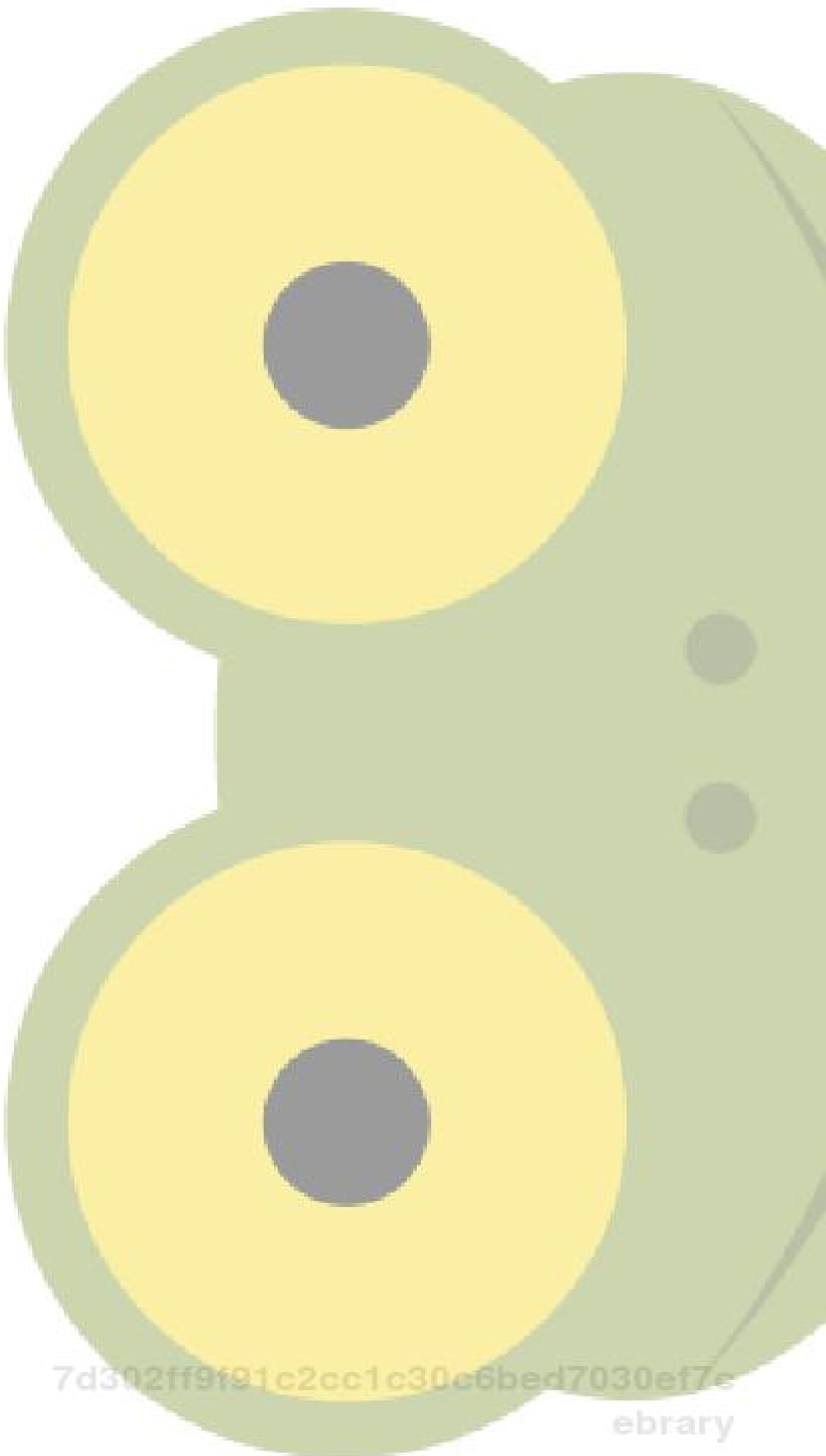
part 3

Enhancing Web Pages with HTML5 and JavaScript



chapter ten

Introducing JavaScript

UP TO THIS point in the book, you focused on how to create web pages using HTML. In the remaining chapters, you are going to be using JavaScript and HTML5 to create interactive features for your Joe's Pizza Co. website. But first, you need to get up to speed about the JavaScript programming language.

The JavaScript programming language is a very large topic, possibly even bigger than HTML and CSS combined. In this chapter I introduce you to the basic concepts of JavaScript so that you can build a foundation on which to enhance your JavaScript knowledge in the future. You will step away from the Joe's Pizza Co. website for a while, and focus instead on building a number of small JavaScript programs. As you progress through each of these exercises you will learn about the key concepts of programming. In the following chapters, you'll return to the Joe's Pizza Co. website and put your new knowledge to use.

You'll start by learning how to add JavaScript to your web pages, by either putting the code inline with your existing HTML or in separate JavaScript files. You will learn how to write your own programs in JavaScript, how JavaScript programs are constructed, and the various tools available to you as a developer. These include control and loop structures as well as how to select and manipulate elements within your pages.

You'll also look at the Document Object Model (DOM), what it is, how it is created, and why it is important. You will also learn about DOM nodes and the relationships between them.

Finally, you will look at a JavaScript library called jQuery and how it can be used to make writing JavaScript programs easier.

Treehouse offers a great JavaScript Foundations course that I recommend you work through in conjunction with this chapter. You can find it at `http://teamtreehouse.com/library/websites/javascript-foundations`.

What is JavaScript?

JavaScript is a general purpose programming language that is used to create dynamic and interactive websites. Brendan Eich originally designed the language while working on the Netscape browser in 1995; he's now the Chief Technology Officer at Mozilla (the non-profit organization that develops, among other things, the Firefox web browser).

So what are *dynamic websites*? JavaScript enables developers to write programs that can change how a website looks and behaves once it has loaded into the user's browser. This means that you could update some of the content in the page without having to reload the page. You can even request new content from a web server and display it on the page without having to refresh it. This is referred to as *Asynchronous JavaScript and XML*, or *AJAX*. Many popular web applications such as Gmail and Facebook make heavy use of AJAX to create fast interfaces.

JavaScript is also very useful for listening for events such as mouse clicks and key presses on the keyboard. Using what are known as *event listeners*, a JavaScript developer can specify some code that should be executed when an event occurs. Remember when you looked at the `<button>` element in Chapter 5 and I said that to make "dumb" buttons smart, you had to use JavaScript? Well, this is how it is done.

An advantage that JavaScript has over other languages is that it runs directly in the user's web browser. This means that the code will often execute very fast. It also helps to lighten the load on the web server that already has enough to deal with, responding to page requests from other users.

Here are some demos of what can now be done using JavaScript and HTML5. Take a look at them—and enjoy:

These demos take advantage of some really advanced HTML5 technologies that are not yet supported in all browsers. I recommend that you view these demos in Google Chrome to ensure that they will work correctly.

- **The Wilderness Downtown by Google**—`http://www.thewildernessdowntown.com/`
- **Water/Ocean by OutsideOfSociety**—`http://oos.moxiecode.com/js_webgl/water_noise/`
- **Rome by Google**—`http://www.ro.me/`

JavaScript Terminology

Throughout this chapter you may encounter some terminology that is new to you. So that you understand what is going on, here are some key terms.

- **variable**—Variables allow you to store pieces of data so that you can use them in your programs. You will look at variables in more detail later in this chapter. The following example creates a variable called `age` and stores in it the integer value `20`.

  ```
  var age = 20;
  ```

- **function**—A function consists of a block of JavaScript code that is executed when the function is *called*. You call a function by adding the function name to your code at the position which you want the code within the function block to be executed. The following example shows a simple function called `printName`.

  ```
  function printName() {
     document.write("Matt West");
    }
  ```

  You will learn about functions in more detail later in this chapter.
- **object**—A JavaScript *object* consists of a collection of properties and functions. Each of these properties stores some data about the object. A `person` object, for example, might have properties such as `name`, `height`, `age`, and `weight`. This concept is similar to the concept of *microdata items* described in Chapter 8.

  JavaScript objects can also have functions attached to them. Every web browser has a `document` object that has a number of properties containing information about the page. This `document` object also has a number of functions attached to it. These functions are called on an object using the `object.function()` syntax. The benefit of attaching functions to objects is that the function will be able to access all of the object's properties, whereas if the function were not attached to an object the desired properties would have to be passed to the function through parameters.
- **event**—An event is triggered when certain conditions are met. For example, the `onload` event is triggered when the page loads. There are also events for things like mouse clicks, key presses, and form submissions. You can use *event listeners* to attach functions to events so that a function is executed when an event is triggered.

Awesome, aren't they! If you aren't brimming with excitement at what is possible now, don't worry. You soon will be.

So by now you should have a general understanding of the purpose of JavaScript and its position within the web ecosystem. Now it's time to start learning how to use JavaScript so that one day you will be able to create websites as amazing as those demos.

The <script> Element

Before you can start writing JavaScript code, you need to know about a few housekeeping issues. The first of these is where to put your JavaScript code. You have two options: You can either put it inline (embedding it within an HTML file), or you can put it in a completely separate file and link to that file in your HTML. The latter method is generally considered the better approach, and this is how you will be adding JavaScript to the Joe's Pizza Co. website. However, there are times when inline scripts are needed, so in this section I cover both methods. Whichever method you end up using in your projects, you are going to use the `<script>` element to let the browser know where your JavaScript code is located.

Inline Scripts

JavaScript code that is embedded directly into your HTML page is referred to as *inline* scripts. Inline scripts are best if you have only a short piece of JavaScript; otherwise, it will be easier to maintain your code over the long run if you put it in its own dedicated file.

To embed JavaScript code in your HTML files, you need to place it within a `<script>` element. Follow these steps:

The example code used in this chapter is available from the Chapter 10 folder in the download code files.

1. Create a new folder on your desktop called `javascript-examples`.
2. Now create a new file in your text editor.
3. Save this file in the `javascript-examples` folder as `example10-1.html`.
4. Copy the following code into the `example10-1.html` file.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Inline Scripts</title>
</head>
<body>
```

```
    <script>
      window.onload = function() {
        document.write('Hello World!');
      };
    </script>
  </body>
  </html>
```

5. Save the file.

Now view this file in your web browser. You should see that `Hello World!` is displayed on the page. Let's dissect this code a little.

JavaScript code will be executed only when you tell the browser to execute it. In this case, you want the code to run as soon as the page has loaded, so you need to tell that to the browser. The first line of your code does this.

JavaScript uses objects to keep track of things like the browser window, document, and HTML elements. When a page finishes loading, the `window` object triggers an event called `onload`. To execute code when the page loads, you need to attach this code to the `onload` event. In the code in the preceding Step 4, you do this by assigning a new function to the `window.onload` event using the = (equals sign) operator. All the code that you place within this function will now be executed when the `onload` event triggers:

```
window.onload = function() {
  document.write('Hello World!');
}
```

To ensure that things are working correctly, you included a line of code that will write the text `Hello World!` onto the screen. To do this, you grabbed the `document` object and used its built-in `write` function. The text that you want to be displayed is placed within the parentheses of the `write` function. You learn more about JavaScript functions later in this chapter.

That's it for inline scripts—pretty straightforward. As you can see, if you had a lot of HTML and JavaScript code in one page, it could be a bit of a pain to maintain, not to mention the fact that if you want to use this code on more than one page you would have to copy and paste it into each page. Then if you wanted to make a change, you would have to make the update in numerous different files. Things can get messy very quickly. This is why it is often better to put your JavaScript code into an external file.

Linking External JavaScript Files

Dedicated JavaScript files should use the `.js` file extension. Make sure that your text editor has syntax highlighting for JavaScript, because this can help to catch any errors (or *bugs*) in your code.

As with external stylesheets, you need to link JavaScript files to your HTML files. You do this using the `<script>` element.

You can use the `src` attribute on the `<script>` element to specify an absolute or relative path to your JavaScript file, in the same way that you do for images.

Unlike the `<img>` element, `<script>` is not a void element and therefore you always have to specify both start and end tags.

The following steps show how to modify `example10-1` to use external JavaScript files instead of inline scripts:

1. Create a new file in your text editor.
2. Save this file in your `javascript-examples` folder as `example10-2.js`.
3. Move the code from your `<script>` element into the new JavaScript file. Don't include the `<script>` element itself.
4. Save the `example10-2.js` file.
5. Now add a `src` attribute to the `<script>` element and set its value to `example10-2.js`.

```
<script src="example10-2.js"></script>
```

6. Save this file as `example10-2.html`.

Load the HTML file in your browser; you should see that the `Hello World!` text is still displayed onscreen. Try changing this text in the JavaScript file to make sure that everything is working properly.

To ensure that your web pages load fast, it's best to always put your `<script>` elements at the end of your HTML file, just before the `</body>` tag.

Now that you know how to get your JavaScript code to run in your web pages, you are going to learn the basics of JavaScript programming.

JavaScript Basics

This section covers the basics of the JavaScript programming language. I explain how to create programs using variables, functions, event listeners, control structures, and loops. These are key programing concepts that will give you the knowledge you need to start using JavaScript with HTML5. As I mentioned before, this is not a comprehensive examination of

JavaScript, but it will give you the skills you need to complete the rest of the exercises in this book and build small JavaScript programs of your own.

A Simple Program

You've already encountered the standard Hello World! program, so let's skip ahead to something a little more interactive. The following program asks users for their names and then displays the input on the screen:

1. Create a new file in your text editor called `example10-3.js`.
2. Copy the following code into this new file.

```
window.onload = function() {
  var name = prompt('Your name please');

  document.write('Hello ' + name + '!');
};
```

3. Save the `example10-3.js` file.
4. Create a new `example10-3.html` file in your text editor.
5. Copy the following code into this HTML file.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Say Hello</title>
</head>
<body>
  <script src="example10-3.js"></script>
</body>
</html>
```

6. Save the `example10-3.html` file.

Now view the `example10-3.html` file in your browser. You should be prompted for your name. Type this into the text box and click OK (or press Enter). A hello message should display on the screen.

This small program addresses a number of different concepts from JavaScript. First, you declared a variable (name) to hold some data (more on variables soon), and then you told the browser to prompt the user for her name, passing in some text to be displayed in the pop-up window. You then combined the name that you collected with some other text and printed it all out to the screen.

This may seem like a fairly useless little program, but here you have covered the key programming concept of input and output. Next up: taking a closer look at variables.

Variables

Variables are used to store pieces of data so that you can play around with them in your program. Declaring a variable in JavaScript is easy; unlike some other programming languages, you do not need to specify what type of data the variable will hold (for example, a number or text). Programming languages like JavaScript are referred to as *dynamically* typed languages.

To declare a variable, you start by using the JavaScript keyword `var`. This tells the browser that you are creating a new variable. You then add the name of your variable. Variable names should not contain spaces, and you should use *camel-case* to join multiple words together. For example, "guitar stand" becomes `guitarStand`. Note the capital letter of the second word: That's where camel-case gets its name—like the hump on a camel's back.

Once you have declared the name of your variable, you can assign it an initial value. This is known as *initializing* a variable. To do this, you use the equal sign (=) followed by your data. The statement below shows a variable `leafColor` that is initialized with the value `green`.

```
var leafColor = "green";
```

In JavaScript, you add a semicolon to the end of each line to indicate that the statement is finished. A *statement* is a command, such as calling a function or creating a variable.

You will be using variables extensively in the JavaScript programs you build throughout the remainder of this book. For example, in Chapter 11 you will be using variables to store references to buttons on your web page so that you can build custom playback controls for a video.

Reserved Words

There are some words that are *reserved* for the JavaScript language; you should not use them as variable names. If you try to do so, your program likely will not execute correctly.

Here is a list of all the reserved words for JavaScript. Have a quick read so that you are familiar with them.

```
break, case, catch, continue, debugger, default, delete, do, else,
finally, for, function, if, in, instanceof, new, return, switch,
this, throw, try, typeof, var, void, while, with
```

Null and Undefined

When you declare a variable, you do not have to initialize it with a value. You can simply declare a variable, as shown here:

```
var age;
```

Variables that are declared but not initialized are known as *undefined*. Because no value is associated with them, the browser cannot give them a type for you (for example, string or number).

Variables can also be `null`. This is similar to being undefined, but these do have a type. A null variable simply has no value. It is empty, but it does have an implied type that was determined by its value in the past.

Let's write a little program to test the difference between `undefined` variables and `null` variables. Here are the steps:

1. Create a new file in your text editor.
2. Save this file as `example10-4.js`.
3. Copy the following code into this file:

```
window.onload = function() {
  // Undefined variable
  var foo;
  document.write('The foo variable is ' + foo + '<br>');

  // Null variable
  var bar = "Hello";
  bar = null;
  document.write('The bar variable is ' + bar);
};
```

4. Save the `example10-4.js` file.
5. Create a new HTML file called `example10-4.html`.
6. Add the following code to this new HTML file:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Undefined vs Null</title>
</head>
<body>
  <script src="example10-4.js"></script>
</body>
</html>
```

7. Save the `example10-4.html` file.

Now open this file in your browser. You should see the following output:

```
The foo variable is undefined
The bar variable is null
```

Because the `foo` variable was declared but not initialized, it never actually existed. Therefore, it is `undefined`. The bar variable, on the other hand, was initialized with the text `Hello`. However, when you set the `bar` variable to be empty using the `null` keyword, it lost its value but still continued to exist. Therefore, when it is written to the screen, the browser sees that it exists but has no value for it, and so displays `null`.

Many developers struggle to grasp the difference between `undefined` and `null`. Take a look at Jim's video on the Treehouse website, where he explains this in a little more detail: `http://teamtreehouse.com/library/websites/javascript-foundations/variables/null-and-undefined`.

Functions

Functions (sometimes called *methods*) are code structures that you can use to store code that might be needed several times in your script. Instead of having to write this code out multiple times, you can create a function and then call that function wherever you want the code to run. This makes your code much more maintainable. In Chapter 13 you will be creating a function that uses the GeoLocation API to find the Joe's Pizza Co. restaurant that is nearest to the user.

You can pass inputs to a function using parameters. These are placed within parentheses after the function name.

Take a look at a simple function that will take in a name and write a hello message to the screen.

```
function sayHello(name) {
  document.write("Hello " + name);
}
```

Here you have defined a parameter called name. When the function is called, the browser will create a variable name and initialize it with the content that you pass to the function, as shown here:

```
sayHello("Joe");
```

If all goes to plan, this will output the following on your screen:

```
Hello Joe
```

Let's write a little program that makes use of the `sayHello()` function. Here you are going to write out a hello message to four different people.

1. Create a new file in your text editor called `example10-5.js`.
2. Copy the following code into this new file:

   ```
   window.onload = function() {
     sayHello('Joe');
     sayHello('Beth');
     sayHello('Steve');
     sayHello('James');
   };

   // Print out a hello message.
   function sayHello(name) {
     document.write('Hello ' + name + '<br>');
   }
   ```

3. Save the `example10-5.js` file.

4. Create a new `example10-5.html` file.
5. Copy the following HTML into this file:

   ```
   <!DOCTYPE html>
   <html>
   <head>
     <meta charset="utf-8">
     <title>Say Hello</title>
   </head>
   <body>
     <script src="example10-5.js"></script>
   </body>
   </html>
   ```

6. Save the `example10-5.html` file.

Open the `example10-5.html` file in your browser. You should see the following output on the screen:

```
Hello Joe
Hello Beth
Hello Steve
Hello James
```

In this example, you used the `sayHello()` function that you created to output a hello message to Joe and his friends.

Functions are extremely useful for creating maintainable code. Try to create functions whenever you can; they will make your life much easier as your programs grow.

Event Listeners

You have already encountered event listeners multiple times in this chapter. Event listeners are used to attach functions to a particular event, such as a page load, mouse click, or key press.

There are two ways of creating event listeners. The first is to attach a function to an event that occurs on a JavaScript object, using the following syntax. This should look somewhat familiar to you.

```
window.onload = function() {
  // Do something.
}
```

Here an empty function block is attached to the `onload` event of the `window` object. This `onload` event is called when the page loads. You don't necessarily have to use an empty function block here. If you have defined a function in your JavaScript code you could use that too.

```
window.onload = sayHello("Joe");
```

The second method for creating an event listener is to use the `addEventListener()` function. This function has two parameters. The first is the event that should trigger the event listener and the second is a function that should be executed when the event is triggered. The `addEventListener()` function should be called on an object as in the following example.

```
document.getElementById("btn").addEventListener("click",
function(event){
  alert("Boo!");
)};
```

In this example an alert dialog would be displayed to the user when they click the button with the ID `btn`.

Note the `event` parameter that is passed into the function block in the previous example. When the event is triggered, details about the event will be passed to the function block through this parameter. You don't have to define a parameter for the event data; it is optional.

You will be using event listeners many times in the remaining chapters of this book, especially in Chapters 11 and 12 when you will use them to listen for button clicks and form submissions.

Let's write a little program that uses what you have learned here.

1. Create a new file in your text editor.
2. Save this file as `example10-6.js`.
3. Add the following JavaScript code to this file.

   ```
   window.onload = function() {
     var button = document.getElementById("btn");

     button.addEventListener("click", function() {
       alert("Boo!");
     });
   }
   ```

4. Save the `example10-6.js` file.
5. Now create a new HTML file called `example10-6.html`.

6. Copy the following HTML code into this file.

   ```
   <!DOCTYPE html>
   <html>
   <head>
     <meta charset="utf-8">
     <title>Event Listeners</title>
   </head>
   <body>
     <button id="btn">Click me!</button>
     <script src="example10-6.js"></script>
   </body>
   </html>
   ```

7. Save this file.

Now open up the `example10-6.html` file in your web browser. If you click the button you should be confronted by an alert dialog that contains the text `Boo!`

Making Decisions

When writing programs, you will meet scenarios in which you need to make a decision before executing code. Maybe you want to check that the user has provided valid data, or that a number is within a certain range. You can make these decisions in your code using `if` and `else` statements.

An `if` statement should contain a condition that evaluates to either `true` or `false`. If the condition evaluates to `true`, the code within the block is executed; if it is `false`, the code is skipped. Here is an example of a simple `if` statement:

```
if(a < b) {
  document.write("a is smaller than b!");
}
```

In this example, the condition would evaluate to `true` if the value of the a variable is smaller than the value of the `b` variable. This means that the code would execute and the text would be output to the screen.

You can find a comprehensive list of JavaScript operators that can be used in conditions at `https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Expressions_and_Operators#Comparison_operators`.

You can also define code that should run if the condition evaluates to `false`. This is done by placing the `else` keyword after the closing curly brace of the `if` statement and then placing the code within two new curly braces. This is shown in the following example.

```
if(a < b) {
  document.write("a is smaller than b!");
} else {
  document.write("a is not smaller than b!");
}
```

This sort of control structure is useful when writing computer programs in any programming language. In later chapters you will be using `if` and `else` statements to check whether a user's web browser supports new technologies like GeoLocation, LocalStorage, and Canvas.

Looping

When coding, you will find that you need to repeat a task multiple times. An example use case would be if you had a collection of structured data about 100 users. The task is to display that data on the page, once for each user. Now you certainly don't want to write out the same code 100 times. This is where loop structures come in. Using them to repeat an action a set number of times can greatly decrease the amount of code you have to write.

There are two types of loop structures, `for` loops and `while` loops. In this section, you learn how to use these in your programs and when you should use one instead of the other.

For Loops

`For` loops are used when you have a set number of times that you want to execute some code. A `for` loop has three parameters.

The *initialization* parameter comes first. This is usually used to create a counter variable that can track how many times the `for` loop has executed.

Next up is the *condition*. This will be evaluated before each iteration of the loop and should evaluate to either `true` or `false`, just like conditions in `if` statements. It is commonly used to check whether the counter is within a desired range.

The final parameter is the *final-expression*. This expression is executed after each iteration of the loop. This typically is used to increase the value of the counter variable by one.

Let's write a little program that uses a `for` loop:

1. Create a new file in your text editor.
2. Save this file as `example10-7.js`.
3. Add the following code to this file.

```
window.onload = function() {
  for( var i = 0; i < 5; i++ ) {
    document.write(i + '<br>');
  }
}
```

4. Save the `example10-7.js` file.
5. Create a new HTML file called `example10-7.html`.
6. Copy the following code into this file.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>For Loops</title>
</head>
<body>
  <script src="example10-7.js"></script>
</body>
</html>
```

7. Save this file.

Here you have first initialized a counter variable `i` and given it the value `0`.

The condition then states that this loop should continue to execute for as long as the counter variable `i` is less than 5.

Finally, you declare that after each iteration of this loop, the counter variable should be increased by one. The ++ syntax here is simply a shorthand way of writing `i=i+1`.

If you run this example in your browser, you should see the following:

```
0
1
2
3
4
```

Notice that the output starts from 0. This is because you initialized the counter variable at 0. In programming, you almost always start counting from 0. This can sometimes lead to the infamous *off-by-one* bug, so be careful to check your numbers if your output is slightly off.

While Loops

`While` loops are similar to `for` loops; however, `while` loops can be used when you don't know how many times you want the loop to iterate. A `while` loop takes only one parameter, the condition. It will keep iterating until this condition evaluates to true. You want to be sure that it will evaluate to `true` at some point; otherwise, it will keep iterating until the end of time, or your computer dies, whichever comes first. This is known as an *infinite* loop.

Let's have a bit more fun with `while` loops. It's time to create a little guessing game. As always, the code below can be found on the book's website in the Chapter 10 folder.

1. Create a new file in your text editor called `example10-8.js`.
2. Add the following code to this new file:

```
window.onload = function() {
  // Generate random number between 0 and 10.
  var randomNumber = Math.floor(Math.random()*10);

  // Initialize a variable for the guess.
  var guess;

  // Keep asking the user to guess until he gets the number.
  while(guess != randomNumber) {
    guess = prompt('What is your guess?');
  }

  // Let the user know that he guessed correctly.
  alert('Congratulations! You guessed correctly. The number
  was ' +
  randomNumber + '.');
}
```

3. Save the `example10-8.js` file.
4. Create a new file called `example10-8.html`.
5. Copy the following code into this new file:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>While Loops</title>
</head>
```

```
    <body>
      <h1>The Guessing Game</h1>
      <p>
        Guess a number between 0 and 10.
      </p>
      <script src="example10-8.js"></script>
    </body>
    </html>
```

6. Save the `example10-8.html` file.

Open this example in your browser and give the game a go.

This code is a little more complex than earlier JavaScript you looked at, so I will take you through each part. Lines that begin with `//` are comments.

The objective of this game is for the user to guess a random number that is generated by the program. This number will be between 0 and 10.

First you need to generate the random number. To do this, you initialize a new variable `randomNumber`. You use JavaScript's built-in `Math` library to help generate the number. Let's take a closer look at this segment.

```
Math.floor(Math.random()*10)
```

Here you first use the Math library's `random` function to generate a decimal number between 0 and 1. Your program needs a number between 0 and 10, though, so you multiply this random number by 10 to make it larger. Finally, you only want that random number as an integer so you can use the `floor` function to round your decimal number to the closest integer. The example below shows how the random number is calculated:

```
Math.random() = 0.4382
Math.random()*10 = 4.382
Math.floor(Math.random()*10) = 4
```

Okay, I'm sorry about the math lesson, but it couldn't be avoided. Moving on . . .

Now that you have your `randomNumber,` you create another variable, `guess`, to store the user's latest attempt to beat the game.

The next step is to create the `while` loop that will check to see if the user's guess is correct and prompt her to make another guess if it is not.

```
while(guess != randomNumber) {
  guess = prompt('What is your guess?');
}
```

The condition of the `while` loop compares the `guess` variable to the `randomNumber` variable using the not-equal operator (`!=`). If the guess is correct, it will continue to execute the rest of the JavaScript. However, if it is incorrect, then you will prompt the user to make another guess and store this new attempt in the `guess` variable. The next time the `while` loop condition evaluates, it will check against the new guess.

You may have noticed that you enter the `while` loop before the user has made a guess at all. Because you have declared the `guess` variable but not initialized it with a value, it will be `undefined`. This means that when the `while` loop compares it to `randomNumber`, the result will be false and the user will be prompted to make a guess. This eliminates the need to duplicate the code for the prompt outside of the `while` loop.

Once the user guesses correctly, the rest of the code will be executed, and therefore you add a little congratulations message to let the user know that she won.

```
alert('Congratulations! You guessed correctly. The number was ' +
  randomNumber + '.');
```

Here you have used the `alert` function. This will display the text in a pop-up similar to the one displayed by the `prompt` function.

That's it for loops. Hopefully, you now understand how you can use `for` and `while` loops to make your code much more maintainable by reducing the amount of code that you have to write.

The Document Object Model (DOM)

The Document Object Model (or DOM for short) is a structural representation of a web page. The DOM is generated by taking all your HTML code and all your CSS code and putting them together to create a master blueprint of the page (the DOM). This is then presented on the screen by the browser. As a developer, you can then manipulate the DOM (and therefore the page that is displayed) using JavaScript.

The DOM Tree

The DOM tree consists of all the elements that you have defined in your HTML code. These elements are converted to what are known as *nodes*. Each node represents an object, and these objects are what you will interact with in your JavaScript programs. Remember all of those attributes that you were placing on your HTML elements? These have now become properties of the new DOM objects.

The DOM tree follows the same structure as your HTML document, and therefore elements that are nested in your markup will appear as children of their parent element in the DOM.

This parent-child relationship is important to the way that the DOM tree works. Let's look at an example.

The following code example shows a `<div>` element that contains an unordered list, which in turn contains a number of list items.

```
<div>
    <ul id="fruits">
      <li>Apples</li>
      <li>Oranges</li>
      <li>Bananas</li>
    </ul>
</div>
```

When the DOM is generated, it first creates an element node for the `<div>` element. As the unordered list is nested within this element, it would become a child node of the `<div>` element node. This `<ul>` element also has an `id` attribute. This would become a property of the unordered list node. Each list item then becomes a child node of the unordered list node. Figure 10-1 shows how this segment of the DOM tree would be visualized (not including properties).

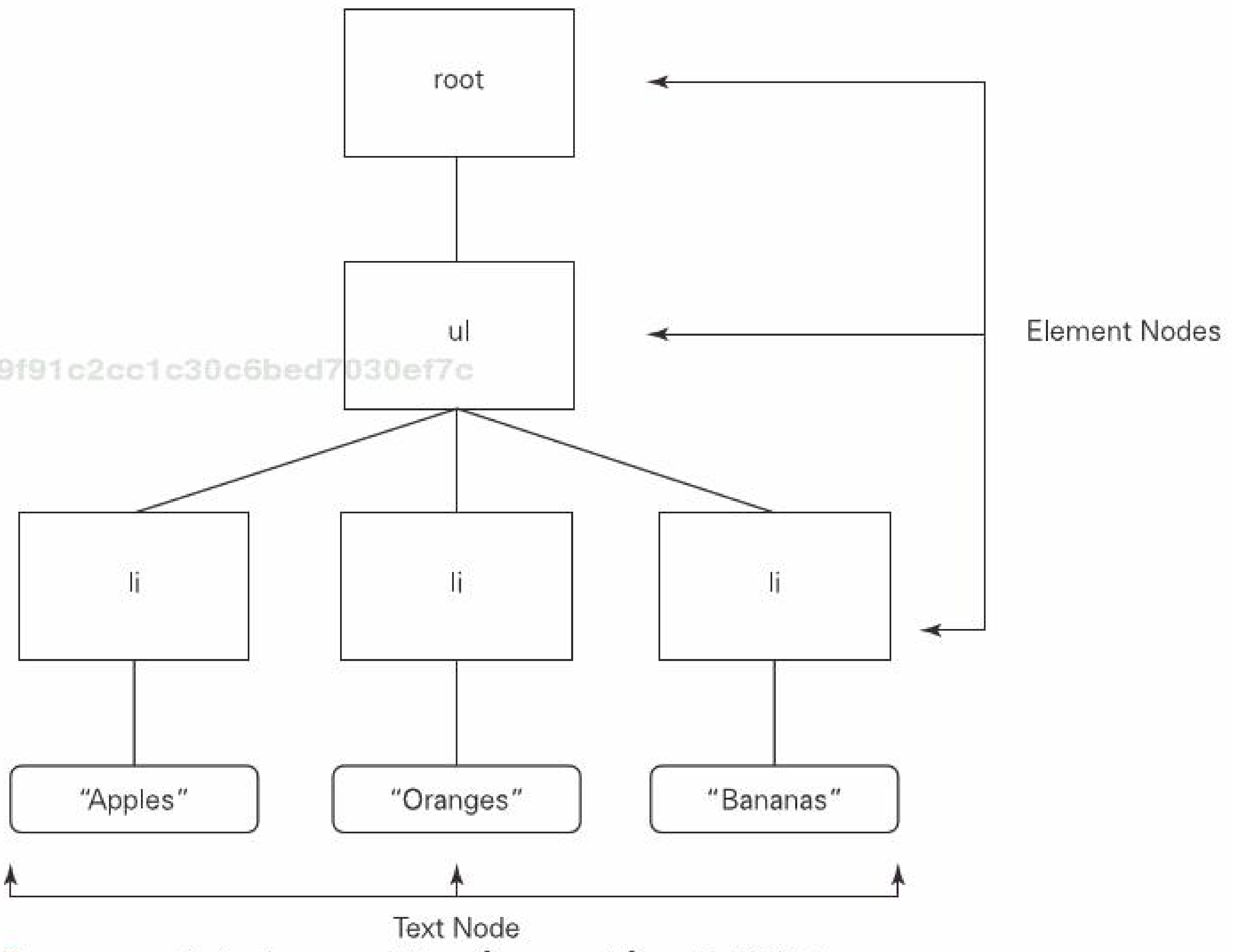


FIGURE 10-1 A visual representation of a segment from the DOM tree.

As you can see in this diagram, there is more than one type of node. The main two types that you will need to worry about are element nodes and text nodes. As the names suggest, *element nodes* are created for HTML elements and *text nodes* are created for the text contained within HTML elements.

Selecting Page Elements

In this section, you discover how to select elements in the DOM using JavaScript. You can use a number of different functions to select elements. Here you are going to look at the four main ones. You use what you learn here in Chapter 11 when building custom playback controls for a video, and in Chapter 12 when you store data entered into a web form.

Up to now, you haven't made much use the developer tools that you installed at the beginning of the book. That's going to change. You could do this work in JavaScript files, as you have done up to now, but I want you to get used to using the JavaScript console that comes with your developer tools. The skills that you learn here will be very useful when it comes to debugging your JavaScript programs in the future.

Debugging refers to the process of examining your code to find and correct errors (or ***bugs***) that are causing your program to behave in a way that it was not intended to.

The following examples use the developer tools supplied with Google Chrome, but feel free to use other tools if you are more familiar with them.

Before starting, you need to set up a test page from which you can select elements. Create a new file using the following code, or alternatively you can use the `example10-9.html`, which can be found in the Chapter 10 folder of the downloadable code resources.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Selectors Test Page</title>
</head>
<body>
  <header>
    <h1>This is a test page</h1>
  </header>

  <section id="text">
    <h1>A Text Section</h1>
    <p id="firstParagraph" class="paragraph">
      This text is within the first paragraph element.
    </p>
```

```
      <p class="paragraph">
        This text is in the second p element.
      </p>
    </section>

    <section id="form">
      <h1>A Web Form</h1>
      <form id="webForm" action="#" method="post">
        <p>
          <label for="name">Name</label>
          <input type="text" id="name" name="name">
        </p>
        <p>
          <label for="email">Email</label>
          <input type="email" id="email" name="email">
        </p>
        <p>
          <button type="submit">Submit</button>
        </p>
      </form>
    </section>
  </body>
  </html>
```

Load this page in your web browser and launch the developer tools. Not sure how to do this? They usually can be found in the Tools menu within your browser. Alternatively, take a quick look at Chapter 1, where I cover the most popular developer tools. Once you have them open, you need to launch the console view. In Chrome, this is done by selecting the Console tab.

Good to go? Awesome.

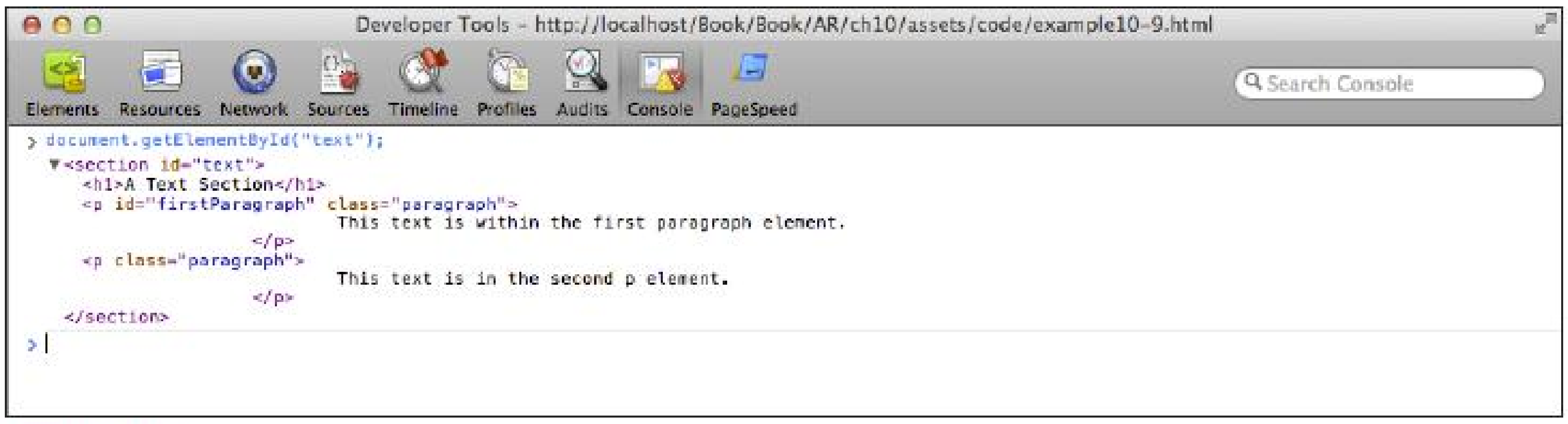
getElementById()

The first selector function that you are going to look at is `getElementById()` (note the lowercase d in Id). This function takes the ID of your element and will return that element's object in the DOM. You call this function on the `document` object.

Let's use this function to select the `<section>` element with the ID `text`. In your console, type the following and then press Enter:

```
document.getElementById("text");
```

Figure 10-2 shows what should be displayed in your console window.



```
> document.getElementById("text");
  ▼<section id="text">
     <h1>A Text Section</h1>
     <p id="firstParagraph" class="paragraph">
                                  This text is within the first paragraph element.
                    </p>
     <p class="paragraph">
                                  This text is in the second p element.
                    </p>
   </section>
> |
```

FIGURE 10-2 Selecting an element getElementById().

Executing this function will return the target element and all its child elements.

getElementsByClassName()

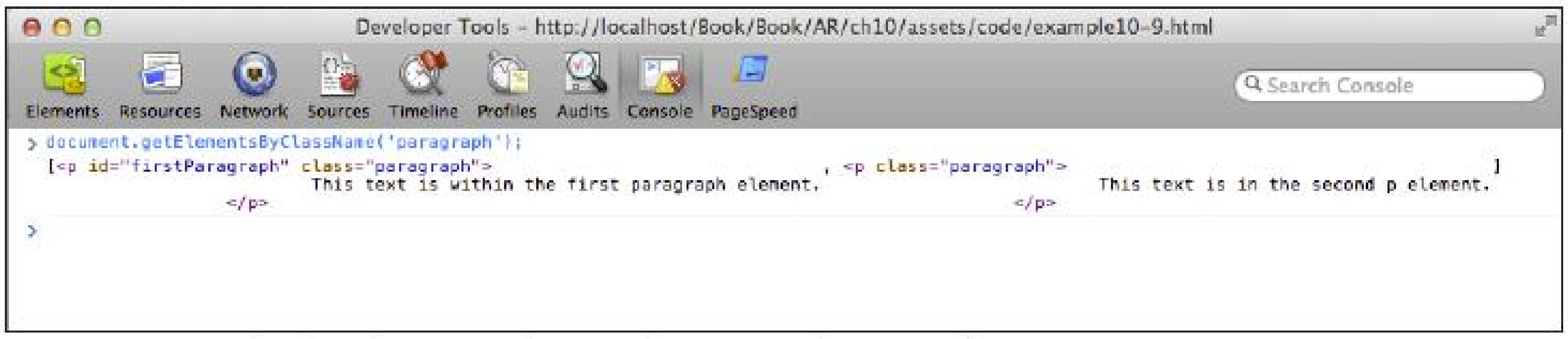
The `getElementsByClassName()` function can be used to simultaneously select multiple elements based on a class that has been assigned to them. This function will return an array of element objects. An *array* is very much like a list, where each item is separated by a comma. Arrays are enclosed within square brackets, as shown in the following example:

```
["red", "green", "blue"]
```

Try using the `getElementsByClassName()` function to select all elements that have the class `paragraph`.

```
document.getElementsByClassName("paragraph");
```

This should return an array of two element objects. These are the paragraphs from the text section. Figure 10-3 shows the output in the console window.



```
> document.getElementsByClassName('paragraph');
  [<p id="firstParagraph" class="paragraph">                              , <p class="paragraph">                              ]
                             This text is within the first paragraph element.                     This text is in the second p element.
                   </p>                                                        </p>
>
```

FIGURE 10-3 Selecting elements using getElementsByClassName().

Note that the `getElementsByClassName()` function is not supported in Internet Explorer 6 to 8.

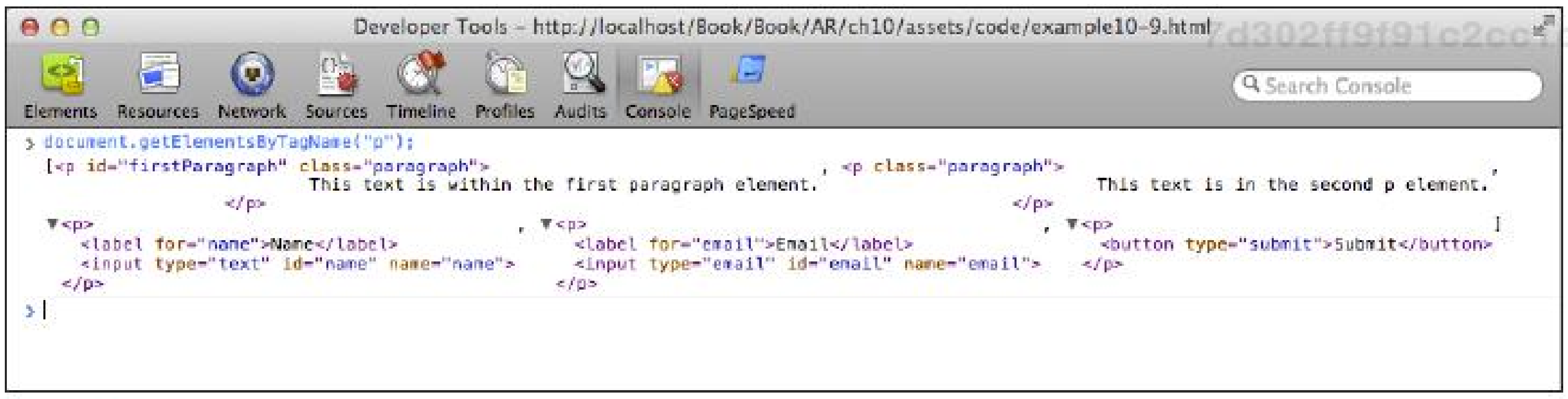
getElementsByTagName()

There may also be scenarios in which you would like to select a number of elements by their tag name. For such occasions, you can use the `getElementsByTagName()` function.

Try selecting all the paragraph elements on the page:

```
document.getElementsByTagName("p");
```

Figure 10-4 shows the expected output for this selection. In this figure I have expanded the final three elements so that you can see their contents by clicking the little gray triangle next to the element in the output.



```
> document.getElementsByTagName("p");
  [<p id="firstParagraph" class="paragraph">
                      This text is within the first paragraph element.
            </p>                                                      , <p class="paragraph">
                      This text is in the second p element.
            </p>                                                      ,
  ▼<p>
     <label for="name">Name</label>
     <input type="text" id="name" name="name">
   </p>                                   , ▼<p>
     <label for="email">Email</label>
     <input type="email" id="email" name="email">
   </p>                                   , ▼<p>
     <button type="submit">Submit</button>
   </p>                                   ]
> |
```

FIGURE 10-4 Selecting Elements Using getElementsByTagName().

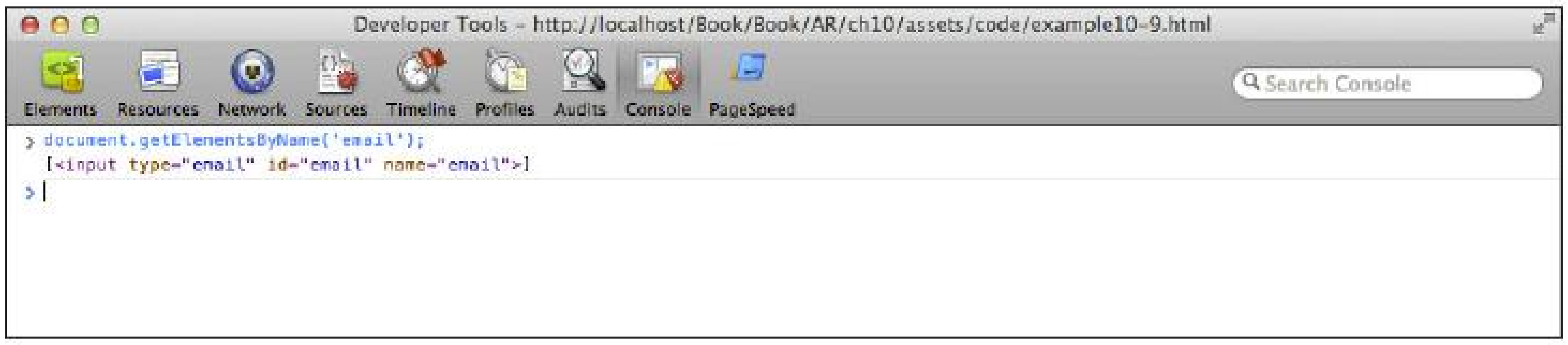
getElementsByName()

Remember those name attributes that you were specifying on form fields? You can also select elements based on those. The `getElementsByName()` function takes in a name and returns an array of all the elements that have that name.

Try selecting the e-mail field using `getElementsByName()`:

```
document.getElementsByName("email");
```

This should return an array with one object, the `<input>` element for the e-mail field, as shown in Figure 10-5.



```
> document.getElementsByName('email');
  [<input type="email" id="email" name="email">]
> |
```

FIGURE 10-5 Selecting elements using getElementsByName().

Interacting with Page Elements

Now that you understand how to select elements using JavaScript, let's look at how you can manipulate their content and attributes.

Manipulating Text Content

First try changing the text of the first <p> element. To do this, select the element and update its `innerHTML` property to `This text was changed using JavaScript`.

```
document.getElementById("firstParagraph").innerHTML = "This text
  was changed using JavaScript";
```

Execute this in your console. You now should see the text on the page change.

What if you wanted to just add more text after what is already there? To do this, you could simply use the += operator instead of =. Here's an example:

```
document.getElementById("firstParagraph").innerHTML += " This text
  was added to the end of the current text.";
```

Manipulating Attributes and Properties

To inspect all the properties that an element has, you can use the `dir()` function in the developer tools console. This will show you a list of all the element's properties as well as its child elements.

Take a look at the properties of the `email` input by selecting it using `getElementById`.

```
dir(document.getElementById("email"));
```

This should display a really long list of properties. You can ignore most of them at this stage, but the key properties are those that link to the element attributes, such as `id`, `class`, `placeholder`, and `value`.

Let's update the value of this field. To do this, select the `<input>` element, specify the property you would like to change, and then assign it a value. Oh, by the way: Did I mention that you can create variables in the console? Let's also create a variable to store the `email` field so that you don't have to type the selection code every time.

```
var emailField = document.getElementById("email");
emailField.value = "test@example.com";
```

You should now see that the value of the `email` field has been updated to the e-mail address you specified.

Throughout the remainder of this book, you will write a lot of JavaScript code that interacts with page elements, updating content and properties based on user input and events.

JavaScript Libraries

JavaScript libraries are collections of code that you can use to make writing JavaScript programs easier by enabling you to write less code. Some of these libraries address cross-browser compatibility issues, and some give you easy ways to create interactive widgets for your pages. Some even do both.

To use a JavaScript library, all you need to do is to include the library script in your HTML file using a `<script>` element. Make sure that you include this script before your JavaScript files, because the browser will load scripts in the order in which it discovers them. You need the library to be seen first so that you can use the functions that it provides.

In this section, you learn some of the basics about the popular JavaScript library called jQuery. You look at some of the ways the jQuery simplifies writing JavaScript programs and how it can differ from writing programs in pure JavaScript.

I strongly recommend that you learn pure JavaScript before getting too involved in libraries like jQuery. That way you have a better understanding of what JavaScript itself is responsible for and what the library is doing to help you out.

jQuery Basics

Before you can use jQuery, you need to download the library and include it in your HTML file. Let's set up another test file so that you can get familiar with jQuery. Follow these steps:

1. Download the latest version from the jQuery website (`http://jquery.com`). At the time of writing this, the latest version is 1.8.1
2. Place this file in your `javascript-examples` folder.
3. Create a new file in your text editor.
4. Save this file as `example10-10.html`.
5. Add the following code to this file:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>jQuery Test Page</title>
</head>
<body>
```

```
    <header>
      <h1>This is a test page</h1>
    </header>

    <section id="text">
      <h1>A Text Section</h1>
      <p id="firstParagraph" class="paragraph">
        This text is within the first paragraph element.
      </p>
      <p class="paragraph">
        This text is in the second p element.
      </p>
    </section>

    <section id="form">
      <h1>A Web Form</h1>
      <form id="webForm" action="#" method="post">
        <p>
          <label for="name">Name</label>
          <input type="text" id="name" name="name">
        </p>
        <p>
          <label for="email">Email</label>
          <input type="email" id="email" name="email">
        </p>
        <p>
          <button type="submit">Submit</button>
        </p>
      </form>
    </section>

    <script src="jquery-1.8.1.js"></script>
  </body>
  </html>
```

6. Save the file.

Now that you have a test page set up, let's look at some of the ways that using jQuery can make writing JavaScript programs easier.

Executing Code on Page Load

In pure JavaScript, you can tell the browser to execute some code when the page loads by attaching a function to the `window.onload` event. The following code shows how this would be done:

```
window.onload = function() {
    // Do something here.
}
```

jQuery provides an alternative way of doing this, as shown here:

```
$(function() {
    // Do something here.
});
```

The dollar ($) symbol at the beginning of this code represents the jQuery object. This object provides a number of helper functions that can take care of common tasks, such as selecting elements and updating the content of those elements.

Many developers (including me) prefer to use the jQuery method of executing code when the page has loaded, even though the syntax itself does not make much sense to humans.

Selecting Elements

The jQuery library provides a simpler way of selecting page elements that is very popular with developers. Instead of using the `getElementsBy...` functions, you can simply pass a selector to the jQuery object and it will return the page element or an array of page elements. If you are familiar with CSS, jQuery uses the same syntax as CSS selectors for referring to IDs and classes. Let me show you some examples comparing the pure JavaScript selectors to jQuery selectors. Load your new test page and try these out in the developer tools console.

To select an element by its ID in jQuery, you use the # sign followed by the ID:

```
// Select an element by its ID.
document.getElementById("text"); // Pure JavaScript
$("#text"); // jQuery
```

Using jQuery, you can select all elements within a certain class by placing a period before the class name:

```
// Select all elements with a class.
document.getElementsByClassName("paragraph"); // Pure JavaScript
$(".paragraph"); // jQuery
```

To select elements by their tag name, simply use the name of the tag. No special characters are needed:

```
// Select elements by their tag name.
document.getElementsByTagName("p"); // Pure JavaScript
$("p"); // jQuery
```

jQuery has other nifty selectors that you can use, too. You can find a full list of these on the jQuery website: `http://api.jquery.com/category/selectors/`.

Other Benefits of Using jQuery

As well as providing handy ways to select elements, jQuery offers a lot of other useful functions. There are functions for easily updating the properties and content of elements as well as manipulating CSS styling. There are even functions available to help you add effects such as fades and animations to your web pages. The jQuery library also provides a series of functions that make it easier to set up event listeners.

One of the biggest benefits of using jQuery is that the library accounts for the inconsistencies in browsers so that you don't have to. As explained earlier in this book, some browser vendors decided to implement technologies slightly differently, whether it is a variation in how a particular function is named or whether a browser vendor decided to include a function at all. This means that it is often necessary to write multiple variations of the same code in order to ensure that it will work in all web browsers. The jQuery library takes care of a lot of these inconsistencies for you so that you can just focus on writing your programs.

That concludes the brief tour of jQuery. If you like the look of jQuery's simpler syntax, check out the documentation on the jQuery website (`http://jquery.com`).

Summary

Congratulations! You just learned the basics of a new programming language.

JavaScript is one of the most important languages for creating modern web applications, especially if you want to take advantage of the awesome new features introduced in HTML5.

In this chapter, you learned the basics of JavaScript and created a number of programs of varying complexity. You also learned how the DOM works and what happens to your HTML and CSS code when the browser processes it. You were introduced to a number of key programming concepts in this chapter, but it takes months (if not years) to truly master all that JavaScript has to offer. Now that you have the basic foundations in place I recommend that you explore JavaScript some more by getting a book on the subject or completing the JavaScript courses on Treehouse. You might want to start with the *JavaScript Foundations* course at `http://teamtreehouse.com/library/websites/javascript-foundations`.

In the next chapter, you dig into some slightly more advanced HTML5 features, starting with native video and audio. You learn how to embed videos and audio clips into your web pages and how to use the new skills that you acquired in this chapter to create custom controls to manage playback.

chapter eleven

Adding Video and Audio

THE WEBSITE YOU have been building for Joe's Pizza Co. looks pretty good so far, but it would look even better if it included a video. In this chapter, you update the About page, adding a short video with some custom playback controls that you build yourself. When the video is in place, you learn how you can make video and audio content more accessible by adding subtitles to your multimedia content.

Until recently, embedding video and audio within a web page required the use of a third-party plug-in, such as Adobe Flash or Microsoft Silverlight. The result was a fragmented ecosystem of plug-ins that users had to install, manage, and upgrade, often confusing less tech-savvy individuals. This approach also meant that developers had less control over their video content.

The capability to create custom playback controls or special effects required the use of proprietary software such as Adobe Flash Professional. Even when developers did take the leap and purchase this considerably expensive software, they still had to learn a whole new programming language, ActionScript, in order to achieve what they desired.

Fortunately, HTML5 introduces a new way of embedding video and audio into web pages. Using the new `<video>` and `<audio>` elements, developers can now add multimedia content to their websites as easily as they can add images or text. Hurray!

Converting Video and Audio Files

Before you can start adding video and audio to your web pages, you first need to make sure browsers will be able to play your multimedia files. In this section, you learn about the various video and audio formats that are natively supported by modern web browsers. (*Natively* means there is no need for a plug-in.) You also create the video files you will use in the Joe's Pizza Co. website.

Video File Formats

5

Three main video formats are used for HTML5 video: WebM, MP4, and OGV. Unfortunately, the browser vendors each had their own opinions on which format should become the standard; therefore, no one format is supported by all modern web browsers. This means you must convert your video file into all three formats to ensure that everyone can see it. I cover how to detect whether a browser supports a certain video format later in this chapter.

Table 11-1 shows browser support for the three main video formats. Notice that by using the WebM and MP4 formats, you can cover all modern web browsers. MP4 files, however, are usually quite large, so it is often useful to provide an OGV file, too, because OGV files are much smaller and take less time to download.

Table 11-1 Video Codec Support in Modern Browsers

	WebM (VP8 codec)	MP4 (H.264 codec)	OGV (Ogg Theora codec)
Chrome	Yes	Yes	Yes
Firefox	Yes	No	Yes
Opera	Yes	No	Yes
Safari	No	Yes	No
IE9 +	Yes	Yes	No

For WebM to work in IE and Safari, the user has to manually install the WebM codec. You can download the installer here: `https://tools.google.com/dlpage/webmmf`.

Browser support for these codecs may change over time. You can check the latest browser support for these codecs using the following links:

- **WebM** - `http://caniuse.com/#feat=webm`
- **MP4** - `http://caniuse.com/#feat=mpeg4`
- **OGV** - `http://caniuse.com/#feat=ogv`

Audio File Formats

Just like with video formats, not a single accepted audio format is supported in all browsers. This means that, once again, you must convert your audio file into a number of different formats to ensure that everyone can enjoy it.

Table 11-2 lists the four main audio formats. The best way to cover all modern web browsers is to provide your audio in MP3, M4A (AAC), and WAV formats.

Table 11-2 Audio Codec Support in Modern Browsers

	OGG (Vorbis codec)	MP3	M4A (AAC codec)	WAV
Chrome	Yes	Yes	No	Yes
Firefox	Yes	No	No	Yes
Opera	Yes	No	No	Yes
Safari	No	Yes	Yes	Yes
IE 9 +	No	Yes	Yes	No

Converting the Video File

Now that you know about the various audio and video formats supported by modern web browsers, it's time to prepare the video files for the Joe's Pizza Co. website. In this section, you use a free tool called Firefogg to convert an MP4 file to OGV and WebM.

If you have trouble converting these video files, you can find copies of the converted files in the `videos` subfolder within folder 1 of the download code for Chapter 11.

To save you from having to download multiple copies of these video files (one set for each code folder), I placed the videos in folder 1 of the download files. If you need to view the About page from any of the other code folders in your browser, please copy the video files from folder 1 into the desired code folder; otherwise the video will not load.

Also, I have a confession to make. The video file you will use is actually a sketch involving Mike the Frog, the mascot from TeamTreehouse.com. Ideally, when you create websites for businesses in the wider world, you will use a more relevant video. This one, however, is just fine for teaching you all about HTML5 video.

Here are the steps to convert an MP4 file:

1. Create a new folder in your project directory and name it `videos`.
2. Download the `mikethefrog.mp4` file from this book's website at `http://wiley.com/go/treehouse/html5foundations` (video courtesy of © Treehouse Island, Inc.). You can find the file in the `chapter11/1/videos` folder of the download files.

3. Place the `mikethefrog.mp4` file in the `videos` folder that you created within your project directory.
4. The Firefogg converter requires that you are using Firefox. If you do not already have Firefox installed, you can download it from `http://www.mozilla.org/en-US/firefox/new/`.
5. Open Firefox and go to `http://firefogg.org`.
6. Click the button that reads Install Firefogg. This will install a Firefox extension. Click Allow and Install Now on the dialogs that appear. You may need to restart Firefox when the installation is complete.
7. Navigate to `http://firefogg.org/make/index.html` using Firefox.
8. Click Select File and select the `mikethefrog.mp4` file from your `videos` folder.
9. On the page that loads, make sure WebM (VP8/Vorbis) is selected in the format drop-down menu. Select the 1080p option from the preset drop-down.
10. Click the Encode button and save the video file in the `videos` folder as `mikethe-frog.webm`. Firefogg may take a little while to encode the new file. Once Firefogg completes the encoding, you should be presented with the video; the application also saves a copy of the new video in your videos folder.
11. Repeat Steps 7 to 10, this time selecting the Ogg (Theora/Vorbis) option from the format drop-down menu.
12. Check that the two new video files (`mikethefrog.webm` and `mikethefrog.ogv`) are in your `videos` folder; then close Firefox.

You now have all the video files you need for the rest of this chapter. In the next section, you embed these video files into your About page.

Adding a Video to the About Page Using the <video> Element

5

The new HTML5 `<video>` element has become very popular recently, with many companies implementing support for HTML5 video within their websites. At the time of this writing, YouTube is running a trial where you can opt-in to being shown HTML5 video (you can sign up at `http://youtube.com/html5`). Vimeo has a similar scheme running. Treehouse also uses HTML5 video to display the lessons on its website.

The `<video>` element, in its simplest form, is similar to the `<img>` element in that it uses a `src` attribute to specify a path to the video.

```
<video src="videos/mikethefrog.webm"></video>
```

Also note that the `<video>` element has a complete set of start and end tags. You can place some content between these tags that should be displayed if the browser does not support HTML5 video (more on this in the "Ensuring Compatibility" section coming up).

In this chapter, you mainly use the `<video>` element; however, the `<audio>` element is nearly identical. What you learn in this chapter can be transferred to the `<audio>` element when embedding audio content.

Adding the <video> Element

In this section, you add a basic `<video>` element to your About page. To start, you use only one video file, the WebM file. It's best, therefore, to use a browser that supports this format (such as Google Chrome). In the next section, you learn how to provide your video in multiple formats to ensure that everyone can view it.

The code for this exercise can be found in folder 2.

Follow these steps to add a `<video>` element:

1. Open the `about.html` file in your favorite text editor.
2. Between the main page heading (`<h1>`) and the page text, add a new `<div>` element with the ID `video`. This element will contain your `<video>` element as well as the controls that you build later in this chapter.

   ```
   <div id="video"></div>
   ```

3. Now add a `<video>` element to this new `<div>`.

   ```
   <video></video>
   ```

4. Set the `src` attribute on the `<video>` element to point to the `mikethefrog.webm` file in your `videos` folder.

   ```
   <video src="videos/mikethefrog.webm"></video>
   ```

5. Set the `width` attribute to 400 and the `height` attribute to 225.

   ```
   <video src="videos/mikethefrog.webm" width="400" height="225">
   </video>
   ```

6. Add an `id` attribute to the `<video>` element and set its value to `myVideo`.

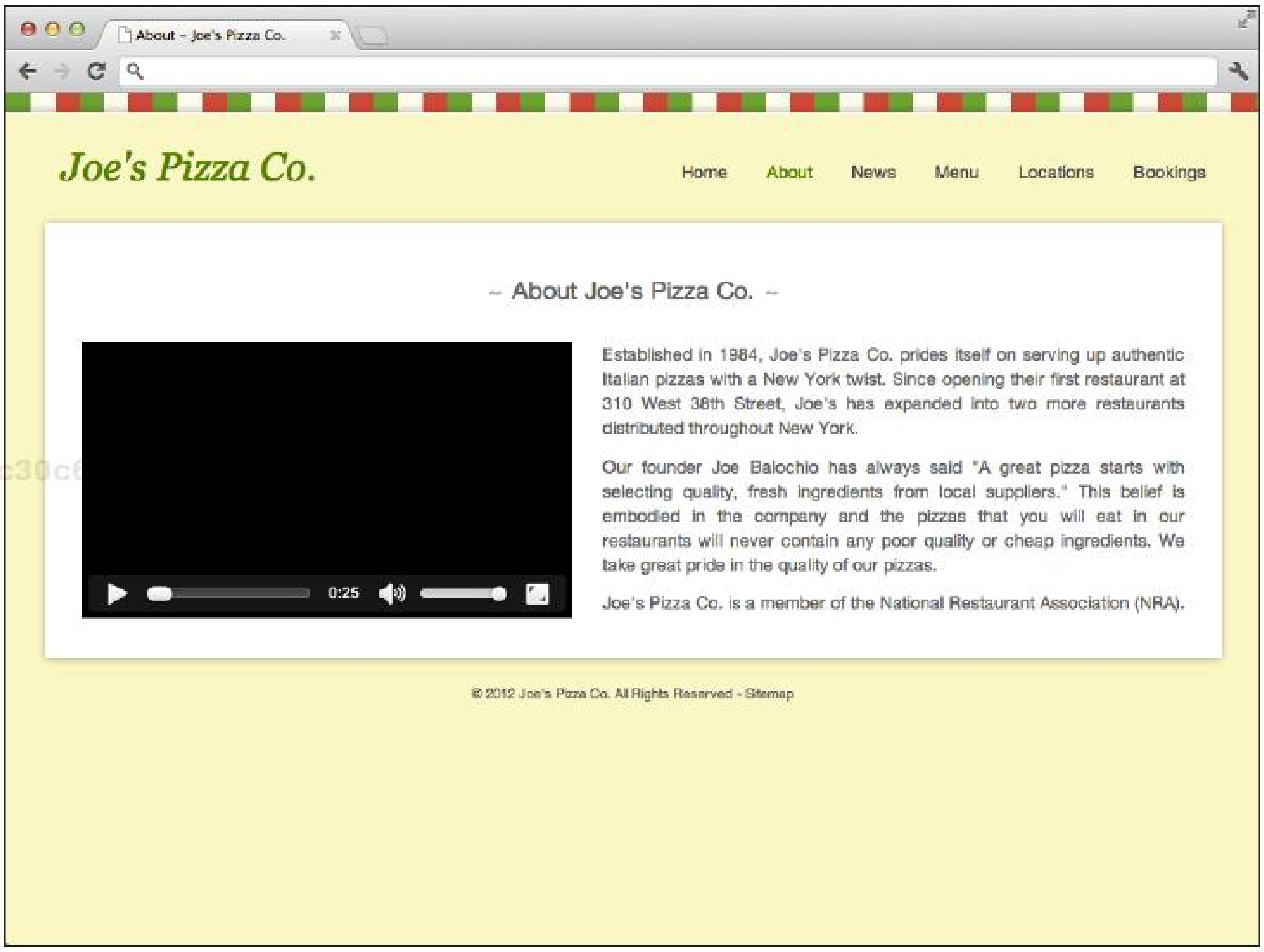
```
<video src="videos/mikethefrog.webm" id="myVideo"
       width="400" height="225">
</video>
```

7. Now add a `controls` attribute to the `<video>` element. This prompts the browser to display its default video controls. Later in this chapter, you will replace these with your own custom controls.

```
<video src="videos/mikethefrog.webm" id="myVideo" width="400"
       height="225" controls>
</video>
```

8. Save the `about.html` file.

If you load up the About page in your web browser, you should see that your video appears to the left of the page text, as shown in Figure 11-1. Click the Play button to ensure that the video will start playing.



~ About Joe's Pizza Co. ~

Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

© 2012 Joe's Pizza Co. All Rights Reserved - Sitemap

FIGURE 11-1 The About page with video.

Styling for the default browser controls has not been standardized, so these controls may look slightly different depending on which browser the video is being viewed in.

Ensuring Compatibility

As I mention earlier in this chapter, there is no single universal video format. If you were to open your About page in a browser that doesn't support the WebM format (such as Safari), you would see that the video never loads. You can solve this problem by providing multiple video files to the browser. The browser then chooses the one it can play and ignores the rest.

You provide multiple file formats using `<source>` elements. These elements go inside the `<video>` element and should have `src` and `type` attributes.

```
<video id="myVideo" width="400" height="225" controls>
  <source src="videos/mikethefrog.webm" type="video/webm">
  <source src="videos/mikethefrog.ogv" type="video/ogv">
  <source src="videos/mikethefrog.mp4" type="video/mp4">
  <p>
    Your browser doesn't support HTML5 video.
    <a href="videos/mikethefrog.mp4">Download</a> the video
    instead.
  </p>
</video>
```

You can also add some fallback content to the `<video>` element that will display if the browser does not support HTML5 video (such as the `<p>` element in the preceding example). This fallback content should usually include a link for the user to download the video file directly.

The `<video>` element is not supported in Internet Explorer versions 8 and earlier. You can check cross-browser support for this element here: `http://caniuse.com/#feat=video`.

Now it's time to modify your About page and add the other video files you created earlier in this chapter. You also add in some fallback content.

The code for this exercise can be found in folder 3.

Here are the steps for modifying your About page to add other video files:

1. Open the `about.html` file in your text editor.
2. Remove the `src` attribute from the `<video>` element.
3. Add a `<source>` element for the WebM file within the `<video>` element.

   ```
   <source src="video/mikethefrog.webm" type="video/webm">
   ```

4. Now add a `<source>` element for the OGV file.

   ```
   <source src="video/mikethefrog.ogv" type="video/ogv">
   ```

5. Finally, add a `<source>` element for the MP4 file.

   ```
   <source src="video/mikethefrog.mp4" type="video/mp4">
   ```

6. Now add some fallback content.

   ```
   <p>
     Your browser doesn't support HTML5 video.
     <a href="videos/mikethefrog.mp4">Download</a> the video
     instead.
   </p>
   ```

7. Save the `about.html` file.

Now, if you open the About page in Safari, you would see that the video plays perfectly. This is because the browser chooses a file that it can play from the source options you give it.

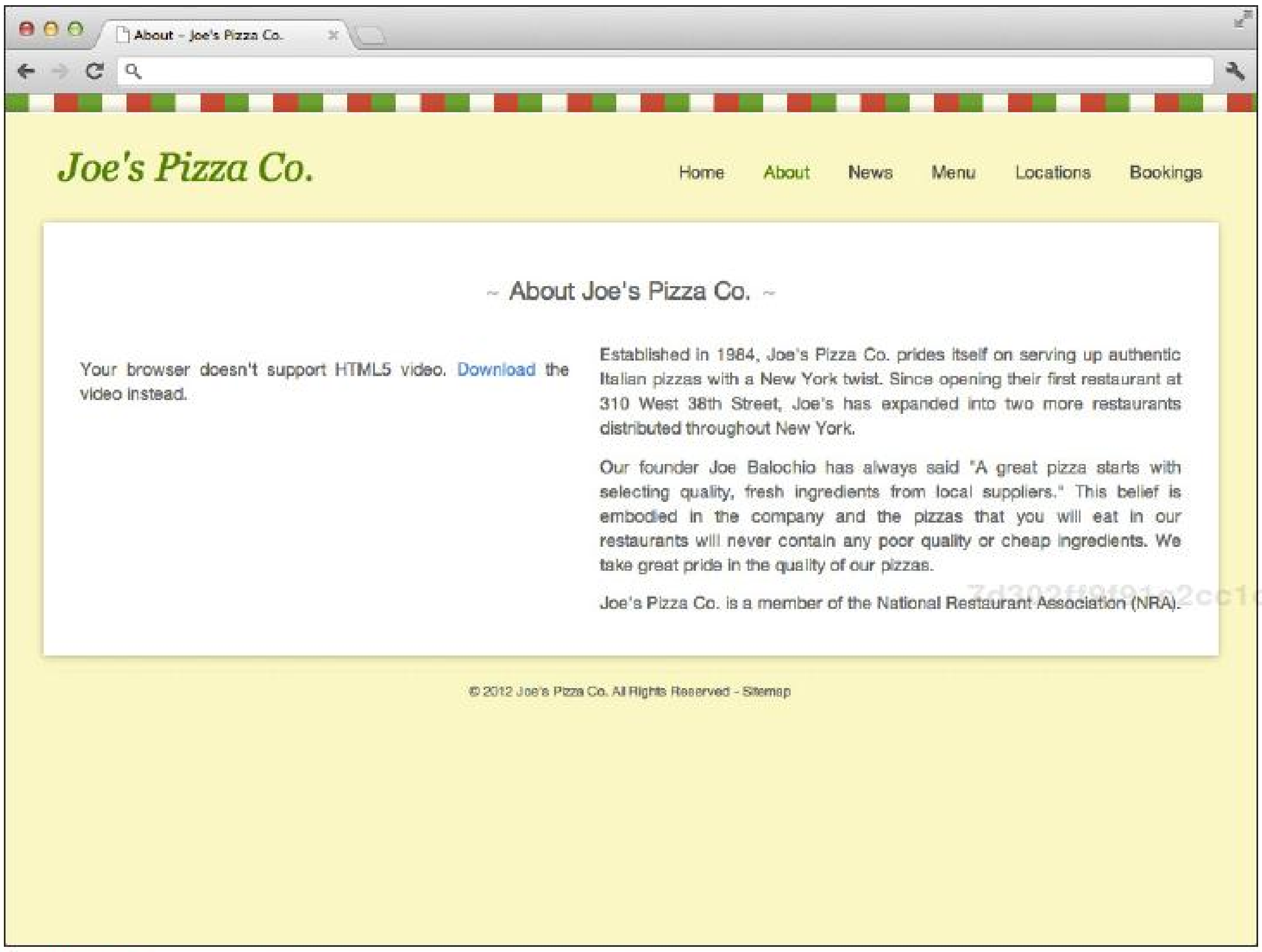
You also added some fallback content, just in case the user's browser doesn't support HTML5 video. Figure 11-2 shows how the page looks if the fallback content is displayed instead of the video.

Adding a Poster Image

5

Depending on your browser, you might have noticed that when the page first loads the video is displayed as a solid black box (refer to Figure 11-1). This may not be the same in all browsers, but there is a way that you can make sure that this black box does not appear.

The `<video>` element has a `poster` attribute that can be used to specify a path to an image file that should be displayed in the video's place until the user clicks the Play button.



~ About Joe's Pizza Co. ~

Your browser doesn't support HTML5 video. Download the video instead.

Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

© 2012 Joe's Pizza Co. All Rights Reserved - Sitemap

FIGURE 11-2 How the About page appears if the user's browser does not support HTML5 video.

In this section, you add a poster image to the video on the About page.

The code in this exercise can be found in folder 4.

Follow these steps to add a poster image:

1. First, download the `poster.png` file from the book's website and place it in your `img` folder. You can find this file in the `chapter11/4/img` folder of the download files.
2. Open the `about.html` file in your text editor.
3. Add a `poster` attribute to your `<video>` element and set its value to `img/poster.png`.

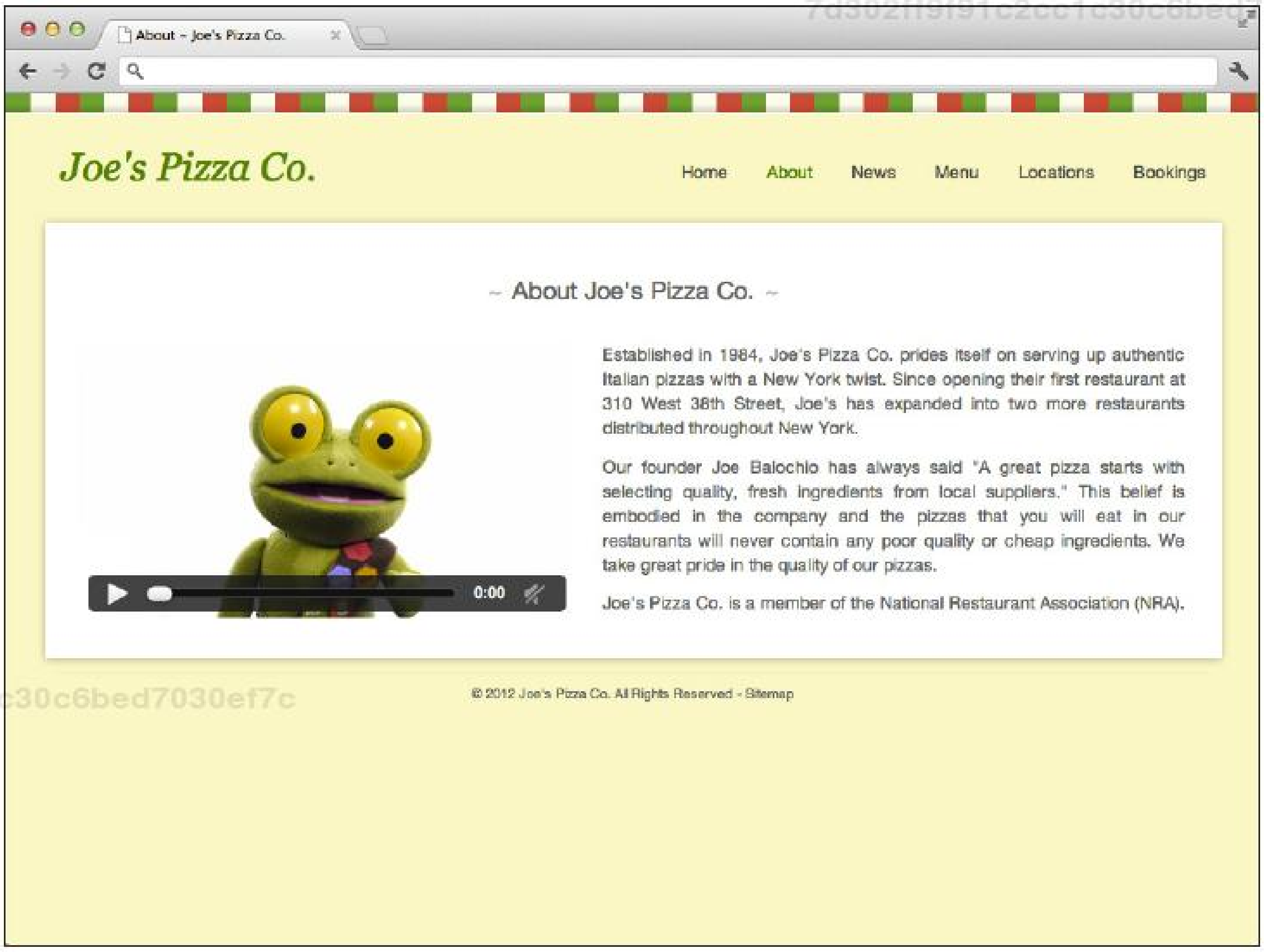
```
<video id="myVideo" width="400" height="225" controls
       poster="img/poster.png">
  ...
</video>
```

4. IE9 contains a bug that prevents the browser from using the poster image. To work around this bug, you need to add a `preload` attribute to the `<video>` element and set its value to none.

```
<video id="myVideo" width="400" height="225" controls
       poster="img/poster.png" preload="none">
  ...
</video>
```

5. Save the `about.html` file.

Now, if you take a look at the About page in your web browser, you should see a nice image of Mike the Frog, as shown in Figure 11-3.



~ About Joe's Pizza Co. ~

Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

© 2012 Joe's Pizza Co. All Rights Reserved - Sitemap

FIGURE 11-3 The video element now has a poster image that replaces the black box that was previously displayed.

The fallback content displays only if a browser does not support HTML5 video at all. If the browser does support HTML5 video but you haven't provided a file in a format it can play, the browser just shows an empty black box.

Other <video> Attributes

In addition to the `poster` attribute, you can use a few other attributes on the `<video>` element. You won't be using the following attributes in the example website, but they are useful to know about.

The `autoplay` attribute tells the browser to start playing the video as soon as the page finishes loading. This is a *Boolean* attribute; it has no value because its presence alone is enough to prompt an action by the browser.

```
<video src="." width="." height="." autoplay></video>
```

Be sure to use the `autoplay` attribute wisely. It can be annoying for users if your video starts playing on its own. If you are going to use the `autoplay` attribute, consider using the `muted` attribute too. The `muted` attribute simply mutes the audio track of the video. Again this is a Boolean attribute.

```
<video src="." width="." height="." muted></video>
```

The `muted` attribute is not supported in Safari and IE9 (and earlier versions).

The `loop` attribute causes a video or audio clip to repeat itself every time it finishes. This attribute is particularly useful when applied to `<audio>` elements.

```
<audio src="." width="." height="." loop></audio>
```

Creating Custom Controls Using JavaScript

So far in this chapter, you have relied on the browser's default controls for controlling playback. In this section, you get rid of these default controls and build your own custom controls that allow the user to play, pause, seek, and mute your video as well as control the volume.

The `<video>` and `<audio>` elements both have the same underlying interface, `HTMLMediaElement`. This interface allows you to interact with the video and audio content in your pages using JavaScript. As you progress through this chapter, you learn about the various methods and properties that you can use to control your media content. Again, in this section, you focus on video—but everything you learn here applies to audio, too.

Before you can start building your custom playback controls, you first need to add a JavaScript file to your project. This file contains all the JavaScript code that controls the video.

The code for this exercise can be found in folder 5.

Here are the steps for adding a JavaScript file:

1. Create a folder named `js` in the root of your project folder.
2. Within this new `js` folder, create a new file called `video.js`.
3. Open the `video.js` file in your text editor.
4. Copy the following code into this file. Here you create a variable (`video`) that contains a reference to the `<video>` element on the page.

```
window.onload = function() {
  // Get the video.
  var video = document.getElementById("myVideo");
}
```

5. Save the `video.js` file.
6. Open the `about.html` file in your text editor.
7. Add the following `<script>` element just before the `</body>` tag. This tells the browser to load the `video.js` file into the About page.

```
  ...
  <script src="js/video.js"></script>
</body>
```

8. Remove the `controls` attribute from the `<video>` element.
9. Save the `about.html` file.

That's it! Everything is set up, and you're ready to start building those custom controls. Let's get started.

Debugging Your JavaScript Code

If you encounter any problems when building the controls in this chapter, try using your browser's developer tools to help you find any bugs that might be in your code.

Most modern web browsers have a JavaScript console as part of their developer tools. If the browser encounters an error when trying to execute your JavaScript code, it prints the error message to this console. To access the JavaScript console, click the Console tab in your developer tools.

When something is not working as you want it to, check this console for any errors. The errors can help you to pinpoint the problem with your code. It is often something as simple as a misspelling in a method name or a missing semicolon from the end of a line.

Creating the Play Button

To start playing a video, you can call the `play()` method on the `<video>` element through JavaScript.

The Play button uses a `<button>` element that hooks up to an event listener in your JavaScript code. When the button is clicked, the event listener calls the `play()` method on your `<video>` element.

The code for this exercise can be found in folder 6.

Start by adding the new Play button to your HTML code. Follow these steps:

1. Open the `about.html` file in your text editor.
2. Underneath the `<video>` element, create a new `<div>` with the ID `video-controls`.

```
<video id="myVideo" width="400" height="225"
       poster="img/poster.png" preload="none">
  ...
</video>
<div id="video-controls"></div>
```

3. Within this new `<div>` element, create a `<button>` element. Set the type to `button` and the ID to `playBtn`.

```
<div id="video-controls">
  <button type="button" id="playBtn">Play</button>
</div>
```

4. Save the `about.html` file.

Next, you need to set up an event listener that fires when the Play button is clicked:

1. Open the `video.js` file in your text editor.
2. Create a new variable called `playBtn` and initialize it by fetching the Play button in your HTML.

```
window.onload = function() {
  // Get the video.
  var video = document.getElementById("myVideo");

  // Get the buttons.
  var playBtn = document.getElementById("playBtn");
}
```

3. Now create an event listener on the Play button that fires when the `click` event occurs.

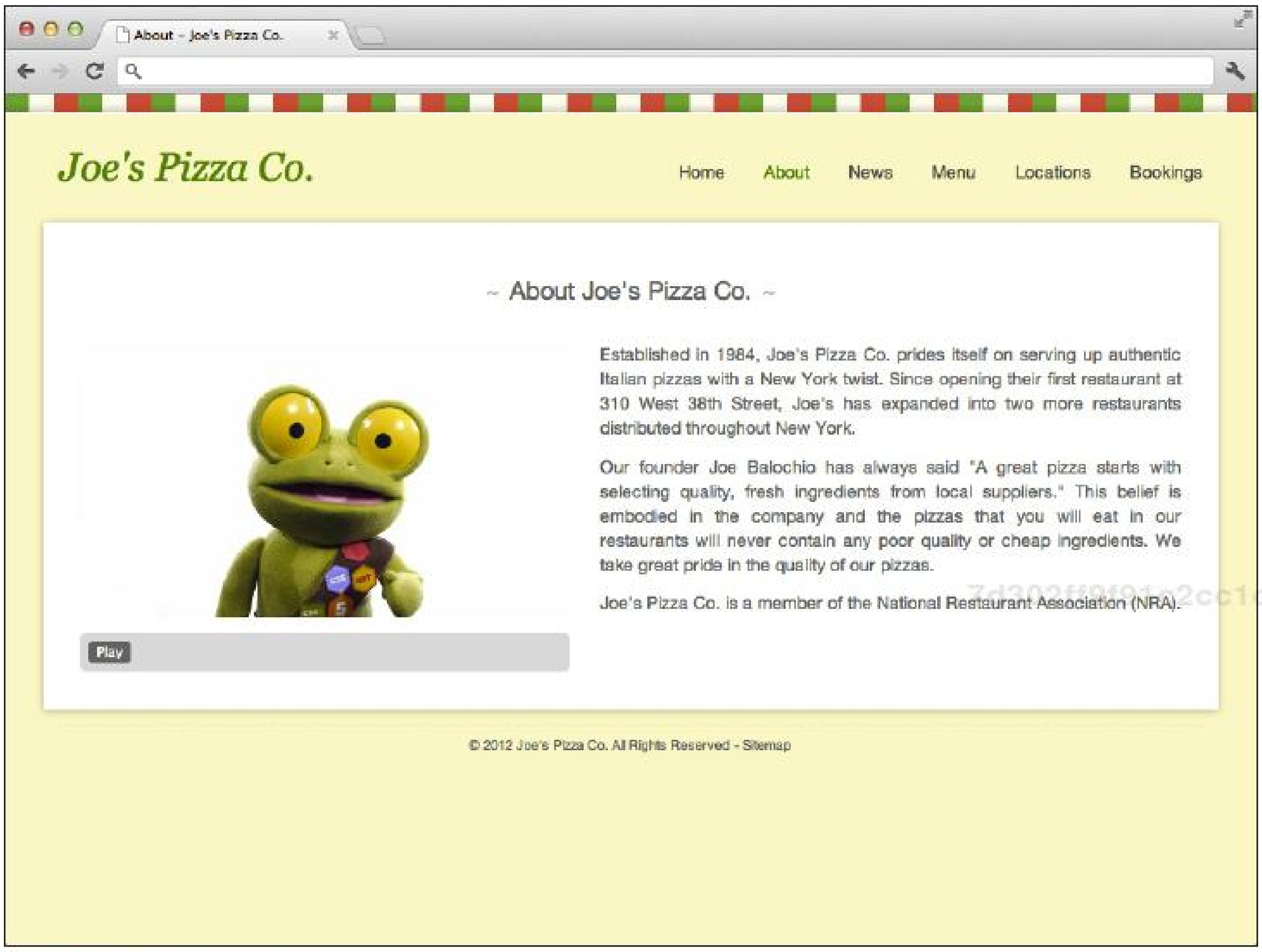
```
window.onload = function() {
  ...
  // Add an event listener for the play button.
  playBtn.addEventListener("click", function(e) {

  });
}
```

4. Within the function block of this event listener, add a statement that calls the `play()` method on the `video` variable.

```
// Add an event listener for the play button.
playBtn.addEventListener("click", function(e) {
  // Play the video.
  video.play();
});
```

5. Save the `video.js` file.

Open the About page in your web browser. If all went according to plan, a Play button should be beneath the video, as shown in Figure 11-4. Click the button and the video should start playing. Success!


About – Joe's Pizza Co.
Joe's Pizza Co.
Home
About
News
Menu
Locations
Bookings
~ About Joe's Pizza Co. ~
Play
© 2012 Joe's Pizza Co. All Rights Reserved - Sitemap

Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

FIGURE 11-4 The new Play button.

If clicking the Play button doesn't start your video, check your code against the files in folder 6 of the download code. There is probably a typo or a missing semi-colon somewhere.

There is one problem: You don't have a way to stop the video yet. Let's fix that.

Creating the Pause Button

The `<video>` and `<audio>` elements have a method you can call that causes playback to pause. This is the `pause()` method.

In this section, you follow a process similar to the one you used when you created the Play button.

The code for this exercise can be found in folder 7.

Start by adding the new Pause button to your HTML code:

1. Open the `about.html` file in your text editor.
2. Create a new `<button>` element underneath the Play button. Set the type to `button` and the ID to `pauseBtn`.

```
<div id="video-controls">
  <button type="button" id="playBtn">Play</button>
  <button type="button" id="pauseBtn">Pause</button>
</div>
```

3. Save the `about.html` file.

Now, you need set up an event listener that fires when the Pause button is clicked:

1. Open the `video.js` file in your text editor.
2. Create a new variable called `pauseBtn` and initialize it by fetching the Pause button in your HTML.

```
window.onload = function() {
  ...
  // Get the buttons.
  var playBtn = document.getElementById("playBtn");
  var pauseBtn = document.getElementById("pauseBtn");
}
```

3. Create an event listener on the Pause button that fires when the `click` event occurs. You should add this below the event listener you created for the Play button.

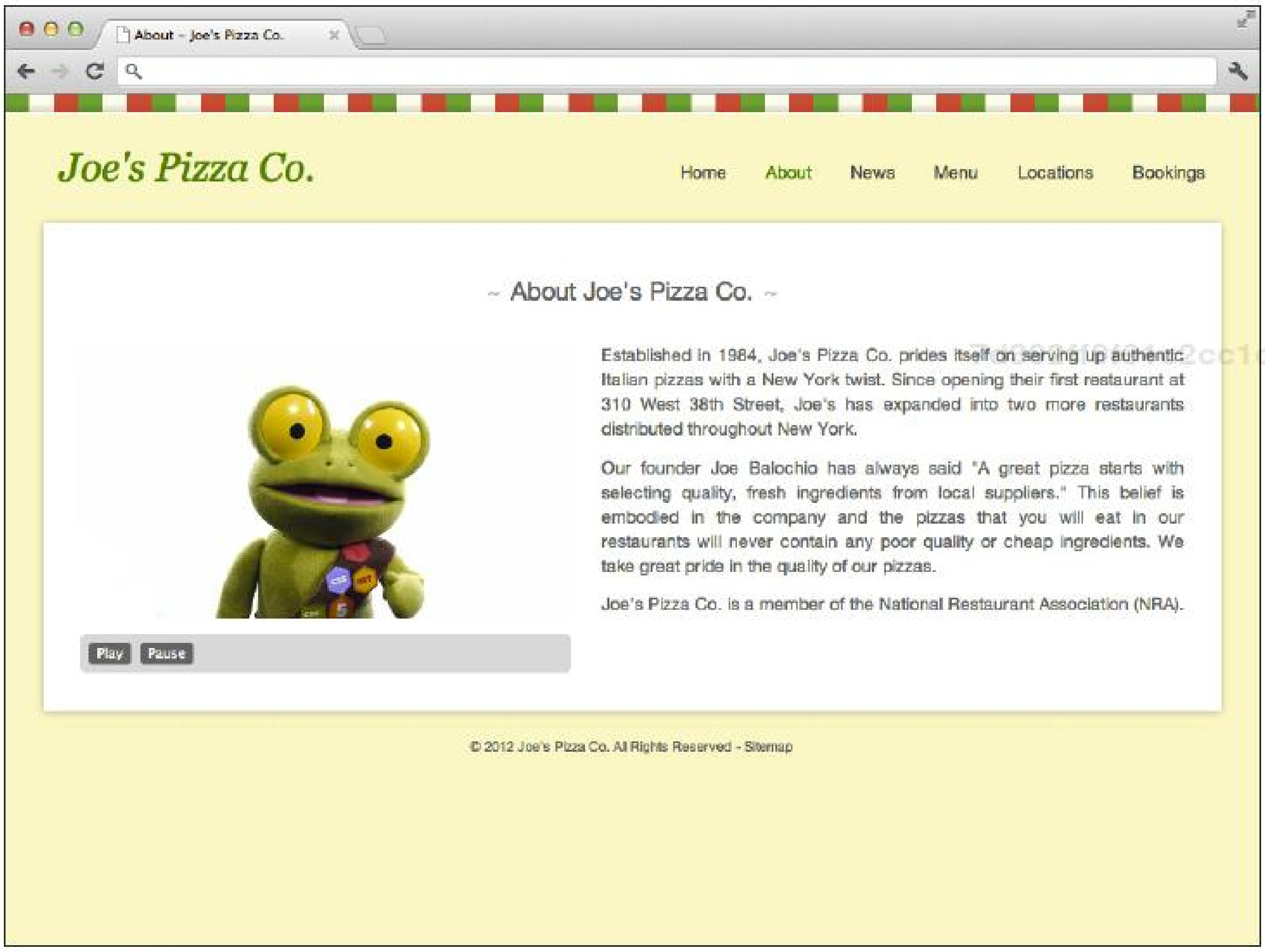
```
window.onload = function() {
  ...
  // Add an event listener for the pause button.
  pauseBtn.addEventListener("click", function(e) {

  });
}
```

4. Within the function block of this event listener, add a statement that will call the `pause()` method on the `video` variable.

```
// Add an event listener for the pause button.
pauseBtn.addEventListener("click", function(e) {
  // Pause the video.
  video.pause();
});
```

5. Save the `video.js` file.

Open the About page. You should see your new Pause button, as shown in Figure 11-5. Try playing the video and then using the Pause button to pause playback. If the Pause button doesn't work, try using your browser's developer tools to diagnose what is causing the problem.



~ About Joe's Pizza Co. ~

Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

© 2012 Joe's Pizza Co. All Rights Reserved - Sitemap

FIGURE 11-5 The new Pause button.

Seeking by Using a Slider

You can now control the basic playback of your video, but it would be kind of cool if you could also seek through the video. To do this, you create a Seek bar using a range slider that's hooked up to an event listener. Every time the user moves the Seek bar, the action manipulates the video's `currentTime` property, which moves playback to the desired point.

The code for this exercise can be found in folder 8.

Let's start by adding the range slider to your HTML:

1. Open the `about.html` file in your text editor.
2. Create a new `<input>` element below the Pause button. Set its type to `range`, its ID to `seekBar`, and its `value` to 0.

   ```
   <input type="range" id="seekBar" value="0">
   ```

3. Save the `about.html` file.

Next, you need to hook this range slider up to the video using an event listener:

1. Open the `video.js` file.
2. Create a new variable called `seekBar` and initialize it by fetching the `<input>` element you just created.

   ```
   window.onload = function() {
     ...
     var pauseBtn = document.getElementById("pauseBtn");
     var seekBar = document.getElementById("seekBar");
     ...
   }
   ```

3. Create a new event listener on the `seekBar` that listens for the `change` event.

   ```
   window.onload = function() {
     ...
     // Add an event listener for the seek bar.
     seekBar.addEventListener("change", function(e) {

     });
   }
   ```

4. Within the function block of this event listener, you first need to calculate the time the video must be moved to. You do this by multiplying the duration of the video by the value of the Seek bar divided by 100. You access the video's `duration` property to get the video duration.

   ```
   // Add an event listener for the seek bar.
   seekBar.addEventListener("change", function(e) {
     // Calculate the time in the video that playback
     // should be moved to.
     var time = video.duration * ( seekBar.value / 100 );
   });
   ```

5. Now that you have the time variable calculated, you need to update the video's currentTime property.

```
// Add an event listener for the seek bar.
seekBar.addEventListener("change", function(e) {
  // Calculate the time in the video that playback
  // should be moved to.
  var time = video.duration * ( seekBar.value / 100 );

  // Update the current time in the video.
  video.currentTime = time;
});
```

6. Save the video.js file.

Now if you open the About page and play the video, you can skip to different parts of the video using the new Seek bar. Great work so far! There are still a few little problems, however. As you drag the bar, the video continues to attempt to play, causing jerky playback. The position of the slider handle also does not move along the slider as the video plays.

Later, we address the problem of jerky playback. First, let's fix the Seek bar so that the position of the slider handle updates as the video plays:

1. In the video.js file, create a new event listener on the video variable (yes, videos fire events too!) that listens for the timeupdate event. This event is fired repeatedly as the video plays.

```
window.onload = function() {
  ...
  // Update the seek bar as the video plays.
  video.addEventListener("timeupdate", function(e) {

  });
}
```

2. Calculate the correct position for the slider handle relative to the playback of the video. To do this, you divide the video duration by 100 and then multiply the result by the current time in the video.

```
// Update the seek bar as the video plays.
video.addEventListener("timeupdate", function(e) {
  // Calculate the slider value.
  var value = ( 100 / video.duration ) * .
  video.currentTime;
});
```

3. Update the slider value using the `value` variable you just created. This moves the slider handle along the slider.

```
// Update the seek bar as the video plays.
video.addEventListener("timeupdate", function(e) {
  // Calculate the slider value.
  var value = ( 100 / video.duration ) *
  video.currentTime;

  // Update the slider value.
  seekBar.value = value;
});
```

4. Save the `video.js` file.

Open the About page again and click the Play button. You should now see that the slider handle moves along the slider as the video plays. Awesome!

Let's fix that problem with jerky playback as you seek the video.

To fix this problem, you use two event listeners. The first event listener pauses the video when the user clicks the slider handle; the second event listener plays the video when the user releases the mouse button. This prevents the video from trying to play as the user moves the slider handle. Follow these steps:

1. Create a new event listener on the `seekBar` that listens for the mousedown event, and place a statement inside its function block that will pause the video.

```
window.onload = function() {
  ...
  // Pause playback when the user starts seeking.
  seekBar.addEventListener("mousedown", function(e) {
    video.pause();
  });
}
```

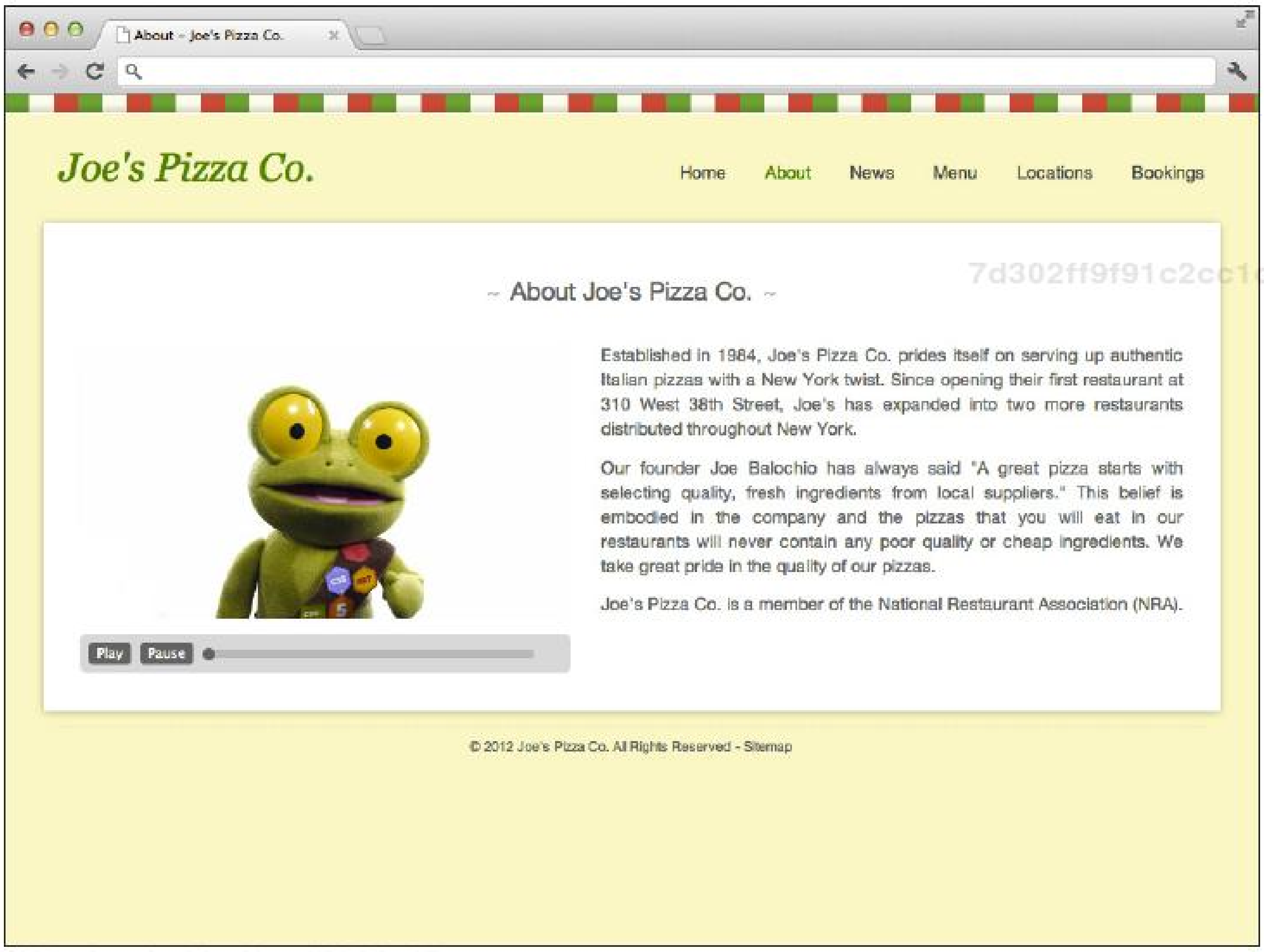
2. Create the other event listener that starts the video playing again when the user releases the mouse button (the mouseup event).

```
window.onload = function() {
  ...
  // Continue playback when the user stops seeking.
  seekBar.addEventListener("mouseup", function(e) {
    video.play();
  });
}
```

3. Save the `video.js` file.

That's it! You're all done with the Seek bar.

Open the About page and play around a bit with the Seek bar. You should see that the video pauses when you start to drag the slider handle and then starts playing again when you release it. The frame displayed in the video still changes as you drag the handle to show where you are moving to in the video. Figure 11-6 shows how your page should look with the new Seek bar.



Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

© 2012 Joe's Pizza Co. All Rights Reserved - Sitemap

FIGURE 11-6 The new Seek bar.

Next, you tackle the volume control.

Creating the Volume Control

The Volume control also uses a slider. This time it is hooked up to an event listener that manipulates the video's `volume` property. The value of the `volume` property should be a number between 0 and 1.

The code in this exercise can be found in folder 9.

Here are the steps for creating a volume control:

1. Open the `about.html` file.
2. Add an `<input>` element and `<label>` below the Seek bar. Note the attributes on the `<input>` element.

   ```
   <label for="volume">Volume: </label>
   <input type="range" id="volume" min="0" max="1" step="0.1"
          value="1">
   ```

3. Save the `about.html` file.
4. Open the `video.js` file.
5. Create a new variable `volumeControl` and initialize it by fetching the volume slider in your HMTL.

   ```
   window.onload = function() {
     ...
     var seekBar = document.getElementById("seekBar");
     var volumeControl = document.getElementById("volume");
     ...
   }
   ```

6. Create a new event listener on `volumeControl` that listens for the `change` event.

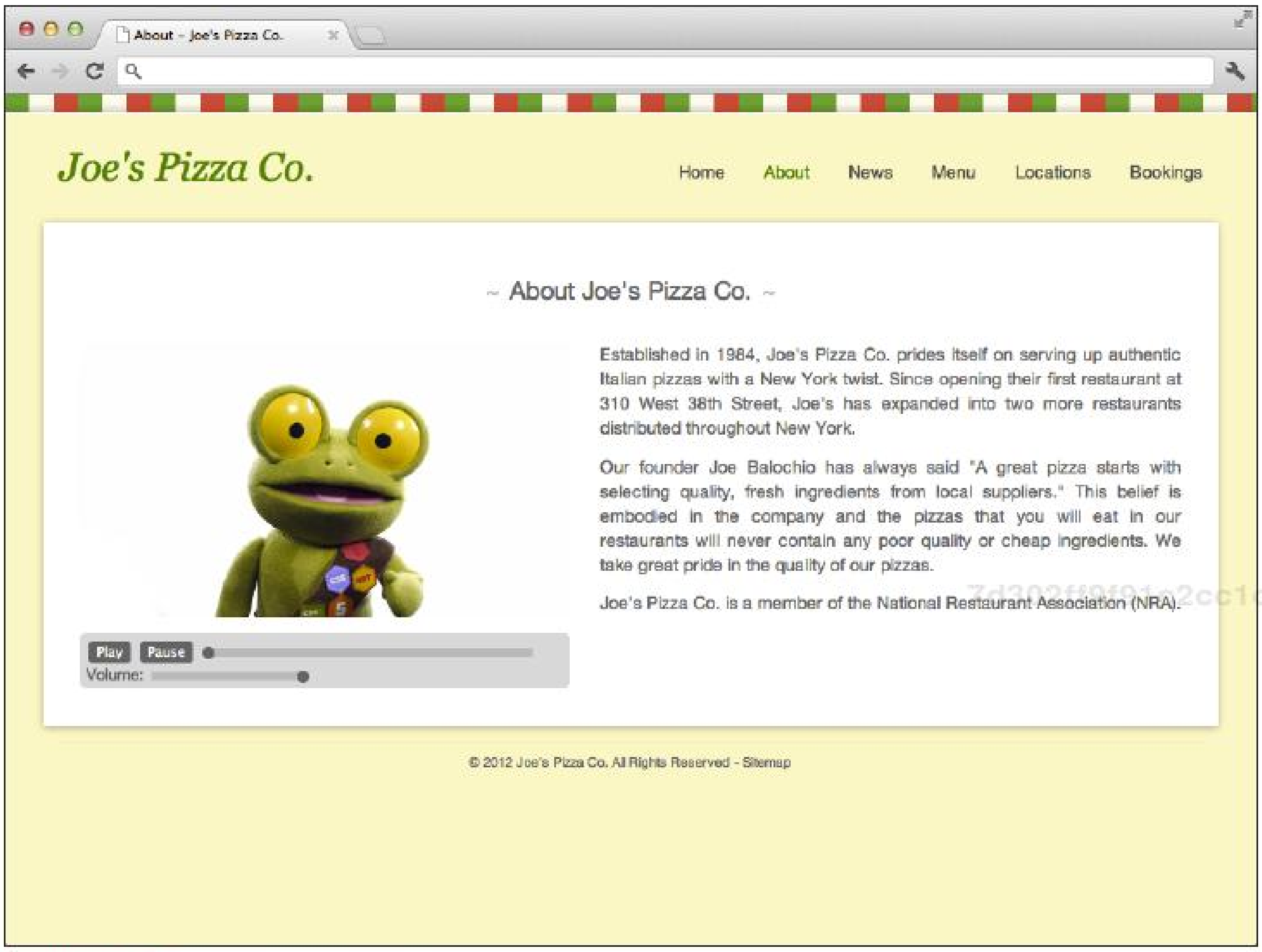
   ```
   window.onload = function() {
     ...
     // Add an event listener for the volume control.
     volumeControl.addEventListener("change", function(e) {

     });
   }
   ```

7. Inside the function block of this event listener, update the video's `volume` property using the current value of the `volumeControl`.

   ```
   // Add an event listener for the volume control.
   volumeControl.addEventListener("change", function(e) {
     // Update the videos volume property.
     video.volume = volumeControl.value;
   });
   ```

8. Save the `video.js` file.

Open the About page and start playing the video. As you drag the volume slider, the volume changes. Figure 11-7 shows how your new Volume control should look.



~ About Joe's Pizza Co. ~

Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

© 2012 Joe's Pizza Co. All Rights Reserved - Sitemap

FIGURE 11-7 The new Volume control.

Creating a Mute Button

The volume slider you just created works great, but sometimes it can be useful to have a Mute button, too. You can mute the audio for your video by setting the `muted` property to `true`.

You should always use the `muted` property rather than just setting the volume to 0. That way, you do not have to write code that remembers the volume level before you mute and retrieves that value when you unmute.

The code for this exercise can be found in folder 10.

Here are the steps for creating a Mute button:

1. Open the `about.html` file in your text editor.
2. Create a new `<button>` element below the volume slider and give it the ID `muteBtn`.

```
<button type="button" id="muteBtn">Mute</button>
```

3. Save the `about.html` file.
4. Open the `video.js` file in your text editor.
5. Add a variable for the Mute button called `muteBtn`.

```
window.onload = function() {
  ...
  var volumeControl = document.getElementById("volume");
  var muteBtn = document.getElementById("muteBtn");
  ...
}
```

6. Create a new event listener for `muteBtn` that listens for the `click` event.

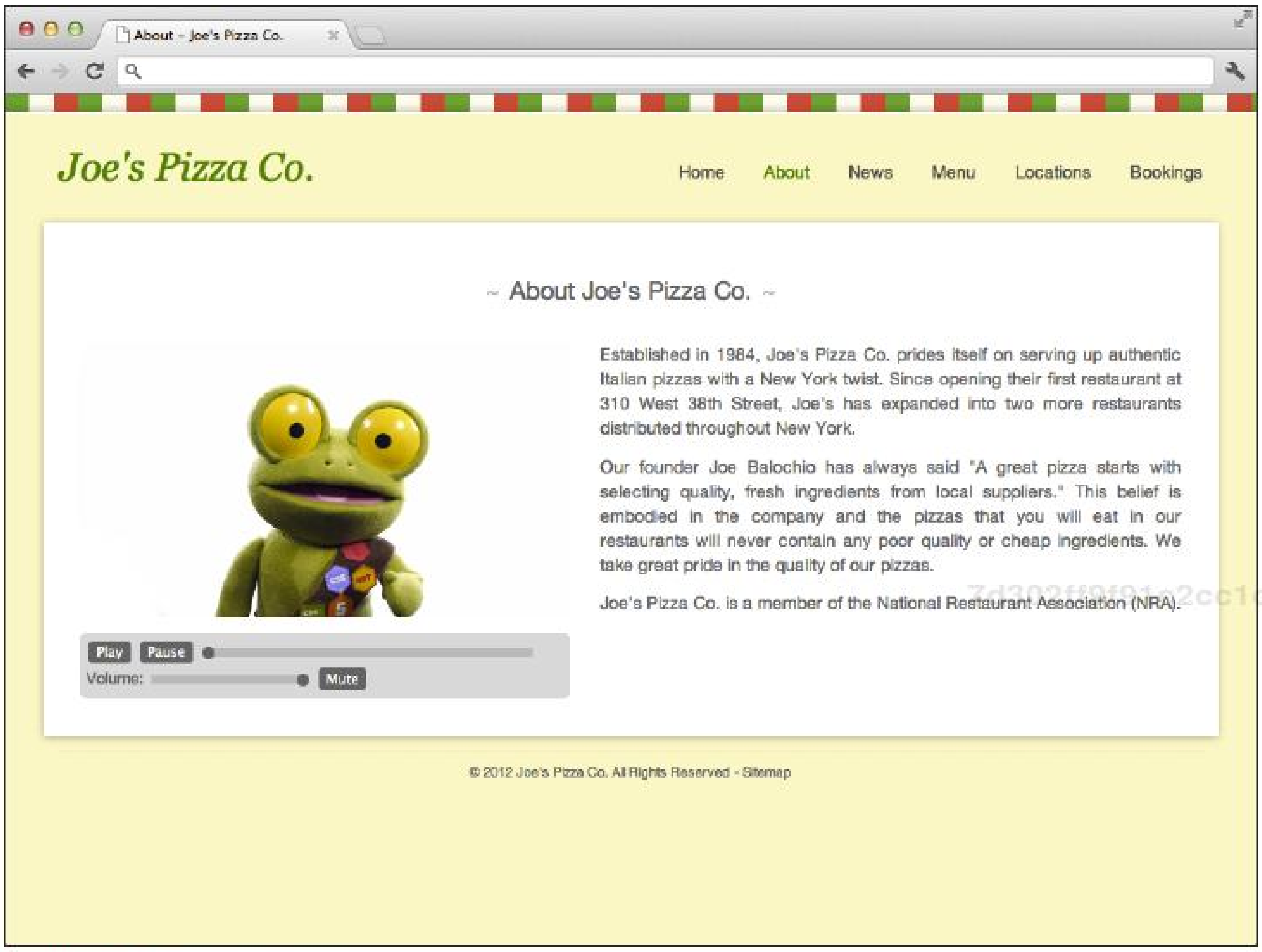
```
window.onload = function() {
  ...
  // Add an event listener for the mute button.
  muteBtn.addEventListener("click", function(e) {

  });
}
```

7. Inside the function block of this event listener, you use an if/else statement to check whether the video is currently muted. If it is, you set the `muted` property to `false` and update the button text to `Mute`; otherwise, set the `muted` property to `true` and the button text to `Unmute`.

```
// Add an event listener for the mute button.
muteBtn.addEventListener("click", function(e) {
  // Toggle the muted value.
  if (video.muted == true) {
    video.muted = false;
    muteBtn.textContent = "Mute";
  } else {
    video.muted = true;
    muteBtn.textContent = "Unmute";
  }
);
```

8. Save the `video.js` file.

Open the About page in your web browser. If you start playing the video and then click the Mute button, the audio is muted. Click the button again and the audio comes back. Figure 11-8 shows how the Mute button should look in your browser.



~ About Joe's Pizza Co. ~

Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

FIGURE 11-8 The Mute button.

You've now created all your custom video controls. The CSS stylesheet you added in Chapter 2 takes care of styling them appropriately. Next, you find out how to make your video more accessible by adding subtitles.

Making Your Video Accessible

It's important to remember that not everyone is lucky enough to experience the rich media that the web has to offer. HTML5 has been designed with this in mind. A number of technologies are available that you can use to ensure that people with disabilities can enjoy your multimedia content. In this section, you learn about the different types of subtitles and how you can use the HTML5 `<track>` element to add subtitles to your video and audio content.

You can package subtitles with your videos in two ways. The first method, *in-band* subtitles, involves embedding the subtitle file within the video file itself. The MP4 file format, for example, is a container for an H.263 video component and an AAC audio component. This format also has the facility for you to store more metadata files within it (including subtitle files). After a subtitle file has been embedded within a file, it is still up to the media player (read: browser) to find it and put it to use.

The second method you can use to add subtitles to a video is *out-of-band* subtitles. These are not directly packaged within the media file; instead, they are provided as separate .VTT files. The new `<track>` element can be used to associate these files with a video.

Later in this section you use the HTML5 `<track>` element to add subtitles to your video. First, however, take a quick look at some of the attributes of the `<track>` element:

- **src**—This attribute provides a path to the subtitles file.
- **kind**—This attribute describes what kind of content the file referenced in the `src` attribute contains. We will be using `subtitles`, but other possible values are `captions`, `descriptions`, `chapters`, and `metadata`.
- **label**—The attribute should be used to provide a title for the track. If you have more than one `<track>` element, the browser may present a list of subtitles for the user to choose from; this label will appear in that list.
- **srclang**—This attribute specifies the content language of the subtitles file. Browsers may try to select the correct subtitle's file based on the value of this attribute and the user's preferences.
- **default**—This attribute sets the current track to be used as default unless the user's browser preferences specify otherwise.

I think you've learned enough about the `<track>` element now. Let's add some subtitles to your video.

The code for this exercise can be found in folder 11.

Follow these steps to add subtitles:

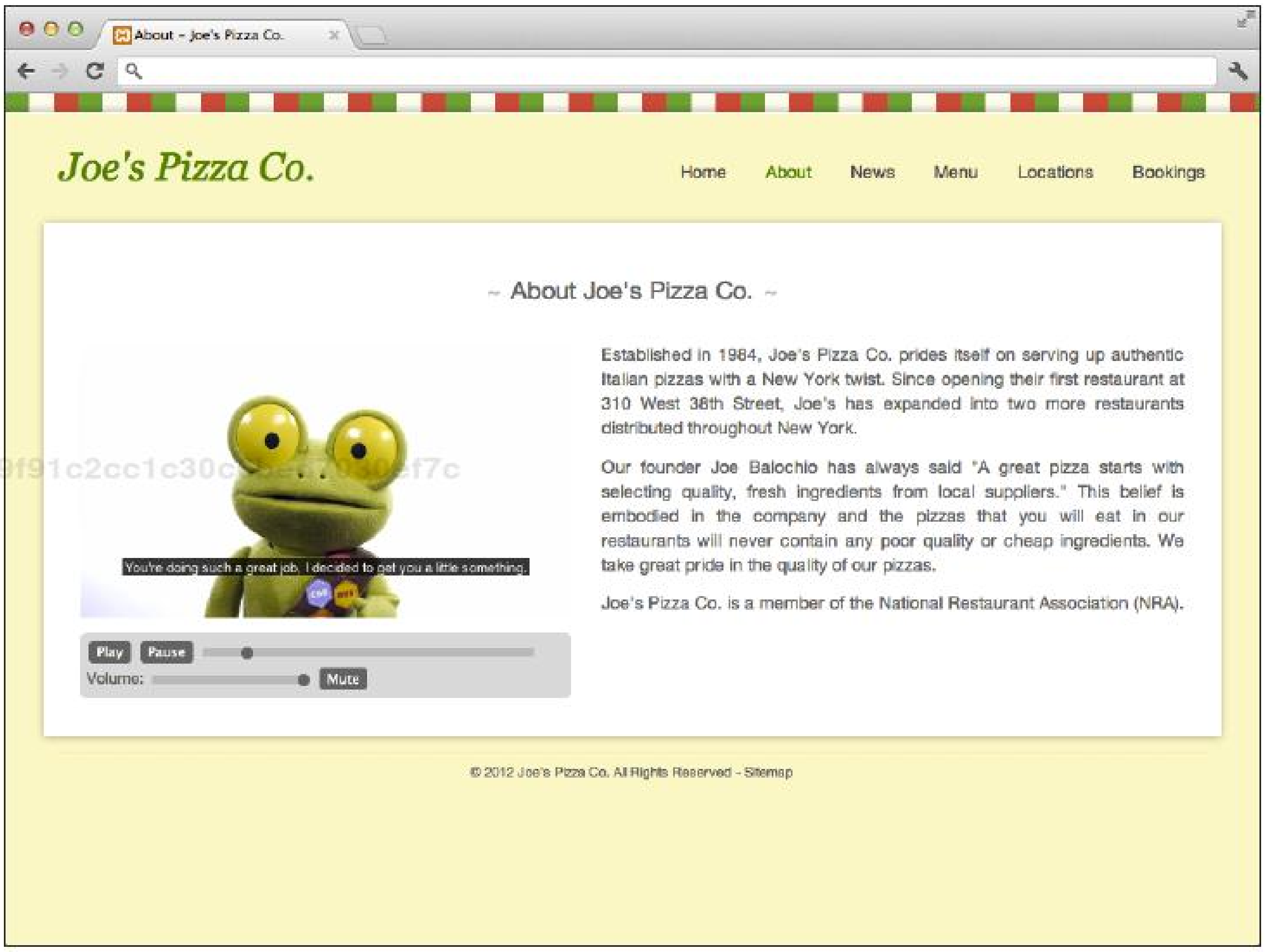
1. Download the subtitle's file (`subtitles_en.vtt`) from the book's website and place it in your `videos` folder. You can find it in the `chapter11/11/videos` folder.
2. Open the `about.html` file in your text editor.
3. Underneath the last `<source>` element, add the following `<track>` element.

```
<video id="myVideo" width="400" height="225"
       poster="img/poster.png" preload="none">
  <source src="videos/mikethefrog.webm" type="video/webm">
  <source src="videos/mikethefrog.ogv" type="video/ogv">
  <source src="videos/mikethefrog.mp4" type="video/mp4">
```

```
        <track kind="subtitles" label="English subtitles"
               src="videos/subtitles_en.vtt" srclang="en"
               default>
        </track>

        <p>
          Your browser doesn't support HTML5 video.
          <a href="videos/mikethefrog.mp4">Download</a> the
          video instead.
        </p>
      </video>
```

4. Save the `about.html` file.

To test out the subtitles, you need to install a local development server on your computer. If you do not feel comfortable doing this, you can skip the remainder of this section and trust me when I say the subtitles will work. Figure 11-9 shows how the page looks when subtitles are enabled.



~ About Joe's Pizza Co. ~

You're doing such a great job. I decided to get you a little something.

Established in 1984, Joe's Pizza Co. prides itself on serving up authentic Italian pizzas with a New York twist. Since opening their first restaurant at 310 West 38th Street, Joe's has expanded into two more restaurants distributed throughout New York.

Our founder Joe Balochio has always said "A great pizza starts with selecting quality, fresh ingredients from local suppliers." This belief is embodied in the company and the pizzas that you will eat in our restaurants will never contain any poor quality or cheap ingredients. We take great pride in the quality of our pizzas.

Joe's Pizza Co. is a member of the National Restaurant Association (NRA).

© 2012 Joe's Pizza Co. All Rights Reserved - Sitemap

FIGURE 11-9 The About page with video subtitles.

Browser support for the `<track>` element is still lacking a bit, so I recommend that you test your About page in Google Chrome. To enable support for the `<track>` element in Chrome, type **chrome://flags** into the address bar and press Enter; then enable the Enable <track> Element option.

At the time of this writing, the `<track>` element still does not work unless the website is on a web server. So far in this book, you have been accessing the web pages directly from your hard drive using the `file://` protocol. (Take a look at the address bar in your browser; it's there.) If you want to test the subtitles, you need to install a local development server using a tool such as XAMPP (`http://www.apachefriends.org/en/xampp.html`). When you have a local development server installed, make sure that your project files are in a folder accessible by the web server (such as the `xampp/htdocs` folder if using XAMPP on Mac) and open the About page to test your subtitles.

Summary

Native video and audio are some of the nicest new features of HTML5. In this chapter, you learned how to convert and embed video files into your web pages. You also learned about the available file formats and which browsers support them.

You put your JavaScript skills to the test in this chapter, creating custom controls for controlling the playback of the video you embedded on the About page of Joe's Pizza Co.'s website—learning all about the `HTMLMediaElement` interface in the process.

Finally, you took steps to make your video more accessible by adding subtitles using the new `<track>` element. This element can help people understand the video, even if they are hearing impaired or do not speak the language you are using for your website.

In Chapter 12, you learn how to store data on a user's computer by using the new client-side storage APIs introduced in HTML5.

chapter twelve

Storing Data

ALMOST ALL APPLICATIONS involve storing data of some kind. This can be data about users, products, videos, or even page visits. Used correctly, this data can be a powerful tool in the decision-making process. That's why companies like Google and Facebook try to collect as much data as they can about their users and their behaviors.

Web applications that handle a lot of data have traditionally used databases for storage. These databases are stored on servers that are often hundreds, if not thousands, of miles away from the user (hence the term *server-side* storage). An alternative option is *client-side* storage. Storing data on the client side means that it is stored on the user's computer or device—the *client*—as opposed to being stored on a web server. In this chapter, you learn about the strengths and weaknesses of using client-side storage. You explore two new technologies that HTML5 introduces for storing data on the client side and learn how these technologies can be used in modern web applications and websites to boost performance and make websites function offline.

More pros and cons of using client-side storage are discussed in the last portion of this chapter.

Why Use Client-Side Storage?

There are many reasons why you might want to store data on the client. In this section, I explore some of the most important, such as the following:

- **Performance**—With a traditional website, every time an application wants to fetch some data, it has to send a request to the server, wait for it to respond, and then download the response. This all takes time and can slow down the user's browsing experience. If you store data on the client, it can be accessed much more quickly because there is no need to make lengthy HTTP requests. Just to clarify: By lengthy, I mean 200 to 400 milliseconds (more on mobile devices). This might not sound like a lot, but it all adds up.
- **Offline access**—The capability to access your data offline is another reason why many developers prefer to keep data on the client side. This is especially prudent for web applications built for mobile phones; these devices are more likely to be without a stable Internet connection at some point in the day. If the data is stored on the device, it can be accessed without having to talk to a web server.
- **No cookies**—Before HTML5 came along, most client-side data was stored using *cookies,* small pieces of data stored in the browser. Cookies have several problems that make them unsuitable for storing a relatively large quantity of data. The first problem is that cookies are limited in size—to be specific, 4096 bytes. That's not much space for storing data. The other big drawback is that cookies are sent with every request your browser makes to the web server, whether needed or not. This increases the amount of time each request takes to complete, and therefore affects your application's performance.

HTML5 introduces two new APIs that you can use to store data on a client: LocalStorage and SessionStorage. In the next few sections, you learn how to use LocalStorage and SessionStorage to store data on a client.

Note that when I refer to client-side storage in this chapter, I am referring to LocalStorage and SessionStorage, not IndexedDB, which is also classed as a client-side storage API but is a little too advanced for this book.

LocalStorage

LocalStorage is a simple way of storing data using key/value pairs. The LocalStorage API consists of an object that provides several functions to store, access, and delete data from the client. You access the API using JavaScript.

All the data that your web application stores is placed in a datastore that is accessible only through your application. This means that the data your application saves cannot be accessed by other applications, and vice versa. Applications are identified by their domain names. If you have sub-domains, such as `a.example.com` and `b.example.com`, they are treated as separate applications and are each given their own datastore.

Using Key/Value Pairs

The concept of key/value pairs is explained in Chapter 5, which examines how form data is passed via parameters in a URL.

If you want to store a simple piece of data, such as a user's name, you create a key that describes the data (in this case, `name`) and then use the data (that is, a specific user's name) as that key's value. This mechanism is good for storing simple unstructured data; however; if you want to store data about multiple users at once, you might be better off utilizing a system that supports structured data, such as a database.

The technical term for assigning datastores to specific domains is *sandboxing*.

To start using LocalStorage, you first need to get familiar with the `localStorage` object and its associated functions. Fire up your developer tools console; you're going to need it in this section. After you work through the examples in the next few pages, you use what you learn here to add LocalStorage capabilities to the bookings form on the Joe's Pizza Co. website.

setItem(key, value)

The `setItem()` function has two parameters. The first parameter, `key`, is used to assign the name by which you access the data. The second, `value`, is where you put the data you want to save.

Try saving some data. Start by opening up a blank HTML file in your browser and naming it `test.html`. Next, fire up the console in your developer tools and enter the following code:

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

If all went well, that data should now be stored in the datastore that the browser created for your domain.

Next let's retrieve your data.

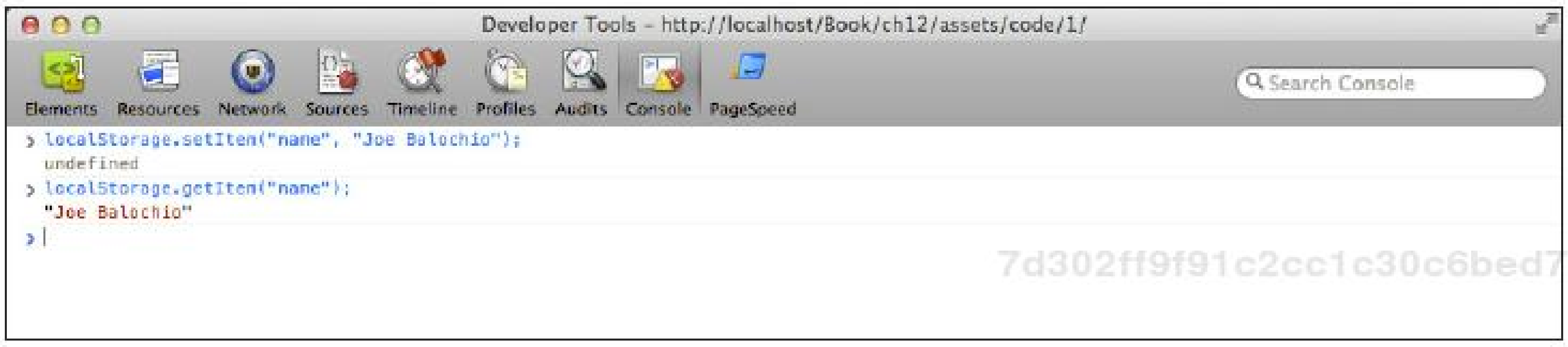
getItem(key)

The `getItem()` function has just one parameter, the `key` that is used to identify your data in the datastore.

Try retrieving the data you saved in the preceding example. Enter the following code into your console:

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

You should see that the data you stored is displayed in the console window, as shown in Figure 12-1.



```
> localStorage.setItem("name", "Joe Balochio");
  undefined
> localStorage.getItem("name");
  "Joe Balochio"
> |
```

FIGURE 12-1 Retrieving data in localStorage using the getItem() function.

The data that you store in LocalStorage is *persistent*. This means that if you close your browser window and then reopen it, the data is still there. Try closing your browser and then loading the same HTML file again. Can you still access the data?

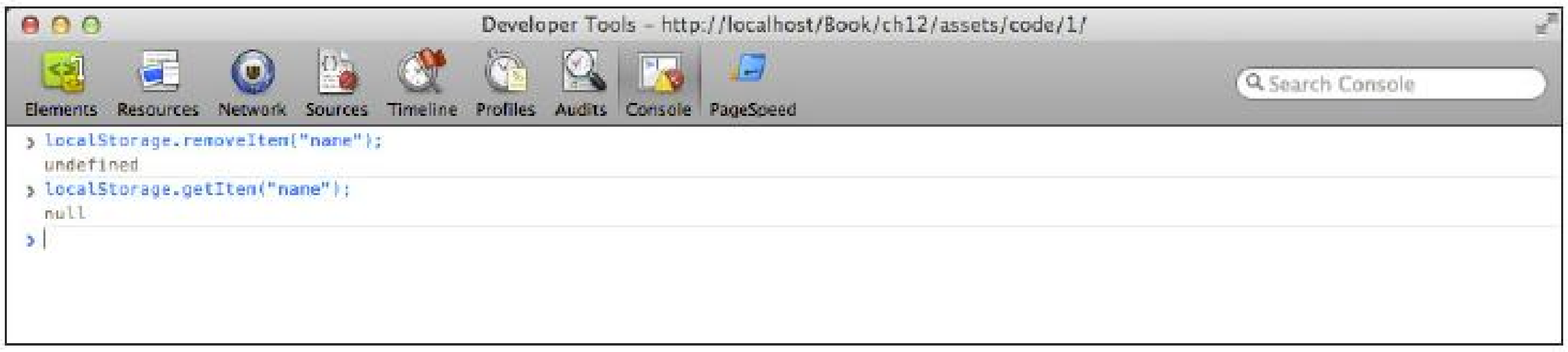
removeItem(key)

Naturally you need to be able to remove the data that you place in LocalStorage. This is done using the `removeItem()` function, passing in the `key` of the data you want to remove.

Try removing the data you saved in the previous example and then attempt to fetch it again. If all goes well, it should be gone.

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

You should see that the console returns `null` when you try to fetch the data. This is shown in Figure 12-2.



```
> localStorage.removeItem("name");
  undefined
> localStorage.getItem("name");
  null
> |
```

FIGURE 12-2 Removing data using the removeItem() function.

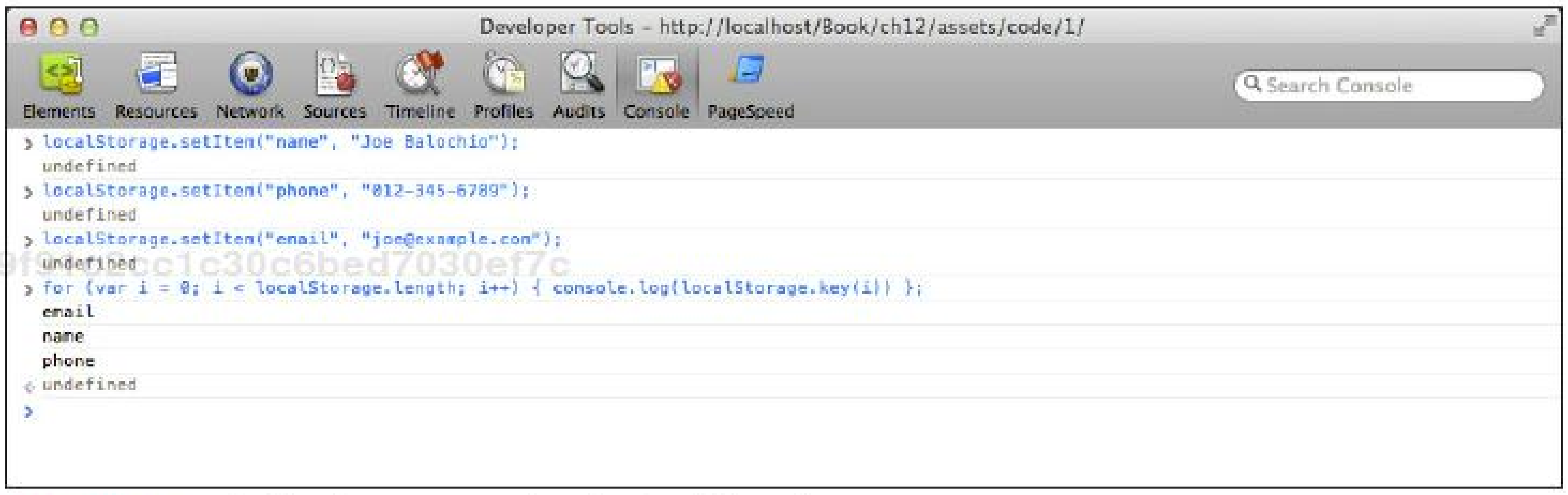
key(index)

The `localStorage` object also has a function you can use to retrieve the name of a key from the datastore. This function is called `key()`, and it takes an integer that represents the position of a key in the key index. The order in which keys are stored in this index varies between browsers. This means that you cannot really rely on this function for finding a specific key, but it does come in handy when trying to retrieve all of the data in LocalStorage; as you will see in the next example.

Try storing a few key/value pairs and then retrieving all the keys in LocalStorage. Enter each line in the code listing below into your console. You need to type the entire `for` loop into the console on one line.

```
// Store some data.
localStorage.setItem("name", "Joe Balochio");
localStorage.setItem("phone", "012-345-6789");
localStorage.setItem("email", "joe@example.com");

// Retrieve the keys for your data.
for (var i = 0; i < localStorage.length; i++) {
console.log(localStorage.key(i)) };
```

The `for` loop in this example will iterate through each piece of data stored in LocalStorage and print the keys to the console, as Figure 12-3 shows.



```
> localStorage.setItem("name", "Joe Balochio");
  undefined
> localStorage.setItem("phone", "012-345-6789");
  undefined
> localStorage.setItem("email", "joe@example.com");
  undefined
> for (var i = 0; i < localStorage.length; i++) { console.log(localStorage.key(i)) };
  email
  name
  phone
< undefined
>
```

FIGURE 12-3 Finding key names using the key() function.

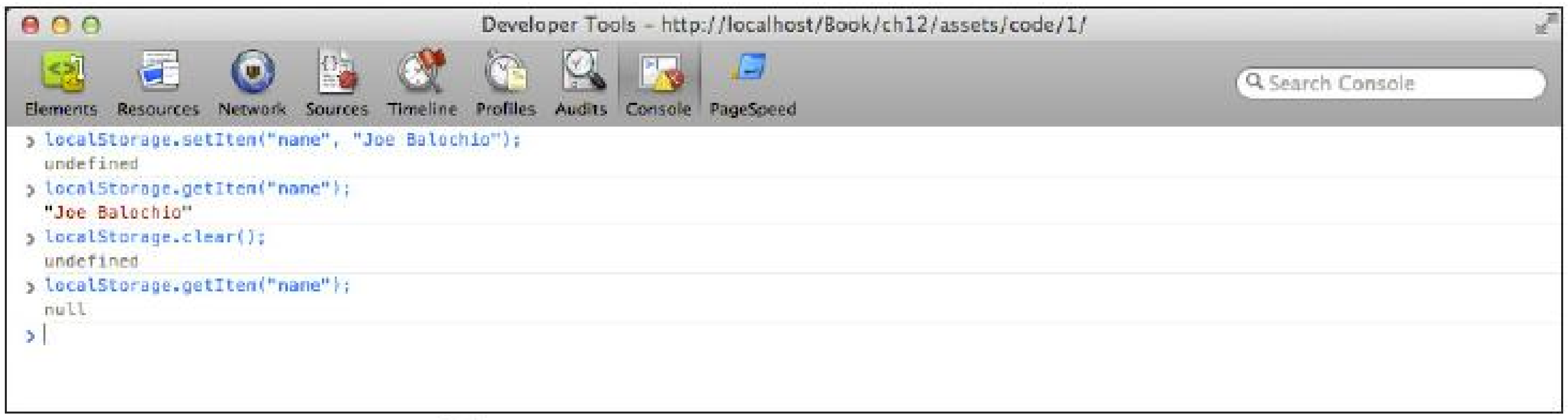
clear()

To clear all the data stored in your application's datastore, you can use the `clear()` function. This deletes every key/value pair associated with your application.

Try deleting all the data you have stored up to now. Enter the following into your console window:

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

Now try to retrieve some data. You should find that the `getItem()` function returns `null`, as shown in Figure 12-4.



```
> localStorage.setItem("name", "Joe Balochio");
  undefined
> localStorage.getItem("name");
  "Joe Balochio"
> localStorage.clear();
  undefined
> localStorage.getItem("name");
  null
> |
```

FIGURE 12-4 Using the clear() function to delete all the stored data.

length

You can find out how many key/value pairs have been stored by accessing the `length` property of the `localStorage` object.

Store some key/value pairs and access the length in your console.

```
localStorage.length;
```

Note that this is a property, not a function, so it has no parentheses or parameters.

Saving Customer Data from the Bookings Form

A use case that suits LocalStorage particularly well is saving the data that a user commonly inputs into a web form. In this section, you use the LocalStorage API to store data from the bookings form on the Joe's Pizza Co. website. You then add some JavaScript code that attempts to pre-load the form with this stored data when the page loads. Specifically, you are storing data from the Name, Phone, and Email fields of the form.

The code for this exercise can be found in the download code for Chapter 12, folder 1.

Here are the steps for saving user data:

1. Open the `bookings.html` file in your text editor.
2. Load the `storage.js` file (you create this shortly) into the Bookings page by adding a new `<script>` element just above the closing tag of the `<body>` element.

```
  <script src="js/storage.js"></script>
</body>
```

3. Locate the `<form>` element and add an `id` attribute to it with the value `bookingsForm`.
4. Save and close the `bookings.html` file.
5. Create a new file in your text editor called `storage.js` and save it in the `js` folder.

6. Attach an empty function to the `onload` event of the `window`, as you did for the `video.js` file in Chapter 11.

```
window.onload = function() {

}
```

7. Check that LocalStorage is supported by the user's web browser. To do this, create an `if` statement that looks for the presence of the `localStorage` object. Remember, lines that start with `//` are comments.

```
window.onload = function() {
  // Check for LocalStorage support.
  if (localStorage) {

  }
}
```

8. Create a new variable called `form` and initialize it by fetching the `<form>` element from your Bookings page using its ID.

```
window.onload = function() {
  // Check for LocalStorage support.
  if (localStorage) {
    // Get the form
    var form = document.getElementById("bookingsForm");
  }
}
```

9. Set up an event listener on the form that is executed when the form is submitted (or the `submit` event is fired).

```
// Get the form
var form = document.getElementById("bookingsForm");

// Event listener for when the bookings form is submitted.
form.addEventListener("submit", function(e) {

});
```

10. Inside the function block of this event listener, add a call to `saveData(form)`. You create this function in the next step.

```
form.addEventListener("submit", function(e) {
  saveData(form);
});
```

11. At the bottom of the `storage.js` file, create a new function called `saveData()`.

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

// Save the form data in LocalStorage.
function saveData() {

}
```

12. You now need to get the values from the Name, Phone, and Email `<input>` elements. Within the `saveData()` function, create three new variables (`name`, `phone`, and `email`), and initialize them by fetching the `<input>` elements using their IDs.

```
function saveData() {
  // Fetch the input elements.
  var name = document.getElementById("name");
  var phone = document.getElementById("phone");
  var email = document.getElementById("email");
}
```

13. Now that you have references to the `<input>` elements, you can get their values and store them in LocalStorage. Use the `setItem()` function to store the values.

```
function saveData() {
  // Fetch the input elements.
  var name = document.getElementById("name");
  var phone = document.getElementById("phone");
```

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

    // Store the values.
    localStorage.setItem("name", name.value);
    localStorage.setItem("phone", phone.value);
    localStorage.setItem("email", email.value);
  }
```

14. You have now added all the code needed to store a user's contact details when the form is submitted. Save the `storage.js` file and try submitting some data using the bookings form. If you go back to the Bookings page, you can use the developer tools console to check that data is being stored correctly. Use the `localStorage.getItem(key)` function you learned about earlier in this chapter to retrieve the data.

15. Write code to populate the form fields with the data in LocalStorage when the page loads. Create a new function called `populateForm()` below the `saveData()` function:

```
// Attempt to populate the form using data stored in
LocalStorage.
function populateForm() {

}
```

16. Copy the code that you used to fetch the form fields in your `saveData()` function to the new `populateForm()` function.

```
// Attempt to populate the form using data stored in
LocalStorage.
function populateForm() {
  // Fetch the input elements.
  var name = document.getElementById("name");
  var phone = document.getElementById("phone");
  var email = document.getElementById("email");
}
```

17. Set the `value` property of each of these three inputs using the data stored in LocalStorage. You can use the `getItem(key)` function to retrieve data from the datastore. If no contact data has been saved yet, the `getItem()` function returns `null`. Most browsers recognize that `null` is not the desired value and won't update the value of the input. However, Internet Explorer will populate the fields with the string `null`. This means that you need to first check that the result of `getItem()` is not null before updating an input's value. Use multiple `if` statements to do this.

```
// Attempt to populate the form using data stored in
LocalStorage.
function populateForm() {
    // Fetch the input elements.
    var name = document.getElementById("name");
    var phone = document.getElementById("phone");
    var email = document.getElementById("email");

    // Retrieve the saved data and update the values of the
    // form fields.
    if (localStorage.getItem("name") != null) {
      name.value = localStorage.getItem("name");
    }

    if (localStorage.getItem("phone") != null) {
      phone.value = localStorage.getItem("phone");
    }

    if (localStorage.getItem("email") != null) {
      email.value = localStorage.getItem("email");
    }
  }
```

18. To finish up, add a call to the `populateForm()` function at the top of the `window.onload` event listener.

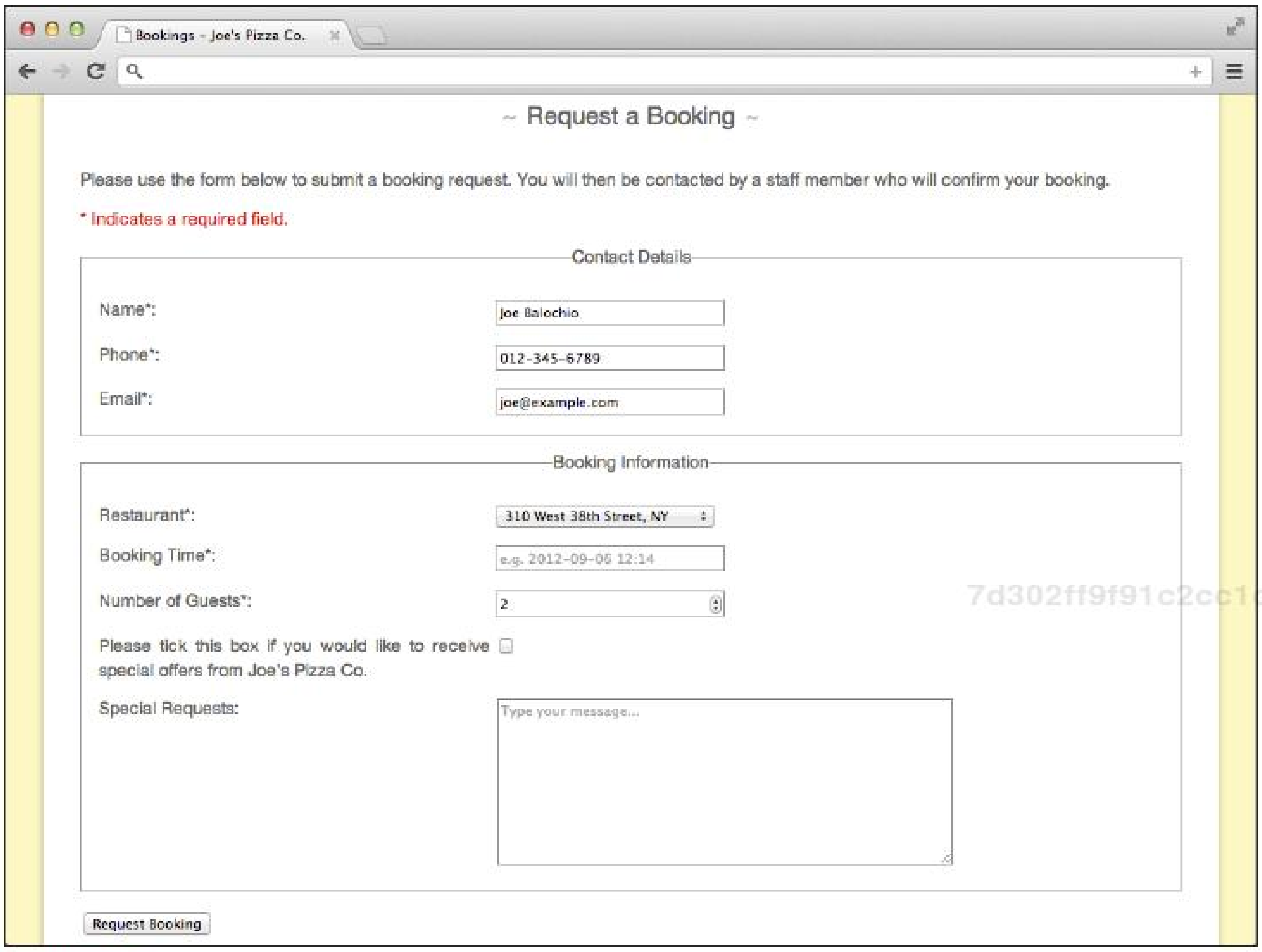
```
window.onload = function() {
  // Check for LocalStorage support
  if (localStorage) {
    // Populate the form fields
    populateForm(form);
    ...
  }
}
```

19. Save the `storage.js` file.

That's it! You have now successfully implemented LocalStorage into your website.

Try opening up the Bookings page and entering some data into the form. When you click the Request Booking button, the contact details will be saved. Go back to the Bookings page and you should see that the Name, Phone, and Email fields are already populated when the page loads. Figure 12-5 shows how this looks in your web browser.

Using the LocalStorage API to enhance your web forms in this way can be beneficial to users. In the next section, you learn how to store more complex data in LocalStorage, such as JavaScript objects and arrays.



~ Request a Booking ~

Please use the form below to submit a booking request. You will then be contacted by a staff member who will confirm your booking.

* Indicates a required field.

Contact Details

Name*:	Joe Balochio
Phone*:	012-345-6789
Email*:	joe@example.com

Booking Information

Restaurant*:	310 West 38th Street, NY
Booking Time*:	e.g. 2012-09-06 12:14
Number of Guests*:	2
Please tick this box if you would like to receive special offers from Joe's Pizza Co.	☐
Special Requests:	Type your message...

Request Booking

FIGURE 12-5 The Name, Phone, and Email fields are populated from data stored in LocalStorage.

Storing Objects and Arrays

Until now, you looked only at how to store text and numbers using LocalStorage. In this section, let's look at how you can store more complex data such as objects and arrays.

A key component in storing these types of data is the use of a data interchange format called JSON.

Introducing JSON

JavaScript Object Notation, or *JSON*, is a lightweight data interchange format used to transmit data over a network. It uses a human-readable, standardized syntax to define data objects that can be parsed by many computer programs.

Here is an example of how to define a simple JavaScript object in JSON.

```
{
    "name": "Joe Balochio",
    "age": "28",
    "gender": "male"
}
```

You define a new object in JSON using curly braces `{}` and then use key/value pairs to define the object's properties. A colon is used to separate a key from its value, and commas separate each of the properties in the object.

JSON objects can easily be serialized and parsed by libraries present in many programming languages. *Serialization* is the process of taking an object in your code and converting it to a JSON object, like the one in the example above. *Parsing* is the opposite; it takes a JSON object and converts it to an object that can be used in your JavaScript code.

The JSON Object

Many programming languages have built-in libraries that provide support for JSON. In JavaScript, you can use the JSON object to make use of the JSON library present in most browsers (excluding IE7 and below). This object has several available functions that you can use to help handle JSON data.

stringify(object)

The first function that we are going to look at is `stringify()`. This function converts an object (or objects) into a JSON object(s) and then returns that JSON as a string. Let's test this out.

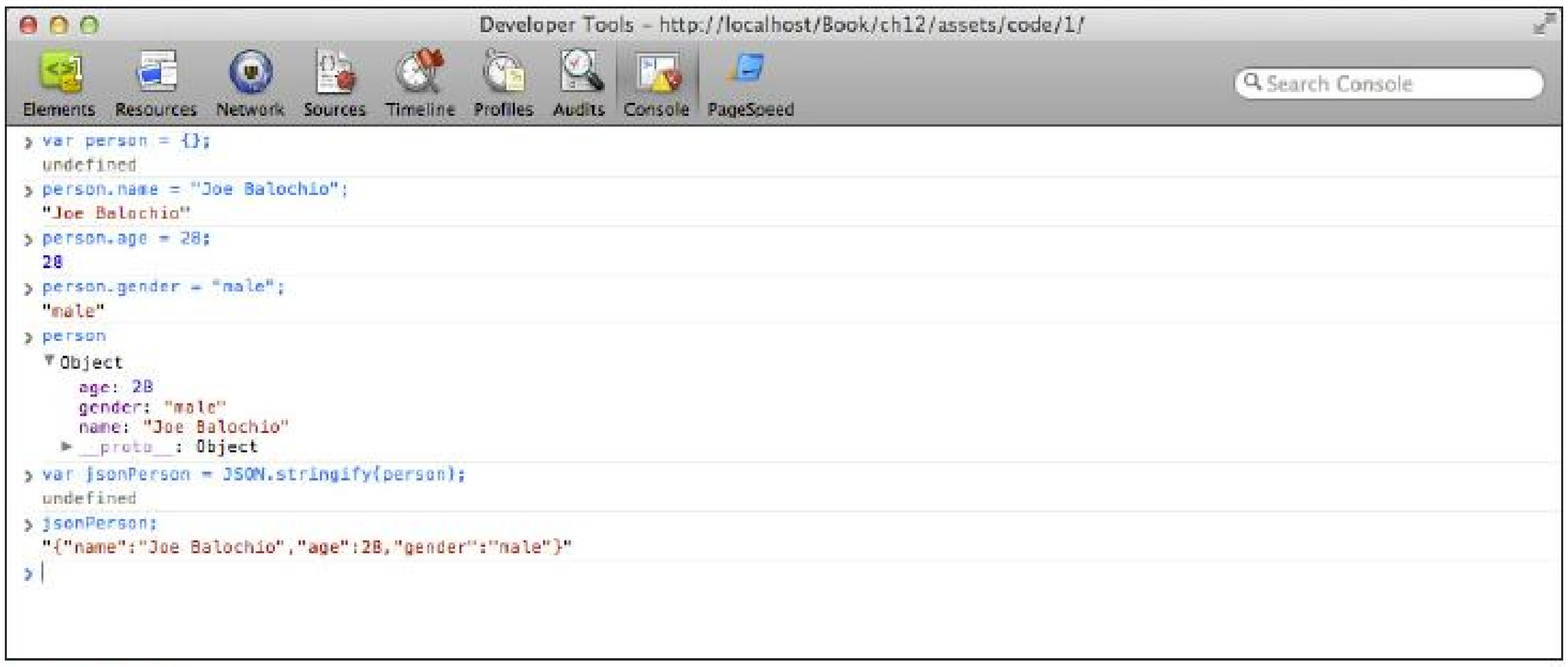
Open your console and create a new JavaScript object by entering the following statements:

```
var person = {};
person.name = "Joe Balochio";
person.age = 28;
person.gender = "male";
```

Now that you have created a JavaScript object, use the `stringify()` function to convert it to JSON.

```
var jsonPerson = JSON.stringify(person);
```

This converts the `person` object to JSON and saves the result to the `jsonPerson` variable. You can output the JSON by inspecting the `jsonPerson` variable in the console (just type `jsonPerson` and press Enter). Figure 12-6 shows the expected output for this example.



```
> var person = {};
  undefined
> person.name = "Joe Balochio";
  "Joe Balochio"
> person.age = 28;
  28
> person.gender = "male";
  "male"
> person
  ▼Object
      age: 28
      gender: "male"
      name: "Joe Balochio"
    ▶ __proto__: Object
> var jsonPerson = JSON.stringify(person);
  undefined
> jsonPerson;
  "{"name":"Joe Balochio","age":28,"gender":"male"}"
> |
```

FIGURE 12-6 Converting a JavaScript object to JSON.

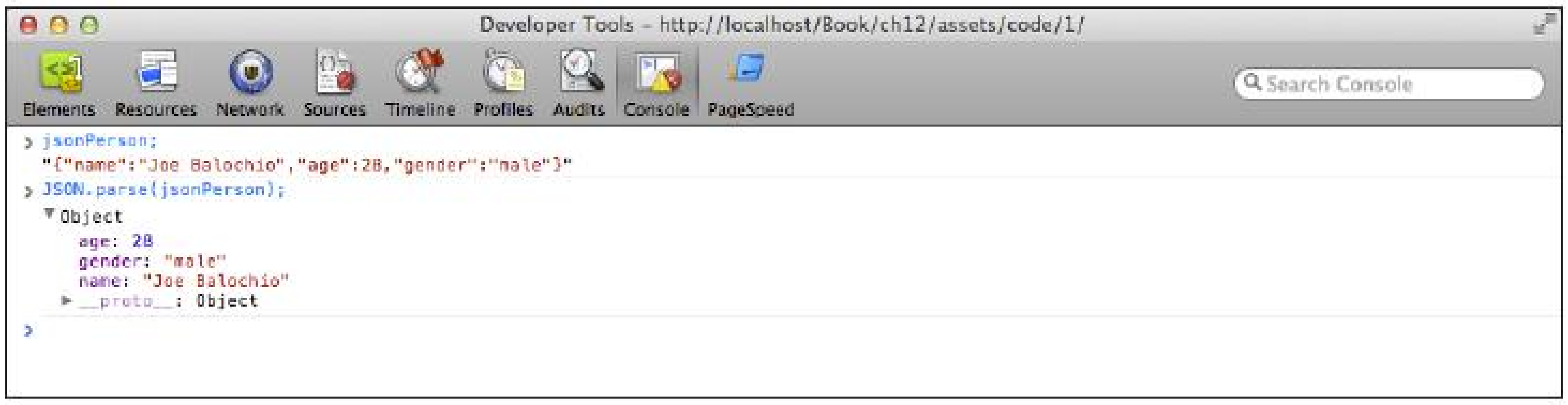
parse(json)

Now that you know how to convert JavaScript objects to JSON, let's look at how you can convert JSON back to JavaScript objects. This is done using the `parse()` function. Simply pass this function your JSON, and it will return your object(s).

Try converting the JSON you have stored in the `jsonPerson` variable back to a JavaScript object. Enter the following into your console.

```
JSON.parse(jsonPerson);
```

This outputs an object into the console. You could also initialize a new variable to hold this object if you so desired. Figure 12-7 shows the JavaScript object that has been parsed from the JSON.



```
> jsonPerson;
  "{"name":"Joe Balochio","age":28,"gender":"male"}"
> JSON.parse(jsonPerson);
  ▼Object
      age: 28
      gender: "male"
      name: "Joe Balochio"
    ▶ __proto__: Object
>
```

FIGURE 12-7 Converting a JSON string to a JavaScript object.

Objects in LocalStorage

More complex types of data such as JavaScript objects cannot be stored in the same way as text and numbers. If you tried to store a normal JavaScript object, it would be converted to a string and saved. The problem is that you cannot easily get this data back to an object once it has been saved. This is where JSON comes in. Using JSON, you can convert a JavaScript object to a JSON string and then save that. Then when you want to retrieve your object, you simply retrieve the JSON and parse it.

Storing Objects

Storing your objects using JSON is fairly simple. The following example shows how to do it. Fire up your console and give it a go.

```
// Create an object.
var myObject = {};
myObject.name = "book";
myObject.color = "green";
myObject.pages = 292;

// Convert the object to JSON.
var json = JSON.stringify(myObject);

// Store the JSON using localStorage.
localStorage.setItem("myObject", json);
```

In this example, you created a new JavaScript object, converted it to JSON, and then saved it on the client using LocalStorage.

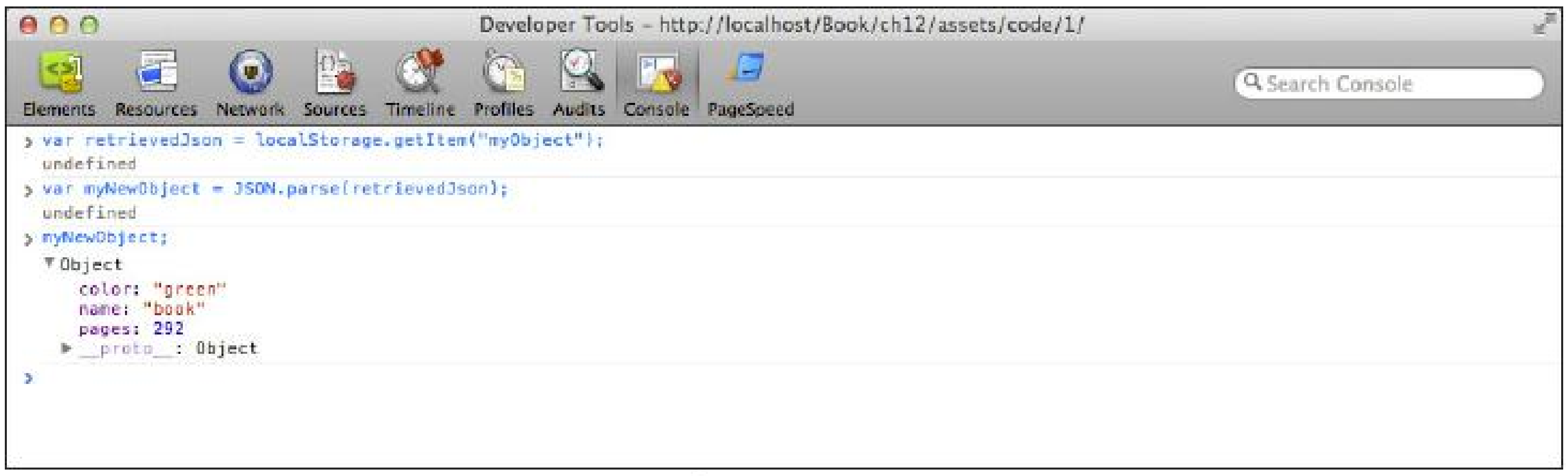
Retrieving Objects

So that's great: You have your JavaScript object converted to JSON and safely stored in LocalStorage. But what if you want to retrieve that object? This is where the `JSON.parse()` function comes in handy.

The example below shows how you can retrieve your object by retrieving the JSON from the datastore and converting it to a JavaScript object.

```
// Get the JSON from the datastore.
var retrievedJson = localStorage.getItem("myObject");

// Convert the JSON to a JavaScript Object.
var myNewObject = JSON.parse(retrievedJson);
```

Now, if you inspect the `myNewObject` variable in the console, you should see that your JavaScript object is back! This is shown in Figure 12-8.



```
> var retrievedJson = localStorage.getItem("myObject");
  undefined
> var myNewObject = JSON.parse(retrievedJson);
  undefined
> myNewObject;
  ▼Object
     color: "green"
     name: "book"
     pages: 292
   ▶ __proto__: Object
>
```

FIGURE 12-8 Retrieving and parsing an object from LocalStorage.

Arrays in LocalStorage

Arrays can be stored in the same way that objects are. First you must convert the array to JSON, and then you can store it as a string. To retrieve your array, you just need to parse the JSON that has been stored. The following example code shows how you could store and retrieve arrays using LocalStorage:

```
// Create an array.
var myArray = ["Mike", "Jim", "Becky", "Jo", "Steph"];

// Convert the array to JSON.
var myJsonArray = JSON.stringify(myArray);

// Store the JSON.
localStorage.setItem("myArray", myJsonArray);
```

. . . . and to retrieve the array:

```
// Retrieve the JSON.
var retrievedJson = localStorage.getItem("myArray");

// Parse the JSON.
var myNewArray = JSON.parse(retrievedJson);
```

Using JSON allows you to maintain your data structures when using a simple datastore such as LocalStorage. You should consider, however, whether you are using this facility correctly. LocalStorage is designed only to hold simple data. If you want to store multiple objects that have the same structure, you are better off using a database.

HTML5 does introduce a way of maintaining a client-side database through the IndexedDB API, which isn't covered in this book. If you're interested, I encourage you to look into it. You can find a great tutorial on how to use IndexedDB on HTML5Rocks at `http://www.html5rocks.com/en/tutorials/indexeddb/todo/`.

SessionStorage

As I mention earlier in this chapter, LocalStorage is persistent. The data you save stays there when you close your browser. This is exactly what you want for most applications; but there may be instances when you want to temporarily store some data, such as session IDs (what applications use to identify different users). It can be a pain to remember to remove all this from LocalStorage when the user leaves your page and ends the session.

SessionStorage was developed to solve this problem. This API has the exact same functions as LocalStorage (under the hood, it uses the same basic `Storage` object), but the data you save in SessionStorage is wiped by the browser when the user closes the window and ends his or her session on your website.

To use SessionStorage, you simply call your functions on the `sessionStorage` object instead of `localStorage`, as shown in the following examples:

```
sessionStorage.setItem("name", "Joe Balochio");
sessionStorage.getItem("name");
sessionStorage.key(0);
sessionStorage.removeItem("name");
sessionStorage.clear();
```

Additional Considerations on Using Client-Side Storage

The new client-side storage APIs introduced in HTML5 are truly wonderful things, but you should carefully consider whether using them adds value to your web application. For some applications, using client-side storage is a no-brainer; for others, you may be better off storing your data on a web server as people have done for the last few decades. The purpose of client-side storage is to enhance the user experience. Be careful that you don't become too eager to use it and end up diminishing it.

Browser vendors have already done a lot of work to help make client-side storage secure; however, it is important that you understand what has and has not been taken care of for you, so that you can build secure applications. In the following sections, you explore some of the factors to consider when using client-side storage in your applications.

Storage Limits

How much data will your application require? With client-side storage, all that data will be taking up valuable space on the user's hard drive, and you don't want to allow websites to store gigabytes of data and fill up your user's drive.

To prevent this problem, browser vendors have imposed a limit on the amount of storage space that each application can use. The official specification outlines that 5 megabytes is an acceptable level of storage space; ultimately, however, it is up to the browser vendor to decide how much storage space to make available. In reality, browsers currently allow between 2.5 and 5 megabytes of data to be stored. You have to think worst-case in these sorts of scenarios, so that means you can only rely on 2.5 megabytes being available. Design your applications with this in mind.

You can use the Web Storage Support Test tool to track the amount of storage available in various browsers, as well as test the amount available in the one you are using. You can find this tool at `http://dev-test.nemikor.com/web-storage/support-test/`.

User Tracking

User tracking refers to the practice of tracking what a user is doing on a website (for example, the pages and products the person views). This data is often used by a third party to carry out targeted advertising. This practice is commonplace today and is achieved by placing a unique identifier in a cookie that can then be used to track a user as he or she navigates the website.

However, some privacy concerns surround this sort of activity. User tracking can also be carried out using the LocalStorage API. Learning from the lessons of the past, the standards organizations have begun to think about how this sort of use case can be handled in a way that is more respectful of the user's privacy.

The WebStorage specification published by the W3C makes recommendations that browser vendors should follow to help protect the privacy of their users. These include making it easier for users to delete data that has been stored on the client and allowing them to blacklist websites that they do not want to have access to client-side storage. The W3C is also considering the suggestion that these blacklists should be combined so that all the browser vendors can see which sites are abusing client-side storage and take necessary precautions to protect their users.

Sensitive Data

Client-side storage is a big bonus for web applications that need to work offline and boost performance. Many of these applications, however, store personal information about users and therefore you should carefully consider how secure this data is on the client.

The job of securing data once it has been saved falls more in the realm of browser vendors because as a developer, you don't really have much control over what's "under the hood." One of the best things that browser vendors can do, and in most cases are doing, is to ensure that when you delete data from LocalStorage, it is also promptly deleted from the underlying system storage (such as the hard drive).

Cross-Directory Attacks

Assigning datastores to individual domains does wonders to stop websites from accessing each other's data. It falls down, however, when it comes to websites like the now deceased geocities.com that allowed people to create their own customized pages. All these pages were under the same domain name and, therefore, would have shared the same datastore on the client. This means that everyone could access, change, or delete the data stored by everybody else's pages. This is known as a *cross-directory attack*. At the moment, there is no way to protect against it. It is recommended that you don't use client-side storage if you are building an application that enables users to create customized web pages under the same domain.

Summary

As HTML5 matures and more companies come on board, we are going to see more and more web applications leverage both the performance boost and added offline capabilities that the new storage APIs provide. Google, for example, has already taken advantage of these APIs to build a version of Gmail that works offline.

In this chapter, you learned how to use the new storage APIs in order to save data on the client-side rather than sending it up to a server to be stored in a database. You updated the Joe's Pizza Co. website, adding some JavaScript code that saves users' contact details when they first submit the bookings form and then automatically populates the Name, Phone, and Email fields for them when they use the form in the future.

You also explored some of the privacy and security concerns surrounding client-side storage and learned what browser vendors have done to make client-side storage more secure.

Chapter 13 introduces you to the GeoLocation API. Although this is not strictly part of HTML5, it's awesome—and so I have included it in this book. You learn all about how to use the API to pinpoint your user's location and how you can use the information to personalize your websites.