of different data types in any way that makes sense, and different browsers can add different little luxuries. For example, mobile browsers take advantage of this information to customize their virtual keyboards, hiding keys that don't apply (see Figure 4-8).

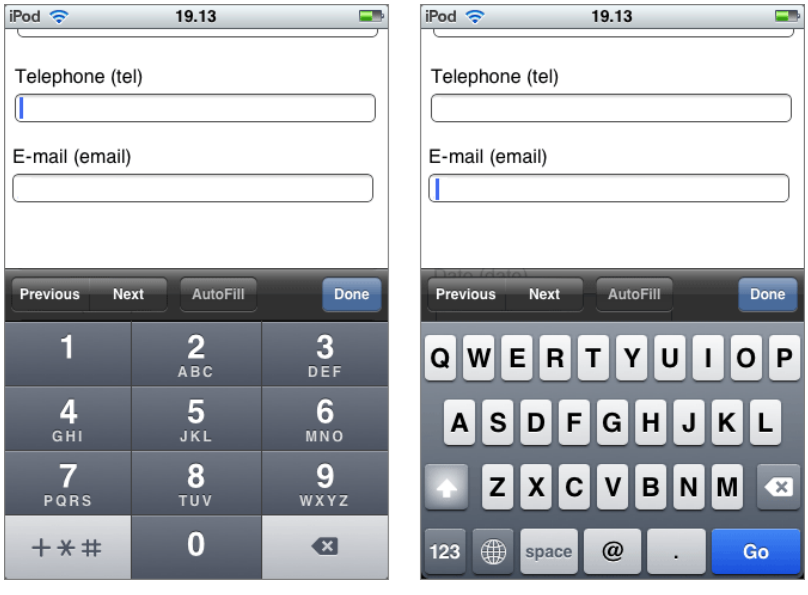


FIGURE 4-8

When people use a mobile device to fill out a form, they don't have the luxury of entering information on a full keyboard. The iPod makes life easier by customizing the virtual keyboard depending on the data type, so telephone numbers get a telephone-style numeric keypad (left), while email addresses get a dedicated @ button and a smaller space bar (right).

The error-prevention and error-checking features are more important. At a bare minimum, a browser that supports HTML5 web forms must prevent a form from being submitted if it contains data that breaks the data type rules. So if the browser doesn't prevent errors (according to the second point in the previous list), it must validate them when the user submits the data (that's the third point).

Unfortunately, not all current browsers live up to this requirement. Some recognize the new data types and provide some sort of editing niceties but no validation. Many understand one data type but not another. Mobile browsers are particularly problematic—they provide some editing conveniences but currently have none of the validation.

Table 4-3 lists the new data types and the browsers that support them completely—meaning that they prevent forms from being submitted when the data type rules are broken.

TABLE 4-3 *Browser compatibility for new input types*

DATA TYPE	IE	FIREFOX	CHROME	SAFARI	OPERA
email	10	4	10	5	10.6
url	10	4	10	5	10.6
search*	n/a	n/a	n/a	n/a	n/a
tel*	n/a	n/a	n/a	n/a	n/a

DATA TYPE	IE	FIREFOX	CHROME	SAFARI	OPERA
`number`	10	-	10	5	9
`range`	10	23	6	5	11
`date`, `month`, `week`, `time`	-	-	10	-	11
`color`	-	-	20	-	-**

* The HTML5 standard does not require validation for this data type.
** Opera supported the color input type in versions 11 and 12, but removed this support in more recent versions.

TIP Incidentally, you can test for data type support in Modernizr using the properties of the `Modernizr.inputtypes` object. For example, `Modernizr.inputtypes.range` returns `true` if the browser supports the `range` data type.

Email Addresses

Email addresses use the `email` type. Generally, a valid email address is a string of characters (with certain symbols not allowed). An email address must contain the @ symbol and a period, and there needs to be at least one character between them and two characters after the period. These are, more or less, the rules that govern email addresses. However, writing the right validation logic or regular expression for email addresses is a surprisingly subtle task that has tripped up many a well-intentioned developer. Which is why it's great to find web browsers that support the `email` data type and perform validation automatically (see Figure 4-9).

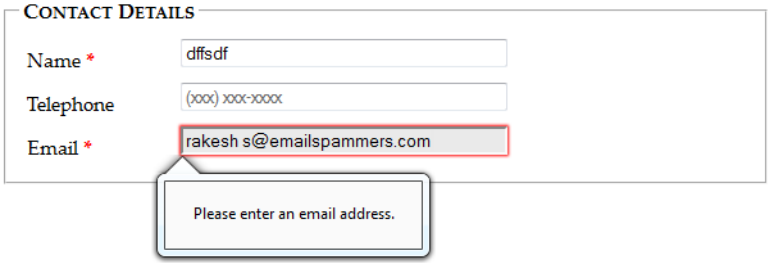


FIGURE 4-9
Firefox refuses to accept the space in this spurious email address.

Email boxes support the `multiple` attribute, which allows the field to hold multiple email addresses. However, these multiple email addresses still look like a single piece of text—you just separate each one with a comma.

NOTE Remember, blank values bypass validation. If you want to force someone to enter a valid email address, you need the `email` data type combined with the `required` attribute (page 112).

URLs

URLs use the `url` type. What constitutes a valid URL is still a matter of hot debate. But most browsers use a relatively lax validation algorithm. It requires a URL prefix (which could be legitimate, like *http://*, or made up, like *bonk://*), and accepts spaces and most special characters other than the colon (:).

Some browsers also show a drop-down list of URL suggestions, which is typically taken from the browser's history of recently visited pages.

Search Boxes

Search boxes use the `search` type. A search box is generally meant to contain keywords that are then used to perform some sort of search. It could be a search of the entire Web (as with Google, in Figure 4-1), a search of a single page, or a custom-built search routine that examines your own catalog of information. Either way, a search box looks and behaves almost exactly like a normal text box.

On some browsers, like Safari, search boxes are styled slightly differently, with rounded corners. Also, as soon as you start typing in a search box in Safari or Chrome, a small X icon appears on the right side that you can click to clear the box. Other than these very minor differences, search boxes *are* text boxes. The value is in the semantics. In other words, you use the `search` data type to make the purpose of the box clear to browsers and assistive software. They can then guide visitors to the right spot or offer other smart features—maybe, someday.

Telephone Numbers

Telephone numbers use the `tel` type. Telephone numbers come in a variety of patterns. Some use only numbers, while others incorporate spaces, dashes, plus signs, and parentheses. Perhaps it's because of these complications that the HTML5 standard doesn't ask browsers to perform any telephone number validation at all. However, it's hard to ignore the feeling that a `tel` field should at least reject letters (which it doesn't).

Right now, the only value in using the `tel` type is to get a customized virtual keyboard on mobile browsers, which focuses on numbers and leaves out letters.

Numbers

HTML5 defines two numeric data types. The `number` type is the one to use for ordinary numbers.

The `number` data type has obvious potential. Ordinary text boxes accept anything: numbers, letters, spaces, punctuation, and the symbols usually reserved for cartoon character swearing. For this reason, one of the most common validation tasks is to check that a given value is numeric and falls in the right range. But use the `number` data type, and the browser automatically ignores all non-numeric keystrokes. Here's an example:

```
<label for="age">Age<em>*</em></label>
<input id="age" type="number"><br>
```

Of course, there are plenty of numbers, and they aren't all appropriate for every kind of data. The markup shown above allows ages like 43,000 and -6. To fix this, you need to use the `min` and `max` attributes. Here's an example that limits ages to the reasonable range of 0 to 120:

```
<input id="age" type="number" min="0" max="120"><br>
```

Ordinarily, the `number` data type accepts only whole numbers, so a fractional age like 30.5 isn't accepted. (In fact, some browsers won't even let you type the decimal point.) However, you can change this too by setting the `step` attribute, which indicates the acceptable intervals for the number. For example, a minimum value of 0 and a step of 0.1 means you can use values like 0, 0.1, 0.2, and 0.3. Try to submit a form with 0.15, however, and you'll get the familiar pop-up error message. The default step is 1.

```
<label for="weight">Weight (in pounds)</label>
<input id="weight" type="number" min="50" max="1000"
step="0.1"value="160"><br>
```

The `step` attribute also affects how the spin buttons work in the number box, as shown in Figure 4-10.

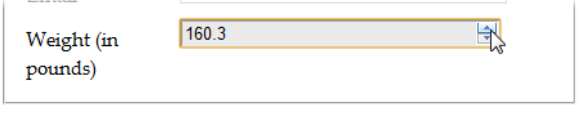


FIGURE 4-10

Many browsers add spin buttons to number boxes. Every time you click the up arrow, the number increases by the step value (unless you reach the maximum allowed number). Similarly, when you click the down arrow, the number decreases by one step.

Sliders

The `range` type is HTML5's other numeric data type. Like the `number` type, it can represent whole numbers or fractional values. It also supports the same attributes for setting the range of allowed values (`min` and `max`). Here's an example:

```
<label for="weight">Weight (in pounds)</label>
<input id="weight" type="range" min="50" max="1000" value="160"><br>
```

The difference is the way the `range` type presents its information. Instead of asking you to type the value you want in a text box, intelligent browsers show a slider control (Figure 4-11).

FIGURE 4-11

You've probably used range-like controls to set things like sound volume. They work well for information where there's a narrow range of values between a known minimum and a known maximum, and the specific numeric value isn't important (but the placement of that value relative to the minimum and maximum is).

To set a `range` type, you simply pull the tab to the position you want, somewhere between the minimum and maximum values at either end of the slider. Browsers that support the `range` type don't give any feedback about the specific value that's set. If you want that piece of information, you need to add a scrap of JavaScript that reacts when the slider changes (perhaps by handling the `onChange` event) and then displays the value nearby. Of course, you'd also want to check if the current browser supports the `range` type (using a tool like Modernizr). If the browser doesn't support the `range` type, there's no need to take any extra steps, because the value will show up in an ordinary text box.

Dates and Times

HTML5 defines several date-related types. Browsers that understand the date types can provide handy drop-down calendars for people to pick from. Not only does this clear away confusion about the right way to format the date, but it also prevents people from accidentally (or deliberately) picking a date that doesn't exist. And smart browsers can go even further—for example, adding integration with a personal calendar.

Right now, the date types are poorly supported, despite their obvious usefulness. Chrome and Opera are the only browsers that provide drop-down calendars (see Figure 4-12). Other browsers ignore date data types and show ordinary, unvalidated text boxes instead.

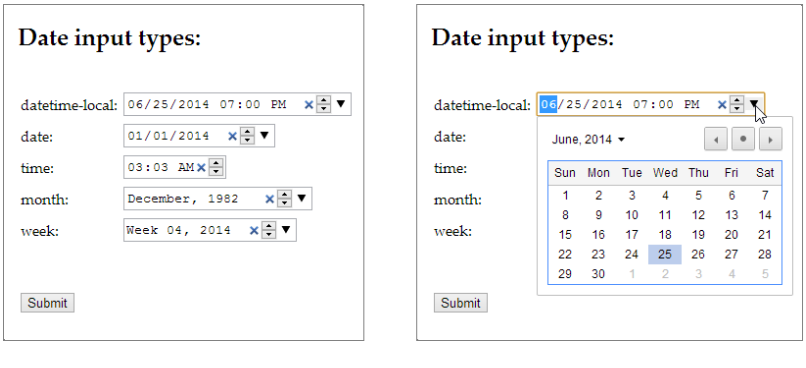


FIGURE 4-12

The `<input>` boxes look slightly different when storing date and time information (left). But the real convenience that supporting browsers provide is the drop-down calendar that lets you set these values with a proper date, and no formatting headaches (right).

TIP If you decide to use one of the date types, consider using a polyfill like HTML5Forms library (page 121) for older browsers. That's because it's easy for people on non-supporting browsers to enter dates in the wrong format, and it's tedious for you to validate date data and provide the appropriate guidance. (That's also why custom JavaScript date controls already exist—and why they're all over the Web.)

Table 4-4 explains the six date formats.

TABLE 4-4 *Date data types*

DATE TYPE	DESCRIPTION	EXAMPLE
`date`	A date in the format *YYYY-MM-DD*.	January 25, 2014: *2014-01-25*
`time`	A 24-hour time with an optional seconds portion, in the format *HH:mm:ss.ss*.	2:35 p.m. (and 50.2 seconds): *14:35* or *14:35:50.2*
`datetime-local`	A date and a time, separated by a capital T (so the format is *YYYY-MM-DDTHH:mm:ss*).	January 25, 2014, 2:35 p.m: *2014-01-15T14:35*
`datetime`	A date and a time, like the `datetime-local` data type, but with a `time-zone` offset. This uses the same format (*YYYY-MM-DD HH:mm:ss-HH:mm*) as the `<time>` element you considered on page 78. However, the `datetime` format is not supported reliably in any browser and may be removed in the future, so use `datetime-local` instead.	January 25, 2014, 2:35 p.m., in New York: *2014-01-15 14:35-05:00*
`month`	A year and month number, in the format *YYYY-MM*.	First month in 2014: *2014-01*
`week`	A year and week number, in the format *YYYY-Www*. Note that there can be 52 or 53 weeks, depending on the year.	Second week in 2014: *2014-W02*

TIP Browsers that support the date types also support the `min` and `max` attributes with them. That means you can set maximum and minimum dates, as long as you use the right date format. So, to restrict a `date` field to dates in the year 2014, you would write `<input type="date" min="2014-01-01" max="2014-12-31">`.

Colors

Colors use the `color` data type. The `color` type is an interesting, albeit little-used, frill that lets a web page visitor pick a color from a drop-down color picker, which looks like what you might see in a desktop paint program. Currently, Chrome is the only browser to add one. Opera had one briefly, in versions 10 and 11, but removed it after deciding it was too experimental.

In browsers that don't support the `color` type, form-fillers will need to type a hexadecimal color code on their own (or you can use the Html5Forms library described on page 121).

New Elements

So far, you've learned how HTML5 extends forms with new validation features and how it gets smarter about data by adding more input types. These are the most practical and the most widely supported new features, but they aren't the end of the HTML5 forms story.

HTML5 also introduces a few entirely new elements to fill gaps and add features. Using these nifty new elements, you can add a drop-down list of suggestions, a progress bar, a toolbar, and more. The problem with new elements is that old browsers are guaranteed not to support them, and even new browsers are slow to wade in when the specification is still changing. As a result, these details include some of the *least* supported features covered in this chapter. You may want to see how they work, but you'll probably wait to use them, unless you're comfortable inching your way out even further into the world of browser quirks and incompatibilities.

Input Suggestions with <datalist>

The `<datalist>` element gives you a way to fuse a drop-down list of suggestions to an ordinary text box. It gives people filling out a form the convenience to pick an option from the list, or the freedom to type exactly what they want (see Figure 4-13).

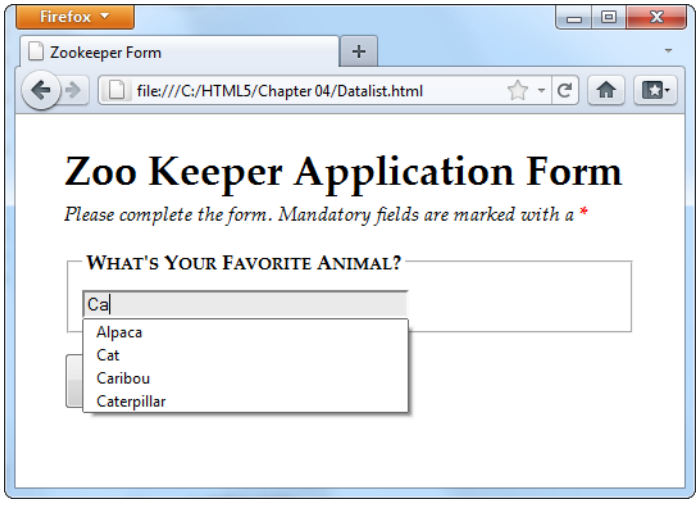


FIGURE 4-13

As you type, the browser shows you potential matches. For example, type in the letters "ca," and the browser shows you every animal that has that letter sequence somewhere in its name (and not necessarily at the beginning).

To use a datalist, you must first start with a standard text box. For example, imagine you have an ordinary <input> element like this:

```
<legend>What's Your Favorite Animal?</legend>
<input id="favoriteAnimal">
```

To add a drop-down list of suggestions, you need to create a datalist. Technically, you can place that <datalist> element anywhere you want. That's because the datalist can't display itself—instead, it simply provides data that an input box will use. However, it makes logical sense to place the <datalist> element just after or just before the <input> element that uses it. Here's an example:

```
<datalist id="animalChoices">
  <option label="Alpaca" value="alpaca">
  <option label="Zebra" value="zebra">
  <option label="Cat" value="cat">
  <option label="Caribou" value="caribou">
  <option label="Caterpillar" value="caterpillar">
  <option label="Anaconda" value="anaconda">
  <option label="Human" value="human">
  <option label="Elephant" value="elephant">
  <option label="Wildebeest" value="wildebeest">
  <option label="Pigeon" value="pigeon">
  <option label="Crab" value="crab">
</datalist>
```

The datalist uses <option> elements, just like the traditional <select> element. Each <option> element represents a separate suggestion that may be offered to the form filler. The label shows the text that appears in the text box, while the value tracks the text that will be sent back to the web server, if the user chooses that option. On its own, a datalist is invisible. To hook it up to a text box so it can start providing suggestions, you need to set the list attribute to match the ID of the corresponding datalist:

```
<input id="favoriteAnimal" list="animalChoices">
```

Current versions of Chrome, Internet Explorer, Firefox, and Opera support the datalist. They'll show the list of possible matches shown in Figure 4-13. But Safari, older versions of Internet Explorer (IE 9 and before), and mobile browsers will ignore the list attribute and the datalist markup, rendering your suggestions useless.

However, there's a clever fallback trick that makes other browsers behave properly. The trick is to put other content inside the datalist. This works because browsers that support the datalist pay attention to <option> elements only, and ignore all other content. Here's a revised example that exploits this behavior. (The bold parts are the markup that datalist-supporting browsers will ignore.)

```
<legend>What's Your Favorite Animal?</legend>
<datalist id="animalChoices">
  <span class="Label">Pick an option:</span>
```

```
    <select id="favoriteAnimalPreset">
     <option label="Alpaca" value="alpaca">
      <option label="Zebra" value="zebra">
      <option label="Cat" value="cat">
      <option label="Caribou" value="caribou">
      <option label="Caterpillar" value="caterpillar">
      <option label="Anaconda" value="anaconda">
      <option label="Human" value="human">
      <option label="Elephant" value="elephant">
      <option label="Wildebeest" value="wildebeest">
      <option label="Pigeon" value="pigeon">
      <option label="Crab" value="crab">
    </select>
    <br>
    <span class="Label">Or type it in:</span>
  </datalist>
  <input list="animalChoices" name="list">
```

If you remove the bold markup, you end up with the same markup you had before. That means browsers that recognize the datalist still show the single input box and the drop-down suggestion list, as shown in Figure 4-13. But on other browsers, the additional details wrap the datalist suggestion in a traditional select list, giving users the option of typing in what they want or picking it from a list (Figure 4-14).

FIGURE 4-14

You can still use your suggestions on browsers that don't support the datalist. But you need to wrap them in a `<select>` *list first.*

This effect isn't completely seamless. When you receive the form data on the web server, you need to check for data from the list (in this example, that's `favoriteAnimalPreset`) and from the text box (that's `favoriteAnimal`). But other than this minor wrinkle, you've got a solid way to add new conveniences without leaving anyone behind.

NOTE When the datalist was first created, it had a feature that let it fetch suggestions from somewhere else—for example, it could call a web server, which could then pull a list of suggestions out of a database. This feature might still be added in a future version of the HTML standard, but for now it's possible only if you write JavaScript code to handle the fetching, with the help of the `XMLHttpRequest` object (page 377).

Progress Bars and Meters

The `<progress>` and `<meter>` element are two new graphical widgets that look similar but serve different purposes (Figure 4-15).

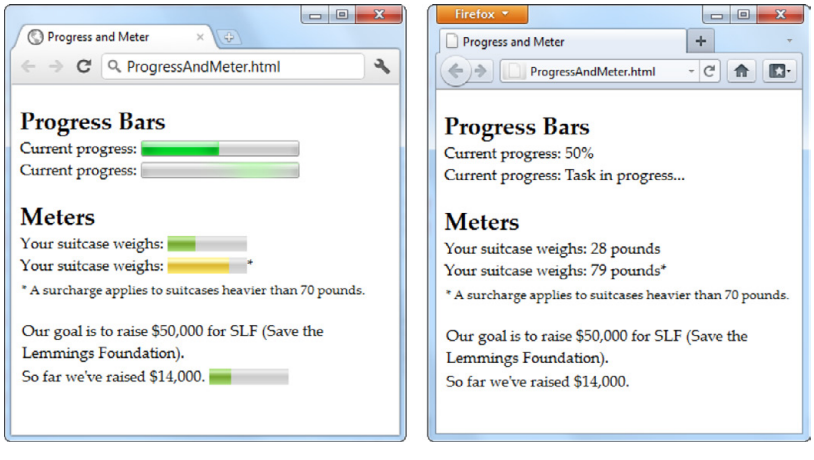


FIGURE 4-15
The `<progress>` and `<meter>` elements are nice visual conveniences, on the browsers that support them (left). Other browsers simply substitute the fallback content you've set (right).

The `<progress>` element indicates how far a task has progressed. It uses a gray background that's partially filled in with a pulsating green bar. The `<progress>` element resembles the progress bars you've probably seen before (for example, when the Windows operating system is copying files), although its exact appearance depends on the browser being used to view the page.

The `<meter>` element indicates a value within a known range. It looks similar to the `<progress>` element, but the green bar is a slightly different shade, and it doesn't pulse. Depending on the browser, the meter bar may change color when a value is classified as "too low" or "too high"—for example, in the latter case Chrome changes the bar from green to yellow. But the most important difference between `<progress>` and `<meter>` is the semantic meaning that the markup conveys.

NOTE Technically, the new <meter> and <progress> elements don't need to be in a form. In fact, they aren't even real controls (because they don't collect input from the web page visitor). However, the official HTML5 standard lumps them all together, because in some respects the <progress> and <meter> elements *feel* like form widgets, probably because they display bits of data in a graphical way.

The latest versions of all major browsers support the <progress> and <meter> elements. However, you'll run into trouble on older versions of Internet Explorer (IE 9 and before) and some mobile browsers. To guarantee support for everyone, you'll need to polyfill this feature with something like HTML5Forms (page 121).

Using the <progress> and <meter> elements is easy. First, consider the <progress> element. It takes a value attribute, which sets the percentage of progress (and thus the width of the green fill) as a fractional value from 0 to 1. For example, you could set the value to 0.25 to represent a task that's 25 percent complete:

```
<progress value="0.25"></progress>
```

Alternatively, you can use the max attribute to set an upper maximum and change the scale of the progress bar. For example, if max is 200, your value needs to fall between 0 and 200. If you set it to 50, you'd get the same 25 percent fill as in the previous example:

```
<progress value="50" max="200"></progress>
```

The scale is simply a matter of convenience. The web page viewer doesn't see the actual value in the progress bar.

NOTE The <progress> element is simply a way to display a nicely shaded progress bar. It doesn't actually *do* anything. For example, if you're using the progress bar to show the progress of a background task (say, using web workers, as demonstrated on page 414), it's up to you to write the JavaScript code that grabs hold of the <progress> element and changes its value.

Browsers that don't recognize the <progress> element simply ignore it. To deal with this problem, you can put some fallback content inside the <progress> element, like this:

```
<progress value="0.25">25%</progress>
```

Just remember that the fallback content won't appear in browsers that *do* support the <progress> element.

There's one other progress bar option. You can show an *indeterminate* progress bar, which indicates that a task is under way, but you aren't sure how close it is to completion. (Think of an indeterminate progress bar as a fancy "in progress" message.) An indeterminate progress bar looks like an empty gray bar but has a periodic green flash travel across it, from left to right. To create one, just leave out the value attribute, like this:

```
<progress>Task in progress ...</progress>
```

The <meter> element has a similar model, but it indicates any sort of measurement. The <meter> element is sometimes described as a *gauge*. Often, the specific meter value you use will correspond to something in the real world (for example, an amount of money, a number of days, an amount of weight, and so on). To control how the <meter> element displays this information, you're able to set both a minimum and maximum value (using the min and max attributes):

```
Your suitcase weighs: <meter min="5" max="70" value="28">28 pounds</meter>
```

Once again, the content inside the <meter> element is shown only if the browser doesn't know how to display a meter bar. Of course, sometimes it's important to show the specific number that the <meter> element uses. In this case, you'll need to add it to the page yourself, and you don't need the fallback content. The following example uses this approach. It provides all the information up front and adds an optional <meter> element on browsers that support it:

```
<p>Our goal is to raise $50,000 for SLF (Save the Lemmings Foundation).</p>
<p>So far we've raised $14,000. <meter max="50000" value="14000"></meter>
```

The <meter> element also has the smarts to indicate that certain values are too high or too low, while still displaying them properly. To do this, you use the low and high attributes. For example, a value that's above high (but still below max) is higher than it should be, but still allowed. Similarly, a value that's below low (but still above min) is lower than it should be:

```
Your suitcase weighs:
<meter min="5" max="100" high="70" value="79">79 pounds</meter>*
<p><small>* A surcharge applies to suitcases heavier than 70 pounds.
</small></p>
```

Browsers may or may not use this information. For example, Chrome shows a yellow bar for overly high values (like the one in the previous example). It doesn't do anything to indicate low values. Finally, you can flag a certain value as being an optimal value using the optimum attribute, but it won't change the way it shows up in today's browsers.

All in all, <progress> and <scale> are minor conveniences that will be useful once their browser support improves just a bit.

Toolbars and Menus with <command> and <menu>

Count this as the greatest feature that's not yet implemented. The idea is to have an element that represents actions the user can trigger (that's <command>) and another one to hold a group of them (that's <menu>). Depending on how you put it together and what styling tricks you use, the <menu> element could become anything from a toolbar docked to the side of the browser window to a pop-up menu that appears when you click somewhere on the page. But right now, no browser supports these elements, and so you'll have to wait to find out if they're really as cool as web developers hope.

An HTML Editor in a Web Page

As you learned in Chapter 1, HTML5 believes in paving cowpaths—in other words, taking the unstandardized features that developers use today and making them part of the official HTML5 standard. One of the single best examples is its inclusion of two odd attributes, named `contenteditable` and `designMode`, which let you turn an ordinary browser into a basic HTML editor.

These two attributes are nothing new. In fact, they were originally added to Internet Explorer 5 in the dark ages of the Internet. At the time, most developers dismissed them as more Windows-only extensions to the Web. But as the years wore on, more browsers began to copy IE's practical but quirky approach to rich HTML editing. Today, every desktop browser supports these attributes, even though they have never been part of an official standard.

UP TO SPEED

When to Use HTML Editing

Before you try out rich HTML editing, it's worth asking what the feature is actually for. Despite its immediate cool factor, HTML editing is a specialized feature that won't appeal to everyone. It makes most sense if you need a quick-and-easy way for users to edit HTML content—for example, if you need to let them add blog posts, enter reviews, post classified ads, or compose messages to other users.

Even if you decide you need this sort of feature, the `contenteditable` and `designMode` attributes might not be your first choice. That's because they don't give you all the niceties of a real web design tool, like markup-changing commands, the ability to view and edit the HTML source, spell-checking, and so on. Using HTML's rich editing feature, you *can* build a much fancier editor, with a bit of work. But if you really need rich editing functionality, you may be happier using someone else's ready-made editor, which you can then plug into your own pages. Popular examples include TinyMCE (*www.tinymce.com*) and CKEditor (*http://ckeditor.com*).

Using contenteditable to Edit an Element

The first tool you have for HTML editing is the `contenteditable` attribute. Add it to any element and set its value to `true` to make the content of that element editable:

```
<div id="editableElement" contenteditable="true">You can edit this text, if
you'd like.</div>
```

You probably won't notice the difference at first. But if you load your page in a browser and click inside that `<div>`, a text-editing cursor (called a *caret*) will appear (Figure 4-16).

In this example, the editable `<div>` contains nothing but text. However, you could just as easily put other elements inside. In fact, this `<div>` element could wrap your entire page, making the whole thing editable. Similarly, you could use `contenteditable` on multiple elements to make several sections of a page editable.

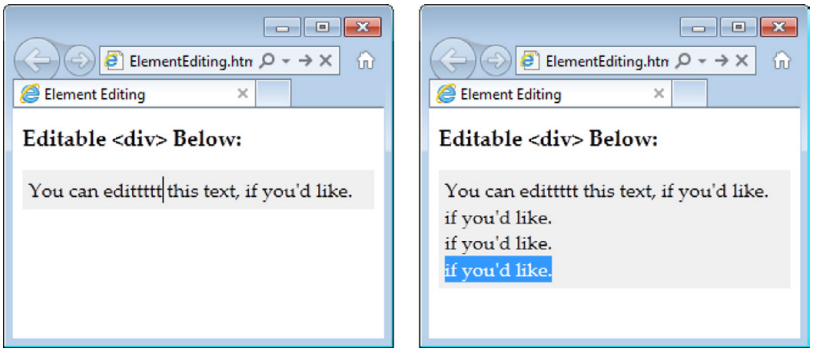


FIGURE 4-16

When you click in an editable region, you can move around using the arrow keys, delete text, and insert new content (left). You can also select text with the Shift key and then copy, cut, and paste it (right). It's a bit like typing in a word processor, only you won't be able to escape the confines of the to get to the rest of the page.

TIP Some browsers support a few built-in commands. For example, you can get bold, italic, and underline formatting in IE using the shortcut keys Ctrl+B, Ctrl+I, and Ctrl+U. Similarly, you can reverse your last action in Firefox by pressing Ctrl+Z, and you can use all of these shortcuts in Chrome. To learn more about these editing commands and how you can create a custom toolbar that triggers them, see Opera's two-part article series at *http://tinyurl.com/htmlEdit1* and *http://tinyurl.com/htmlEdit2*.)

Usually, you won't set `contenteditable` in your markup. Instead, you'll turn it on using a bit of JavaScript, and you'll turn it off when you want to finish editing. Here are two functions that do exactly that:

```
function startEdit() {
  // Make the element editable.
  var element = document.getElementById("editableElement");
  element.contentEditable = true;
}

function stopEdit() {
  // Return the element to normal.
  var element = document.getElementById("editableElement");
  element.contentEditable = false;

  // Show the markup in a message box.
  alert("Your edited content: " + element.innerHTML);
}
```

And here are two buttons that use them:

```
<button onclick="startEdit()">Start Editing</button>
<button onclick="stopEdit()">Stop Editing</button>
```

Just make sure you don't place the buttons in the editable region of your page, because when a page is being edited, its elements stop firing events and won't trigger your code.

Figure 4-17 shows the result after the element has been edited and some formatting has been applied (courtesy of the Ctrl+B shortcut command).

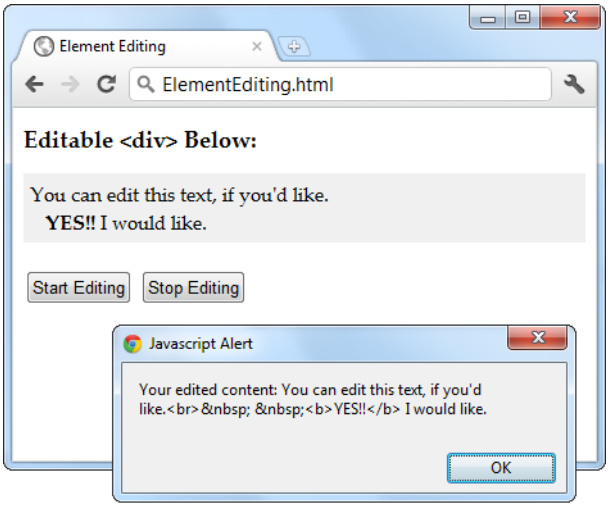


FIGURE 4-17

This proves it—editing an element really does change the in-memory content of the page. In this example, the new content is simply shown in a message box. However, a more typical page would send this data to a web server, probably using the `XMLHttpRequest` *object described on page 377.*

NOTE There are subtle differences in the way rich HTML editing works in different browsers. For example, pressing Ctrl+B in Chrome adds a `<b>` element, while pressing it in IE adds the `<strong>` element. Similar variations occur when you hit the Enter key to add a new line or Backspace to delete a tag. One of the reasons that it makes sense for HTML5 to standardize the rich HTML editing feature is the ability to enforce better consistency.

Using designMode to Edit a Page

The `designMode` property is similar to `contenteditable`, except it allows you to edit an entire web page. This may seem like a bit of a problem—after all, if the whole page is editable, how will the user click buttons and control the editing process? The solution is to put the editable document inside an `<iframe>` element, which then acts as a super-powered editing box (Figure 4-18).

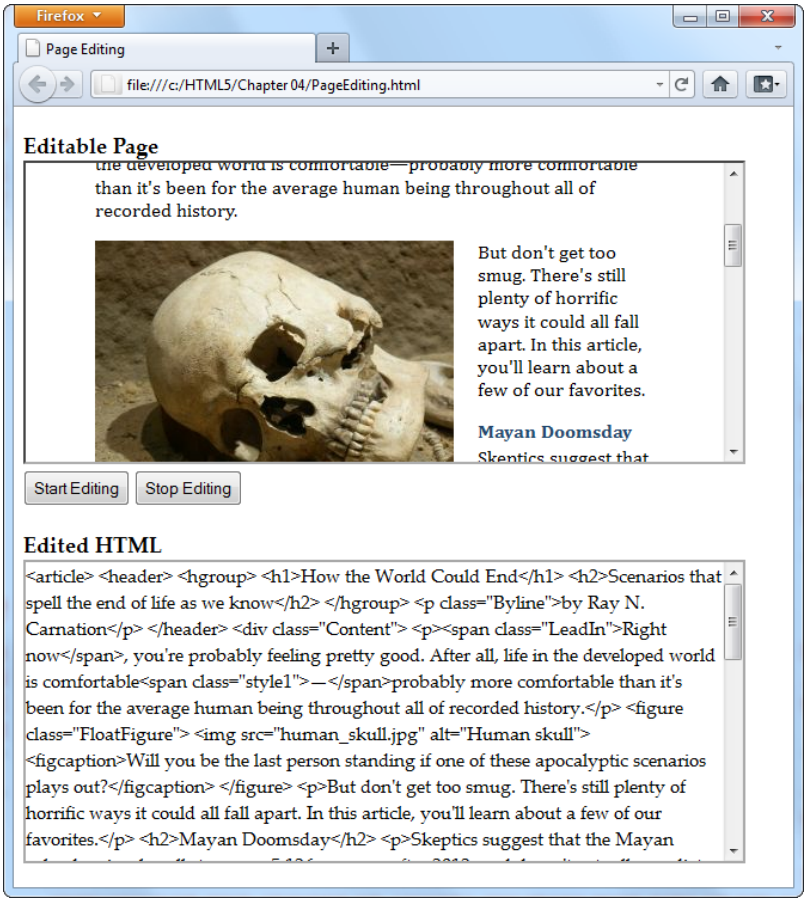


FIGURE 4-18

This page contains two boxes. The first is an `<iframe>` that shows the apocalypse page example from Chapter 2. The second is an ordinary `<div>` that shows the HTML markup of the page after it's been edited. The two buttons at the top of the page control the show, switching the `<iframe>` into design mode when the user is ready to work.

The markup in this page is refreshingly simple. Here are the complete contents of the <body> element in the page:

```
<h1>Editable Page</h1>
<iframe id="pageEditor" src="ApocalypsePage_Revised.html"></iframe>
<div>
  <button onclick="startEdit()">Start Editing</button>
  <button onclick="stopEdit()">Stop Editing</button>
</div>

<h1>Edited HTML</h1>
<div id="editedHTML"></div>
```

As you can see, this example relies on the `startEdit()` and `stopEdit()` methods, much like the previous example. However, the code is tweaked so that it sets the `designMode` attribute rather than the `contenteditable` attribute:

```
function startEdit() {
  // Turn on design mode in the <iframe>.
  var editor = document.getElementById("pageEditor");
  editor.contentWindow.document.designMode = "on";
}

function stopEdit() {
  // Turn off design mode in the <iframe>.
  var editor = document.getElementById("pageEditor");
  editor.contentWindow.document.designMode = "off";

  // Display the edited HTML (just to prove it's there).
  var htmlDisplay = document.getElementById("editedHTML");
  htmlDisplay.textContent =   editor.contentWindow.document.body.innerHTML;
}
```

This example gives you a better idea of the scope of the rich editing feature. For example, click on a picture and you'll see how the browser lets you manipulate it. You can resize it, drag it to a new place, or delete it completely with a single click of the Delete button. You'll have similar power over form controls, if they're in the page you're editing.

Of course, there's still a significant gap you'll need to cross if you want to turn this example into something practical. First, you'll probably want to add better editing controls. Once again, the helpful folks at Opera have your back if you're ready to make a deeper exploration into the command model, which is beyond the scope of this chapter (see *http://tinyurl.com/htmlEdit1* and *http://tinyurl.com/htmlEdit2*). Second, you'll need to do something useful with your edited markup, like sending it to your web server via `XMLHttpRequest` (page 377).

There's one more caveat to note. If you run this example locally from your hard drive, it won't work in all browsers. (Internet Explorer and Chrome run into security restrictions, while Firefox sails ahead without a problem.) To avoid the problem, you can run it from the try-out site at *http://prosetech.com/html5*.

PART 2

Video, Graphics, and Glitz

CHAPTER 5:
Audio and Video

CHAPTER 6:
Fancy Fonts and Effects with CSS3

CHAPTER 7:
Responsive Web Design with CSS3

CHAPTER 8:
Basic Drawing with the Canvas

CHAPTER 9:
Advanced Canvas: Interactivity and Animation

CHAPTER

5

Audio and Video

There was a time when the Internet was primarily a way to share academic research. Then things changed and the Web grew into a news and commerce powerhouse. Today the Internet's state-of-the-art networking technology is used less for physics calculations and more for spreading viral videos of piano-playing kittens across the planet. And network colossus Cisco reports that the trend isn't slowing down, estimating that a staggering *80 percent* of all Internet traffic will be video by 2017.

Amazingly, this monumental change happened despite the fact that—up until now—the HTML language had no built-in support for video or audio. Instead, Web surfers of the recent past relied on the Flash plug-in, which worked for most people, most of the time. But Flash has a few key gaps, including the fact that Apple devices (like iPhones and iPads) refuse to support it.

HTML5 solves these problems by adding the `<audio>` and `<video>` elements that HTML has been missing all these years. However, the transition to HTML5 audio and video has been far from seamless. Browser makers spent a few years locked in a heated name-calling, finger-pointing format war. The good news today is that much of the dust has settled, and HTML5 audio and video have become good choices for even the most cautious web developer.

The Evolution of Web Video

Without HTML5, you have a couple of ways to add video to a web page. One old-fashioned approach is to shoehorn it into a page with the `<embed>` element. The browser then creates a video window that uses Windows Media Player, Apple QuickTime, or some other video player, and places it on the page.

The key problem with this technique is that it puts you in a desolate no-man's-land of browser support. You have no way to control playback, you may not be able to buffer the video to prevent long playback delays, and you have no way of knowing whether your video file will be playable at all on different browsers or operating systems.

The second approach is to use a browser plug-in—like Microsoft's relative newcomer, Silverlight, or the overwhelming favorite, Adobe Flash. Up until recently, Flash had the problem of browser support solved cold. After all, Flash video works everywhere the Flash plug-in is installed, and currently that's on more than 99 percent of Internet-connected computers. Flash also gives you nearly unlimited control over the way playback works. For example, you can use someone else's prebuilt Flash video player for convenience, or you can design your own and customize every last glowy button.

But the Flash approach isn't perfect. To get Flash video into a web page, you need to throw down some seriously ugly markup that uses the `<object>` and `<embed>` elements. You need to encode your video files appropriately, and you may also need to buy the high-priced Flash developer software and learn to use it, and the learning curve can be steep. But the worst problem is Apple's mobile devices—the iPhone and iPad. They refuse to tolerate Flash at all, slapping blank boxes over the web page regions that use it.

NOTE Plug-ins also have a reputation for occasional unreliability. That's because of the way they work. For example, when you visit a page that uses Flash, the browser lets the Flash plug-in take control of a rectangular box somewhere on the web page. Most of the time, this hands-off approach works well, but minor bugs or unusual system configurations can lead to unexpected interactions and glitches, like suddenly garbled video or pages that suck up huge amounts of computer memory and slow your web surfing down to a crawl.

Still, if you watch video on the Web today, and you aren't using an iPhone or iPad, odds are that it's wrapped in a Flash mini-application. If you're not sure, try right-clicking the video player. If the menu that pops up includes a command like "About Flash Player 11," then you know you're dealing with the ubiquitous Flash plug-in. And even when you move to HTML5, you'll probably still need a Flash-powered fallback for browsers that aren't quite there yet, like Internet Explorer 8.

NOTE YouTube provides a trial HTML5 video player at *www.youtube.com/html5*. Everywhere else, YouTube sticks exclusively with Flash. The exception is if you visit YouTube using an iPhone or iPad, in which case YouTube is smart enough to switch to properly supported HTML5 video automatically.

Introducing HTML5 Audio and Video

A simple idea underpins HTML5's audio and video support. Just as you can add images to a web page with the <img> element, you should be able to insert sound with an <audio> element and video with a <video> element. Logically enough, HTML5 adds both.

UP TO SPEED

Turn Back Now If...

Unfortunately, some things are beyond HTML5's new audio and video capabilities. If you want to perform any of these tricks, you'll need to scramble back to Flash (at least for now):

- **Licensed content.** HTML5 video files don't use any sort of copy protection system. In fact, folks can download HTML5 videos as easily as downloading pictures—with a simple right-click of the mouse. That said, digital rights management features are currently under development and slated for inclusion in HTML 5.1.
- **Video or audio recording.** HTML5 has no way to stream audio or video from your computer to another computer. So if you want to build a web chat program that uses the microphones and webcams of your visitors, stick with Flash. The creators of HTML5 are experimenting with a <device> element that might serve the same purpose, but for now there's no HTML-only solution, in any browser.
- **Adaptive video streaming.** Advanced, video-heavy websites like YouTube need fine-grained control over video streaming and buffering. They need to provide videos in different resolutions, stream live events, and adjust the video quality to fit the bandwidth of the visitor's Internet connection. Until HTML5 can provide these features, video-sharing sites may add HTML5 support, but they won't completely switch from Flash.
- **Low-latency, high-performance audio.** Some applications need audio to start with no delay or they need to play multiple audio clips in perfect unison. Examples include a virtual synthesizer, music visualizer, or a real-time game with plenty of overlapping sound effects. And while browser makers are hard at work improving HTML5's audio performance, it still can't live up to these demands.
- **Dynamically created or edited audio.** What if you could not just play recorded audio, but also analyze audio information, modify it, or generate it in real time? New specifications, like the experimental Web Audio API (*http://tinyurl.com/web-audio-API*), are competing to add on these sorts of features to HTML5 audio, but they aren't here yet.

Making Some Noise with <audio>

Here's an example of the <audio> element at its absolute simplest:

```
<p>Hear us rock out with our new song,
<cite>Death to Rubber Duckies</cite>:</p>
<audio src="rubberduckies.mp3" controls></audio>
```

The src attribute provides the file name of the audio file you want to play. The controls attribute tells the browser to include a basic set of playback controls. Each browser has a slightly different version of these controls, but they always serve the same purpose—to let the user start and stop playback, jump to a new position, and change the volume (Figure 5-1).

NOTE The <audio> and <video> elements must have both a start and an end tag. You can't use empty element syntax, like <audio />.

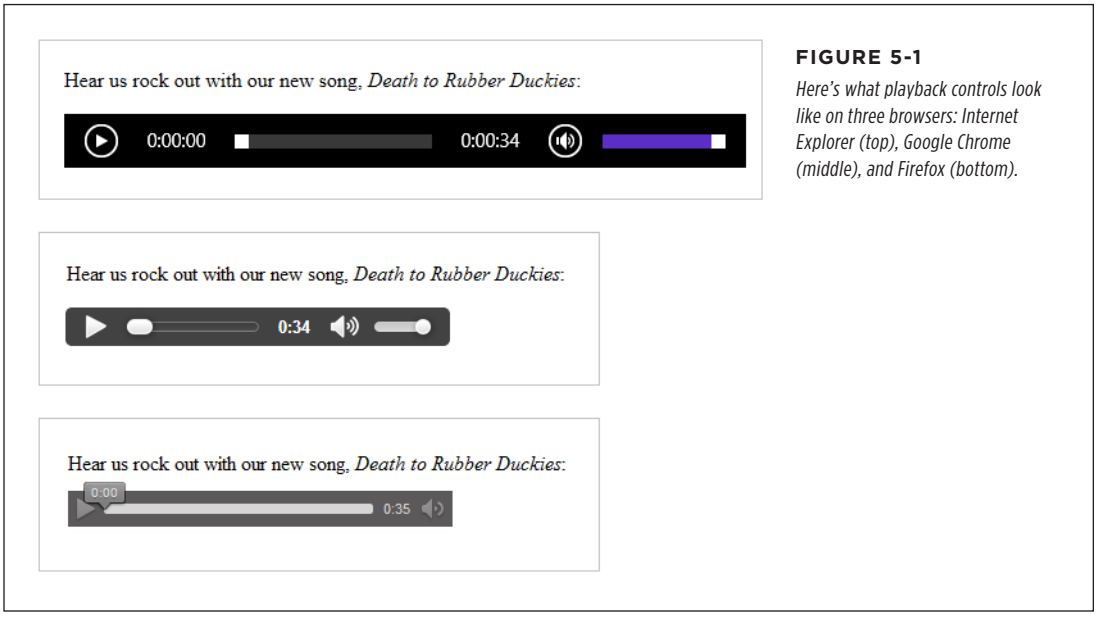


FIGURE 5-1
Here's what playback controls look like on three browsers: Internet Explorer (top), Google Chrome (middle), and Firefox (bottom).

Along with the basic src and controls attributes, the <audio> element supports several other attributes, which are detailed in the following sections.

Preloading Media Files

One useful attribute is preload, which tells the browser how it should download a media file. Set preload to auto to tell the browser to start downloading the whole file, so it's available when the user clicks the play button. Of course, this download process unfolds in the background, so your web page visitor can scroll around and read the page without waiting for the download to finish.

The preload attribute also supports two other values. You can use metadata to tell the browser to grab the first small chunk of data from the file, which is enough to determine some basic details (like the total length of the audio). Or, you can use none, which tells the browser to hold off completely. You might use one of these options to save bandwidth, for example, if you have a page stuffed full of <audio> elements and you don't expect the visitor to play more than a few of them.

```
<audio src="rubberduckies.mp3" controls preload="metadata"></audio>
```

When you use the none or metadata values, the browser downloads the audio file as soon as someone clicks the play button. Happily, browsers can play one chunk of audio while downloading the next without a hiccup unless you're working over a slow network connection.

If you don't set the `preload` attribute, browsers can do what they want, and different browsers make different assumptions. Most browsers assume `auto` as the default value, but Firefox uses `metadata`. Furthermore, it's important to note that the `preload` attribute isn't a rigid rule, but a recommendation you're giving to the browser—one that may be ignored depending on other factors. (And some slightly older browser builds don't pay attention to the `preload` attribute at all.)

NOTE If you have a page stuffed with `<audio>` elements, the browser creates a separate strip of playback controls for each one. The web page visitor can listen to one audio file at a time or start them all playing at once.

Automatic Playback

Next up is the `autoplay` attribute, which tells the browser to start playback immediately once the page has finished loading. It looks like this:

```
<audio src="rubberduckies.mp3" controls autoplay></audio>
```

Without `autoplay`, it's up to the person viewing the page to click the play button.

You can use the `<audio>` element to play background music unobtrusively, or even to provide the sound effects for a browser-based game. To get background music, remove the `controls` attribute and add the `autoplay` attribute (or use JavaScript-powered playback, as described on page 160). But use this approach with caution, and remember that your page still needs some sort of audio shutoff switch.

NOTE No one wants to face a page that blares music or sound effects but lacks a way to shut the sound off. If you decide to use the `<audio>` element without the `controls` attribute, you *must*, at a bare minimum, add a mute button that uses JavaScript to silence the audio.

Looping Playback

Finally, the `loop` attribute tells the browser to start over at the beginning when playback ends:

```
<audio src="rubberduckies.mp3" controls loop></audio>
```

In most browsers, playback is fluid enough that you can use this technique to create a seamless, looping soundtrack. The trick is to choose a loopable piece of audio that ends where it begins. You can find hundreds of free examples at *www.flashkit.com/loops*. (These loops were originally designed for Flash but can also be downloaded in MP3 and WAV versions.)

Getting the Big Picture with <video>

The `<video>` element pairs nicely with the `<audio>` element. Here's a straightforward example that puts it to use:

```
<p>A butterfly from my vacation in Switzerland!</p>
<video src="butterfly.mp4" controls></video>
```

Once again, the `controls` attribute gets the browser to generate a set of handy playback controls (Figure 5-2). In most browsers, these controls disappear when you click somewhere else on the page and return when you hover over the movie.

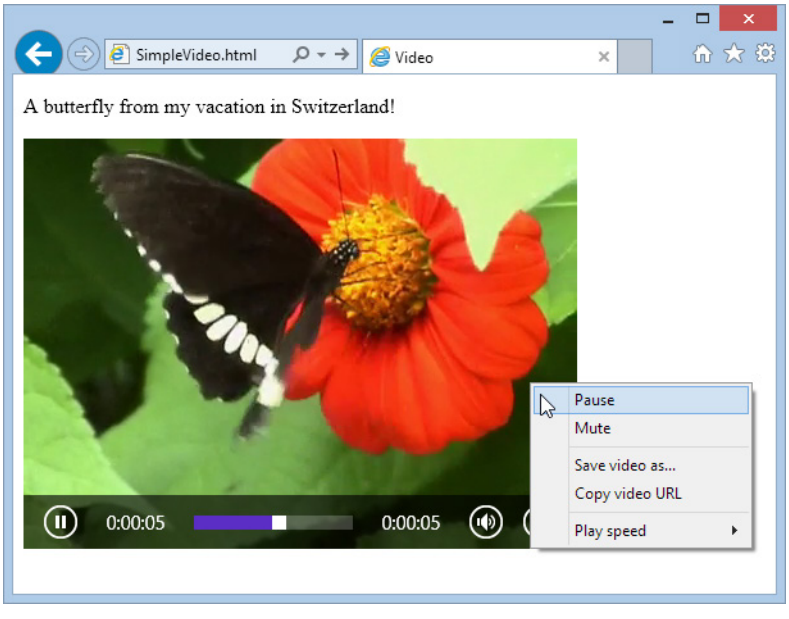


FIGURE 5-2

The `<video>` element could easily be mistaken for a Flash video window. But if you right-click the `<video>` element, you'll get a simpler menu that includes the option to save the video file to your computer. Depending on the browser, it may also include commands for changing the playback speed, looping the video, taking it full screen, and muting the sound.

The `<video>` element has the same `src`, `controls`, `preload`, `autoplay`, and `loop` attributes as the `<audio>` element. However, if you choose to enable automatic playback, you can make it less obnoxious by throwing in the `muted` attribute, which shuts off the sound on most browsers. The viewer can switch the audio back on by clicking the speaker icon, as usual.

The `<video>` element also adds three more attributes: `height`, `width`, and `poster`.

The `height` and `width` attributes set the size of the video window (in pixels). Here's an example that creates a video box that measures 400 x 300 pixels:

```
<video src="butterfly.mp4" controls width="400" height="300"></video>
```

This should match the natural size of the video itself, but you might choose to indicate these details explicitly so your layout doesn't get messed up before the video loads (or if the video fails to load altogether).

NOTE No matter which dimensions you use to size the video box, the video *frame* always keeps its proper proportions. For example, if you take a 400 x 300 pixel video and put it in a 800 x 450 pixel video box, you'll get the biggest video frame that fits in the box without stretching, which is 600 x 450 pixels. This leaves 100 pixels on each side of the video frame, which appear as blank space.

Finally, the `poster` attribute lets you supply an image that should be used in place of the video. Browsers use this picture in three situations: if the first frame of the video hasn't been downloaded yet, if you've set the `preload` attribute to `none`, or if the selected video file wasn't found.

```
<video src="butterfly.mp4" controls poster="swiss_alps.jpg"></video>
```

Although you've now learned everything there is to know about audio and video markup, there's a lot more you can do with some well-placed JavaScript. But before you can get any fancier with the `<audio>` and `<video>` elements, you need to face the headaches of audio and video codec support.

GEM IN THE ROUGH

Media Groups

The HTML5 standard specifies an unusual attribute named `mediagroup` that applies to both the `<audio>` element and the `<video>` element. You can use the `mediagroup` attribute to link multiple media files together, so their playback is synchronized. All you need to do is assign the same `mediagroup` name (which can be whatever you want) to each `<audio>` or `<video>` element:

```
<video src="shot12_cam1.mp4" controls
mediagroup="shot12"></video>
<video src="shot12_cam2.mp4" controls
mediagroup="shot12"></video>
```

Now if the viewer presses play in the first video window (for *shot12_cam1.mp4*), playback begins in both windows at once.

The `mediagroup` attribute might be a useful tool if you need to synchronize concurrent video files—for example, recordings of a sporting event taken from different angles. You can also use it to synchronize audio and video, which is useful if you need to choose different audio tracks based on the visitor's language or accessibility needs. For example, an audio track for visually impaired users might add a voiceover that describes the action that's taking place. To accomplish this wizardry, you'd create a page with several hidden `<audio>` elements, give each one a different `mediagroup` name, and then add a bit of simple JavaScript that sets the `mediagroup` of your `<video>` element to match the `mediagroup` of the right `<audio>` element, based on the visitor's requirements.

Unfortunately, `mediagroup` isn't much use right now, because its browser support is still limited. Chrome and Opera understand it, but the latest versions of Internet Explorer and Firefox ignore it completely.

Understanding the HTML5 Media Formats

If the `<video>` and `<audio>` elements seem too good to be true, well, sometimes they are. The problem is that a media file format that works flawlessly in one browser can flummox another.

The examples you've just considered use two popular standards: MP3 audio and H.264 video. They're enough to keep most browsers happy. But use them on the Opera browser, and they won't work (Figure 5-3).

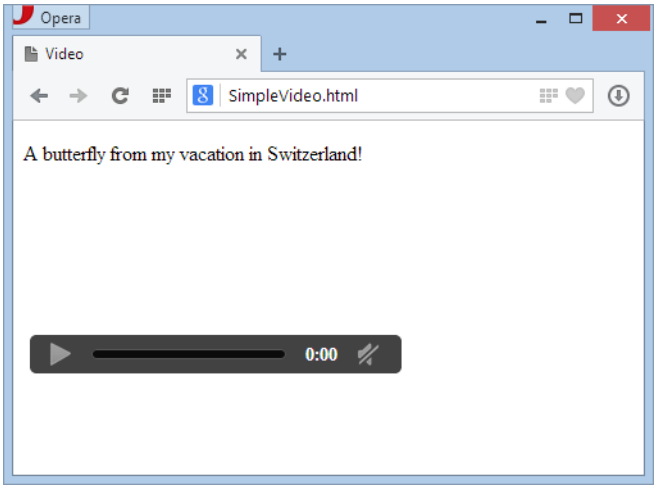


FIGURE 5-3
If you load a page that uses an H.264 video in Opera, the playback controls are disabled, and a blank space appears where the video should be.

Fortunately, you can solve this problem with a format fallback, as you'll see on page 155. But before you learn how to do that, you need to take a closer look at the range of audio and video formats on the Web today, and the current state of browser support.

Meet the Media Formats

The official HTML5 standard doesn't require support for any specific video or audio format. (Early versions did, but the recommendation was dropped after intense lobbying.) As a result, browser makers are free to choose the formats *they* want to support, despite the fact that they're congenitally unable to agree with one another.

Table 5-1 shows the standards that they're using right now.

TABLE 5-1 *Some of the audio and video standards that HTML5 browsers may support*

FORMAT	DESCRIPTION	COMMON FILE EXTENSION	MIME TYPE
MP3	The world's most popular audio format.	.mp3	audio/mp3
Ogg Vorbis	A free, open standard that offers high-quality, compressed audio comparable to MP3.	.ogg	audio/ogg
WAV	The original format for raw digital audio. Doesn't use compression, so files are staggeringly big and unsuitable for most web uses.	.wav	audio/wav

FORMAT	DESCRIPTION	COMMON FILE EXTENSION	MIME TYPE
H.264	The industry standard for video encoding, particularly when dealing with high-definition video. Used by consumer devices (like Blu-ray players and camcorders), web sharing websites (like YouTube and Vimeo), and web plug-ins (like Flash and Silverlight).	.mp4	video/mp4
Ogg Theora	A free, open standard for video by the creators of the Vorbis audio standard. Byte for byte, the quality and performance doesn't match H.264, although it's still good enough to satisfy most people.	.ogv	video/ogg
WebM	The newest video format, created when Google purchased VP8 and transformed it into a free standard. Critics argue that the quality isn't up to the level of H.264 video—yet—and that it may have unexpected links to other people's patents, which could lead to a storm of lawsuits in the future.	.webm	video/webm

Table 5-1 also lists the recommended file extensions your media files should use. To realize why this is important, you need to understand that there are actually three standards at play in a video file. First, and most obviously, is the *video codec*, which compresses the video into a stream of data (examples include H.264, Theora, and WebM). Second is the *audio codec*, which compresses one or more tracks of audio using a related standard. (For example, H.264 generally uses MP3, while Theora uses Vorbis.) Third is the *container format*, which packages everything together with some descriptive information and, optionally, other frills like still images and subtitles. Often, the file extension refers to the container format, so .mp4 signifies an MPEG-4 container, .ogv signifies an Ogg container, and so on.

Here's the tricky part: Most container formats support a range of different video and audio standards. For example, the popular Matroska container (.mkv) can hold video that's encoded with H.264 or Theora. To keep your head from exploding, Table 5-1 puts each video format with the container format that's most common and has the most reliable web support.

Table 5-1 also indicates the proper MIME type, which must be configured on your web server. If you use the wrong MIME type, browsers may stubbornly refuse to play a perfectly good media file. (If you're a little fuzzy on exactly what MIME types do and how to configure them, see the box on page 152.)

UP TO SPEED

MIME Types and Why to Use Them

A *MIME type* (sometimes called a *content type*) is a piece of information that identifies the type of content in a web resource. For example, the MIME type of a web page is *text/html*.

Before a web server sends a resource to a browser, it sends the MIME type. For example, if a browser asks for the page SuperVideoPlayerPage.html, the web server sends the *text/html* MIME type, a few other pieces of information, and the actual file content. When the browser receives the MIME type, it knows what to do with the content that comes next. It doesn't need to try to make a guess based on a file name extension or some other sort of hackery.

For common file types—for example, HTML pages and images—you don't need to worry about MIME types, because every web server already handles them properly. But some web servers might not be configured with the MIME types for audio and video. That's a problem, because browsers will be thrown off course if the web server sends a media file with the wrong MIME type. Usually, they won't play the file at all.

To avoid this problem, make sure your web server is set up with the MIME types listed in Table 5-1, and use the corresponding file extensions for your audio and video files. (It's no use configuring the MIME type and then using the wrong file extension, because the web server needs to be able to pair the two together. For example, if you configure .mp4 files to use the MIME type *video/mp4*, but then you give your video file the extension *.mpFour*, the web server won't have a clue what you're trying to do.)

Configuring MIME types is an easy job, but the exact steps depend on your web hosting company (or your web server software, if you're hosting your site yourself). If your web hosting company uses the popular cPanel interface, then look for an icon named MIME Types and click it. You'll then see a page like the one shown in Figure 5-4. And if you're in any doubt, contact your web hosting company for help.

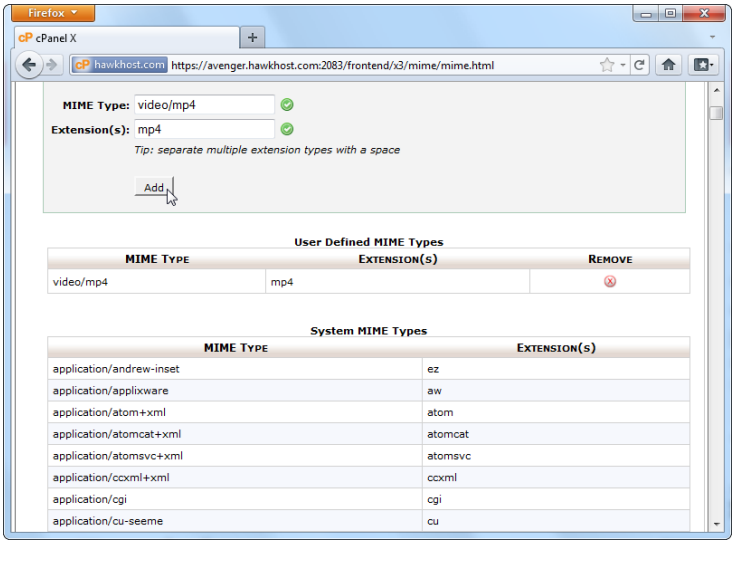


FIGURE 5-4

Here a new MIME type is being added to support H.264 video files. In many cases, you won't need to take this step, because your website will already be configured correctly.

Browser Support for Media Formats

The format headaches in HTML5 have a long history. The conflict is fueled by the different needs of browser makers. Small companies, like Mozilla (the creators of Firefox) and Opera (the creators of the Opera browser) don't want to pay stiff licensing costs for popular standards like MP3 audio and H.264 video. And it's hard to blame them—after all, they are giving away their work for free.

But bigger companies, like Microsoft and Apple, have their own reasonable-sounding excuses for shunning unlicensed standards. They complain that these standards won't perform as well (they currently lack hardware acceleration) and aren't as widespread (unlike H.264, which is used in camcorders, Blu-ray players, and a host of other devices). But the biggest problem is that these unlicensed standards may have obscure ties to someone else's intellectual property. If they do, and if big companies like Microsoft and Apple start using them, they open themselves up to pricey patent lawsuits that could drag on for years.

Fortunately, the situation is improving. In 2013, Firefox gave in and agreed to support MP3 and H.264. Google, despite threatening to remove support for H.264 in Chrome, has never taken that step and now seems unlikely to do so. Opera remains the last holdout on the desktop—for now. For the full details of browser media support, see Table 5-2 (for audio formats) and Table 5-3 (for video formats).

TABLE 5-2 *Browser support for HTML5 audio formats*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	ANDROID
MP3	9	21	5	3.1	-	3	2.3
Ogg Vorbis	-	3.6	5	-	10.5	-	-
WAV	-	3.6	8	3.1	10.5	-	-

TABLE 5-3 *Browser support for HTML5 video formats*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	ANDROID
H.264 Video	9	21	5	3.1	-	4*	2.3
Ogg Theora	-	3.5	5	-	10.5	-	-
WebM	-	4	6	-	10.6	-	2.3

* iOS 3.x supports video, but there are subtle video bugs hiding in older versions of the Safari browser. For example, if you set the `poster` attribute (page 149), you may find that the video becomes unplayable.

Mobile browsers have their own quirks. Some don't support features like autoplay and looping, because these features can drain batteries and use up valuable bandwidth. But even if you don't plan to use these features, mobile devices need special consideration to ensure good video playback performance and to minimize data

usage. To make mobile-friendly videos, you should encode them with lower quality settings and, possibly, with a lower resolution.

TIP As a general rule of thumb, if you want a video to be playable on a mobile device, you should encode it using the H.264 Baseline Profile (rather than High Profile). For iPhone and Android phones, use a size of 640 x 480 or smaller (and stick to 480 x 360 if you want to play it on a BlackBerry). Many encoding programs (see the box on page 156) have presets that let you prepare mobile-optimized video.

FREQUENTLY ASKED QUESTION

H.264 Licensing

I'm using H.264 for my videos. Do I have to pay licensing costs?

If you're using an H.264 decoder in your product (for example, you're creating a browser that can play H.264-encoded video), you definitely need to pay. But if you're a video provider, it's less clear cut.

First, the good news. If you're using H.264 to make free videos, you won't be asked to pay anything, ever. If you're creating videos that have a commercial purpose but aren't actually being sold (say, you're shooting a commercial or promoting yourself in an interview), you're also in the clear.

If you're *selling* H.264-encoded video content on your website, you may be on the hook to pay license fees to MPEG-LA, either now or in the future. Right now, the key detail is the number of subscribers. If you have fewer than 100,000, there's no licensing cost, but if you have 100,000 to 250,000, you're expected to cough up $25,000 a year. This probably won't seem like much bank for a video- selling company of that size, and it may pale in comparison to other considerations, like the cost of professional encoding tools. However, these numbers could change when the licensing terms are revised in 2016. Big companies looking to make lots of money in web video might prefer to use an open, unlicensed video standard like Theora or WebM.

For the full licensing legalese on H.264, visit *http://tinyurl.com/h264-lic*.

Fallbacks: How to Please Every Browser

At this writing, an H.264 video file presented by the HTML5 `<video>` element works for over 80 percent of the people surfing the Web. This percentage is impressive, but it isn't good enough on its own. To create a video that *everyone* can see, you need the help of a fallback.

There are two types of fallbacks that web developers use with HTML5 video. The first is a *format fallback*. This mechanism, which is built into HTML5, lets you swap out one type of media file—say, an MP3 file—and replace it with a file in another format (for example, Ogg Vorbis). This type of fallback solves the Opera problem shown on page 150. However, it won't help when your page meets an old browser that doesn't support HTML5's media features, like Internet Explorer 8.

The second type of fallback is a *technology fallback*. If the browser that's processing your page doesn't support the `<video>` and `<audio>` element, your page can substitute a time-tested Flash player that does the job.

Conscientious web developers use *both* types of fallback. More time-constrained (or lazier) web developers sometimes omit the format fallback, in order to eliminate the work of re-encoding their video files. After all, the Opera browser (the lone desktop browser that doesn't support H.264) accounts for a mere 1 percent of worldwide browser use, and developers speculate that Opera may eventually be forced to add H.264 support. On the other hand, the Flash fallback is easier to implement, because it uses the same media file, and it fills in the support gap for a larger portion of browsers, like that dinosaur IE 8. So ignore the Flash fallback at your own peril.

The following sections explain both types of fallbacks.

Supporting Multiple Formats

The `<audio>` and `<video>` elements have a built-in format fallback system. To use it, you must remove the `src` attribute from the `<video>` or `<audio>` element, and replace it with a list of nested `<source>` elements inside. Here's an example with the `<audio>` element:

```
<audio controls>
  <source src="rubberduckies.mp3" type="audio/mp3">
  <source src="rubberduckies.ogg" type="audio/ogg">
</audio>
```

Here, the same `<audio>` element holds two `<source>` elements, each of which points to a separate audio file. The browser then chooses the first file it finds that has a format it supports. Firefox and Opera will grab *rubberduckies.ogg*. Internet Explorer, Safari, and Chrome will stick with *rubberduckies.mp3.* Unfortunately, it's up to you to encode your content in every alternate format you want to support—a process that wastes time, CPU power, and disk space.

In theory, a browser can determine whether or not it supports a file by downloading a chunk of it. But a better approach is to use the `type` attribute to supply the correct MIME type (see page 152). That way, the browser will attempt to download only a file it believes it can play. (To figure out the correct MIME type, consult Table 5-1.)

The same technique works for the `<video>` element. Here's an example that supplies the same video file twice, once encoded with H.264 and once with WebM, guaranteeing support for all HTML5-aware browsers:

```
<video controls width="700" height="400">
  <source src="beach.mp4" type="video/mp4">
  <source src="beach.webm" type="video/webm">
</video>
```

In this example, there's one new detail to note. When using multiple video formats, the H.264-encoded file should always come first. Otherwise, it won't work for old iPads running iOS 3.x. (The problem has since been fixed in iOS 4, but there's no disadvantage to keeping H.264 in the top spot.)

NOTE Just because a browser believes it supports a specific type of audio or video doesn't necessarily mean it can play it. For example, you may have used an insanely high bitrate, or a strange codec in a recognized container format. You can deal with issues like these by supplying type *and codec* information through the `type` attribute, but it'll make a mess of your markup. The HTML5 spec has all the gruesome details at *http://tinyurl.com/media-types*.

If you're really ambitious, you may opt to create a single video page that's meant for both desktop browsers and mobile devices. In this case, you not only need to worry about the H.264 and WebM video formats, but you also need to think about creating low-bandwidth versions of your video files that are suitable for devices that have less hardware power and use slower Internet connections. To make sure mobile devices get the lighter-weight video files while desktop browsers get the higher-quality ones, you need to write some crafty JavaScript or use *media queries*, as explained on page 231.

UP TO SPEED

Encoding Your Media

Now you know what combination of formats to use, but you don't necessarily know how to transform your media files into those formats. Don't despair, as there are plenty of tools. Some work on entire batches of files at once, some have a reputation for professional-grade quality (and a price tag to match), and some do their work on powerful web servers so you don't have to wait. The trick is picking through all the choices to get the encoder that works for you.

Here are some of your options:

- **Audio editors.** If you're looking to edit WAV files and save them in the MP3 or Vorbis formats, a basic audio editor can help out. Audacity (*http://audacity.sourceforge.net*) is a free editor for Mac and Windows that fits the bill, although you'll need to install the LAME MP3 encoder to get MP3 support (*http://lame1.buanzo.com.ar*). Goldwave (*www.goldwave.com*) is a similarly capable audio editor that's free to try, but sold for a nominal fee.
- **Miro Video Converter.** This free, open-source program runs on Windows and Mac OS X. It can take virtually any video file and convert it to WebM, Theora, or H.264. It also has presets that match the screen sizes and supported formats for mobile devices, like iPads, iPhones, or Android phones. The only downside is that you can't tweak more advanced options to control how the encoding is done. To try it out, go to *www.mirovideoconverter.com*.
- **Firefogg.** This Firefox plug-in (available at *http://firefogg.org*) can create Theora or WebM video files, while giving you a few more options than Miro. It also runs right inside your web browser (although it does all its work locally, without involving a web server).
- **HandBrake.** This open-source, multi-platform program (available at *http://handbrake.fr*) converts a wide range of video formats into H.264 (and a couple of other modern formats).
- **Zencoder.** Here's an example of a professional media encoding service that you can integrate with your website. Zencoder (*http://zencoder.com*) pulls video files off your web server, encodes them in all the formats and bitrates you need, gives them the names you want, and places them in the spot they belong. A big player (say, a video sharing site) would pay Zencoder a sizable monthly fee.

Adding a Flash Fallback

The format fallback system has a key limitation: It works only on browsers that understand the <audio> and <video> elements (which is almost every browser in circulation today, except IE 8). To get your pages to work on non-HTML5 browsers, you need to add a Flash fallback.

To understand how the Flash fallback works, you first need to know that every web browser since the dawn of time deals with the tags it doesn't recognize in the same way—it ignores them. For example, if Internet Explorer 8 comes across the opening tag for the <video> element, it barrels merrily on, without bothering to check the src attribute. However, browsers don't ignore the *content* inside an unrecognized element, which is a crucial difference. It means if you have markup like this:

```
<video controls width="400" height="300">
  <source src="discoParty.mp4" type="video/mp4">
  <source src="discoParty.webm" type="video/webm">
  <p>We like disco dancing.</p>
</video>
```

Browsers that don't understand HTML5 will act as though they saw this:

```
<p>We like disco dancing.</p>
```

This fallback content provides a seamless way to deal with older browsers.

NOTE Browsers that support HTML5 audio ignore the fallback section, even if they can't play the media file. For example, consider what happens if Opera finds a <video> element that uses an H.264 video file but doesn't support Theora. In this situation, the video player won't show anything at all.

So now that you know how to add fallback content, you need to decide what your fallback content should include. One example of bad fallback content is a text message (as in, "Your browser does not support HTML5 video, so please upgrade."). Website visitors consider this sort of comment tremendously impolite, and they're likely never to return when they see it.

The proper thing to include for fallback content is another working video window—in other words, whatever you'd use in an ordinary, non-HTML5 page. One possibility is a YouTube video window. If you use this approach, you need to meet YouTube's rules (make sure your video is less than 15 minutes and doesn't contain offensive or copyrighted content). You can then upload your video to YouTube in the best format you have on hand, and YouTube will re-encode the video into the formats it supports. To get started, head to *www.youtube.com/my_videos_upload*.

Another possibility is to use a Flash video player. (Or, if you're playing audio, a Flash audio player.) Happily, the world has plenty of Flash players. Many of them are free, at least for noncommercial uses. And best of all, most support H.264, a format you're probably already using for HTML5 video.

Here's an example that inserts the popular Flowplayer Flash (*http://flash.flowplayer.org*) into an HTML5 <video> element:

```
<video controls width="700" height="400">
  <source src="beach.mp4" type="video/mp4">
  <source src="beach.webm" type="video/webm">

  <object id="flowplayer" width="700" height="400"
   data="flowplayer-3.2.16.swf"
   type="application/x-shockwave-flash">
    <param name="movie" value="flowplayer-3.2.16.swf">
    <param name="flashvars" value='config={"clip":"beach.mp4"}'>
  </object>
</video>
```

Here, the bold part is a parameter that the browser passes to the Flowplayer Flash, with the file name of the video file. As you can see, even though this example has three possible outcomes (HTML5 video with H.264, HTML5 video with WebM, or Flash video with H.264), it needs only *two* video files, which saves on the encoding work. Figure 5-5 shows the result in action.

Of course, some people won't have Flash or a browser that supports HTML5. You can offer them another fallback, such as a link to download the video file and open it in an external program. You place that fallback after the Flash content, but still inside the <object> element, like this:

```
<video controls width="700" height="400">
  <source src="beach.mp4" type="video/mp4">
  <source src="beach.webm" type="video/webm">

  <object id="flowplayer" width="700" height="400"
   data="http://releases.flowplayer.org/swf/flowplayer-3.2.16.swf"
   type="application/x-shockwave-flash">
    <param name="movie" value="beach.mp4">
    <img src="beach_thumbnail.jpg" alt="A lazy day at the beach">
    <p>Your browser does not support HTML5 video or Flash.</p>
    <p>You can download the video in <a href="beach.mp4">MP4 H.264</a>
    or <a href="beach.webm">WebM</a> format.</p>
  </object>
</video>
```

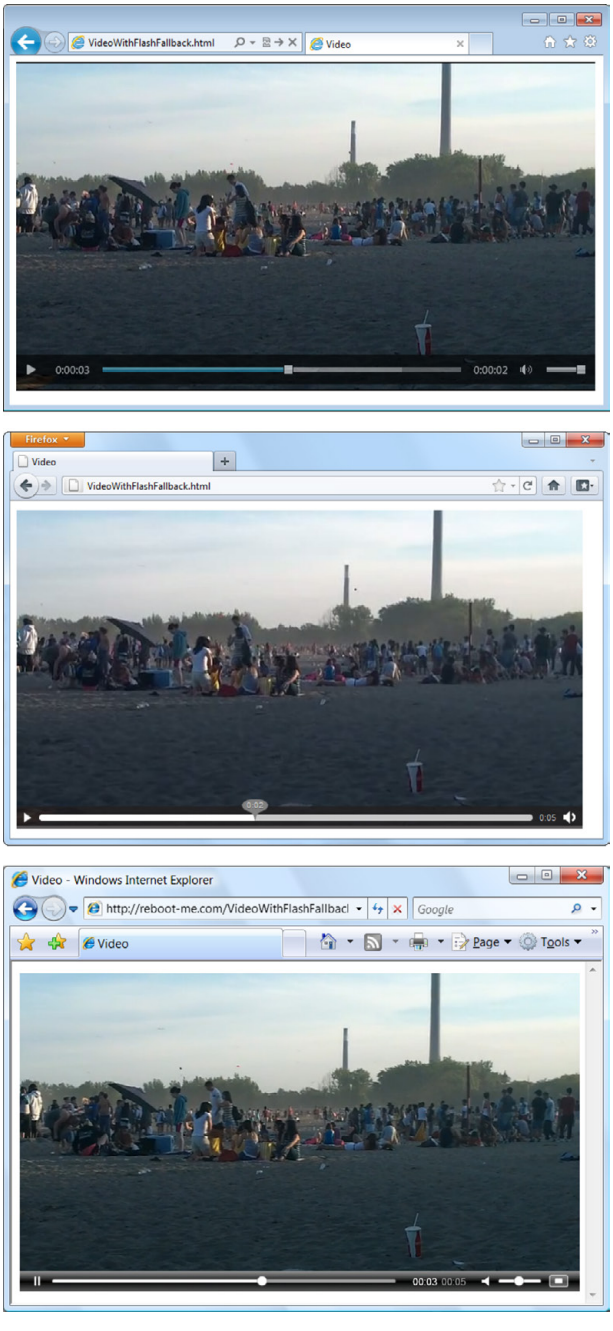


FIGURE 5-5

One video, served three ways: in IE 9 (top), in Firefox (middle), and in IE 7, with Flash (bottom).

Interestingly, there's another way to implement a Flash fallback. The examples you've seen so far use HTML5 with a Flash fallback, which gives everybody HTML5 video (or audio) except for people with older browsers, who get Flash. However, you can invert this approach and use Flash first, with an HTML5 fallback. This gives everybody Flash, except for those who don't have it installed. This strategy makes sense if you're already showing video content on your website with a mature Flash video player, but you want to reach out to iPad and iPhone users. You might also choose this approach if your media requirements go beyond what HTML5 currently supports (as detailed in the box on page 145).

If you want a Flash player with an HTML fallback, you simply need to invert the previous example. Start with the `<object>` element, and nestle the `<video>` element inside, just before the closing `</object>` tag. Place the fallback content just after the last `<source>` element, like this:

```
<object id="flowplayer" width="700" height="400"
 data="http://releases.flowplayer.org/swf/flowplayer-3.2.16.swf"
 type="application/x-shockwave-flash">
  <param name="movie" value="butterfly.mp4">

  <video controls width="700" height="400">
    <source src="beach.mp4" type="video/mp4">
    <source src="beach.webm" type="video/webm">

    <img src="beach_thumbnail.jpg" alt="A lazy day at the beach">
    <p>Your browser does not support HTML5 video or Flash.</p>
    <p>You can download the video in <a href="beach.mp4">MP4 H.264</a>
    or <a href="beach.webm">WebM</a> format.</p>
  </video>
</object>
```

Incidentally, there are a number of JavaScript players that support HTML5 directly and have a built-in Flash fallback. For example, Flowplayer provides another version called Flowplayer HTML5 (get it at *http://flowplayer.org*), which uses the HTML5 `<video>` element if the browser supports it and performs a Flash fallback automatically if needed. The advantage is that this approach simplifies your markup, because one ingredient (the JavaScript-powered media player) handles everything. The disadvantage is that it takes you further away from a pure HTML5 solution, which is what you'll want to use one day soon when HTML5-loving browsers are ubiquitous.

Controlling Your Player with JavaScript

So far, you've covered some heavy ground. You've learned how to take the new `<audio>` and `<video>` elements and turn them into a reasonably supported solution that works on *more* web pages than today's Flash-based players. Not bad for a bleeding-edge technology.

That's about the most you can do with the <audio> and <video> elements if you stick to markup only. But both elements have an extensive JavaScript object model, which lets you control playback with code. In fact, you can even adjust some details—like playback speed—that aren't available in the browser's standard audio and video players.

In the following sections, you'll explore the JavaScript support by considering two practical examples. First, you'll add sound effects to a game. Next, you'll create a custom video player. And finally, you'll consider the solutions that other people have developed using this potent mix of HTML5 and JavaScript, including supercharged, skinnable players and accessible captioning.

Adding Sound Effects

The <audio> element doesn't just let web visitors play songs and voice recordings. It's also a useful tool for playing sound effects, whenever you need them. This makes it particularly useful if you need to add music and sound effects to a game.

Figure 5-6 shows a very simple example, with an interactive ball-dropping animation. You'll see the code that makes this example work when you consider the <canvas> element in Chapter 8. But for now, the only important detail is how you can add a suitable sonic backdrop.

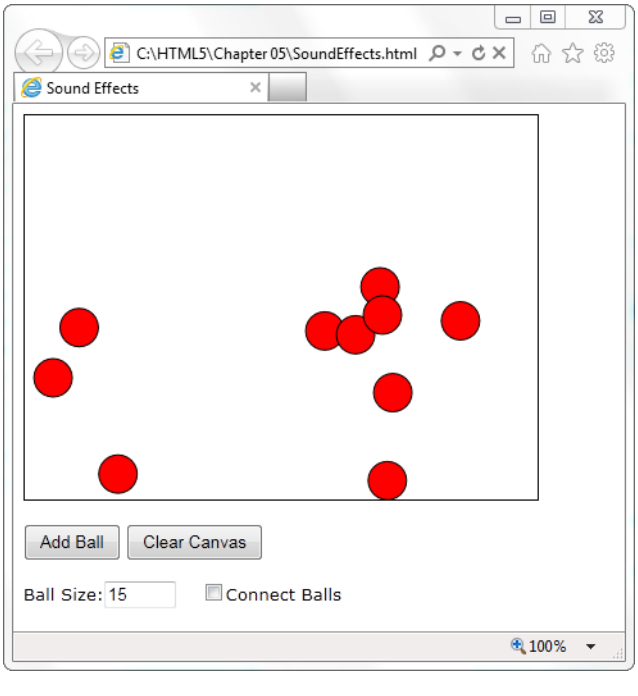


FIGURE 5-6

This web page runs a simple animation on a canvas. The visitor can click a button to add a new ball (which then falls down and bounces around the drawing surface). Or, the visitor can click a ball to send it bouncing in a new direction.

This example combines a background music track with sound effects. The background music track is the easiest part. To create it, you start by adding an invisible <audio> element to your page, like this:

```
<audio id="backgroundMusic" loop>
  <source src="TheOwlNamedOrion.mp3" type="audio/mp3">
  <source src="TheOwlNamedOrion.ogg" type="audio/ogg">
</audio>
```

This audio player doesn't include the autoplay or controls attributes, so initially it's silent and invisible. It does use the loop attribute, so once it starts playing it will repeat the music track endlessly. To control playback, you need to use two methods of the audio (or video) object: play() and pause(). Confusingly, there's no stop method—for that, you need to pause the video and then reset the currentTime property to 0, which represents the beginning of the file.

With this in mind, it's quite easy to start playback on the background audio when the first ball is created:

```
var audioElement = document.getElementById("backgroundMusic");
audioElement.play();
```

And just as easy to stop playback when the canvas is cleared:

```
var audioElement = document.getElementById("backgroundMusic");
audioElement.pause();
audioElement.currentTime = 0;
```

As you learned earlier, there's no limit on the amount of audio you can play at once. So while the background audio is playing its tune, you can concentrate on the more interesting challenge of adding sound effects.

In this example, a "boing" sound effect is played every time a ball ricochets against the ground or a wall. To keep things interesting, several slightly different boing sounds are used. This is a stand-in for a more realistic game, which would probably incorporate a dozen or more sounds.

There are several ways to implement this design, but not all of them are practical. The first option is to add a single new <audio> element to play sound effects. Then, every time a collision happens, you can load a different audio file into that element (by setting the src property) and play it. This approach hits two obstacles. First, a single <audio> element can play only a single sound at once, so if more than one ball hits the ground in quick succession, you need to either ignore the second, overlapping sound, or interrupt the first sound to start the second one. The other problem is that setting the src property forces the browser to request the audio file. And while some browsers will do this quickly (if the audio file is already in the cache), Internet Explorer doesn't. The result is laggy audio—in other words, a boing that happens half a second after the actual collision.

A better approach is to use a group of <audio> elements, one for each sound. Here's an example:

```
<audio id="audio1">
  <source src="boing1.mp3" type="audio/mp3">
  <source src="boing1.wav" type="audio/wav">
</audio>
<audio id="audio2">
  <source src="boing2.mp3" type="audio/mp3">
  <source src="boing2.wav" type="audio/wav">
</audio>
<audio id="audio3">
  <source src="boing3.mp3" type="audio/mp3">
  <source src="boing3.wav" type="audio/wav">
</audio>
```

NOTE Even though these three <audio> elements use different audio files, that isn't a requirement. For example, if you wanted to have the same boing sound effect but allow overlapping audio, you'd still use three audio players.

Whenever a collision happens, the JavaScript code calls a custom function named boing(). That method grabs the next <audio> element in the sequence and plays it.

Here's the code that makes it happen:

```
// Keep track of the number of <audio> elements.
var audioElementCount = 3;

// Keep track of the <audio> element that's next in line for playback.
var audioElementIndex = 1;

function boing() {
  // Get the <audio> element that's next in the rotation.
  var audioElementName = "audio" + audioElementIndex;
  var audio = document.getElementById(audioElementName);

  // Play the sound effect.
  audio.currentTime = 0;
  audio.play();

  // Move the counter to the next <audio> element.
  if (audioElementIndex == audioElementCount) {
    audioElementIndex = 1;
  }
  else {
    audioElementIndex += 1;
  }
}
```

TIP To get an idea of the noise this page causes with its background music and sound effects, visit the try-out site at *http://prosetech.com/html5*.

This example works well, but what if you want to have a much larger range of audio effects? The easiest choice is to create a hidden `<audio>` element for each one. If that's impractical, you can dynamically set the `src` property of an existing `<audio>` element. Or, you can create a new `<audio>` element on the fly, like this:

```
var audio = document.createElement("audio");
audio.src = "newsound.mp3";
```

Or use this shortcut:

```
var audio = new Audio("newsound.mp3");
```

However, there are two potential problems with both approaches. First, you need to set the source well before you play the audio. Otherwise, playback will be noticeably delayed, particularly on Internet Explorer. Second, you need to know what the supported audio formats are, so you can set the right file type. This requires using the clunky `canPlayType()` method. You pass in an audio or video MIME type, and `canPlayType()` tells you if the browser can play that format—sort of. It actually returns a blank string if it can't, the word "probably" if it thinks it can, and the word "maybe" if it hopes it might but just can't make any promises. This rather embarrassing situation exists because supported container formats can use unsupported codecs, and supported codecs can still use unsupported encoding settings.

Most developers settle on code like this, which attempts playback if `canPlayType()` gives any answer other than a blank string:

```
if (audio.canPlayType("audio/ogg")) {
  audio.src = "newsound.ogg";
}
else if (audio.canPlayType("audio/mp3")) {
  audio.src = "newsound.mp3";
}
```

Creating a Custom Video Player

One of the most common reasons to delve into JavaScript programming with the `<audio>` and `<video>` elements is to build your own player. The basic idea is pure simplicity—remove the `controls` attribute, so that all you have is a video window, and add your own widgets underneath. Finally, add the JavaScript code that makes these new controls work. Figure 5-7 shows an example.

Every video player needs a basic complement of playback buttons. Figure 5-7 uses plain-Jane buttons:

```
<button onclick="play()">Play</button>
<button onclick="pause()">Pause</button>
<button onclick="stop()">Stop</button>
```

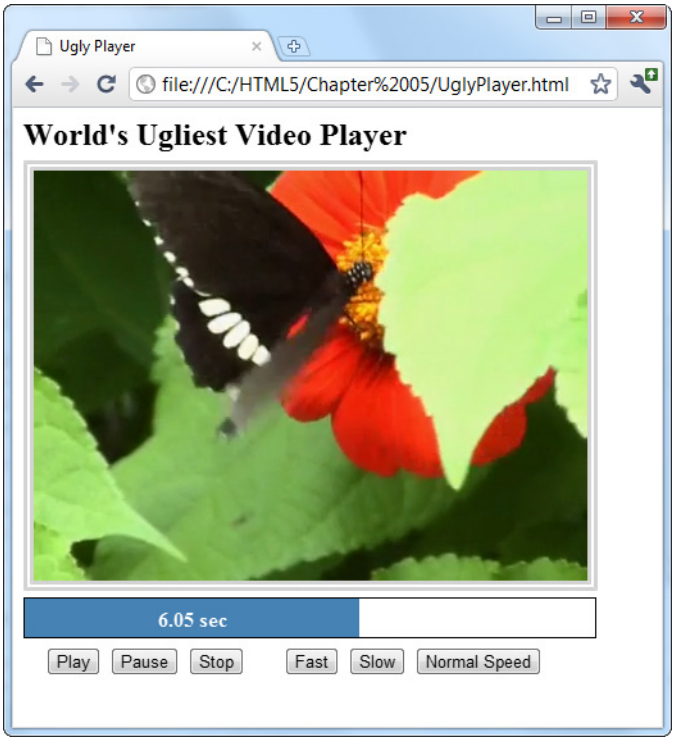


FIGURE 5-7

Making a custom HTML5 video player is easy (making it pretty is not). This example includes the standard playback buttons, a playback progress bar, and a few extra buttons that show off what JavaScript can do with the <video> element.

These buttons trigger the following super-simple functions:

```
function play() {
  video.play();
}

function pause() {
  video.pause();
}
function stop() {
  video.pause();
  video.currentTime = 0;
}
```

The other three playback buttons are more exotic. They adjust the `playbackRate` property to change the speed. For example, a `playbackRate` of 2 plays video at twice the normal speed, but with pitch correction so the audio sounds normal, just accelerated. This is a great feature for getting through a slow training video

in a hurry. Similarly, a playbackRate of 0.5 plays video at half normal speed, and a playbackRate of -1 should play video at normal speed, backward, but browsers have trouble smoothly implementing this behavior.

```
function speedUp() {
  video.play();
  video.playbackRate = 2;
}

function slowDown() {
  video.play();
  video.playbackRate = 0.5;
}

function normalSpeed() {
  video.play();
  video.playbackRate = 1;
}
```

Creating the playback progress bar is a bit more interesting. From a markup point of view, it's built out of two elements, one inside the other:

```
<div id="durationBar">
  <div id="positionBar"><span id="displayStatus">Idle.</span></div>
</div>
```

TIP The playback progress bar is an example where the <progress> element (page 133) would make perfect sense. However, the <progress> element still has limited support—far less than the HTML5 video feature—so this example builds something that looks similar using two elements.

The outer element (named durationBar) draws the solid black border, which stretches over the entire bar and represents the full duration of the video. The inner element (named positionBar) indicates the current playback position, by filling in a portion of the black bar in blue. Finally, a element inside the inner holds the status text, which shows the current position (in seconds) during playback.

Here are the style sheet rules that size and paint the two bars:

```
#durationBar {
  border: solid 1px black;
  width: 100%;
  margin-bottom: 5px;
}

#positionBar {
  height: 30px;
  color: white;
  font-weight: bold;
```

```
  background: steelblue;
  text-align: center;
}
```

When video playback is under way, the <video> element triggers the onTimeUpdate event continuously. You can react to this event to update the playback bar:

```
<video id="videoPlayer" ontimeupdate="progressUpdate()">
  <source src="butterfly.mp4" type="video/mp4">
  <source src="butterfly.webm" type="video/webm">
</video>
```

Here, the code gets the current positioning in the video (from the currentTime property), divides that into the total time (from the duration property), and turns that into a percentage that sizes the element named positionBar:

```
function progressUpdate() {
  // Resizing the blue positionBar, from 0 to 100%.
  var positionBar = document.getElementById("positionBar");
  positionBar.style.width = (video.currentTime / video.duration * 100) + "%";

  // Display the number of seconds, using two decimal places.
  displayStatus.innerHTML = (Math.round(video.currentTime*100)/100) + " sec";
}
```

TIP To get fancier, you could superimpose a download progress bar that shows how much current content has been downloaded and buffered so far. Browsers already add this feature to their built-in players. To add it to your own player, you need to handle the onProgress event and work with the seekable property. For more information about the many properties, methods, and events provided by the <video> element, check out Microsoft's reference at *http://tinyurl.com/video-obj-js*.

JavaScript Media Players

If you're truly independent-minded, you can create your own audio or video player from scratch. But it's not a small project, especially if you want nifty features, like an interactive playlist. And if you don't have a small art department to back you up, there's a distinct possibility that your final product will look just a little bit ugly.

Happily, there's a better option for web authors in search of the perfect HTML5 player. Instead of building one yourself, you can pick up a free, JavaScript-customized media player from the Web. Two solid choices are VideoJS (*http://videojs.com*) and, for jQuery fans, jPlayer (*www.jplayer.org*). Both of these players are lightweight, easy to use, and *skinnable*, which means you can change the look of the playback controls by plugging in a different style sheet.

Most JavaScript media players (including VideoJS and jPlayer) have built-in Flash fallbacks, which saves you from needing to find a separate Flash player. And jPlayer includes its own handy playlist feature, which lets you queue up a whole list of audio and video files (Figure 5-8).

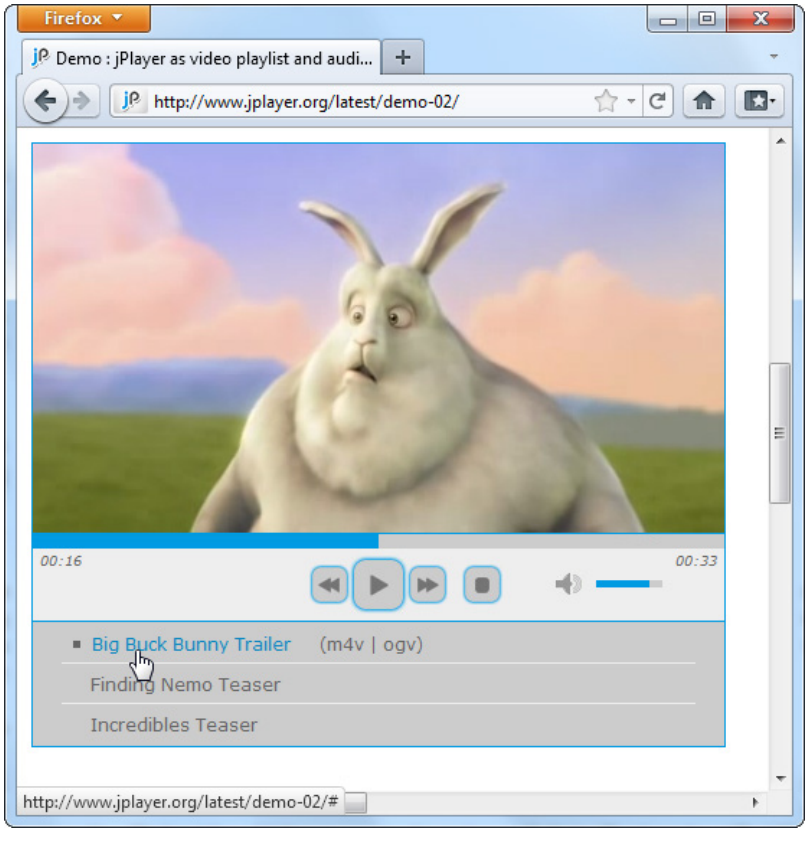


FIGURE 5-8

Using jPlayer's playlist feature, you can offer a series of audio or video files. The user can then play them all in sequence or click to play a specific one. The example playlist here has three videos.

To use VideoJS, you start by downloading the JavaScript files from the VideoJS website. Then you add the JavaScript reference and style sheet reference shown here:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>...</title>
  <script src="video.js"></script>
  <link rel="stylesheet" href="video-js.css">
</head>
...
```

Then you use the exact same <video> element you'd normally use, with the multiple source elements and Flash fallback. (The VideoJS player sample code has the Flowplayer already slotted in for the Flash fallback, but you can remove it and use a different Flash player instead.) In fact, the only difference between a normal HTML5 video page and one that uses VideoJS is the fact that you must use a special element to wrap the video player, as shown here:

```
<div class="video-js-box">
  <video class="video-js" width="640" height="264" controls ...>
    ...
  </video>
</div>
```

It's nice to see that even when extending HTML5, life can stay pretty simple.

Video Captions

As you've seen in previous chapters, the creators of HTML5 were often thinking about web accessibility—in other words, how people with disabilities can use rich web pages easily and efficiently.

Adding accessibility information to images is easy enough. You simply need to bolt on some suitably descriptive text with the `alt` attribute. But what's the equivalent to alt text for a video stream? The consensus is to use *subtitles*, text captions that pop up at the right point during playback. Subtitles can be similar to television closed-captioning, by simply transcribing dialogue, or they can add descriptive and supplementary information of their own. The point is that they give people an avenue to follow the video even if they have hearing difficulties (or if they just don't want to switch on their computer speakers to play the *Iron Man 4* movie trailer for the entire office).

Figure 5-9 shows an example of a captioned video.

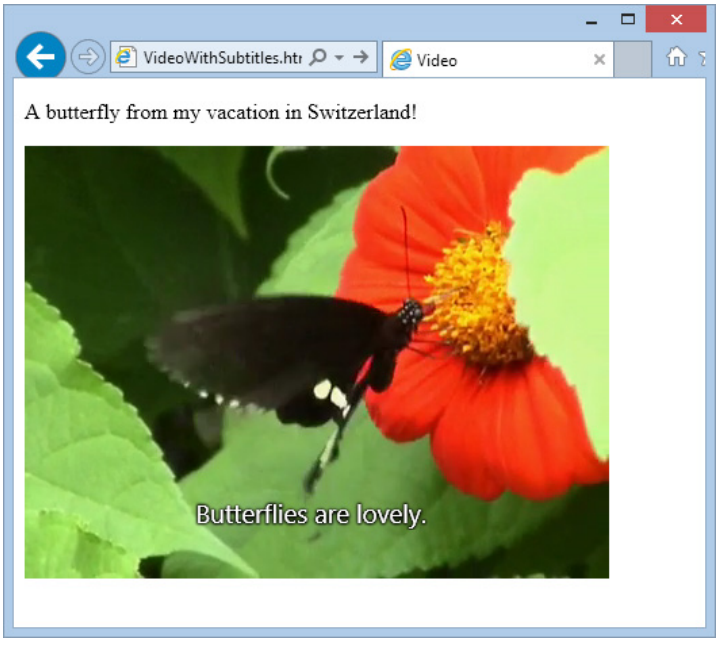


FIGURE 5-9

The subtitles in this video appear in their usual place: centered, and near the bottom of the video window.

Timed Text Tracks and WebVTT

In technical video speak, a *subtitle* is a caption that appears superimposed on a video, and a sequence of subtitles is a *timed text track*. There are a number of different formats for timed text tracks, but they all have fundamental similarities. All of them are recorded as ordinary text with time markings and are placed in chronological order in an ordinary text file. Here's an example of a timed tracks file written in WebVTT (Web Video Text Tracks Format), which is the format favored by HTML5. It holds four captions:

```
WEBVTT

00:00:05.000 --> 00:00:10.000
This caption appears 5 seconds in and lingers until the 10 second mark.

00:01:00.000 --> 00:01:10.000
Now 1 minute has passed. Think about that for 10 seconds.

00:01:10.000 --> 00:01:15.000
This caption appears immediately after the second caption disappears.

00:01:30.000 --> 00:01:35.000
Captions can use <i>line breaks</i> and <b>simple</b> HTML markup.
```

As you can see, every entry in a WebVTT file specifies three details: the time the caption should appear (in *hour:minute:seconds* format), the time it should disappear, and the associated text. Save this content in a text file with the extension *.vtt* (as in *subtitles.vtt*), and you have a ready-to-go timed track file.

Although captioning seems simple, there are a number of fiddly details. For example, you may want to control line breaks, format your text, move the text captions to another position in the video window, or show karaoke-style captions that fill in one word at a time. That's why there are close to 50 different timed track formats. In fact, the struggle between timed text standards has been as ugly as the format war over audio and video codecs.

Currently, the official HTML5 specification doesn't specify what timed text format you should use. However, browser makers have united around WebVTT, a still-evolving specification inspired by the simple SRT format used for subtitles in desktop media players. Browser makers have chosen to ignore the more mature TTML (Timed Text Markup Language) standard, which the W3C has honed over a decade, because it's too complex. (Currently, IE 10 [and later] are the only browsers that give TTML any measure of support.)

TIP You can learn more about the WebVTT standard—including the techniques you need to format and style captions—from the specification at *http://dev.w3.org/html5/webvtt*. If you'd like some help writing your captions, you can try out Microsoft's nifty Caption Maker page (Figure 5-10), which you can find at *http://tinyurl.com/capmaker*.

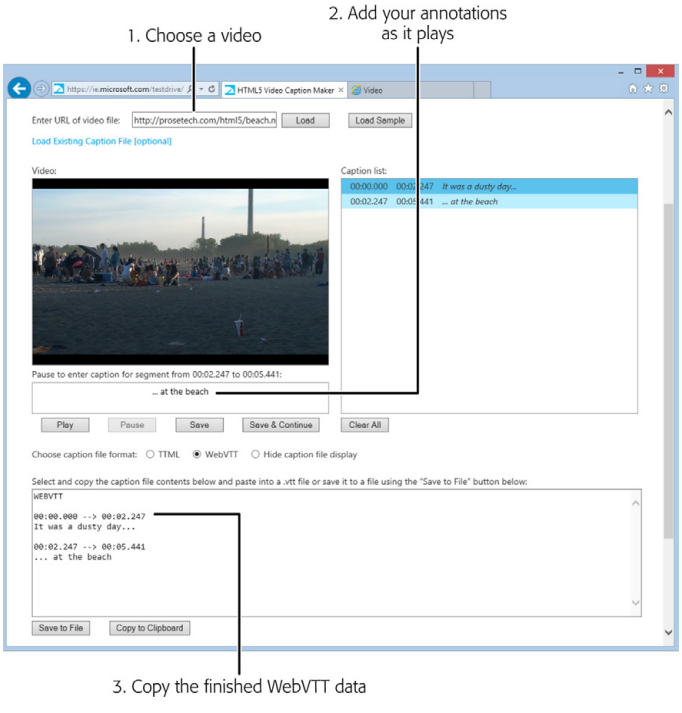


FIGURE 5-10

To use the Caption Maker, you must first point it to a video file that's already on the Web (type in the URL and click Load). While you're playing the video, it's easy to make a quick caption (type it in the box under the video window and then click Save). Then, when you're all done, you can copy the complete WebVTT listing from the text box below.

Adding Captions with <track>

Once you have a WebVTT file that contains your captions, you need a way to pair them up with your video file. The element that works this magic is named `<track>`. You add it inside your `<video>` element, after any `<source>` elements:

```
<video controls width="700" height="400">
  <source src="butterfly.mp4" type="video/mp4">
  <source src="butterfly.webm" type="video/webm">
  <track src="butterfly.vtt" srclang="en" kind="subtitles" label="English"
   default>
</video>
```

The `<track>` element takes several attributes. First is the `src` attribute, which identifies the timed track file. The `srclang` attribute identifies the language code of your subtitle file, for accessibility tools. Use `en` for English (or get the code for a more exotic language at *http://tinyurl.com/l-codes*).

The `kind` attribute describes the type of content in your captions. The HTML5 specification gives you five choices, but only two result in the pop-up captions you expect. Specify `subtitles` if your text consists of transcriptions or translations of dialogue), or `captions` if your text includes dialogue *and* descriptions for sound effects and musical cues.

TIP Subtitles make sense when you can hear the audio but not understand it—for example, when watching a movie in another language. Captions make sense when no audio is available at all—for example, you've muted your player so you won't wake up your officemate in the cubicle beside you.

More specialized values for the `kind` attribute are `descriptions` (text that can replace the video when it's not available and may be spoken by speech synthesis), `chapters` (chapter titles, which viewers can use as a navigation aid), and `metadata` (bits of information that you can retrieve in your JavaScript code). If you choose one of these values, the video player won't show the text. It's up to another tool—or your JavaScript code—to retrieve this information and act on it.

The `label` attribute sets the text that's shown in the video player's Caption menu, which you can call up by clicking a small button under the video window. The label text is particularly important if you want to let viewers choose from multiple tracks. For example, here's a video that has two tracks:

```
<video controls width="700" height="400">
  <source src="butterfly.mp4" type="video/mp4">
  <source src="butterfly.webm" type="video/webm">
  <track src="butterfly.vtt" srclang="en" kind="subtitles" label="English"
   default>
  <track src="butterfly_fr.vtt" srclang="fr" kind="subtitles" label="French">
</video>
```

The first track has the `default` attribute, so it's the one that's initially picked. But the viewer can click the caption button and pick the other track (Figure 5-11).

Even if your video has just a single track, the caption list still includes two choices: your track and an Off option that turns off the captions. If you want to give viewers the option of captions, but you want your captions to be off initially, just make sure none of your tracks have the `default` attribute. That way, the video player starts with the Off option.

NOTE Track files aren't just for accessibility and soundless playback. Search engines can also mine the information in a track file and use it to improve search results. If fact, a super-smart search engine of the future might use WebVTT information to lead a searcher directly to a specific playback location *within* a video file, by matching the search text with a timed caption.

Browser Support for Video Captions

Browsers have been slow to support the <track> element. At this writing, Firefox doesn't support it at all, but the developers at Mozilla plan to include it in the future. Table 5-4 details the current state of affairs.

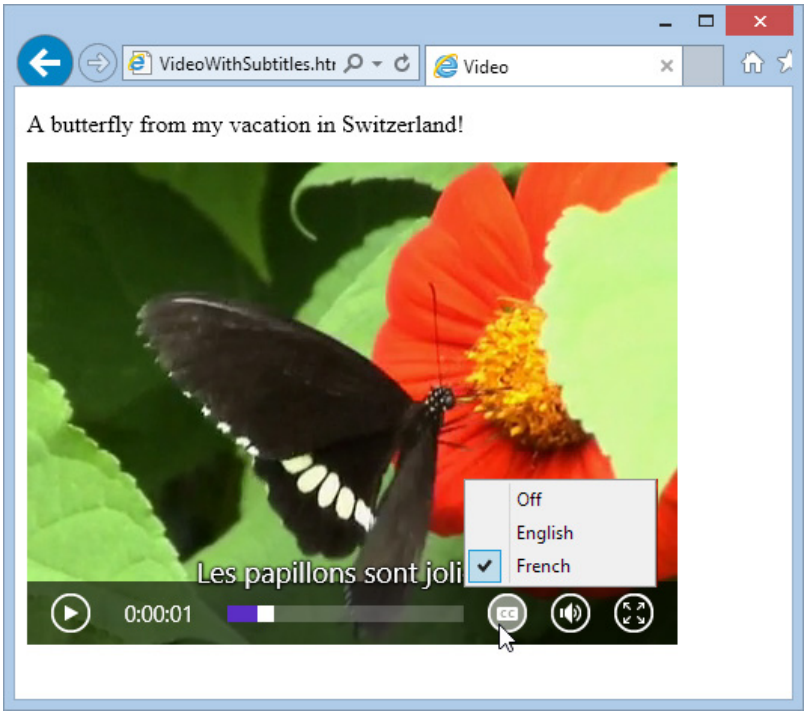


FIGURE 5-11
Browsers that support the <track> element add a caption-picking button like this one. Using it, the viewer can switch tracks or turn captions off altogether.

TABLE 5-4 *Browser support for the <track> element*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	ANDROID
Minimum Version	10	-	26	6*	15	-	2

* Safari doesn't provide a caption button for switching tracks or turning captions on and off.

NOTE If you're testing videos that use captions in Chrome, you'll need to upload your files first. If you simply launch the file from your computer, Chrome can play the video file, but it can't fetch the matching WebVTT file.

Fortunately, you can use the <track> element without worry. Browsers that don't support it simply ignore it, without a hiccup.

If you need a way to provide captions that work on every HTML5 browser, there's an easy workaround. You can use the <track> element as you would normally, in conjunction with a JavaScript polyfill, such as Captionator.js (*http://captionatorjs.com*). Captionator works by placing a floating element over the video window. Then, when playback reaches the appropriate points, Captionator.js retrieves the caption text from the WebVTT file and inserts it into the floating element.

CHAPTER

6

Fancy Fonts and Effects with CSS3

It would be ludicrous to build a modern website without CSS. The standard is fused into the fabric of the Web almost as tightly as HTML. Whether you're laying out pages, building interactive buttons and menus, or just making things look pretty, CSS is a fundamental tool. In fact, as HTML has increasingly shifted its focus to content and semantics (page 38), CSS has become the heart and soul of web *design*.

Along the way, CSS has become far more detailed and far more complex. When CSS evolved from its first version to CSS 2.1, it quintupled in size, reaching the size of a modest novel. Fortunately, the creators of the CSS standard had a better plan for future features. They carved the next generation of enhancements into a set of separate standards, called *modules*. That way, browser makers were free to implement the most exciting and popular parts of the standard first—which is what they were already doing, modules or not. Together, the new CSS modules fall under the catchall name *CSS3* (note the curious lack of a space, as with *HTML5*).

CSS3 has roughly 50 modules in various stages of maturity. They range from features that provide fancy eye candy (like rich fonts and animation) to ones that serve a more specialized, practical purpose (for example, speaking text aloud or varying styles based on the capabilities of the computer or mobile device). They include features that are reliably supported in the most recent versions of all modern browsers and features so experimental that no browser yet supports them.

In this chapter, you'll tour some of the most important (and best supported) parts of CSS3. First, you'll see how to use shadows, rounded corners, and other refinements to make your boxes look better. Next, you'll learn how you can use transitions to create subtle effects when the visitor hovers over an element, clicks on it, or tabs over to a control. (And you'll make these effects even better with two more CSS3

features: transforms and transparency.) Finally, you'll learn how to jazz up ordinary text with a rich variety of web fonts.

But first, before you get to any of these hot new features, it's time to consider how you can plug in the latest and most stylin' features without leaving a big chunk of your audience behind.

Using CSS3 Today

CSS3 is the unchallenged future of web styling, and it's not finished yet. Many modules are still being refined and revised, and no browser supports them all. You can see the current state of this giant family of specifications at *http://tinyurl.com/CSS3-stages*.

Because CSS3 is still being fine-tuned, it has the same complications as HTML5. Website authors like yourself need to decide what to use, what to ignore, and how to bridge the support gaps.

There are essentially three strategies you can use when you start incorporating CSS3 into a website. The following sections describe them.

NOTE CSS3 is not part of HTML5. The standards were developed separately, by different people working at different times in different buildings. However, even the W3C encourages web developers to lump HTML5 and CSS3 together as part of the same new wave of modern web development. For example, if you check out the W3C's HTML5 logo-building page at *www.w3.org/html/logo*, you'll see that it encourages you to advertise CSS3 in its HTML5 logo strips. Furthermore, many hallmarks of modern web design with HTML5—such as the mobile-friendly layout techniques you'll learn about in the next chapter—require CSS3.

Strategy 1: Use What You Can

It makes sense to use features that already have solid browser support across all browser brands. One example is the web font feature (page 206). With the right font formats, you can get it working with browsers all the way back to IE 6. Unfortunately, very few CSS3 features fall into this category. The `word-wrap` property works virtually everywhere, and older browsers can do transparency with a bit of fiddling, but just about every other feature leaves the still-popular IE 8 browser in the dust.

NOTE Unless otherwise noted, the features in this chapter work on the latest version of every modern browser, including Internet Explorer, provided you use IE 9 or later. However, they don't work on older versions of IE.

Strategy 2: Treat CSS3 Features as Enhancements

CSS3 fans have a rallying cry: "Websites don't need to look exactly the same on every browser." Which is certainly true. (They have a one-page website, too—see *http://DoWebsitesNeedToBeExperiencedExactlyTheSameInEveryBrowser.com*, which picks up a few frills on modern browsers but remains functional on laggards like IE 7.)

The idea behind this strategy is to use CSS3 to add fine touches that won't be missed by people using less-capable browsers. One example is the `border-radius` property that you can use to gently round the corners of a floating box. Here's an example:

```
header {
  background-color: #7695FE;
  border: thin #336699 solid;
  padding: 10px;
  margin: 10px;
  text-align: center;
  border-radius: 25px;
}
```

Browsers that recognize the border-radius property will know what to do. Older browsers will just ignore it, keeping the plain square corners (Figure 6-1).

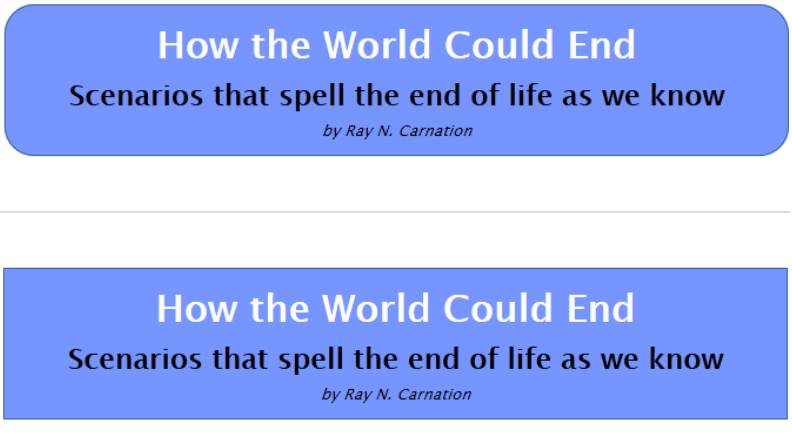


FIGURE 6-1
On Internet Explorer 9, this header box has rounded corners (top). IE 8 ignores the border-radius property but applies the rest of the style sheet properties (bottom).

This backward compatibility allows web designers to play with the latest frills in the newest version of CSS without breaking their sites on older browsers. However, there's a definite downside if you go too far. No matter how good a website looks in the latest version of your favorite browser, it can be deeply deflating if you fire up an older browser that's used by a significant slice of your clientele and find that it looks distinctly less awesome. After all, you want your website to impress everyone, not just web nerds with the best browsers.

For this reason, you may want to approach some CSS3 enhancements with caution. Limit yourself to features that are already in most browsers, even if they require the latest browser versions. And don't use CSS3 features in ways that change the experience of your website so dramatically that some people will get second-rate status.

TIP When it comes to CSS3, Internet Explorer is the straggler. There's a militant minority of web designers who believe that web designers should ignore backward browsers like IE 8 and start using CSS3 features as soon as other browsers support them. Otherwise, who will keep pressure on Microsoft and encourage the Web to get better? That philosophy makes sense, *if* the primary purpose of your website is the political one of promoting advanced web standards. But otherwise, keep in mind that dismissing a large segment of the web world will reflect poorly on you—because no matter how much you dislike someone's browser, that person is still using it to look at *your* work.

Strategy 3: Add Fallbacks with Modernizr

Using a partially supported CSS3 feature is a great idea if the website still looks great without it. But sometimes, a vital part of your website design can go missing, or the downgraded version of your website just looks ugly. For example, consider what happens if you use the Firefox-only multicolored border settings, as shown in Figure 6-2.

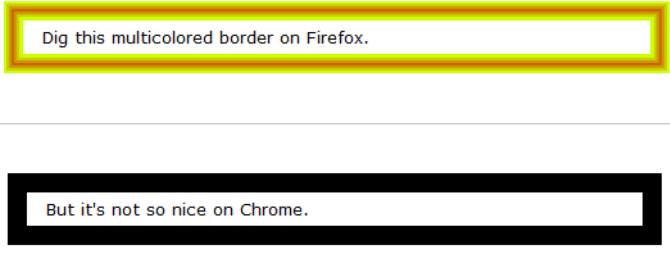


FIGURE 6-2
This multicolored border looks snazzy in Firefox (top). But try the same thing out in Chrome, and you'll get a thick, plain black border (bottom)—and that never looks good.

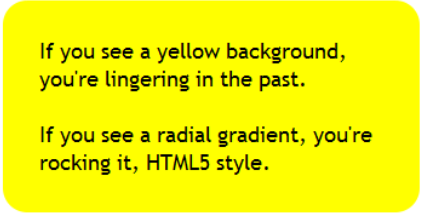
Sometimes, you can solve the problem by stacking properties in the right order. The basic technique is to start with more general properties, followed by new properties that override these settings. When this works, it satisfies every browser—the old browsers get the standard settings, while the new browsers override these settings with newer ones. For example, you can use this technique to replace an ordinary background fill with a gradient:

```
.stylishBox  {
  ...
  background: yellow;
  background: radial-gradient(ellipse, red, yellow);
}
```

Figure 6-3 shows the result.

In some cases, overriding style properties doesn't work, because you need to set properties in combination. The multicolored border in Figure 6-2 is an example. The multicolored effect is set with the `border-colors` property, but it appears only if the border is made thick with the `border-thickness` property. On browsers that

don't support multicolored borders, the thick border is an eyesore, no matter what single color you use.



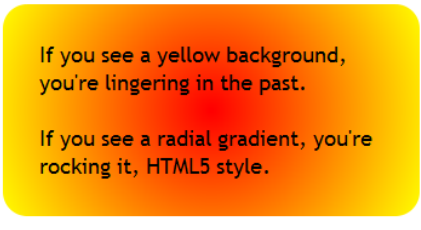


FIGURE 6-3

Top: In browsers that don't understand CSS3, the `stylishBox` *rule paints a yellow background.*

Bottom: In browsers that do understand CSS3, the yellow background is replaced with a radial gradient that blends from a red center point to yellow at the edges.

One way to address problems like these is with Modernizr, the JavaScript library that tests HTML5 feature support (page 31). It lets you define alternate style settings for browsers that don't support the style properties you really want. For example, imagine you want to create two versions of the header box shown in Figure 6-1. You want to use rounded corners if they're supported, but substitute a double-line border if they aren't. If you've added the Modernizr script reference to your page, then you can use a combination of style rules, like this:

```
/* Settings for all headers, no matter what level of CSS3 support. */
header {
  background-color: #7695FE;
  padding: 10px;
  margin: 10px;
  text-align: center;
}

/* Settings for browsers that support border-radius. */
.borderradius header {
  border: thin #336699 solid;
  border-radius: 25px;
}
```

```
/* Settings for browsers that don't support border-radius. */
.no-borderradius header {
  border: 5px #336699 double;
}
```

So how does this nifty trick work? When you use Modernizr in a page, you begin by adding the `class="no-js"` attribute to the root `<html>` element:

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

When you load Modernizr on a page, it quickly checks if a range of HTML5, JavaScript, and CSS3 features are supported. It then applies a pile of classes to the root `<html>` element, separated by spaces, changing it into something like this:

```
<html class="js flexbox canvas canvastext webgl no-touch geolocation
postmessage no-websqldatabase indexeddb hashchange history draganddrop
no-websockets rgba hsla multiplebgs backgroundsize borderimage borderradius
boxshadow textshadow opacity no-cssanimations csscolumns cssgradients
no-cssreflections csstransforms no-csstransforms3d csstransitions fontface
generatedcontent video audio localstorage sessionstorage webworkers
applicationcache svg inlinesvg smil svgclippaths">
```

If a feature appears in the class list, that feature is supported. If a feature name is prefixed with the text "no-" then that feature is not supported. Thus, in the example shown here, JavaScript is supported (`js`) but web sockets are not (`no-websockets`). On the CSS3 side of things, the border-radius property works (`borderradius`) but CSS3 reflections do not (`no-cssreflections`).

You can incorporate these classes into your selectors to filter out style settings based on support. For example, a selector like `.borderradius header` gets all the `<header>` elements inside the root `<html>` element—if the browser supports the `border-radius` property. Otherwise, there will be no `.borderradius` class, the selector won't match anything, and the rule won't be applied.

The catch is that Modernizr provides classes for only a subset of CSS3 features. This subset includes some of CSS3's most popular and mature features, but the `border-color` feature in Figure 6-2 doesn't qualify because it's still Firefox-only. For that reason, it's a good idea to hold off on using multicolored borders in your pages, at least for now.

NOTE You can also use Modernizr to create JavaScript fallbacks. In this case, you simply need to check the appropriate property of the Modernizr object, as you do when checking for HTML5 support. You can use this technique to compensate if you're missing more advanced CSS3 features, like transitions or animations. However, there's so much work involved and the models are so different that it's usually best to stick with a JavaScript-only solution for essential website features.

Browser-Specific Styles with Vendor Prefixes

When the creators of CSS develop new features, they often run into a chicken-and-egg dilemma. In order to perfect the feature, they need feedback from browser makers and web designers. But in order to get this feedback, browser makers and web designers need to implement these new-and-imperfect features. The result is a cycle of trial and feedback that takes many revisions to settle down. As this process unfolds, the syntax and implementation of features change. This raises a very real danger—unknowing web developers might learn about a dazzling new feature and implement it in their real-life websites, not realizing that future versions of the standard could change the rules and break the websites.

To avoid this threat, browser makers use a system of *vendor prefixes* to change CSS property and function names while they're still under development. For example, consider the `radial-gradient()` function described on page 193. In older versions of Firefox, the `radial-gradient()` function wasn't available. However, you could use an "in progress" version of this function called `-moz-radial-gradient`:

```
.stylishBox  {
  background: yellow;
  background: -moz-radial-gradient(ellipse, red, yellow);
}
```

Firefox uses the vendor prefix `-moz-` (which is short for Mozilla, the organization that's behind the Firefox project). Every browser engine has its own vendor prefix (Table 6-1), which complicates life horrendously, but for a good reason. Different browser makers add support at different times, often using different draft versions of the same specification. Although all browsers will support the same syntax for final specification, the syntax of the vendor-specific properties and functions often varies.

TABLE 6-1 *Vendor prefixes*

PREFIX	FOR BROWSERS
`-moz-`	Firefox
`-webkit-`	Chrome, Safari, and the latest versions of Opera (the same rendering engine powers all three browsers)
`-ms-`	Internet Explorer
`-o-`	Old versions of Opera (before version 15)

Here's an example that applies a radial gradient using all four of the browser-specific prefixes:

```
.stylishBox  {
  background: yellow;
  background-image: -moz-radial-gradient(circle, green, yellow);
  background-image: -webkit-radial-gradient(circle, green, yellow);
  background-image: -o-radial-gradient(circle, green, yellow);
  background-image: -ms-radial-gradient(circle, green, yellow);
}
```

Clearly, when dealing with the less mature parts of CSS3, you need some bloated style sheet rules.

The obvious question for every web designer is "When do I need to use vendor prefixes, and when is it safe to use the ordinary, unprefixed property or function name?" You might think you could just fire up your browser for a quick test, but you won't have a conclusive answer unless you run your page through *every* browser out there. For example, the `borderadius` property works on all browsers, with no vendor prefixes required. But the `radial-gradient()` function is a bit trickier: At this writing, it works on most browsers but still requires the `-webkit-` prefix on some mobile browsers. And the `transform` property that you'll consider later in this chapter works with no prefix on IE and Firefox, but still requires the `-webkit-` prefix in Chrome, Safari, and Opera.

Further complicating life, the syntax you use to specify a property value or function argument can change. For example, IE 10 introduced a prefixed version of the `radial-gradient()` function during testing. The final, released version of IE 10 lets developers use either the most recent syntax with the unprefixed `radial-gradient()` function or the slightly older form with the prefixed `-ms-radial-gradient()` function. This setup is good for endless hours of debugging fun.

NOTE In this chapter, you'll learn about the current state of support for all CSS3 parts covered here, including which ones need vendor prefixes. If a style sheet example in this chapter *doesn't* use vendor prefixes, you can assume that it's safe to omit them when you use the feature in your own pages.

If your head is starting to spin, don't worry—help is at hand. To get the latest information about which CSS3 features require vendor prefixes, you can turn to the virtually indispensable site *http://caniuse.com* (which you first saw on page 27). When you look up a CSS3 feature, the "Can I use..." site clearly spells out which browser versions require a vendor prefix (Figure 6-4).

NOTE Using vendor prefixes is a messy business. Web developers are split on whether they're a necessary evil of getting the latest and greatest frills, or a big fat warning sign that should scare clear-thinking designers away. But one thing is certain: If you don't use the vendor prefixes, significant parts of CSS3 will be off limits for now.

Building Better Boxes

From the earliest days of CSS, web designers were using it to format boxes of content. As CSS became more powerful, the boxes became more impressive, creating everything from nicely shaded headers to floating, captioned figures. And when CSS cracked the hovering problem, floating boxes were even turned into rich, glowy buttons, taking over from the awkward JavaScript-based approaches of yore. With this in mind, it's no surprise that some of the most popular and best-supported CSS3 features can make your boxes look even prettier, no matter what they hold.

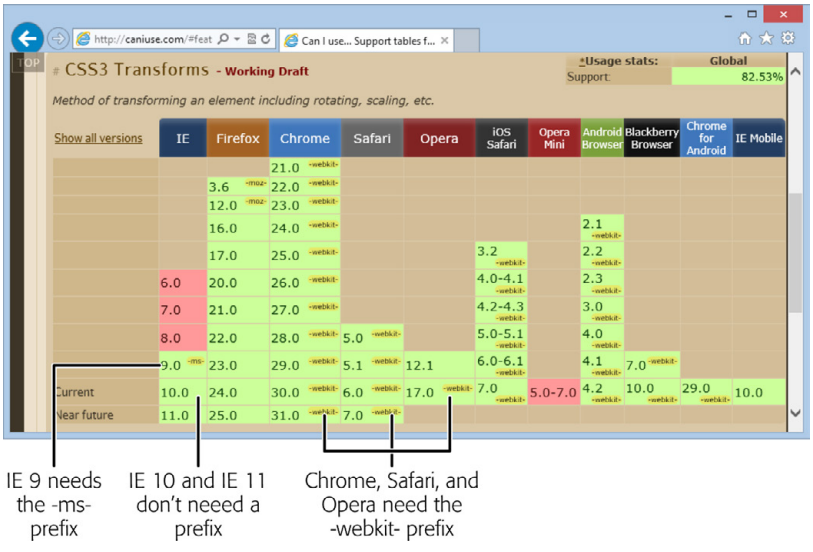


FIGURE 6-4

If you need to use a prefix, the "Can I use..." website lets you know in a tiny yellow text bubble. For example, the CSS3 transforms feature doesn't need a vendor prefix in the latest versions of Firefox or IE, but it does in all other browsers.

GEM IN THE ROUGH

Adding Vendor Prefixes Automagically

If you make heavy use of the parts of CSS3 that still require vendor prefixes, you can quickly get worn down updating massive style sheets and adding multiple versions of the same style sheet property over and over again. Before you lose your sanity, consider a miraculously clever JavaScript tool called *-prefix-free*.

To use -prefix-free, you create an ordinary style sheet, using CSS3 properties as you need them, without worrying about vendor prefixes. Then, in your web pages, you add a reference to the -prefix-free script.

When someone views one of your pages, the -prefix-free script springs into action. It examines the current browser and automatically tweaks your style sheet to suit by adding all the vendor prefixes that that browser needs. (Yes, this automatic tweaking takes a bit of extra time, but you'll probably find it's so fast as to be undetectable.) Of course, -prefix-free can't make a browser support a feature that it otherwise wouldn't, but it can transform ordinary, sensibly named properties into the messy, vendor-specific names that some browsers need to support new and evolving features. For many developers, adding an extra JavaScript file is a small price to pay for managing the chaos of CSS3 prefixes.

To download the -prefix-free library, or play with an interactive page that lets you type some CSS and test the script's prefix-adding ability, visit *http://leaverou.github.io/prefixfree*.

Transparency

The ability to make partially transparent pictures and colors is one of the most basic building blocks in CSS3. There are two ways to do it.

Your first option is to use the rgba() color function, which accepts four numbers. The first three values are the red, green, and blue components of the color, from 0 to 255. The final value is the *alpha*, a fractional value number from 0 (fully transparent) to 1 (fully opaque).

Here's an example that creates a 50 percent transparent lime green color:

```
.semitransparentBox {
  background: rgba(170,240,0,0.5);
}
```

Browsers that don't support rgba() will just ignore this rule, and the element will keep its default, completely transparent background. So the second, and better, approach is to start by declaring a solid fallback color, and then replace that color with a semitransparent one:

```
.semitransparentBox {
  background: rgb(170,240,0);
  background: rgba(170,240,0,0.5);
}
```

This way, browsers that don't support the rgba() function will still color the element's background, just without the transparency.

TIP To make this fallback better, strive to use a color that more accurately reflects the semitransparent effect. For example, if you're putting a semitransparent lime green color over a mostly white background, the color will look lighter because the white shows through. Your fallback color should reflect this fact, if possible.

CSS3 also adds a style property named opacity, which works just like the alpha value. You can set opacity to a value from 0 to 1 to make any element partially transparent:

```
.semitransparentBox {
  background: rgb(170,240,0);
  opacity: 0.5;
}
```

Figure 6-5 shows two examples of semitransparency, one that uses the rgba() function and one that uses the opacity property.

The opacity property is a better tool than the rgba() function if you want to do any of the following:

- Make more than one color semitransparent. With opacity, the background color, text color, and border color of an element can become transparent.
- Make something semitransparent, even if you don't know its color (for example, because it might be set by another style sheet or in JavaScript code).
- Make an image semitransparent.
- Use a transition, an animated effect that can make an element fade away or reappear (page 199).

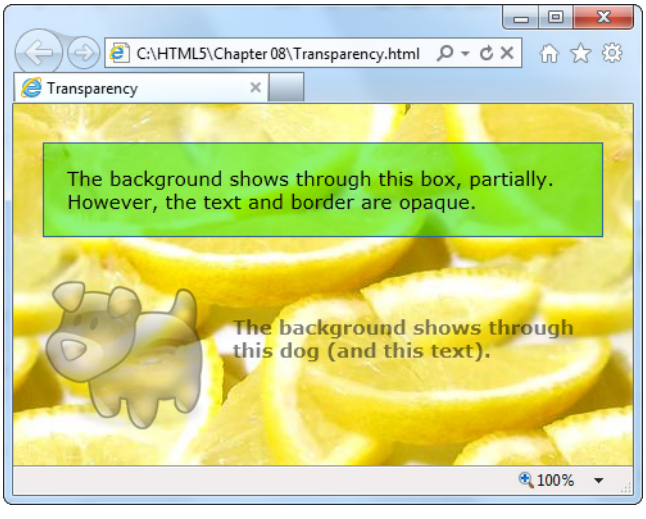


FIGURE 6-5

This page serves up semitransparency two different ways: to fade out a picture (using the `opacity` *property) and to let the background show through a box (using a background color created with the* `rgba()` *function).*

Rounded Corners

You've already learned about the `border-radius` property, which lets you shave the hard corners off boxes. But what you haven't yet seen is how you can tweak this setting to get the curve you want.

First, you can choose a different, single value for the `border-radius` property, since the property specifies the radius of the circle that's used to draw the rounded edge. (You don't see the entire circle—just enough to connect the vertical and horizontal sides of the box.) Set a bigger `border-radius` value, and you'll get a bigger curve and a more gently rounded corner. As with most measurements in CSS, you can use a variety of units, including pixels and percentages. You can also adjust each corner separately by supplying four values:

```
.roundedBox {
  background: yellow;
  border-radius: 25px 50px 25px 85px;
}
```

But that's not all—you can also stretch the circle into an ellipse, creating a curve that stretches longer in one direction. To do this, you need to target each corner separately (using properties like `border-top-left-radius`) and then supply two numbers: one for the horizontal radius and one for the vertical radius:

```
.roundedBox {
  background: yellow;
  border-top-left-radius: 150px 30px;
  border-top-right-radius: 150px 30px;
}
```

Figure 6-6 shows some examples.

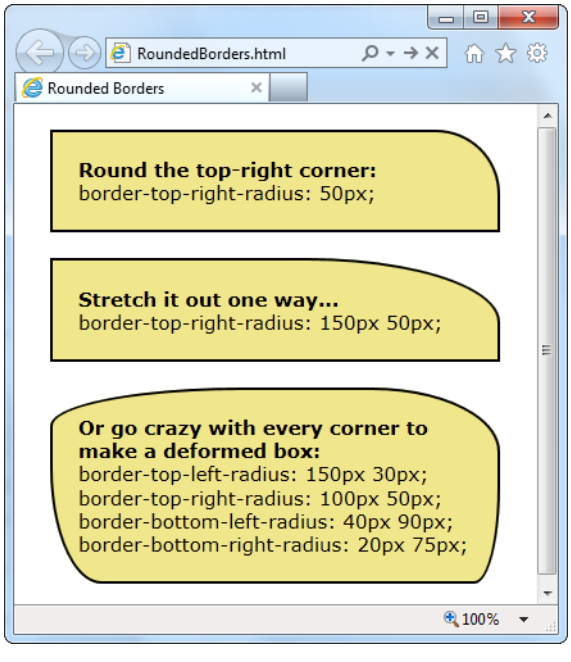


FIGURE 6-6

A clever application of `border-radius` *can create virtually any sort of curve.*

Backgrounds

One shortcut to attractive backgrounds and borders is to use images. CSS3 introduces two new features to help out here. First is multiple background support, which lets you combine two or more images in a single element's background. Here's an example that uses two backgrounds to embellish the top-left and bottom-right corner of a box:

```
.decoratedBox {
  margin: 50px;
  padding: 20px;
  background-image: url('top-left.png'), url('bottom-right.png');
  background-position: left top, right bottom;
  background-repeat: no-repeat, no-repeat;
}
```

This first step is to supply a list with any number of images, which you use to set the `background-image` property. You can then position each image and control whether it repeats, using the `background-position` and `background-repeat` properties. The trick is to make sure that the order matches, so the first image is positioned with the first `background-position` value, the second image with the second `background-position` value, and so on. Figure 6-7 shows the result.

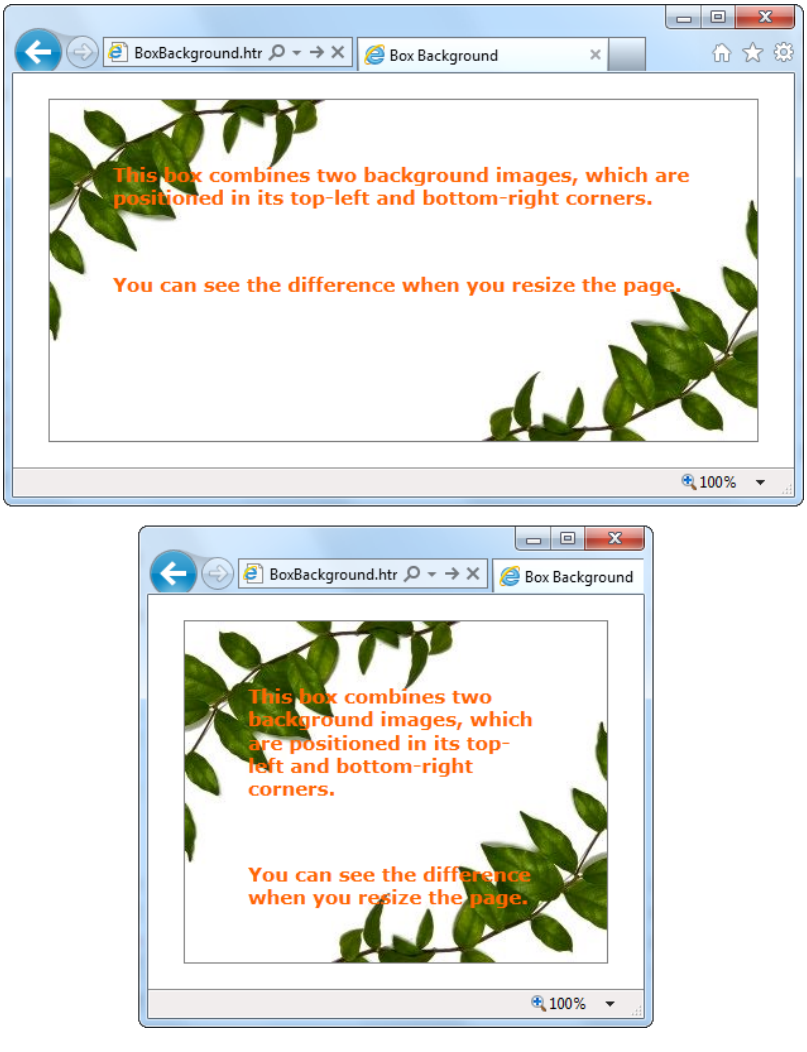


FIGURE 6-7
It doesn't matter how big this box grows—the two background images stay fixed at either corner.

NOTE If browsers don't support multiple backgrounds, they'll completely ignore your attempt to set the background. To avoid this problem, start by setting the `background` or `background-image` property with a fallback color or picture. Then, attempt to set multiple backgrounds by setting `background-image` with a list of pictures.

And here's a revised example that uses the *sliding doors* technique—a time-honored web design pattern that creates a resizable graphic out of three pieces: an image for the left, an image for the right, and an extremely thin sliver that's tiled through the middle:

```
.decoratedBox {
  margin: 50px;
  padding: 20px;
  background-image: url('left.png'), url('middle.png'), url('right.png');
  background-position: left top, left top, right bottom;
  background-repeat: no-repeat, repeat-x, no-repeat;
}
```

You could use markup like this to draw a background for a button. Of course, with all of CSS3's fancy new features, you'll probably prefer to create those using shadows, gradients, and other image-free effects.

Shadows

CSS3 introduces two types of shadows: box shadows and text shadows. Of the two, box shadows are generally more useful. You can use a box shadow to throw a rectangular shadow behind any `<div>` (but don't forget your border, so it still looks like a box). Shadows even follow the contours of boxes with rounded corners (see Figure 6-8).

FIGURE 6-8
Shadows can make text float (top), boxes pop out (middle), or buttons look glowy (bottom).

The two properties that make shadows work are box-shadow and text-shadow. Here's a basic box shadow example:

```
.shadowedBox {
  border: thin #336699 solid;
  border-radius: 25px;
  box-shadow: 5px 5px 10px gray;
}
```

The first two values set the horizontal and vertical offset of the shadow. Using positive values (like 5 pixels for both, in the above example) displaces the shadow down and to the right. The next value sets the *blur* distance—in this example, 10 pixels—which increases the fuzziness of the shadow. At the end is the shadow color. If there's any content underneath the box, consider using the rgba() function (page 186) to supply a semitransparent shadow.

If you want to tweak your shadow, you can tack on two details. You can add another number between the blur and the color to set the shadow *spread*, which expands the shadow by thickening the solid part before the blurred edge starts:

```
box-shadow: 5px 5px 10px 5px gray;
```

And you can add the word inset on the end to create a shadow that reflects inside an element, instead of outside. This works best if you use a shadow that's directly on top of the element, with no horizontal or vertical offset:

```
box-shadow: 0px 0px 20px lime inset;
```

This creates the bottom example in Figure 6-8. You can use inset shadows to add hover effects to a button (page 196).

NOTE You can even supply multiple shadows by separating each one with a comma. But getting shadow-crazy is usually a waste of effort and computing power.

The text-shadow property requires a similar set of values, but in a different order. The color comes first, followed by the horizontal and vertical offsets, followed by the blur:

```
.textShadow  {
  font-size: 30px;
  font-weight: bold;
  text-shadow: gray 10px 10px 7px;
}
```

Box shadows and text shadows don't show up in old versions of Internet Explorer. Box shadows require IE 9 or later, while text shadows require IE 10 or later.

Gradients

Gradients are blends of color that can create a range of effects, from the subtle shading behind a menu bar to a psychedelic button that's colored like a 1960s revival party. Figure 6-9 shows some examples.

NOTE Many web gradients are faked with background images. But CSS3 lets you define the gradient you want and gets the browser to do the work. The advantage is fewer image files to schlep around and the ability to create gradients that seamlessly resize themselves to fill any amount of space.

CSS supports two types of gradients: linear gradients that blend from one band of color to another, and radial gradients that blend from a central point to the outer edges of your region.

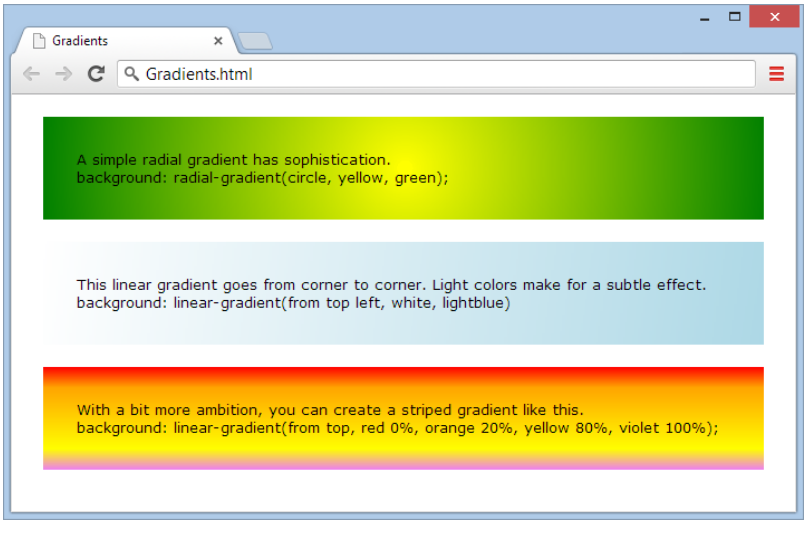


FIGURE 6-9
At the heart of it, gradients are nothing more than blends of two or more colors. But that simple recipe cooks up into plenty of different dishes.

There are no special CSS properties for creating gradients. Instead, you can use a gradient function to set the background property. Just remember to set the property to a solid color first to create a fallback for browsers that don't support gradients (like versions of Internet Explorer before IE 10).

LINEAR GRADIENTS

There are four gradient functions. The first function is `linear-gradient()`. Here it is in one of its simpler forms, shading a region from white at the top to blue at the bottom:

```
.colorBlendBox {
  background: linear-gradient(from top, white, blue);
}
```

The word `from` indicates that the top is the starting point for the first color (white). You can replace this with `to`, which reverses the gradient so it blends from blue at the bottom to white at the top:

```
  background: linear-gradient(to top, white, blue);
```

Similarly, you can replace `top` with `left` to go from one side to another. Or use both to blend diagonally from the top-left corner:

```
background: linear-gradient(from top left, white, lightblue)
```

If you want multiple-color bands, you simply supply a list of colors. Here's how you create a series of three horizontal color stripes, starting with red at the top:

```
background: linear-gradient(from top, red, orange, yellow);
```

Finally, you can control where each color starts (bumping some together or off to one side), using *gradient stops*. Each gradient stop is a percentage, with 0 percent being at the very start of the gradient and 100 percent being at the very end. Here's an example that extends the orangey-yellow section in the middle:

```
background: linear-gradient(from top, red 0%, orange 20%, yellow 80%,
 violet 100%);
```

The syntax of the `linear-gradient()` function is easy to follow. But here's the bad news: To guarantee support on Android browsers and slightly older versions of Safari (before Safari 7), you need to also add the `-webkit-` vendor prefix. And what's worse, the `-webkit-linear-gradient()` function is subtly different from the true `linear-gradient()` function. Unlike `linear-gradient()`, `-webkit-linear-gradient()` doesn't use the `to` or `from` values to specify direction. Instead, `from` is assumed automatically.

Here's a fully outfitted style sheet rule that satisfies slightly older browsers by adding a vendor-prefixed gradient:

```
.colorBlendBox {
  background: lightblue;
  background: -webkit-linear-gradient(top left, white, lightblue);
  background: linear-gradient(from top left, white, lightblue);
}
```

Fortunately, there's no need to add other vendor prefixes (like `-moz-` and `-o-`), unless you want to support *much* older versions of Firefox and Opera.

TIP In all these examples, gradients were used with the `background` property. However, you can also use gradient functions to set the `background-image` property in exactly the same way. The advantage here is that `background-image` lets you use an image fallback. First, set `background-image` to a suitable fallback image for less-equipped browsers, and then set it again using a gradient function. Most browsers are smart enough that they won't download the gradient image unless they need it, which saves bandwidth.

RADIAL GRADIENTS

To set a radial gradient, you use the `radial-gradient()` function. You need to supply a color for the center of the circle and a color for the outer edge of the circle, where it meets the boundaries of the element. Here's a radial gradient that places a white point in the center and fades out to blue on the edges:

```
background: radial-gradient(circle, white, lightblue);
```

Once again, you need to add a `-webkit-` version of the function to be safe:

```
background: -webkit-radial-gradient(circle, yellow, green);
```

Replace the word `circle` with `ellipse` if you want to stretch your gradient out into an oblong shape to better fit its container.

As with a linear gradient, you can supply a whole list of colors. Optionally, you can add percentages to tweak how quickly the gradient blends from one color to the next. Here's an example that starts yellow, blends slowly into green, and then quickly blends through blue, white, and black near the outside edge of the element:

```
background: radial-gradient(circle, yellow 10%, green 70%, blue, white,
black);
```

You can also place the center of your gradient using percentages. For example, if you want the center point of your circle to be near the top-right corner of your element, you might use this sort of radial gradient:

```
background: radial-gradient(circle at 90% 5%, white, lightblue);
```

These percentages tell the browser to start the gradient 90% from the left edge (which is almost all the way to the right side) and 5% from the top edge.

NOTE The syntax of the `radial-gradient()` function has changed since it was first created. The `at` keyword, which positions the gradient's center point, is a relatively new detail. Although it's safe to use `at` with the `radial-gradient()` function, don't attempt to use it with the vendor-specific `-webkit-radial-gradient()` function.

REPEATING GRADIENTS

CSS3 also includes two functions that let you create more dizzying gradients: `repeating-linear-gradient()` and `repeating-radial-gradient()`. Whereas `linear-gradient()` and `radial-gradient()` blend through your list of colors once, the `repeating-linear-gradient()` and `repeating-radial-gradient()` functions cycle through the same set of colors endlessly, until they fill up all the available space in your element with blended stripes of color. The result is a psychedelic tie-dye effect that just might fool you into thinking you've stepped back in time to the '70s.

The syntax of `repeating-linear-gradient()` and `repeating-radial-gradient()` is essentially the same as the syntax of `linear-gradient()` and `radial-gradient()`. The only difference is that you need to make sure you limit the size of your gradient so it can repeat.

For example, this repeating gradient won't look any different from a normal gradient, because its size isn't limited. Instead, it starts with yellow in the center and blends to green at the outer edge:

```
background: repeating-radial-gradient(circle, yellow, green);
```

The following gradient is different. It keeps the yellow in the center, but sets the green to kick in at the 10% mark. After that, the gradient repeats, starting with the yellow color again. The result is a striped effect of blurry yellow and green lines.

```
background: repeating-radial-gradient(circle, yellow, green 10%);
```

You can have as many colors as you like in a repeating gradient. The key detail is to make sure that the final color includes a percentage or pixel value, which sets that color's position. That way, the color won't be placed at the edge of your element.

Instead of using a percentage value, you can use a pixel width, like this:

```
background: repeating-linear-gradient(to top, red, orange, white, yellow,
red 30px);
```

This gradient creates a slightly different effect. Now each stripe always has the same thickness (30 pixels), and the number of stripes depends on the available space. By comparison, the previous example always had the 10 proportionately sized stripes, each one filling 10% of the available space.

TIP Repeating gradients come with two caveats. First, you may include only the `to` keyword but never `from`, because a repeating gradient can be filled in only one direction. Second, if you want your gradient to blend seamlessly without a sharp break between colors each time the gradient repeats itself, make sure the final color in your list is the same as the first color in your list.

GEM IN THE ROUGH

Fancy Gradients with Less Fuss

Creating complex gradients is a fiddly business. To speed up the process, you may want to try an online gradient-generating tool. The idea is simple: You play with the controls in your browser until the gradient looks fabulous, and the tool spits out the markup you need (complete with different vendor-prefixed versions, just in case you need them). Two good gradient-generating tools are the Ultimate CSS Gradient Generator (*www.colorzilla.com/gradient-editor*) and Microsoft's CSS Gradient Background Maker (*http://tinyurl.com/ms-gradient*).

Creating Effects with Transitions

Back in the day when CSS 2.1 was hot stuff, web developers were excited about a new feature called *pseudo-classes* (page 443). Suddenly, with the help of `:hover` and `:focus`, developers could create interactive effects without writing any JavaScript code. For example, to create a hover button, you simply supply a set of new style properties for the `:hover` pseudo-class. These styles kick in automatically when the visitor moves the mouse pointer over your button.

TIP If you're the last web developer on earth who hasn't rolled your own hover button, you can find a detailed tutorial in *Creating a Website: The Missing Manual*, or in an online article at *www.elated.com/articles/css-rollover-buttons*.

Great as they are, pseudo-classes aren't cutting edge any longer. The problem is their all-or-nothing nature. For example, if you use the `:hover` pseudo-class, then your style settings spring into action immediately when someone hovers over an element. But in Flash applications or in desktop programs, the effect is usually more refined. The hovered-over button may shift its color, move, or begin to glow using a subtle animation that takes a fraction of a second to complete.

Some web developers have begun to add effects like these to their pages, but it usually requires the help of someone else's JavaScript animation framework. But CSS3 has a simpler solution—a *transitions* feature that lets you smoothly switch from one group of settings to another.

A Basic Color Transition

To understand how transitions work, you need to see a real example. Figure 6-10 shows a color-changing button that's bolstered with some CSS3 transition magic.

FIGURE 6-10

If this were an ordinary rollover button, its background would jump from green to yellow in one step. But with transitions, the green blends into yellow, taking half a second to make the change. Move the mouse off, and the same transition plays out in reverse, returning the button to its normal state. The result is a button that just feels more polished. (To try if for yourself, head to the try-out page at http://prosetech.com/html5.*)*

First, consider how you'd style this button the ordinary way, without using transitions. This task is basic CSS, requiring one style rule to set the button's initial appearance and a second style rule to change it when it's hovered on:

```
.slickButton {
  color: white;
  font-weight: bold;
  padding: 10px;
  border: solid 1px black;
  background: lightgreen;
  cursor: pointer;
}

.slickButton:hover {
  color: black;
  background: yellow;
}
```

Here's a button that uses these style rules:

```
<button class="slickButton">Hover Here!</a>
```

This approach is all well and good, but it lacks a certain finesse. To smooth out the green-to-yellow color change, you can create a CSS3 transition using the `transition` property. You do this in the normal `slickButton` style (not the `:hover` pseudo-class).

At a minimum, every transition needs two pieces of information: the CSS property that you want to animate and the time the browser should take to make the change. In this example, the transition acts on the `background` property, and the duration is 0.5 seconds:

```
.slickButton {
  color: white;
  font-weight: bold;
  padding: 10px;
  border: solid 1px black;
  background: lightgreen;
  cursor: pointer;
  -webkit-transition: background 0.5s;
  transition: background 0.5s;
}

.slickButton:hover {
  color: black;
  background: yellow;
}
```

As you'll no doubt notice, this example adds two transition properties instead of the promised one. That's because the CSS3 transitions standard is not quite final and some browsers still require the `-webkit-` vendor prefix.

There's one quirk in this example. The hovered-over button changes two details: its background color and its text color. But the transition applies to the background color only. As a result, the text blinks from white to black in an instant, while the new background color fades in slowly.

There are two ways to patch this up. Your first option is to set the transition property with a comma-separated list of transitions, like this:

```
.slickButton {
  ...
  -webkit-transition: background 0.5s, color 0.5s;
  transition: background 0.5s, color 0.5s;
  ...
}
```

But there's a shortcut if you want to set transitions for all the properties that change and you want to use the same duration for each one. In this case, you can simply add a single transition and use `all` for the property name:

```
-webkit-transition: all 0.5s;
transition: all 0.5s;
```

Right now, transitions work in the latest version of every browser. Old versions of Internet Explorer (IE 9 and before) don't have any transition support, and vendor prefixes won't help. However, this lack of support isn't the problem it seems. Even if a browser ignores the transition property, it still applies the effect. It just makes the change immediately, rather than smoothly fading it in. That's good news—it means a website can use transitions and keep the essentials of its visual style intact on old browsers.

More Transition Ideas

It's gratifying to see that CSS transitions can make a simple color change look good. But if you're planning to build a slick rollover effect for your buttons or menus, there are plenty of other properties you can use with a transition. Here are some first-rate ideas:

- **Transparency.** By modifying the `opacity` property, you can make an image fade away into the background. Just remember not to make the picture completely transparent, or the visitor won't know where to hover.
- **Shadow.** Earlier, you learned how the `box-shadow` property can add a shadow behind any box (page 190). But the right shadow can also make a good hover effect. In particular, consider shadows with no offset and lots of blur, which create more of a traditional glow effect. You can also use an inset shadow to put the effect inside the box.
- **Gradients.** Change up a linear gradient or add a radial one—either way, it's hard not to notice this effect.
- **Transforms.** As you'll learn on page 201, transforms can move, resize, and warp any element. That makes them a perfect tool for transitions.

On the flip side, it's usually not a good idea to use transitions with padding, margins, and font size. These operations take more processing power (because the browser needs to recalculate layout or text hinting), which can make them slow and jerky. If you're trying to make something move, grow, or shrink, you're better off using a transform (page 201).

Triggering Transitions with JavaScript

As you've seen, transitions kick in when an element switches from one style to another. If you want a nice, code-free way to make this happen, you can use pseudo-classes like `:hover` and `:focus`. But this approach has obvious limits. For example, it won't work if you want your transition to take place at another time or in response to a different event. It also won't work if you want your transition to be triggered by one element but then *affect* a different element. In situations like these, you need to chip in with a bit of JavaScript code.

Fortunately, it's easy to create a JavaScript-powered transition. As with an ordinary transition, you begin by creating two style rules, one for your element's initial state, and one for its transitioned state. Then you add the JavaScript code that finds your element and changes its style when the time is right.

WORD TO THE WISE

Don't Leave Old Browsers Behind

As you know, browsers that don't support transitions switch between states immediately, which is usually a good thing. However, if you use CSS3 glitter to make your states look different (for example, you're adding a shadow or a gradient to a hovered-over button), old browsers ignore that, too. That's not so good. It means that visitors with less capable browsers get *no* hover effect at all.

To solve this problem, use a fallback that older browsers understand. For example, you might create a hover state that sets a different background color and *then* sets a gradient. This way, older browsers will see the background change to a new solid color when the button is hovered over. More capable browsers will see the background change to a gradient fill. For even more customizing power, you can use Modernizr, which lets you define completely different styles for older browsers (page 31).

Figure 6-11 shows an example that uses a code-powered transition. In this page, two images are layered over each other—an image of a city skyline in the day and an image of the same skyline at night. The buttons use a few simple lines of JavaScript to trigger a transition that hides or shows the night-time image.

The first step to creating this example is adding an image-formatting style sheet rule. It does two things: switches the images to absolute positioning (so they can be placed on top of one another in their containing `<div>`), and defines the type of transition you plan to carry out. In this case, it's a 10-second transition that alters the opacity of the nighttime image.

```
img {
  position: absolute;
  -webkit-transition: opacity 10s;
  transition: opacity 10s;
}
```

You also need two style rules to represent the different possible states for the nighttime image, which begins fully transparent but can become solid at the click of a button:

```
.solid {
  opacity: 1;
}

.transparent {
  opacity: 0;
}
```

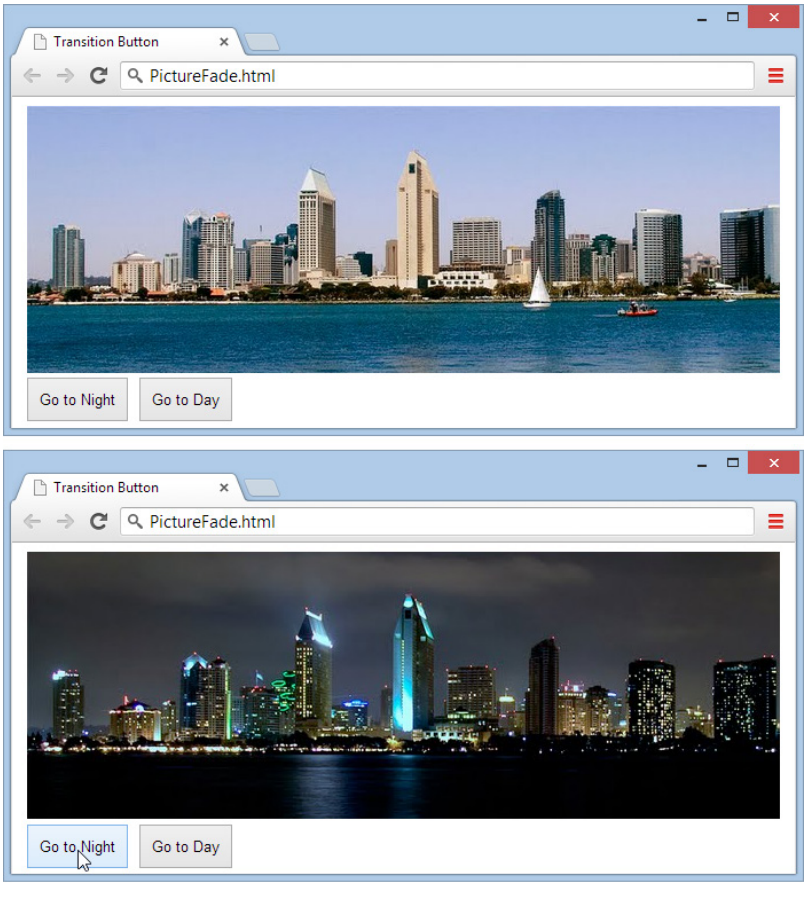


FIGURE 6-11
Initially, the image with the night skyline is completely transparent (top). But click the To Night button and the night image fades in, gradually blotting out the day skyline (bottom).

The page markup places both images into a , taking care to apply the transparent class to the second image, and defines two buttons that are hard-wired into the JavaScript functions you need.

```
<div>
  <img src="day.png" alt="Daytime view">
  <img id="nightImage" src="night.png" alt="Night-time view"
class="transparent">
</div>

<button onclick="toNight()">Go to Night</button>
<button onclick="toDay()">Go to Day</button>
```

The final step is to add the code that reacts to the button clicks, finds the nighttime image, and switches its style:

```
function toNight() {
  var nightImage = document.getElementById("nightImage");
  nightImage.className = "solid";
}

function toDay() {
  var nightImage = document.getElementById("nightImage");
  nightImage.className = "transparent";
}
```

Although this seems like a single abrupt act, the change will phase in steadily over 10 seconds, thanks to the transition that's defined for all `<img>` elements.

Remember, transitions take place only if your visitor has a modern browser. If someone visits your site with IE 9 and clicks the To Night or To Day button, the page will shift abruptly from one style to the other, with no 10-second blending effect. Unfortunately, there's no easy polyfill to patch this gap, and it's all too easy to ignore what your pages will look like on less capable browsers when you start weaving transitions into your code

NOTE If animated effects are an essential part of your pages, you're probably not quite ready to embrace CSS3. Instead, the most practical solution for transitions today is a JavaScript library like jQuery UI or MooTools. But CSS3 is the clear future of web effects, once the standards settle down and modern browsers have colonized the computers of the world.

Transforms

A *transform* is a powerful tool that lets you move, scale, skew, or rotate an element, warping its appearance. With CSS3 transforms, you use them to change the appearance of an element. Like transitions, transforms are a new and experimental feature. To use them, you need to use the `transform` property. Here's an example that rotates an element and all its contents:

```
.rotatedElement {
  transform: rotate(45deg);
}
```

To get your transforms to work on Chrome, Safari, and Opera, you need to add the `-webkit-` vendor prefix. On Internet Explorer 9, you need the `-ms-` prefix (although IE 10 and later don't need any prefix). Firefox doesn't need a prefix. So unless you're using the -prefix-free tool (page 185), the proper way to use a transform looks like this:

```
.rotatedElement {
  -ms-transform: rotate(45deg);
  -webkit-transform: rotate(45deg);
  transform: rotate(45deg);
}
```

POWER USERS' CLINIC

Making More Natural Transitions

The `transition` property is an all-in-one property that combines several details. So far, you've seen how to give a transition a duration and specify the property it acts on. However, there are two more details you can use to fine-tune your transition.

First, you can choose a *timing function* that controls how the transition effect flows—for example, whether it starts slow and then speeds up or starts fast and then decelerates. In a short transition, the timing function you choose doesn't make much of a difference. But in a longer, more complex animation, it can change the overall feel of the effect. Here's an example that uses the `ease-in-out` timing function so that a transition starts slow, then accelerates, and then slows back down at the end:

```
transition: opacity 10s ease-in-out;
```

Other timing functions include `linear` (the transition has a constant rate from start to finish), `ease-in` (the transition starts slow and then goes at a constant rate), `ease-out` (the transition starts at a constant rate but slows at the end), and `cubic-bezier` (the transition goes according to a Bézier curve that you define, you math lover, you).

No matter what timing function you choose, the whole transition takes the same amount of time—the duration you've specified. The difference is in how the transition speeds up or slows down as it takes place. To review the different timing functions and get a feel for how each one alters the pace of a transition, you can see them in action with a helpful series of rolling square animations at *http://css3.bradshawenterprises.com/transitions/*.

The other transition detail that you can add is an optional delay that holds off the start of the transition for some period of time. Here's an example that waits 0.1 seconds:

```
transition: opacity 10s ease-in-out 0.1s;
```

In this example, the `rotate()` function does the work, twisting an element 45 degrees around its center. However, there are many more transform functions that you can use, separately or at the same time. For example, the following style chains three transforms together. It enlarges an element by half (using the `scale` transform), moves it 10 pixels to the left (using the `scaleX` transform), and skews it for effect (using the `skew` transform):

```
.rotatedElement {
  -ms-transform: scale(1.5) scaleX(10px) skew(10deg);
  -webkit-transform: scale(1.5) scaleX(10px) skew(10deg);
  transform: scale(1.5) scaleX(10px) skew(10deg);
}
```

A skew twists an element out of shape. For example, imagine pushing the top edge of a box out to the side, while the bottom edge stays fixed (so it looks like a parallelogram).

Table 6-2 lists all the two-dimensional transform functions you can use. To remove all your transforms, set the `transform` property to none.

NOTE Transforms don't affect other elements or the layout of your web page. For example, if you enlarge an element with a transform, it simply overlaps the adjacent content.

TABLE 6-2 *Transform functions*

FUNCTION	DESCRIPTION
`translateX(x)`	Moves an element horizontally. Use a positive value to shift it to the right, and a negative value to shift it to the left.
`translateY(y)`	Moves an element vertically. Use a positive value to shift it down, and a negative value to shift it up.
`translate(x, y)`	Moves an element vertically and horizontally.
`scaleX(x)`	Scales an element horizontally. Use a value greater than 1.0 to enlarge it (2.0 is twice as big) and a value between 0 and 1.0 to shrink it (0.5 is half as big). Use a negative value to flip the element around the *y*-axis, creating a right-to-left mirror image.
`scaleY(y)`	Scales an element vertically. Use a value greater than 1.0 to enlarge it and a value between 0 and 1.0 to shrink it. Use a negative value to flip the element around the *x*-axis, creating a bottom-to-top mirror image.
`scale(x, y)`	Scales an element horizontally and vertically.
`rotate(angle)`	Rotates an element clockwise around its center. Use a negative value to rotate the element counter-clockwise. If you want to rotate an element around another point, use the CSS `transform-origin` property.
`skewX(angle)`	Tilts an element horizontally. The top and bottom edges remain level, but the sides are pulled out of alignment.
`skewY(angle)`	Tilts an element vertically. The left and right edges remain in place, but the top and bottom are slanted.
`skew(x-angle, y-angle)`	Tilts an element horizontally and vertically.
`matrix(n1, n2, n3, n4, n5, n6)`	Uses matrix multiplication to move each of the corners of the element. The matrix, which is represented by six numbers, can duplicate any other transform (or any combination of transforms). However, you're unlikely to build the matrix you need yourself, even if you're a math nerd. Instead, you'll probably use a tool that provides you with the ready-made matrix you want.

NOTE When you get tired of moving an element around in two dimensions, you can use 3-D transforms to move, rotate, and warp it in three-dimensional space. You'll find several good, interactive examples of 3-D transforms at *http://tinyurl.com/3d-transitions*.

POWER USERS' CLINIC

How to Shift the Starting Point

Ordinarily, transforms are made using the center point of your element as a reference point. You can shift this reference point by using the `transform-origin` property before you apply your transform. For example, here's how you can rotate a shape around its top-left corner:

```
.rotatedElement {
  -ms-transform-origin: 0% 0%;
  -webkit-transform-origin: 0% 0%;
  transform-origin: 0% 0%;

  -ms-transform: rotate(45deg);
  -webkit-transform: rotate(45deg);
  transform: rotate(45deg);
}
```

To rotate around the top-right corner, you'd use a value of 100% 0%. You can even specify a far-off origin that doesn't appear in your element (like 50% 200%, which puts the reference point halfway across the element, and a distance down that's equal to twice the element's height).

By default, the `transform-origin` property is set to 50% 50%, which puts the center point exactly in the middle of your element.

Transitions That Use Transforms

Transforms and transitions make a natural pair. For example, imagine you want to create an image gallery, like the one shown in Figure 6-12.

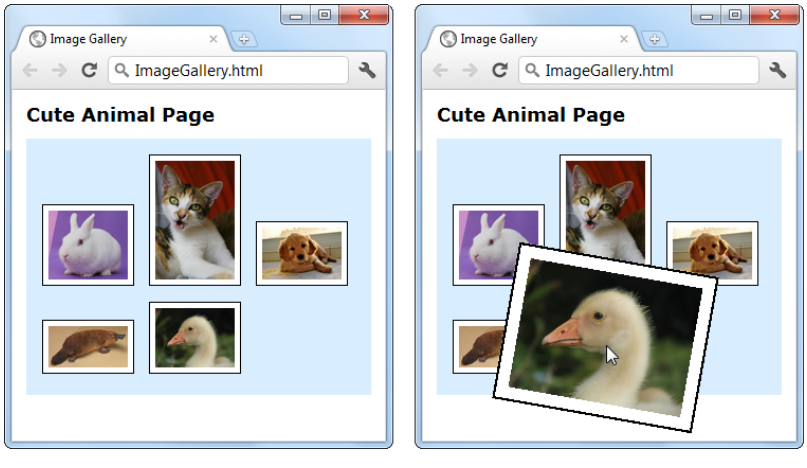


FIGURE 6-12
Here, a transform makes the hovered-over image stand out.

This example starts out simple enough, with a bunch of images wrapped in a `<div>` container:

```
<div class="gallery">
  <img src="bunny.jpg">
  <img src="cat.jpg">
  <img src="dog.jpg">
  <img src="platypus.jpg">
  <img src="goose.jpg">
</div>
```

Here's the style for the that holds all the images:

```
.gallery {
  margin: 0px 30px 0px 30px;
  background: #D8EEFE;
  padding: 10px;
}
```

And here's how each <img> element starts off:

```
.gallery img {
  margin: 5px;
  padding: 5px;
  width: 75px;
  border: solid 1px black;
  background: white;
}
```

Notice that all the images are given explicit sizes with the width property. That's because this example uses slightly bigger pictures that are downsized when they're shown on the page. This technique is deliberate: It makes sure the browser has all the picture data it needs to enlarge the image with a transform. If you didn't take this step, and used thumbnail-sized picture files, the enlarged versions would be blurry.

Now for the hover effect. When the user moves the mouse over an image, the page uses a transform to rotate and expand the image slightly:

```
.gallery img:hover {
  -ms-transform: scale(2.2) rotate(10deg);
  -webkit-transform: scale(2.2) rotate(10deg);
  transform: scale(2.2) rotate(10deg);
}
```

Right now, this transform snaps the picture to its new size and position in one step. But to make this effect look more fluid and natural, you can define an all-encompassing transition in the normal state:

```
.gallery img {
  margin: 5px;
  padding: 5px;
  width: 75px;
  border: solid 1px black;
```

```
    -ms-transition: all 1s;
    -webkit-transition: all 1s;
    transition: all 1s;
    background: white;
}
```

Now the picture rotates and grows over a time span of 1 second. Move the mouse away, and it takes another second to shrink back to its original position.

Web Fonts

With all its pizzazzy new features, it's hard to pick the best of CSS3. But if you had to single out just one feature that opens an avalanche of new possibilities and is ready to use *right now*, that feature may just be web fonts.

In the past, web designers had to work with a limited set of web-safe fonts. These are the few fonts that are known to work on different browsers and operating systems. But as every decent designer knows, type plays a huge role in setting the overall atmosphere of a document. With the right font, the same content can switch from coolly professional to whimsical, or from old-fashioned to futuristic.

NOTE There were good reasons why web browsers didn't rush to implement custom web fonts. First, there are optimization issues, because computer screens offer far less resolution than printed documents. If a web font isn't properly tweaked for onscreen viewing, it'll look like a blurry mess at small sizes. Second, most fonts aren't free. Big companies like Microsoft were understandably reluctant to add a feature that could encourage web developers to take the fonts installed on their computers and upload them to a website without proper permission. As you'll see in the next section, font companies now have good solutions for both problems.

CSS3 adds support for fancy fonts with the `@font-face` feature. Here's how it works:

1. You upload the font to your website (or, more likely, multiple versions of that font to support different browsers).
2. You register each font-face you want to use in your style sheet, using the `@font-face` command.
3. You use the registered font in your styles, by name, just as you use the web-safe fonts.
4. When a browser encounters a style sheet that uses a web font, it downloads the font to its temporary cache of pages and pictures. It then uses that font for just your page or website (Figure 6-13). If other web pages want to use the same font, they'll need to register it themselves and provide their own font files.

NOTE Technically, `@font-face` isn't new. It was a part of CSS 2, but dropped in CSS 2.1 when browser makers couldn't cooperate. Now, in CSS3, there's a new drive to make `@font-face` a universal standard.

The following sections walk you through these essential steps.

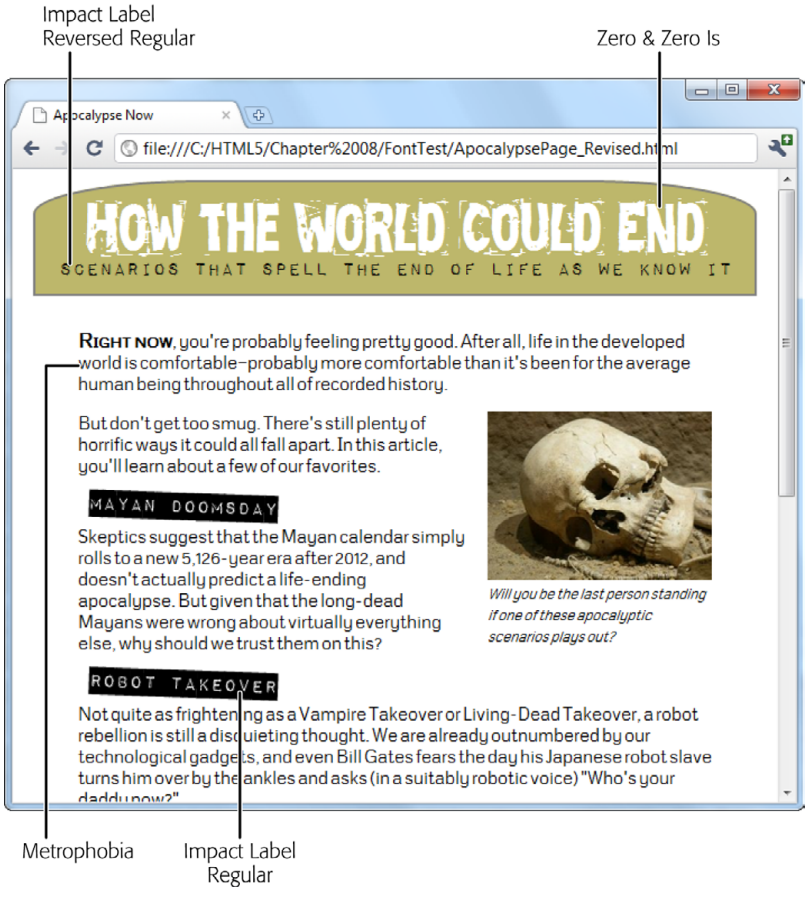


FIGURE 6-13
This revised version of the apocalyptic page combines a hodgepodge of four fonts. All of these fonts are free, and you'll learn how to get them for yourself from Font Squirrel on page 209.

Web Font Formats

Although all current browsers support `@font-face`, they don't all support the same *types* of font files. Internet Explorer, which has supported `@font-face` for years, supports only EOT (Embedded OpenType) font files. This format has a number of advantages—for example, it uses compression to reduce the size of the font file, and it allows strict website licensing so a font can't be stolen from one website and used on another. However, the EOT format never caught on, and no other browser uses it. Instead, other browsers have (until recently) stuck with the more familiar font standards used in desktop computer applications—that's TTF (TrueType) and OTF (OpenType PostScript). But the story's still not complete without two more acronyms—SVG and WOFF. Table 6-3 puts all the font formats in perspective.

TABLE 6-3 *Embedded font formats*

FORMAT	DESCRIPTION	USE WITH
WOFF (Web Open Font Format)	The single font format of the future. Newer browsers support it.	Any browser that supports it, starting with Internet Explorer 9, Firefox 3.6, and Chrome 6.
EOT (Embedded Open Type)	A Microsoft-specific format that never caught on with browsers except Internet Explorer.	Internet Explorer (before IE 9)
TTF (TrueType) OTF (OpenType PostScript)	Your font will probably begin in one of these common desktop formats.	Mobile devices using the Android operating system and (optionally) non-IE browsers, such as Firefox, Chrome, Safari, and Opera
SVG (Scalable Vector Graphics)	An all-purpose graphics format you can use for fonts, with good but not great results (it's slower to display and produces lower-quality text).	Old mobile versions of Safari (before iOS 4.2), and (optionally) mobile devices using the Android operating system.

Bottom line: If you want to use the `@font-face` feature and support a wide range of browsers, you need to distribute your font in several different formats. The best practice is to include a WOFF file (for optimum performance on modern browsers), an EOT file (to fill in the gaps on old versions of IE), and a TTF or OTF file (to fill in support for Android and older non-IE browsers). It's also a good idea to supply a lower-quality SVG file to satisfy old iPads and iPhones.

If you think that's too many font files to manage, you can strip this down to an absolute minimum and cover most browsers with just two files: a font in the TTF or OTF format (either one is fine), and a font in the EOT format. This won't satisfy everyone, but it will give fancy fonts to the vast majority of people who visit your site.

NOTE Fortunately, font vendors and online font services will usually supply you with all four font formats you need, so you can guarantee the best possible level of browser support.

Finding a Font for Your Website

Now that you know where you can get the font files you need for your website, in which formats, you need to get your hands on them. You have two possibilities:

- **Download a free web font.** This way, you don't need to worry about licensing details. You can keep your wallet closed.

- **Convert a desktop font you already have into a web font.** This approach lets you use a font that you've already fallen in love with. It's also great for consistency—for example, if you work in a company that already has a standard set of fonts that it uses in logos, memos, and publications, it makes sense to stick with the same typefaces online. However, you'll need to do a bit of research to figure out the licensing situation, and you may need to cough up some more cash.

In the following sections, you'll try both approaches.

TROUBLESHOOTING MOMENT

Ironing Out the Quirks

Even if you follow the rules and supply all the required font formats, expect a few quirks. Here are some problems that occasionally crop up with web fonts:

- Many fonts look bad on the ancient but still-popular Windows XP operating system, because Windows XP computers often have the anti-aliasing display setting turned off. (And fonts without anti-aliasing look as attractive as mascara on a mule.)
- Some people have reported that some browsers (or some operating systems) have trouble printing certain embedded fonts.
- Some browsers suffer from a problem known as FOUT (which stands for Flash of Unstyled Text). This phenomenon occurs when an embedded font takes a few seconds to download, and the page is rendered first using a fallback font, and then re-rendered using the embedded font. This problem is most noticeable on old builds of Firefox. If it really bothers you, Google provides a JavaScript library that lets you define fallback styles that kick in for unloaded fonts, giving you complete control over the rendering of your text at all times (see *http://tinyurl.com/font-loader*).

Although these quirks are occasionally annoying, most are being steadily ironed out in new browser builds. For example, Firefox now minimizes FOUT by waiting for up to 3 seconds to download an embedded font before using the fallback font.

Getting a Free Font from Font Squirrel

One of the best places to find free web fonts is the Font Squirrel website at *www.fontsquirrel.com*. It provides a catalog of roughly 1,000 free-to-use fonts (see Figure 6-14).

When you find a font you like in Font Squirrel's list, start by checking the tiny icons underneath (Figure 6-15). These icons indicate how the font is licensed. Solid icons indicate that the font can be used in a particular context; white outlines indicate that it cannot.

If a font's licensing details check out (and on Font Squirrel, they almost always do), the next step is to take a closer look at your font. Click on the font text to switch to a font preview page, which shows every letter of the font and lets you test drive it on some text you type in.

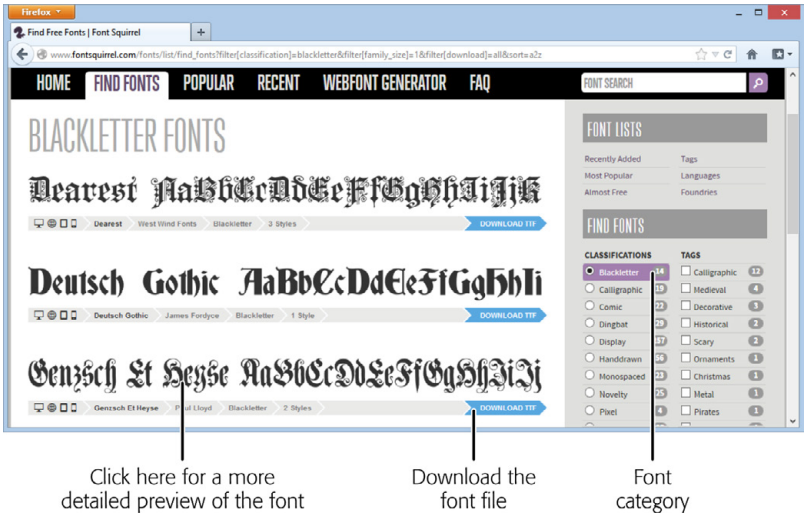


FIGURE 6-14

Font Squirrel gives you several options for font hunting, but the most effective way to find what you want is to browse by type ("Calligraphic," "Novelty," and "Retro," for example). Best of all, most fonts are free to use wherever you want—on your personal computer to create documents, or on the Web to build web pages.

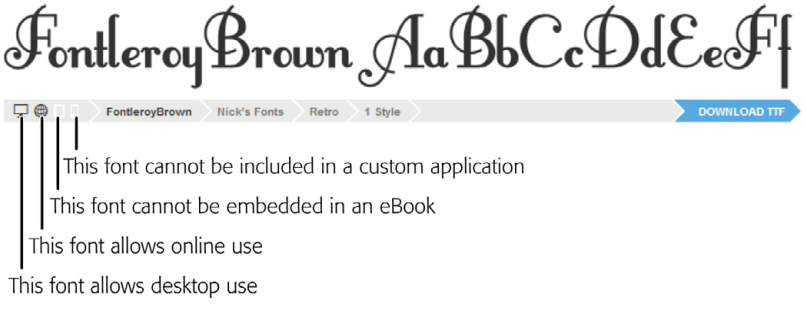


FIGURE 6-15

The first two icons are the most important—they indicate that the font is allowed both on the desktop and on the Web. Virtually all Font Squirrel fonts include these two icons. The next two icons indicate whether you can use the font in ebooks and custom-built applications. (If you're not sure what an icon means, just click it to find out.)

If you like what you see, the final step is to download the font. Depending on the font, you may be able to download a complete web-ready font kit, or—more likely—you'll need to download just the TTF or OTF file and then create your own kit. This quirk is largely due to the messy world of font licensing. Many fonts use the SIL Open Font License, which makes the font free for everyone to use but doesn't allow a service like Font Squirrel to repackage it. Fortunately, creating your own kit is easy—and perfectly legit. You'll learn how to do that in the next section.

FREQUENTLY ASKED QUESTION

Using a Font on Your Computer

Can I use the same font for web work and printed documents?

If you find a hot new font to use in your website, you can probably put it to good use on your computer, too. For example, you might want to use it in an illustration program to create a logo. Or, your business might want to use it for other print work, like ads, fliers, product manuals, and financial reports.

Modern Windows and Mac computers support TrueType (.ttf) and OpenType (.otf) fonts. Every font package includes a font in one of these formats—usually TrueType. To install it in Windows, make sure you've pulled it out of the ZIP download file. Right-click the font file and then choose Install. (You can do this with multiple font files at once.) On a Mac, double-click the font file to open the Font Book utility. Then, click the Install Font button.

Preparing a Font for the Web

Using Font Squirrel, you can convert a standard desktop TTF or OTF format font file into a web-ready font that you can use in any web page. You can do this with any of the free fonts you download from Font Squirrel. You *may* also be able to do this with the fonts on your own computer, but it's important to understand the licensing issues that you'll face first (see the box below). Using an ordinary desktop font in an online website without permission is likely to be a breach of copyright—and, if the font maker has asked Font Squirrel to blacklist the font, you won't be allowed to perform the desktop-to-web conversion anyway.

UP TO SPEED

Understanding the Rules of Font Licensing

Ordinary fonts used in desktop software aren't free. It's not kosher to take a font you have on your computer and use it on your website, unless you have explicit permission from the font's creator.

For example, Microsoft and Apple pay to include certain fonts with their operating systems and applications so you can use them to, say, create a newsletter in a word processor. However, this license doesn't give you permission to put these fonts on a web server and use them in your pages.

If you have a favorite font, the only way to know whether you need to pay for it is to contact the company or individual that made it. Some font makers charge licensing fees based on the amount of traffic your website receives. Other font creators may let you use their fonts for a nominal amount or for free, provided you meet certain criteria (for example, you include some small-print note about the font you're using, or you have a noncommercial website that isn't out to make boatloads of money). There's also a side benefit to reaching out: Skilled font makers often provide versions of their creations that are optimized for web viewing.

Once you know that you're allowed to use a specific font, you can convert it using Font Squirrel's handy web font generator. To get there, click on the Webfont Generator tab near the top of the Font Squirrel site, or just surf directly to *www.fontsquirrel.com/fontface/generator*. Figure 6-16 shows you the three-step process you need to follow.

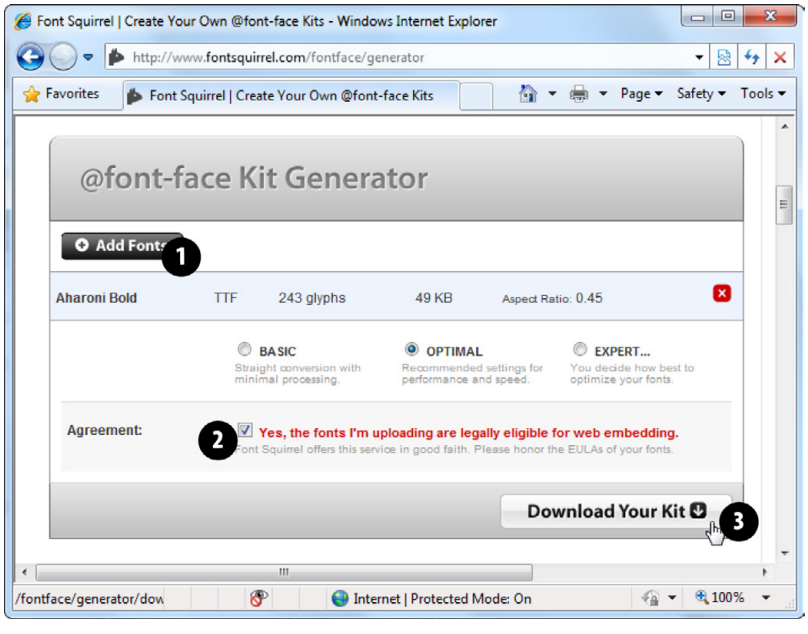


FIGURE 6-16

First, click Add Fonts to upload a font file from your computer (#1). Then, add a checkmark to the setting "Yes, the fonts I'm uploading are legally eligible for web embedding," assuming you've reviewed their license requirements, as described on page 211 (#2). Finally, click Download Your Kit (#3).

When you download a font kit, you get a compressed Zip file that contains a number of files. For example, if you download the Chantelli Antiqua font, then you get these files:

```
Bernd Montag License.txt
Chantelli_Antiqua-webfont.eot
Chantelli_Antiqua-webfont.svg
Chantelli_Antiqua-webfont.ttf
Chantelli_Antiqua-webfont.woff
demo.html
stylesheet.css
```

The text file (*Bernd Montag License.txt*) provides licensing information that basically says you can use the font freely but never sell it. The Chantelli_Antiqua-webfont files provide the font in four different file formats. (Depending on the font you pick, you may get additional files for different variations of that font—for example, in bold, italic, and extra-dark styles.) Finally, the *stylesheet.css* file contains the style sheet rule you need to apply the font to your web page, and *demo.html* displays the font in a sample web page.

To use the Chantelli Antiqua font, you need to copy all the Chantelli_Antiqua-webfont files to the same folder as your web page. Then you need to register the font so that it's available for use in your style sheet. To do that, you use a complex `@font-face` rule at the beginning of your style sheet, which looks like this (with the lines numbered for easy reference):

```
1  @font-face {
2    font-family: 'ChantelliAntiquaRegular';
3    src: url('Chantelli_Antiqua-webfont.eot');
4    src: local('Chantelli Antiqua'),
5      url('Chantelli_Antiqua-webfont.woff') format('woff'),
6      url('Chantelli_Antiqua-webfont.ttf') format('truetype'),
7      url('Chantelli_Antiqua-webfont.svg') format('svg');
8  }
```

To understand what's going on in this rule, it helps to break it down line by line:

- **Line 1.** `@font-face` is the tool you use to officially register a font so you can use it elsewhere in your style sheet.
- **Line 2.** You can give the font any name you want. You'll use this name later, when you apply the font.
- **Line 3.** The first format you specify must be the file name of the EOT file. That's because Internet Explorer gets confused by the rest of the rule and ignores the other formats. The `url()` function is a style sheet technique that tells a browser to download another file at the location you specify. If you put the font in the same folder as your web page, then you can simply provide the file name here.
- **Line 4.** The next step is running the `local()` function. This function tells the browser the font name, and if that font just happens to be installed on the visitor's computer, the browser uses it. However, in rare cases this can cause a problem (for example, it could cause Mac OS X to show a security dialog box, depending on where your visitor has installed the font, or it could load a different font that has the same name). For these reasons, web designers sometimes use an obviously fake name to ensure that the browser finds no local font. One common choice is to use a meaningless symbol like `local('☺')`.
- **Lines 5 to 7.** The final step is to tell the browser about the other font files it can use. If you have a WOFF font file, suggest that first, as it offers the best quality. Next, tell the browser about the TTF or OTF file, and finally about the SVG file.

TIP Of course, you don't need to type the `@font-face` rule by hand (and you definitely don't need to understand all the technical underpinnings described above). You can simply copy the rule from the *stylesheet.css* file that's included in the web font kit.

Once you register an embedded font using the `@font-face` feature, you can use it in any style sheet. Simply use the familiar `font-family` property, and refer to the font family name you specified with `@font-face` (in line 2). Here's an example that leaves out the full `@font-face` details:

```
@font-face {
  font-family: 'ChantelliAntiquaRegular';
  ...
}

body {
  font-family: 'ChantelliAntiquaRegular';
}
```

This rule applies the font to the entire web page, although you could certainly restrict it to certain elements or use classes. However, you must register the font with `@font-face` *before* you use it in a style rule. Reverse the order of these two steps, and the font won't work properly.

Even Easier Web Fonts with Google

If you want a simpler way to use a fancy font on your website, Google has got you covered. It provides a service called Google Fonts (formerly Google Web Fonts), which hosts free fonts that anyone can use. The beauty of Google Fonts is that you don't need to worry about font formats, because Google detects the user's browser and automatically sends the right font file. All you need to do is add a link to a Google-generated style sheet.

To use a Google font in your pages, follow these steps:

1. **Go to *www.google.com/fonts*.**

   Google shows you a long list of available fonts (Figure 6-17).

2. **At the top of the page, click a tab title (Word, Sentence, or Paragraph) to choose how you preview fonts.**

   For example, if you're hunting for a font to use in a heading, you'll probably choose Word or Sentence to take a close look at a single word or line of text. But if you're looking for a font to use in your body text, you'll probably choose Paragraph to study a whole paragraph of text at once. No matter what option you choose, you can type in your own preview text and set an exact font size for your previews.

3. **Set your search options.**

   If you have a specific font in mind, type it into the search box. Otherwise, you'll need to scroll down, and that could take ages. To help you get what you want more quickly, start by setting a sort order and some filtering options, if they apply (for example, you might want to find the most popular bold sans-serif fonts). Figure 6-17 shows you where to find these options.

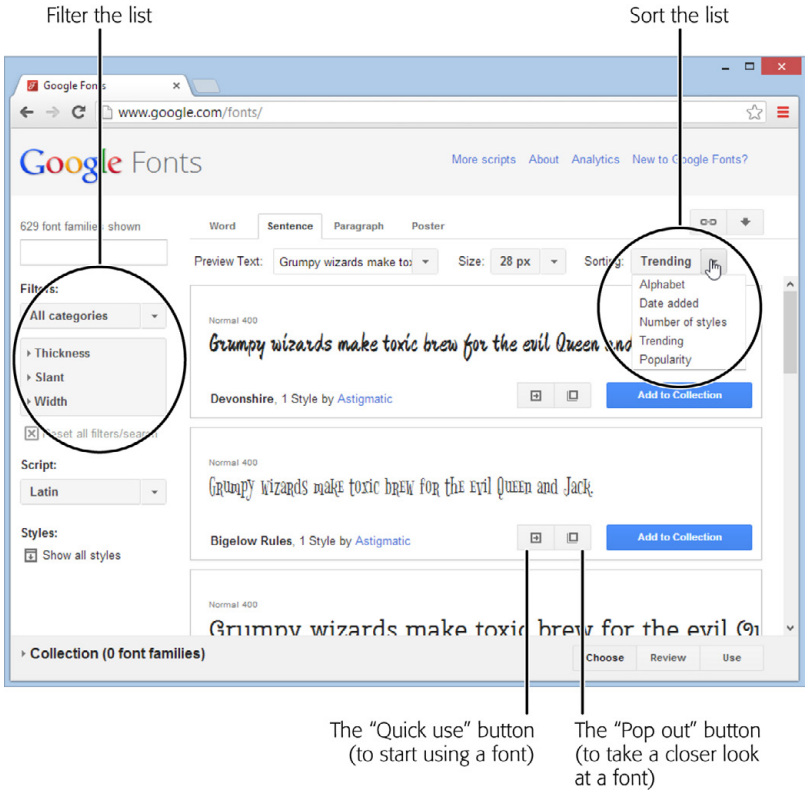


FIGURE 6-17

Google has a relentlessly expanding selection of fonts. When you're looking for a font, you'll probably want to tweak the font list's sorting and filtering options (circled). For example, you can sort alphabetically or put the most popular fonts first, and you can filter out just serif, sans-serif, or hand-written (cursive) fonts.

4. **When you see a font that's a candidate for your site, click the "Pop out" button.**

   Google pops open an informative window that describes the font and shows each of its characters.

5. **If you like the font, click the "Quick-use" button to get the information you need to use it.**

   Google shows you the code you need to use this font. It consists of a style sheet link (which you must add to your web page) and an example of a style sheet rule that uses the font.

6. **Add a style sheet link to your web page.**

   For example, if you picked the Metrophobic font, Google wants you to place the following link in the <head> section of your page:

```
<link href="http://fonts.googleapis.com/css?family=Metrophobic"
rel="stylesheet">
```

This style sheet registers the font, using `@font-face`, so you don't have to. Best of all, Google provides the font files, so you don't need to upload anything extra to your website.

NOTE Remember to put the link for the Google font style sheet before your other style sheet links. That way, your other style sheets can use the Google font.

7. **Use the font, by name, wherever you want.**

   For example, here's how you could use the newly registered Metrophobic font in a heading, with fallbacks in case the browser fails to download the font file:

```
h1 {
  font-family: 'Metrophobic', arial, serif;
}
```

POWER USERS' CLINIC

Creating a Font Collection

These steps show the fastest way to get the markup you need for a font. However, you can get more options by creating a *font collection*.

A font collection is a way to package up multiple fonts. To start creating one, you simply click the "Add to Collection" button next to a font you like. As you add fonts to your collection, each one appears in the fat blue footer at the bottom of the page.

When you're finished picking the fonts you want, click the Use button in the footer. Google then shows a page that's similar to the "Quick-use" page, except that it allows you to create a single style sheet reference that supports *all* the fonts from your custom-picked collection.

When you create a font collection, you can also use two buttons that appear at the top right of the page. Click the Bookmark button (which looks like a link in a chain) to create a browser bookmark that lets you load up the same collection at some point in the future, so you can tweak it. Click the Download button (which looks like a down-pointing arrow) to download copies of the fonts to your computer, so you can install the fonts and use them for print work.

TIP Still looking for the perfect font? Popular font subscription sites like *http://fonts.com* and *http://typekit.com* give you access to thousands of ultra-high-quality typefaces from legendary font foundries like Linotype and Monotype. Font-addicted web developers will need to pay from $10 to $100 a year, with more money needed to outfit super-popular sites that get avalanches of web traffic.

Putting Text in Multiple Columns

Fancy fonts aren't the only innovation CSS3 has for displaying text. It also adds a module for multicolumn text, which gives you a flexible, readable way to deal with lengthy content.

Using multiple columns is almost effortless, and you have two ways to create them. Your first option is to set the number of columns you want using the `column-count` property, like this:

```
.Content {
  text-align: justify;
  column-count: 3;
}
```

As of this writing, the `column-count` property works only in Internet Explorer 10 and 11. Although multiple columns are supported by Chrome, Firefox, Safari, and Opera, you need to use the vendor-prefixed versions of the `column-count` property, like this:

```
.Content {
  text-align: justify;
  -moz-column-count: 3;
  -webkit-column-count: 3;
  column-count: 3;
}
```

This approach—creating a set number of columns—works well for fixed-size layouts. But if you have a space that grows and shrinks with the browser window, your columns may grow too wide and become unreadable. In this situation, it's better *not* to set the exact number of columns. Instead, tell the browser how big each column should be using the `column-width` property:

```
.Content {
  text-align: justify;
  -moz-column-width: 10em;
  -webkit-column-width: 10em;
  column-width: 10em;
}
```

The browser can then create as many columns as it needs to fill up the available space (see Figure 6-18).

NOTE You can use pixel units to size a column, but em units make more sense. That's because em units adapt to the current font size. So if a web page visitor ratchets up the text size settings in her browser, the column width grows proportionately to match. To get a sense of size, 1 em is equal to two times the current font size. So if you have a 12 pixel font, 1 em works out to 24 pixels.

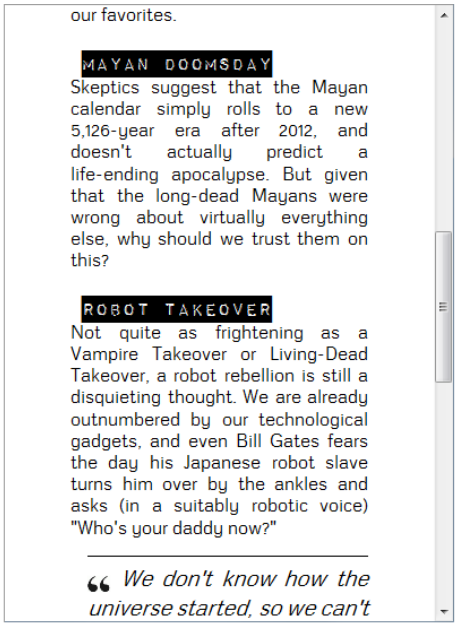


FIGURE 6-18

In a narrow window (top), Firefox can accommodate just one column. But widen the window, and you'll get as many more as can fit (bottom).

CSS3 provides a few more properties for tailoring the look of your columns. You can adjust the size of the spacing between columns with `column-gap`. You can also add a vertical line to separate them with `column-rule`, which accepts a thickness, border style, and color (just like the `border` property). Here's an example that makes a red, 1-pixel-wide column rule:

```
-webkit-column-rule: 1px solid red;
-moz-column-rule: 1px solid red;
column-rule: 1px solid red;
```

You can also use the `column-span` property to let figures and other elements span columns. The default value of `column-span` is 1, which means the element is locked in the single column where it appears. The only other acceptable value is `all`, which lets an element stretch across the entire width of all the columns. There's currently no way to let an element span some but not all columns.

Here's an example (shown in Figure 6-19) that uses column spanning with a figure:

```
.SpanFigure {
  -moz-column-span: all;
  -webkit-column-span: all;
  column-span: all;
}
```

This technique doesn't work for figures that set the `float` property to something other than `none`. That's because floating figures already have the ability to float free of your layout and any columns it contains.

NOTE Columns work well if you need to break up text to make it more readable on wide layouts. However, columns aren't the best choice for truly large amounts of content, since there's currently no way to tie the height of a column to the height of the browser window. So if you split a lengthy essay into three columns, the reader will need to scroll from top to bottom to read the first column, then back to the top, then down to the bottom to read the second column, and again for the third. If the content is more than a screenful or two, all this scrolling gets old fast.

FIGURE 6-19

This tweaked-up multicolumn page adds a rule between columns and lets figures span multiple columns.

CHAPTER

7

Responsive Web Design with CSS3

When web designers first started putting their content into HTML pages, they faced a challenge. Whereas print designers could rely on certain assumptions about how their documents would be arranged on paper and how they would be read by their audiences, the online world was loose and lawless. Depending on the user's browser (and personal preferences), the same HTML page might appear wedged in a tiny window or floating in a giant one. This made complex layouts risky. A layout that looked perfect in one window could easily turn into an awkward and ungainly mess when viewed in a window with different proportions.

Today, this variability has only increased. Not only do web designers need to think about different sizes of browser windows on desktop computers, but they also need to accommodate different sizes of *devices*, like tablets and smartphones. And at the same time, website layouts have become more intricate, with most sites now composed of menus, navigation aids, sidebars, and so on. If your goal is to create a single website that can shift gracefully between different viewing contexts, these details present a significant challenge.

Because web designers have long since outsourced the layout and formatting work of their web pages to CSS, it makes sense for CSS to provide the solution for this problem. Fortunately, CSS3 has the perfect tool: a feature called *media queries*, which lets your website seamlessly switch from one set of styles to another depending on the window size or the viewing device.

Media queries are an essential technique for mobile web development. But even if you don't expect any visitors to surf your site on a smartphone, media queries will still help you ensure that your layout adapts itself to the viewer—for example, dropping an extra column when there's no space to show it comfortably, or moving

the navigation links from the top of a page to its side. This sort of adaptation is part of a wildly popular web design philosophy called *responsive design*, which you'll explore in this chapter.

Responsive Design: The Basics

The problem of varying window sizes has been around since the dawn of the Web. Over the years, web designers have cooked up a variety of complementary techniques—some elegant, some messy—to cope with the challenges of responsive design.

Before you learn how to use media queries, it's important to consider the following traditional tactics. All of them are still important today—but, as you'll see, they don't form a complete solution on their own. Once you recognize their limits, you'll understand how CSS3 patches the gaps.

Fluid Layout

The simplest solution to the problem of resizing windows is to make a *proportional* layout—one that simply sucks up the available space, no matter how large or small it is.

Creating a proportional layout is easy enough in theory. The basic principle is to carve up your page into columns using percentage sizes instead of pixel sizes. Say, for example, you have a two-column layout like this:

```
<body>
  <div class="leftColumn">
    ...
  </div>

  <div class="rightColumn">
    ...
  </div>
</body>
```

The style rules for a fixed layout might look like this:

```
.leftColumn {
  width: 275px;
  float: left;
}

.rightColumn {
  width: 685px;
  float: left;
}
```

```
body {
  margin: 0px;
}
```

But the style rules for a proportionately sized layout would look like this:

```
.leftColumn {
  width: 28.6%;
  float: left;
}

.rightColumn {
  width: 71.4%;
  float: left;
}

body {
  margin: 0px;
}
```

Here, the left column has a width of 28.6%, so it takes 28.6% of the width of its container, which is the <body> element. In this example, the <body> element has no margins, so it takes up the full width of the browser window, and the left column gets 28.6% of that.

As you would expect, the percentages of the two columns combined add up to exactly 100%, filling the page. No matter what the size of the browser window, the columns expand or shrink to match. Proportional layouts are also called *fluid layouts*, because the content flows seamlessly into whatever space is available.

NOTE In this example, the left column width of 28.6% is calculated by dividing the fixed width of the column (275 pixels) into the fixed width of the entire layout (which, previously, was set at the relatively common default width of 960 pixels). Because most layouts are initially planned using fixed widths, web developers are accustomed to using this sort of calculation when they create fluid layouts.

Of course, it's not enough to adjust column sizes alone. You also need to think about margins, padding, and borders. When novice web developers create their first fluid layouts, they often leave in fixed margins and padding (using pixel values), while sizing their columns proportionately. As a result, the columns can occupy only the space that's left over after the margins are subtracted. However, the column width percentages are calculated according to the full page width, without taking the margins into account. This discrepancy can lead to problems in narrow windows, when the fixed-width margins crowd out the proportional columns.

For example, imagine you create styles like this:

```
.leftColumn {
  width: 27%;
```

```
  margin: 5px;
  float: left;
}

.rightColumn {
  width: 68%;
  margin: 5px;
  float: left;
}
```

These two columns occupy a combined 95%, leaving an extra 5% for the margin space. This is enough for mid- to large-sized windows, but if you size the window small enough, the leftover 5% can't accommodate the fixed margin space. To see this problem in action, simply give each column a different background color using the background property and then try resizing the window, as shown in Figure 7-1.

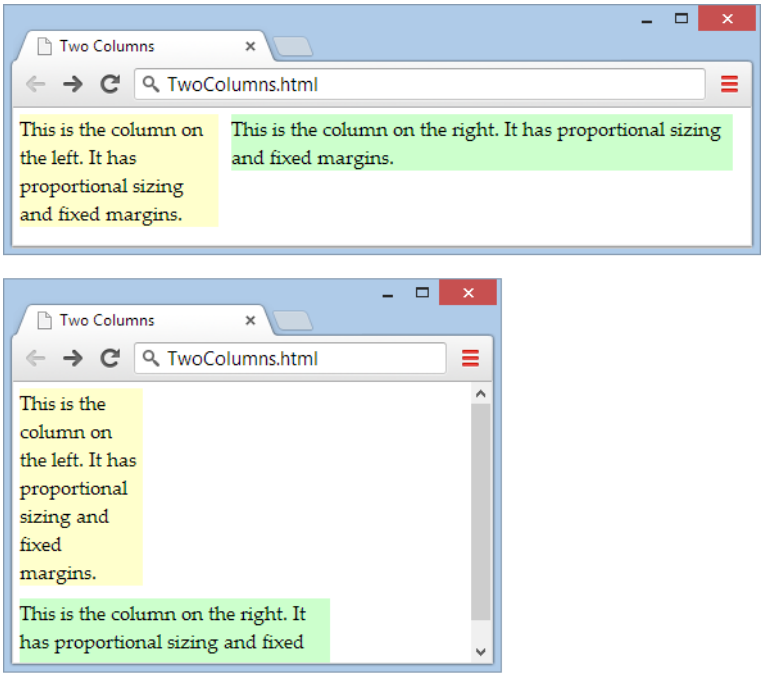


FIGURE 7-1
Proportionally sized columns and fixed margins don't mix. Make the window too small, and the margins crowd out the second column.

To fix this problem, any margins you add between proportional columns must also use proportional sizing. So if the columns leave 5% of the page width unclaimed, you can use that for your margins. Split it into three 1.66% sections—one for the left edge of the window, one for the right edge of the window, and one for the space between the columns, like this:

```
.leftColumn {
  width: 27%;
  margin-left: 1.66%;
  margin-right: 1.66%;
  background: #FFFFCC;
  float: left;
}

.rightColumn {
  width: 68%;
  margin-right: 1.66%;
  background: #CCFFCC;
  float: left;
}
```

Figure 7-2 shows the solution, with both margins and columns sized proportionally using percentages.

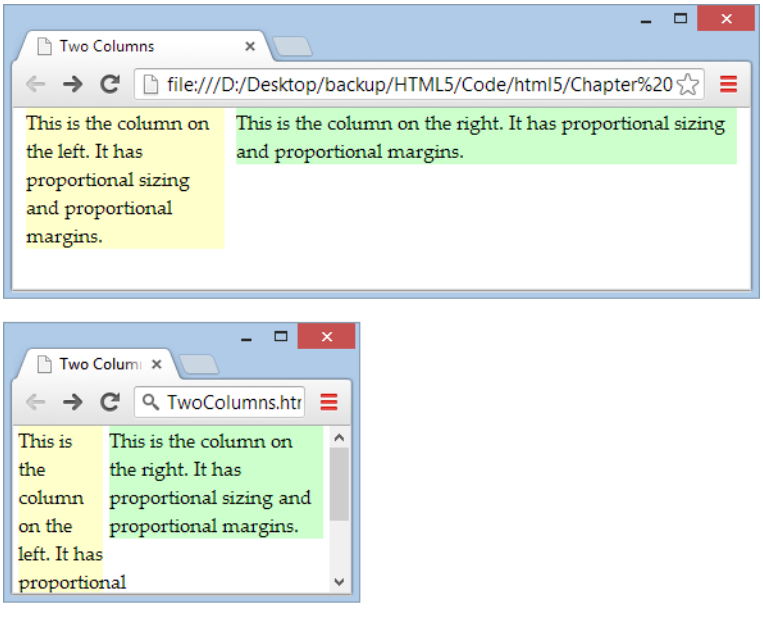


FIGURE 7-2
As long as all the horizontal measurements are done proportionally with percentages, your page's layout will scale to any size.

Depending on the effect you want, you may find that proportional margins don't look quite right. If you don't want your margins to change based on the size of the web browser window, you can use a workaround. For example, you can place another element inside one of your proportional columns and give that element its own fixed margins or padding. Because this element is placed *inside* the top-level layout that you've already established, and because your layout is fully proportional, it will fit snugly into any window size.

Borders present a similar problem. If you add borders to your columns, the extra space they require will break your layout in the same way as the fixed margins shown in Figure 7-1. In this situation, you can't solve the problem with proportional measurements, because borders don't accept percentage widths. Instead, the easiest solution is to use the workaround suggested above: Add a `<div>` element inside your proportional column, and apply the border to the `<div>`. This time-honored technique makes your markup a bit messier (because you need an extra layer of layout), but it ensures that your layout works at any size.

FEATURE FROM THE FUTURE

CSS3 Box-Sizing and calc()

The layout problems you've touched on in this section are common—so common that CSS3 is brimming with potential solutions. Here are two of the most promising (although not perfect, as you'll see).

- **Box-sizing.** Ordinarily, borders are added to the outside of elements, which means you need to subtract the border space from your layout calculations. But CSS3 adds a new `box-sizing` property that, if set to `border-box`, puts the border on the *inside* of your box. The border looks the same, but the size calculation is different. For example, it means that a 67%-wide column stays 67% wide, no matter how thick its border.
- **The `calc()` function.** If you need to combine proportional and fixed measurements, you can ask CSS3 to do the calculations for you—and use the results in your layout—thanks to the nifty `calc()` function. For example, imagine you need to create a column that's 67% wide, less 5px of margin space. Careless web developers might fudge the issue by sizing the column down to 65% (causing the inconsistent spacing issue shown in Figure 7-1). But with CSS3 you can set the `width` property to `calc(67%-5px)`, which makes sure your column mops up exactly all of the available space—and not a pixel more.

Unfortunately, in both cases the cure may not be much better than the disease. The `box-sizing` setting fails on IE 7, and requires the vendor-specific `-moz-` prefix (page 183) on Firefox. The `calc()` function fails on IE 7, IE 8, and the pre-Chrome Android browser, and older versions of Safari require the `-webkit-` prefix. There are polyfills that can smooth out these issues, but for now it's easier to avoid these features altogether until more people inch forward to newer, more modern browsers.

Fluid Images

Achieving a multicolumn, proportional layout is the first step in responsive design. However, there's plenty more to occupy yourself with when you begin to consider the content in those columns.

One issue is images. Ordinarily, image boxes are sized to fit their content—in other words, the exact pixel dimensions of your picture file. However, this arrangement can lead to a problem in small-sized windows. If there's not enough room to accommodate the picture, it will spill out of its column and sprawl over other elements, obscuring them and generally looking sloppy.

The solution to this annoyance is simple. Cap each image at the maximum width of its container, with a style rule that looks like this:

```
img {
  max-width: 100%;
}
```

As always, the 100% is relative to the element's container. In this case, that's the column that contains the image, not the whole page. Now your image can grow until it reaches its full size *or* until it meets the boundaries of its container, whichever comes first (Figure 7-3).

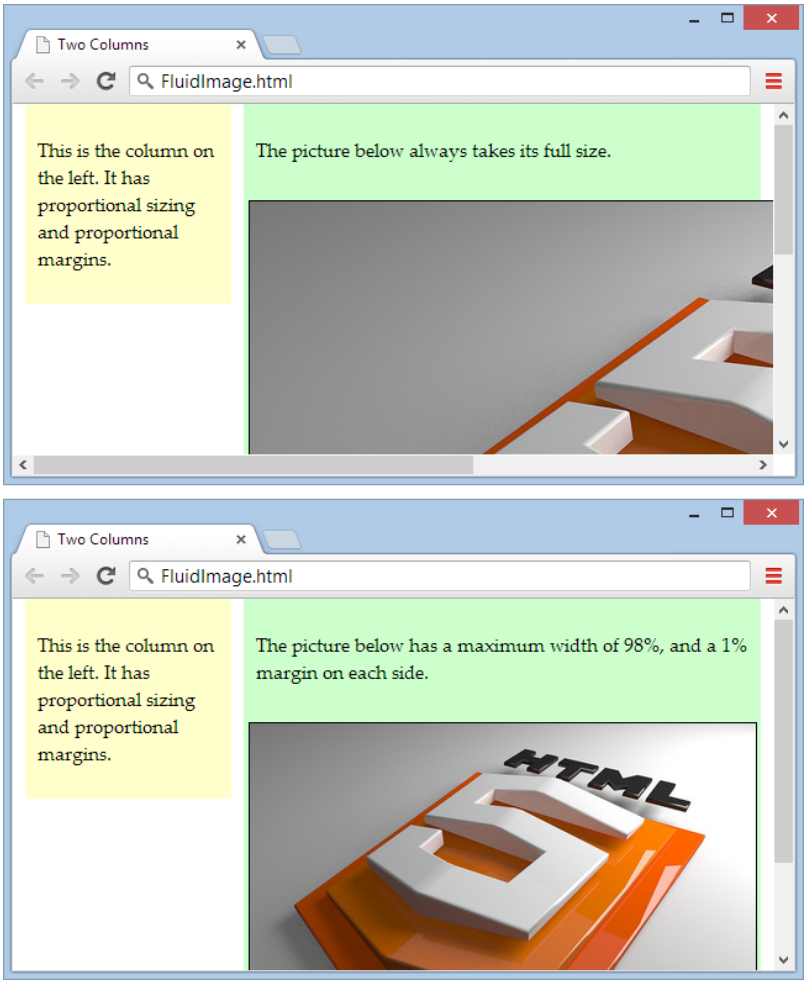


FIGURE 7-3
Without fluid images, any picture can run amok (top). But limit the width to match the container, and images get exactly the right size (bottom).

TIP If you decide to add a margin around your image, make sure the percentages you use for the `margin-left`, `margin-right`, and `max-width` total 100% (and not more).

One limitation of fluid images is that the browser needs to download the full-sized image file no matter what size the image is displayed at. This wastes a small amount of time and bandwidth, which is inconvenient for mobile devices. Sadly, CSS can't correct this problem on its own. But there are other potential solutions that attempt to deal with this issue using some combination of server-side code, web services, and JavaScript libraries. If you're serving up an image-heavy site to large numbers of mobile users, you may want to consider these techniques, which are discussed in a Smashing Magazine article at *http://tinyurl.com/responsive-img* (but most web developers won't bother). Happily, there is a way to solve the analogous but more serious problem of video sizes on mobile devices—see the box on page 244.

Fluid Typography

Now that you have a fluid layout with properly sized pictures, it's time to turn your attention to the text in each column. Casual web developers pick an attractive fixed size for their text (using pixel measurements) and leave it at that. However, such hard-coded sizes break the responsive layout model, because text that's properly sized on a desktop display will be vanishingly tiny on mobile devices. And while website readers can always zoom in to see small text, the goal of responsive design is to make a page that fits any window without requiring excessive zooming and side-to-side scrolling.

Once again, the solution involves avoiding using fixed units of measurement like pixels and points. Instead, you want to set your text sizes relatively, using percentages or em units. The *em* unit, named after the letter M, is the most popular choice.

NOTE The em has a long tradition of representing widths in print typography. For example, the term *em dash* originally referred to a dash that was the width of a capital M in the current typeface.

Percentage sizes and em units have the same result: The size of your text is adjusted relative to the browser's default text size. If you set a text size of 110% or 1.1em, you'll get letters that are 10% bigger than normal, unstyled text. Set a text size of 50% or 0.5em, and you'll get half-sized characters.

Although it doesn't matter whether you use percentages or ems, most responsive web designs follow the same convention. They set the base text size of the page to 100% (just to emphasize that this is the baseline from which all other text is sized), and then tweak that size up or down in other elements with ems:

```
body {
  font-size: 100%
}

p {
  font-size: 0.9em
}
```

```
h1 {
  font-size: 2em
}
```

Experienced web developers don't stop there. Instead, they use ems for all other fixed measurements in their layouts. For example, if you have a border or a bit of margin or padding space deep inside your layout, you're better off to set it with ems than pixels. That way, these sizes are tweaked to match the size of the text. It's a subtle adjustment, but one that creates a more polished appearance.

For example, imagine you've created a two-level layout that places a inside your left column. You use this to get the extra spacing you want around your content, without breaking the proportionally sized column layout:

```
<body>
  <div class="leftColumn">
    <div class="leftColumnContent">
      ...
    </div>
  </div>

  <div class="rightColumn">
    ...
  </div>
</body>
```

You could set the border, margins, and padding of your left column content using pixels. Your layout would still work and it would still be fluid. But it's even better if you use ems, as shown here:

```
.leftColumn {
  width: 28.6%;
  background: #FFFFCC;
  float: left;
}

.rightColumn {
  width: 71.4%;
  background: #CCFFCC;
  float: left;
}

.leftColumnContent {
  border: 0.07em solid gray;
  margin: 0.3em;
  padding: 0.2em 0.3em 0.4em 0.4em;
}
```

NOTE In most layouts, the chief benefit of using ems for borders, margins, and padding is that it prevents these elements from being too large in tiny windows and dominating your layout on mobile devices.

FEATURE FROM THE FUTURE

CSS3: When an Em Becomes a Rem

There's one quirk that faces web designers when using text in complex responsive layouts. Proportionately sized text units, like ems and percentages, size their text with respect to the containing element. That's no problem in a simple example like the one considered on page 229, because the containing element is the <body> element that holds the page, or another element that inherits the font settings of the <body> element. The headaches happen when you apply proportional sizing to *multiple* levels of your layout.

For example, suppose you create a `<div>` and give it a text size of 1.1em. Then, inside that `<div>` you add an `<h1>` heading with a size of 2em. You might expect that the heading is twice the default text size, but it's actually twice the size of its *container*, which is 1.1em. That works out to a heading that's 2.2 times the default text size.

To avoid this compounding effect, you need to be disciplined about where you apply your text sizing. Ideally, you should do it at only one layout level. However, CSS3 has a new unit that neatly solves the problem, called *rems* (which stands for "root em"). Essentially, a rem is a relative measurement just like an em, but with a twist: No matter where you put it, a rem is always calculated relative to the text size of the `<html>` element, not the text size of the containing element. Thus, 2rem is always two times the size of normal text, no matter where you apply it.

Rems have surprisingly good support—they work in every modern browser. The problem is the familiar stragglers, IE 8 and IE 7, which don't understand rems at all. And while it's technically possible to polyfill the gap with JavaScript (see *http://tinyurl.com/rem-polyfill*), most sensible web developers avoid adding yet another script simply to switch their unit system and stick to the slightly inconvenient em units for now.

Of course there's much more to typography than the size of your typeface. To create text that remains readable on a range of displays, you need to think about line-lengths, margins, line height, and even multicolumn text (as demonstrated on page 217). You can't deal with any of these issues using ordinary fluid layouts and proportional sizing. However, you *can* create more flexible style sheets that tweak these other details using media queries, as you'll see shortly. But first, there's one more consideration that you need to unravel: the automatic scaling behavior of mobile phones.

Understanding Viewports: Making Your Layout Work on a Smartphone

In theory, the two-column example you've seen so far can fit into any window size. But in practice there's another complication that comes into play for small mobile devices: the size of the *viewport*.

Apple introduced the viewport concept so its iPhones could do a respectable job displaying the ordinary websites of the time, which didn't use the techniques of responsive web design. Instead of showing just a tiny fragment of a large web page, mobile browsers like Safari show a zoomed-out view that fits in more content. This zoomed-out display area is called the viewport.

The viewport technique is a bit of a tradeoff. It ensures that the page looks more like it would on a desktop browser, but it also makes most ordinary text illegible. It reduces the need to scroll back and forth, but it increases the need to zoom in and out. It makes it easier for viewers to orient themselves in the page, but it prevents them from comfortably reading the content.

NOTE Although Apple introduced the viewport feature, all other mobile developers now follow the same practice. The only difference is how big the viewport is and how much of the web page gets crammed into view at once.

If you're creating a traditional desktop website, you don't have to worry about the devices' viewport settings. They'll ensure that your site looks reasonably good on super-small mobile screens (even though mobile visitors won't find the scaled-down site completely convenient). On the other hand, if you're planning to go all the way with responsive design and create a true, mobile-friendly website, you need to make viewport changes. You need to tell mobile browsers *not* to perform their automatic viewport scaling, which you can easily do by adding the following `<meta>` element to the page's `<head>` section:

```
<meta content="initial-scale=1.0" name="viewport">
```

This line tells mobile devices to use the true scale of your page, with no zooming out. For example, it means a modern iPhone will fit your page into a 320-pixel wide window and display that at full size. Without this scale adjustment, the iPhone will give your page a desktop-sized 980 pixels of width and then shrink that down to fit. Figure 7-4 shows the difference.

NOTE You're probably aware that there are plenty of online simulators that let you see what your website looks like on different mobile devices. For example, on *http://mobiletest.me* you can compare your site's appearance on the latest iPhones, iPads, and Android devices. However, most simulators don't replicate the automatic scaling behavior. In other words, when you preview your site in a simulator, it may look as though you set the initial scale to 1.0 with the `<meta>` element shown above. If you haven't, you won't get an accurate reflection of what you'll see on the device itself, so tread with caution.

Adapting Your Layout with Media Queries

You've now seen how to create a fluid layout that can grow or shrink to fit any browser window. This approach guarantees that your page will fit into any window. However, it doesn't ensure that your page will always look good.

Simple fluid layouts tend to break down at the extremes. In a very tiny window, multiple columns are compacted down to embarrassingly thin dimensions, crowding text and pictures into an unreadable jumble. In a very large window, columns become dauntingly large, and it's hard to follow a line across the vast expanse of the page without losing your place.

FIGURE 7-4

Left: The iPhone's automatic rescaling treats this fluid layout like a desktop-optimized web page. As a result, its text is unreadable without zooming.

Right: Turn off the scaling, and you see your page as it truly is. The next step is to simplify the layout at small sizes using media queries.

One way to deal with these issues is to set limits on how far your layout can expand or contract. You can do that with the `max-width` and `min-width` properties. Expand a page beyond its maximum width, and you'll end up with an extra margin of space on the right. Shrink a page past its minimum width, and the columns will lock into their dimensions, while the browser adds scroll bars to let you move around. Maximum and minimum width settings give you a bit of basic protection against extreme layouts. However, they also reduce the value of your responsive design. For example, if your page can't shrink down to the dimensions of an iPhone window, it's not much use to mobile visitors.

A better solution is to gracefully tweak the *structure* of the layout when your page size changes. For example, a really small window needs a streamlined layout with no sidebars or ad panels. And a really big window presents the opportunity for scaled up text or multiple columns of text (page 217).

Enter *media queries*. This CSS3 feature gives you a simple way to vary styles for different viewing settings. Used carefully, they can help you serve everything from

an ultra-widescreen desktop computer to an iPhone—without altering a single line of HTML.

The Anatomy of a Media Query

Media queries work by latching onto a key detail about the device that's viewing your page (like its size, resolution, color capabilities, and so on). Based on that information, you can apply different styles, or even swap in a completely different style sheet.

At its simplest, a media query is a separate section in your style sheet. That section starts with the word `@media`, followed by a condition in parentheses, and then a series of related styles in curly brackets. Here's the basic structure:

```
@media (media-feature-name: value) {
  /* New styles go here. */
}
```

A media query is similar to a block of conditional JavaScript code. If the current browser meets the condition that you've set out in parentheses, the styles inside come into effect. But if the browser doesn't satisfy the condition, the styles are ignored.

NOTE The styles that lie outside of your `@media` section are always applied, no matter what. The conditional media query styles are applied in *addition* to the other styles. For that reason, the conditional media query styles often have the job of overriding the other style settings—for example, hiding something that was previously visible, moving a section to a new location, applying new text sizes, and so on.

To use a media query, you need to know what sorts of conditions you can construct. The media query standard lets you examine various details, which it calls *media features*. For example, you can find out the width of the display area and then change your styles when it shrinks beyond a certain limit. Table 7-1 lists the most commonly used media features. (There are also several vendor-specific, experimental media features that aren't supported consistently. These aren't included in this table.)

TABLE 7-1 *Most useful media features for building media queries*

FEATURE NAME	VALUE	COMMONLY USED TO...

`width`
`min-width`
`max-width`
	The width of the display area (or rendering surface, on a printer).	Change the layout to accommodate very narrow displays (like a smartphone) or very wide displays.

`height`
`min-height`
`max-height`
	The height of the display area.	Change the layout to accommodate very tall or very short displays.

`device-width`
`min-device-width`
`max-device-width`
	The full width of the screen on the current computer or device (or the full width of a page in a printout).	Adjust the layout to specifically target different devices, like smartphones.

FEATURE NAME	VALUE	COMMONLY USED TO...
device-height min-device-height max-device-height	The full height of the screen or page.	Adjust the layout to specifically target different devices, like smartphones.
orientation	One of two values: landscape or portrait.	Change the layout for different orientations on a table computer.
device-aspect-ratio min-device-aspect-ratio max-device-aspect-ratio	The proportions of the display area, as a ratio. For example, an aspect ratio of 1/1 is completely square.	Adjust styles to fit different window shapes (although this approach quickly gets complicated).
color min-color max-color	The number of color bits. For example, 1-bit color is monochrome, while modern displays typically use 24-bit color, which accommodates millions of colors.	Check for the presence of color (for example, for a printable version of a page), or assess the level of color support.

NOSTALGIA CORNER

CSS Media Types

Interestingly, the creators of CSS took a crack at the multiple-device problem in CSS 2.1, using a feature called *media types*. You may already be using this standard to supply a separate style sheet for printouts:

```
<head>
  ...
  <!-- Use this stylesheet to display the
       page onscreen. -->
  <link  rel="stylesheet" media="screen"
    href="styles.css">

  <!-- Use  this  stylesheet  to  print the
       page. -->
  <link rel="stylesheet" media="print"
    href="print_styles.css">
</head>
```

The media attribute also accepts the value handheld, which is meant for low-bandwidth, small-screen mobile devices. As a result, many modern mobile browsers ignore handheld style sheets anyway, making the media attribute a woefully inadequate tool for dealing with the wide range of web-connected devices that exists today. However, it's still a good way to clean up printouts.

Creating a Simple Media Query

You'll notice that most media query features have several versions, which let you set maximum or minimum limits. These limits are important, because most media queries apply to a range of values.

To use media queries, you must first choose the property you want to examine. For example, if you wanted to create a new set of styles that comes into effect for narrow windows, you'd choose the max-width setting. It's then up to you to choose a suitable

limit. For example, the following media query creates a block of conditional styles that spring into action when the width of the browser window is 480 pixels or less:

```
@media (max-width: 480px) {
  ...
}
```

TIP Right now, the most popular media features are `max-device-width` (for creating mobile versions of your pages), `max-width` (for varying styles based on the current size of the browser window), and `orientation` (for changing your layout based on whether a tablet computer like an iPad is turned horizontally or vertically).

For a simple test, use a media query to make an obvious change. For example, this media query alters the background color of a column:

```
@media (max-width: 480px) {
  .leftColumn {
    background: lime;
  }
}
```

Now you can check whether your media query is working. In your browser, slowly resize the browser window. As soon as the display area of the window shrinks to less than 480 pixels, the new style kicks in and the column changes to a fetching shade of lime green. All the other style properties that you've applied to the `leftColumn` class (for example, its size and positioning) stay in place, because the media query doesn't override them.

NOTE Browsers that don't understand media queries, like Internet Explorer 8, will simply ignore these new styles and keep applying the original styles, no matter how big or small the browser window becomes.

If you want, you can add another media query section that overrides these rules at a still-smaller size. For example, this section will apply new rules when the browser width creeps under 250 pixels:

```
@media (max-width: 250px) {
  ...
}
```

Just remember that these rules are overriding everything that's been applied so far—in other words, the cumulative set of properties that have been set by the normal styles and the media query section for under 450 pixels. If this seems too confusing, don't worry—you'll learn to work around it with more tightly defined media queries on page 239. But first, it's time to consider a more practical example.

Building a Mobile-Friendly Layout

With media queries, you have the essential building blocks you need to create a website that looks just as respectable in a smartphone browser as a desktop browser. All you need to do is apply them.

Figure 7-5 shows a revamped example of the *ApocalypseSite.html* page you first saw in Chapter 2 (page 54). The original page used a fixed layout with hard-coded column widths. The revised version uses all of the techniques explored in this chapter. It has a fluid layout with proportional sizing (page 222) that fits any window width. It uses em units for margins, padding, border widths, and text sizes, ensuring that these details are adjusted in harmony on different devices (page 228). The site header image grows or shrinks to fit the available space, and the ad image in the sidebar uses the fluid image technique to make sure it never oversteps its bounds (page 226). It also uses the `<meta>` element fix to prevent mobile browsers from zooming out (page 231).

FIGURE 7-5

Behind the scenes, this page uses the best practices of responsive web design. You can peruse the complete CSS on the try-out site at http://prosetech.com/html5.

In short, the new *ApocalypseSite.html* page is mobile-ready. However, its layout still isn't mobile-*friendly*. That's because no matter how small the two side-by-side

columns compress themselves, they won't fit cleanly in a tiny window. To correct this oversight, you need to use a media query.

Before you crack open your style sheet, you need to consider what the mobile version of your site should look like. Usually, mobile sites slim themselves down to a single column. Sidebars are either hidden completely or inserted above or below the main content. Figure 7-6 shows a cleaned-up version of the *ApocalypseSite.html* on an iPhone.

FIGURE 7-6

Here are two snapshots of the mobile version of the ApocalypseSite.html page. The page starts with a small site header, followed by the article content (left). The article links and ad block from the sidebar are placed after the article footer (right).

It's surprisingly easy to create the mobile version of the *ApocalypseSite.html* page. The shrunken site header and nicely sized text happen automatically, thanks to the page's use of fluid images and em units. The only task left for your media query is to rearrange the columns.

Initially, the two columns are defined with these two style rules:

```
.NavSidebar {
  float: left;
  width: 22%;
  font-size: small;
}
```

```
.Content {
  float: left;
  width: 78%;
}
```

The sidebar is floated on the left with a width of 22%. The content is floated next to it and given a width of 78%.

Because the layout stops working at small widths, it makes sense to use the popular `max-width` media feature. As you learned in the previous section, `max-width` gets the current size of the page in the browser window. If this value is small, two columns aren't appropriate.

Here's the media query that removes the floats and resizes the section so the columns take the full available width:

```
@media (max-width: 568px) {
  .NavSidebar {
    float: none;
    width: auto;
  }

  .Content {
    float: none;
    width: auto;
  }
}
```

These styles are applied in addition to the normal styles you've already defined. Thus, you may need to reset properties you've already changed to their default values. In this example, the media query styles reset the `float` property to `none`, and the `width` property to `auto` (although it would work equally well if you set the width to 100%). These are the default values, but the original sidebar style changed them. You'll also notice that the original `NavSidebar` style set the font size. The media query doesn't override this detail, so it stays in place.

Technically, this media query creates styles that apply to any narrow window, regardless of whether it's in a mobile browser or in a micro-sized window on a desktop browser. This makes perfect sense, but you can get more particular and create one set of styles for tiny desktop windows and another for mobile devices. To target tiny desktop windows, you'd use `max-width`, and to spot mobile devices, you'd use `max-device-width`, as detailed in Table 7-1.

TIP It's up to you when you want to switch to your simplified layout, but the 568-pixel mark is a good choice. That's because 568 pixels is the width of your page in an iPhone when it's turned sideways, in landscape orientation. (It works for Android devices too, as explained on page 243.)

This example needs one more adjustment. In the original version of the page, the `NavSidebar` section is defined before the `Content` section in the HTML markup. This lets you float them both on the left side, and makes sure the `NavSidebar` is on the left. Unfortunately, when you remove the floating behavior, the mobile version of the site is forced to show the sections in the order they appear, which means the `NavSidebar` appears at the top of the page, followed by the `Content` section underneath. This layout is a bit off-putting to mobile viewers, since they'd rather not scroll through a set of links and advertisements before they get to the meat of the page.

When faced with a challenge like this, the solution is to start by arranging your markup to suit the mobile version of the site, and *then* layer on extra CSS rules to create the more sophisticated multicolumn layout. This gold-standard technique is called *mobile-first* development.

In this example, that means putting the `Content` section before the `NavSidebar` section. This solves the problem in the mobile version of the page, but it also forces the sidebar to the right column on the full-sized version of the page. To correct this quirk, simply tweak the `Content` section so it floats to the right:

```
.Content {
  float: right;
  width: 78%;
}
```

Now the `Content` section returns to its place on the right, while the `NavSidebar` clings to the left, restoring the full-size layout shown in Figure 7-5.

At this point, you may want to consider adding another media query to also change your styles for very wide windows. For example, you could break your text into multiple columns (using the CSS properties described on page 217) to make sure your text remains readable.

TIP Looking for some examples to inspire you? Try out a ready-made responsive template. There are plenty of examples on the Web. To get started, start browsing *http://html5up.net*, *www.typeandgrids.com*, or *http://responsify.it*.

More Advanced Media Query Conditions

Sometimes you might want to make your styles even more specific, so they depend on multiple conditions. Here's an example:

```
@media (min-width: 400px) and (max-width: 700px) {
  /* These styles apply to windows from 400 to 700 pixels wide. */
}
```

POWER USERS' CLINIC

Hiding and Replacing Sections

If you're ambitious, there are many more changes you can make to differentiate the mobile version of your site from its full-size incarnation. For example, you can use the CSS `display` property to hide and show entire *sections* of your page.

Before you use this approach, consider its drawbacks. If you switch large sections of your page, you'll be left with messy markup, which you'll need to maintain, keep consistent, and test on different devices. Also, if your hidden sections contain images, browsers will still download them, even if they're never shown. On a mobile device, this can become a performance drag and a waste of bandwidth.

However, there *is* an appropriate time to use the section-switching technique: when you need to replace a complex navigation aid or menu with a slimmer, simpler mobile version. For example, it's common practice to give mobile users a drop-down list for navigation instead of an unwieldy tree. (There's even a clever style technique that can convert a row of links into a drop-down list, as detailed at *http://css-tricks.com/convert-menu-to-dropdown*).

Sometimes, simple tricks and small alterations aren't enough. You may want to revamp the mobile version of your site more radically. Here, you have a range of options, ranging in complexity and sophistication. At one extreme, you could create a completely separate mobile site and host it on a different web domain (as the New York Times does with its mobile site at *http://mobile.nytimes.com*). This option is a lot more work, and without some sort of content management system running on your web server, you'll never keep your mobile site in sync with your standard site. Another way is to write web server code that checks every request, figures out what web browser is on the other end, and sends the appropriate type of content. This sort of solution is great, if you have the time and skills.

A more modest approach is to use a JavaScript tool that lets you alter your pages dynamically based on the viewer. One example is Modernizr, which provides a method named `Modernizr.mq()` for testing media queries in your code (read about it at *http://modernizr.com/docs*). This approach is more powerful than media queries, but it also introduces more complexity into your page design.

This type of media query comes in handy if you want to apply several sets of mutually exclusive styles, but you don't want the headaches of several layers of overlapping rules. Here's an example:

```
/* Normal styles here */

@media (min-width: 600px) and (max-width: 700px) {
  /* Override the styles for 600-700 pixel windows. */
}

@media (min-width: 400px) and (max-width: 599.99px) {
  /* Override the styles for 400-600 pixel windows. */
}

@media (max-width: 399.99px) {
  /* Override the styles for sub-400 pixel windows. */
}
```

In this case, if the browser window is 380 pixels, exactly two sets of styles will apply: the standard styles and the styles in the final `@media` block. Whether this approach simplifies your life or complicates it depends on what you're trying to accomplish. If you're using complex styles and changing them a lot, the no-overlap approach shown here is often the simplest way to go.

Notice that you have to take care that your rules don't unexpectedly overlap. For example, if you set the maximum width of one rule to 400 pixels and the minimum width of another rule to 400 pixels, you'll have one spot where both style settings suddenly combine. The slightly awkward solution is to use fractional values, like the 399.99 pixel measurement used in this example.

Another option is to use the `not` keyword. There's no functional difference, but if the following style sheet makes more sense to you, feel free to use this approach:

```
/* Normal styles here */

@media (not max-width: 600px) and (max-width: 700px) {
  /* Override the styles for 600-700 pixel windows. */
}

@media (not max-width: 400px) and (max-width: 600px) {
  /* Override the styles for 400-600 pixel windows. */
}

@media (max-width: 400px) {
  /* Override the styles for sub-400 pixel windows. */
}
```

In these examples, there's still one level of style overriding to think about. That's because every `@media` section starts off with the standard, no-media-query style rules. Depending on the situation, you may prefer to separate your style logic completely (for example, so a mobile device gets its own, completely independent set of styles). To do so, you need to use media queries with external style sheets, as described next.

Replacing an Entire Style Sheet

If you have simple tweaks to make, the `@media` block is handy, because it lets you keep all your styles together in one file. But if the changes are more significant, you may decide that it's easier to create a whole separate style sheet. You can then use a media query to create a link to that style sheet:

```
<head>
  <link rel="stylesheet" href="standard.css">
  <link rel="stylesheet" media="(max-width: 568px)" href="small_styles.css">
  ...
</head>
```

The browser will download the second style sheet (*small_styles.css*) with the page but won't apply it unless the browser width falls under the maximum.

As in the previous example, the new styles will override the styles you already have in place. In some cases, you want completely separate, independent style sheets. If so, you first need to add a media query to your standard style sheet to make sure it kicks in only for large sizes:

```
<link rel="stylesheet" media="(min-width: 568.01px)" href="standard.css">
<link rel="stylesheet" media="(max-width: 568px)" href="small_styles.css">
```

The problem with this approach is that browsers that don't understand media queries will ignore *both* style sheets. You can fix this up for old versions of Internet Explorer by adding your main style sheet again, but with conditional comments:

```
<link rel="stylesheet" media="(min-width: 568.01px)" href="standard.css">
<link rel="stylesheet" media="(max-width: 568px)" href="small_styles.css">
<!--[if lt IE 9]>
  <link rel="stylesheet" href="standard.css">
<![endif]-->
```

This example still has one small blind spot. Old versions of Firefox (earlier than 3.5) don't understand media queries and don't use the conditionally commented IE section. You could solve the problem by detecting the browser in your code and then using JavaScript to swap in a new page, but it's messy. Fortunately, old versions of Firefox are becoming increasingly rare.

Incidentally, you can combine media queries with the media types described in the box on page 234. When you do so, always start with the media type, and don't put it in parentheses. For example, here's how you could create a print-only style sheet for a specific page width:

```
<link rel="stylesheet" media="print and (min-width: 25cm)"
 href="NormalPrintStyles.css" >
<link rel="stylesheet" media="print and (not min-width: 25cm)"
 href="NarrowPrintStyles.css" >
```

Recognizing Specific Mobile Devices

As you've already learned, you can distinguish between normal computers and mobile devices by writing a media query that uses `max-device-width`. But what widths should you use?

If you're looking for mobile phones, check for a `max-device-width` of 568 pixels. This is a good rule of thumb, since it catches current iPhone and Android phone models, whether they're in portrait or landscape orientation:

```
<link rel="stylesheet" media="(max-device-width: 568px)"
  href="mobile_styles.css">
```

If you're a hardware geek, this rule may have raised a red flag. After all, modern mobile devices use tiny, super-high-resolution screens. For example, the iPhone 5 crams a grid of 640 x 1136 pixels into view at once. You might think you'd need larger device widths to recognize these devices. Surprisingly, though, that isn't the case.

For example, consider the iPhone 5. It claims that it has a pixel width of 320 pixels (in portrait orientation), even though it actually has twice as many physical pixels. It uses this quirk to prevent websites from concluding that 640-pixel wide iPhone displays should receive the full desktop version of a website. Although the iPhone can certainly display such a site, its tiny pixels would make it all but impossible to read.

Most modern, high-resolution devices behave this way. They add in a fudge factor called the *pixel ratio*. In the iPhone (version 4 and later), every CSS pixel is two physical pixels wide, so the pixel ratio is 2. In fact, you can create a media query that matches the iPhone 4 but ignores older iPhones, using the following media query:

```
<link rel="stylesheet"
 media="(max-device-width: 480px) and (-webkit-min-device-pixel-ratio: 2)"
 href="iphone4.css">
```

Table 7-2 lists the device widths of some popular devices. Keep in mind that there's often a bit of pixel ratio fudgery at work. For example, all versions of the iPad report a device width of 768, even though the number of physical pixels doubled in the iPad 3.

TABLE 7-2 *Common device widths*

DEVICE	DEVICE WIDTH (IN PORTRAIT MODE)	DEVICE WIDTH (IN LANDSCAPE MODE)
Apple iPhone 4	320	480
Apple iPhone 5	320	568
Apple iPad	768	1024
Samsung Galaxy S4	360	640
Google Nexus 4	384	640
Kindle Fire	600	1024

TIP New devices are released all the time. For current information, consult a site like *www.mobitest.me/devices*.

Tablets like the iPad pose a special challenge: Users can turn them to show content vertically or horizontally. And although this changes the `max-width`, it doesn't alter the `max-device-width`. In both portrait and landscape orientation, the iPad reports a device width of 768 pixels. Fortunately, you can combine the `max-device-width` property with the `orientation` property if you want to vary styles based on the iPad's orientation:

```
<link rel="stylesheet"
 media="(max-device-width: 768px) and (orientation: portrait)"
 href="iPad_portrait.css">

<link rel="stylesheet"
 media="(max-device-width: 768px) and (orientation: landscape)"
 href="iPad_landscape.css">
```

Of course, this rule isn't limited to iPads. Other devices that have similar screen sizes (in this case, 768 pixels or less) will get the same style rules.

NOTE On their own, media queries probably aren't enough to turn a normal website into a mobile-friendly one. You'll also need to think about the user experience. You may need to break content down into smaller pieces (so less scrolling is required) and avoid effects and interactions that are difficult to navigate with a touch interface (like pop-up menus).

GEM IN THE ROUGH

Media Queries for Video

One obvious difference between desktop websites and mobile websites is the way they use video. A mobile website may still include video, but it will typically use a smaller video window and a smaller media file. The reasons are obvious—not only do mobile browsers have slower, more expensive network connections to download video, but they also have less powerful hardware to play it back.

Using the media query techniques you've just learned, you can easily change the size of a `<video>` element to suit a mobile user. However, it's not as easy to take care of the crucial second step and link to a slimmed-down video file.

HTML5 has a solution: It adds a `media` attribute directly to the `<source>` element. As you learned in Chapter 5, the `<source>` element specifies the media file a `<video>` element should play. By adding the `media` attribute, you can limit certain media files to certain device types.

Here's an example that hands the *butterfly_mobile.mp4* file out to small-screened devices. Other devices get *butterfly.mp4* or *butterfly.ogv*, depending on which video format they support.

```
<video controls width="400" height="300">
  <source src="butterfly_mobile.mp4"
   type="video/mp4"
   media="(max-device-width: 480px)">
  <source src="butterfly.mp4"
  type="video/mp4">
  <source src="butterfly.ogv"
  type="video/ogg">
</video>
```

It's still up to you to encode a separate copy of your video for mobile users. Encoding tools usually have device-specific profiles that can help you out. For example, they might have an option for encoding "iPad video." It's also still up to you to make sure that you use the right media format for your device (usually, that will be H.264) and supply video formats for every other browser.