Introduction to jQuery

Lesson 1: **O'Reilly School of Technology, CodeRunner, and jQuery**

- Introduction to jQuery
  - Linking to the jQuery Library
  - Comparing jQuery and JavaScript
  - Why Use jQuery
  - jQuery Features
  - The jQuery.com Web Site

  Quiz 1 Project 1

Lesson 2: **Selecting and Manipulating Elements**

- Selecting and Manipulating Elements
  - Selecting Elements by Tag Name
  - Descendant Selectors
  - Class Selectors
  - jQuery selectors: Going Beyond the Basics
  - jQuery Objects and JavaScript Objects
  - jQuery Methods and Chaining
  - Manipulating Element Style with css(), addClass(), and removeClass()

  Quiz 1 Project 1

Lesson 3: **Filtering Elements**

- Filtering Elements
  - Filtering jQuery Selector Results
  - The each() Method
  - Filtering Form Elements
  - Selecting Elements by Content
  - Checking and Unchecking Form Elements with jQuery
  - Adding New Elements to Your Page

  Quiz 1 Project 1

Lesson 4: **Manipulating Element Style**

- Using jQuery to Style your Elements
  - Getting Started with the css() Method
  - Viewing Your Style with the Web Inspector Developer Tool
  - The Box Model
  - Units of Measurement and jQuery css()
  - Setting Multiple CSS Properties with css()
  - Sizing Elements with jQuery
  - Adding, Removing, Querying, and Toggling Classes
  - Showing and Hiding Elements with jQuery

  Quiz 1 Project 1

Lesson 5: **Looping, Functions, and This**

- Looping, Functions, and This
  - A Deeper Dive into each()
  - Comparing each() to a JavaScript Loop
  - Using the each() Function Parameters
  - Comparing Wrapped and Unwrapped Objects
  - Pay Attention to This

O'Reilly School of Technology, CodeRunner, and jQuery

Welcome to the O'Reilly School of Technology's (OST) Introduction to jQuery course.

Course Objectives

When you complete this course, you will be able to:

- demonstrate an ability to use jQuery.
- use jQuery to build applications.
- create an interactive map using jQuery.

Lesson Objectives

In this lesson you will:

- read about the O'Reilly School of Technology's *UserActive* approach to learning.
- navigate in the CodeRunner sandbox environment.
- log into and out of the Linux learning environment.

Before we begin, you need to learn a little about the programming environment you'll be using at the O'Reilly School. Everything you'll need is available through a web browser and you won't need to download or install anything to complete the course. There is an optional section in a later lesson that shows you how to download and use the jQuery libraries locally instead of linking to them; if you want to you can try it on your own computer, or on the OST computers. We'll provide all the instructions you need to complete that optional section of the course.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!
- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.
- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.
- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is

problem solving. Of course, you can always contact your instructor for hints when you need them.

- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.
- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

```
White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove will look like this.

We may also include instructive comments that you don't need to type.
```

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

```
The plain black text that we present in these INTERACTIVE boxes is
provided by the system (not for you to type). The commands we want you to type look lik
e this.
```

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

```
Gray "Observe" boxes like this contain information (usually code specifics) for you to
observe.
```

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

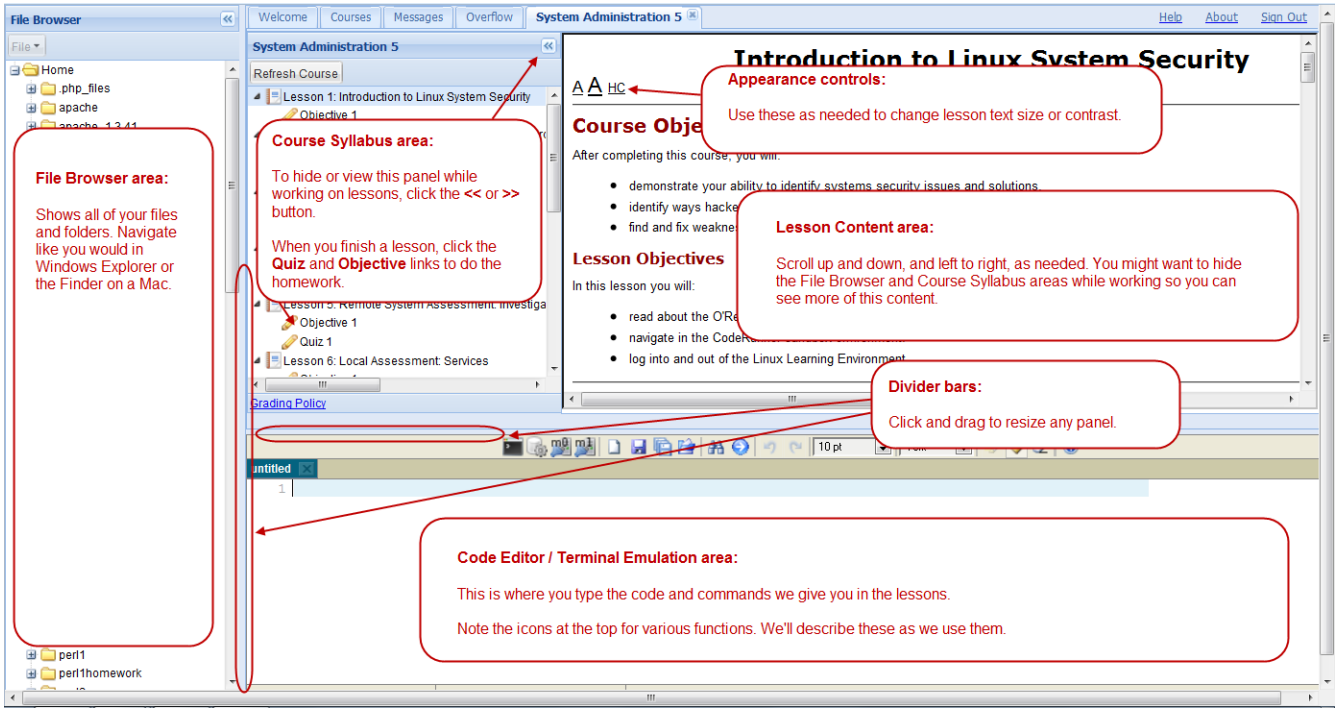
Note Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

Tip Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

WARNING Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:



These videos explain how to use CodeRunner:

File Management Demo

Code Editor Demo

Coursework Demo

Introduction to jQuery

In this course, you'll learn how to use one of the most popular JavaScript libraries on the web: jQuery. jQuery is actually JavaScript. Well to be more specific, JavaScript is a language, and jQuery is a framework built with JavaScript to help JavaScript programmers execute common web tasks. jQuery might look a bit weird to you at first, but everything you'll do in the course is JavaScript. jQuery helps to make some of tasks you do frequently in JavaScript less complicated, like getting elements from the DOM, styling and animating elements, and more. Throughout the course, we'll build a variety of small applications, leading up to a final project where we bring everything together into a fun web application that uses jQuery to create an interactive map that displays data. Let's get started right away by building your first jQuery application.

Linking to the jQuery Library

First, create a jQuery folder for your work. In the File Browser, right-click the **Home** folder, select **New folder...** or press **Ctrl+n**, type the new folder name **jQuery**, and press **Enter**.

The jQuery library is a file that contains JavaScript code. You can either download this file to your own computer (or web server) and link to that file in your code, or you can link to a version that's available at jQuery.com. We'll take the second option so you don't have to download anything now. We'll go through the steps of how to download jQuery (and another library, jQuery UI) in a later lesson though, so you'll know how to do it. Let's get started by creating a new HTML file, and adding this code:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>First jQuery</title>
  <meta charset="utf-8">
  <style>
    div {
      position: relative;
      width: 200px;
      height: 200px;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>
    $(document).ready(function() {
        $("div#hello").html("Hello, world!!");
        $("div#hello").css("background-color", "lightyellow");
    });
  </script>
</head>
<body>
    <h1>First jQuery</h1>
    <div id="hello">
    </div>
</body>
</html>
```

Save this as **first.html** in your **/jQuery** folder.

Preview Preview the file; you see a web page that looks like this:

> **Tip** If you find the light yellow background difficult to see, try a different color.

Let's go through the code, piece by piece. As we build our code, you'll be introduced to concepts that are used in jQuery often, so let's make sure you grasp them fully. If you don't understand them right away though, don't panic! We'll return to them often and you'll have a chance to work through them until you're confident in your knowledge.

To start, we link to the jQuery library in the line:

OBSERVE:

```
<script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
```

Go to that URL, **http://code.jquery.com/jquery-1.8.3.min.js**. You see the JavaScript code that makes up the jQuery library. It's not very pretty, because it's been **minified**—a process by which you can squeeze JavaScript code down to be as small as possible, by removing the white space and changing the names of variables to be as short as possible. This makes the code extremely difficult to read, but allows it to load a lot faster!

Linking to the jQuery library is just like linking to your own code: the browser reads in the JavaScript when it sees the <script> link, and interprets the code right away. That means if you make any references to jQuery objects and methods *below* the link to the jQuery library, the browser will know what they are.

> **Note** You can see the unminified version of jQuery at http://code.jquery.com/jquery-1.8.3.js.

In the file we just created, we also have our own code (below the link to the jQuery library). This code makes use of jQuery, so it's important that we put the link to the jQuery library first.

OBSERVE:

```
$(document).ready(function() {
   $("div#hello").html("Hello, world!!");
   $("div#hello").css("background-color", "lightyellow");
});
```

Start at the top left and take a closer look at the first line:

$** is the main jQuery function. In fact, **$ is a short name for the function, **jQuery()**, so you could write that as jQuery(document).... However, because we use it so often (just about every line of our jQuery code uses this function), the jQuery library defines **$** to be another name for the main jQuery function, which saves us some typing! It takes a little getting used to because we don't usually see $ used as a variable or function name, but here, it really is a function name.

$(document) says, "call the main jQuery function and pass the **document** object to it." Now, **$** is a multi-purpose function, so you'll see it used in a couple of different ways, but by far the most common is to access elements from the DOM or access jQuery objects. In this case **$(document)** returns a jQuery object that represents the document (the page we're in). This is not exactly the same as the JavaScript document object, which also represents the document (the page we're in), so don't confused the two! We'll talk more later about the difference between a jQuery object and a JavaScript object.

$(document).ready(...); is an example of *chaining*. Here, we've got two function calls that are *chained* together; first, the call to get the jQuery document object: **$(document)**, and then the call to **ready(...)**. (Don't worry right now about the argument to **ready(...)**; we'll get to that shortly.) So, we take the object returned by **$(document)** and then call its **ready(...)** method. We could split this into two separate statements like this:

OBSERVE:

```
var do = $(document);
do.ready(...);
```

No one ever does that in jQuery though; instead we chain them together into one statement to make the code more compact and easier to read. To chain the two functions together, place a **.** between them.

Finally, we call **ready()**, and pass in one argument that happens to be a function. The **ready()** method is used in a similar way as we use **window.onload** in JavaScript. Just like **window.onload**, **ready()** is used to define a function that is run when the document is **ready** to be used by jQuery (and JavaScript). In this case **ready** means that the DOM is loaded, so your code can do things like get elements from the DOM, and update the page by modifying those elements or adding new ones. Keep in mind that you don't want to use *both* jQuery **ready()** and **window.onload**. Pick one and stick with it! We'll be using **$(document).ready(...)** throughout the rest of this course.

Now let's take a closer look at the function that we pass into **ready()**:

OBSERVE:

```
$(document).ready(function() {
   $("div#hello").html("Hello, world!!");
   $("div#hello").css("background-color", "lightyellow");
});
```

We're passing an *anonymous function* to ready() which we call the "ready function." Rather than declaring a function and giving it a name, and then using that name as the value we pass to **ready()**, like this:

OBSERVE:

```
$(document).ready(imReady);

function imReady() {

    $("div#hello").html("Hello, world!!");
    $("div#hello").css("background-color", "lightyellow");
}
```

...we skip the declaration and pass the function value instead. You could do it either way, but typically, jQuery programmers use an anonymous function, so get used to seeing the ready function written this way.

In the function itself, we use the main jQuery function, **$**, to access objects; in this case, we use it to access the **element object representing the with the id "hello"**. If you look in the HTML, you'll see this . It's currently empty; it has no content.

In our ready function, we use the **html()** method to set the content of the **"hello" ** to **"Hello world!!"**. **html()** is a convenience method that jQuery includes as part of the jQuery element object.

Again, we chain method calls here. First we call the **$("div#hello")**, which returns an element object (a **jQuery element object**, which is not quite the same as a JavaScript element object), and then we call **html("Hello, world!!")**, which is a method of the jQuery element object. In our code, we use the jQuery object, **$**, to get the **"hello" element**, and call the **html() method** to set that element's content to "Hello world!!".

Knowing that that's how you read that particular line of code, can you read the next one?

We'd read it like this: We use the jQuery object, **$**, to get the **"hello" element**, and call the **css() method** to set the background-color CSS property to "lightyellow".

We end the function with the typical curly bracket: }. Don't forget to add a parentesis) to close the **ready()** method call! This is a common mistake programmers make when they're new to jQuery. You'll be passing a lot of functions around, like we do here, so get in the habit of writing "});" whenever you pass a function to a method call. If you forget the closing parenthesis, your JavaScript won't work, and you'll need to access the error console to debug it; the console indicates the line number where you've left off the closing parenthesis.

Because jQuery is JavaScript, anything you can do in jQuery you can also do in JavaScript. jQuery just makes it a little bit more efficient. Can you think of how you'd write the code above in JavaScript? Think about that for a few minutes and see if you can write that code before you look at the answer below.

Comparing jQuery and JavaScript

As we just mentioned, anything you can do in jQuery you can do in JavaScript. Let's compare the jQuery code we just wrote to set the content and background color of a , to that same code written in JavaScript. Here's the jQuery version:

OBSERVE:

```
$(document).ready(function() {
    $("div#hello").html("Hello, world!!");
    $("div#hello").css("background-color", "lightyellow");
});
```

Here's the JavaScript:

OBSERVE:

```
window.onload = function() {
    var div = document.getElementById("hello");
    div.innerHTML = "Hello, world!!";
    div.style.backgroundColor = "lightyellow";
}
```

In this case, it's almost the same number of lines of code, but you'll find that as you write more complex programs, using jQuery will save you some typing and reduce the size of your code. Compare the two ways of writing this code. Can you see the similarities?

Why Use jQuery

When you first get started with jQuery, it takes a while to get used to the syntax; jQuery looks a little funny at first glance compared to JavaScript.

The ultimate goal of jQuery is to make things you do often in JavaScript just a bit easier; to simplify common tasks. Here's a quote from the jQuery.com website:

jQuery is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Ajax interactions for rapid web development.

Once you begin writing more complex jQuery code, you'll find the jQuery convenience methods extremely handy. You'll also find that many of the things you do using jQuery are much simpler than their equivalents in JavaScript. That's a good thing!

In addition, jQuery is one of the most popular JavaScript libraries in the world! So knowing how to read and write jQuery code will be a huge benefit in your career as a web developer.

jQuery Features

Let's take a closer look at what the library offers.

Think of jQuery as a collection of features that fall into these categories:

Core Functions	The core jQuery functions and some utility methods.
Selection and Traversal	This category of features is the "query" in jQuery. These methods help you navigate through the DOM, finding elements and content.
Manipulation and CSS	Allows you to manipulate and style elements in your page.
Events	Simplifies working with events and makes it easier to create interactive pages.
Effects	Includes some basic effects, like hiding and showing elements, and basic animation of elements.
Ajax	Simplifies working with Ajax and XMLHttpRequest, with utilities for getting data and working with data, including JSON.
User Interface (UI)	Provides a separate library of interface widgets that you can plug in to your page; for example, sliders, progress bars, menus, and such.
Extensibility	Enables you to create jQuery plug-ins to add onto the base jQuery library. (This category is beyond the scope of this course.)

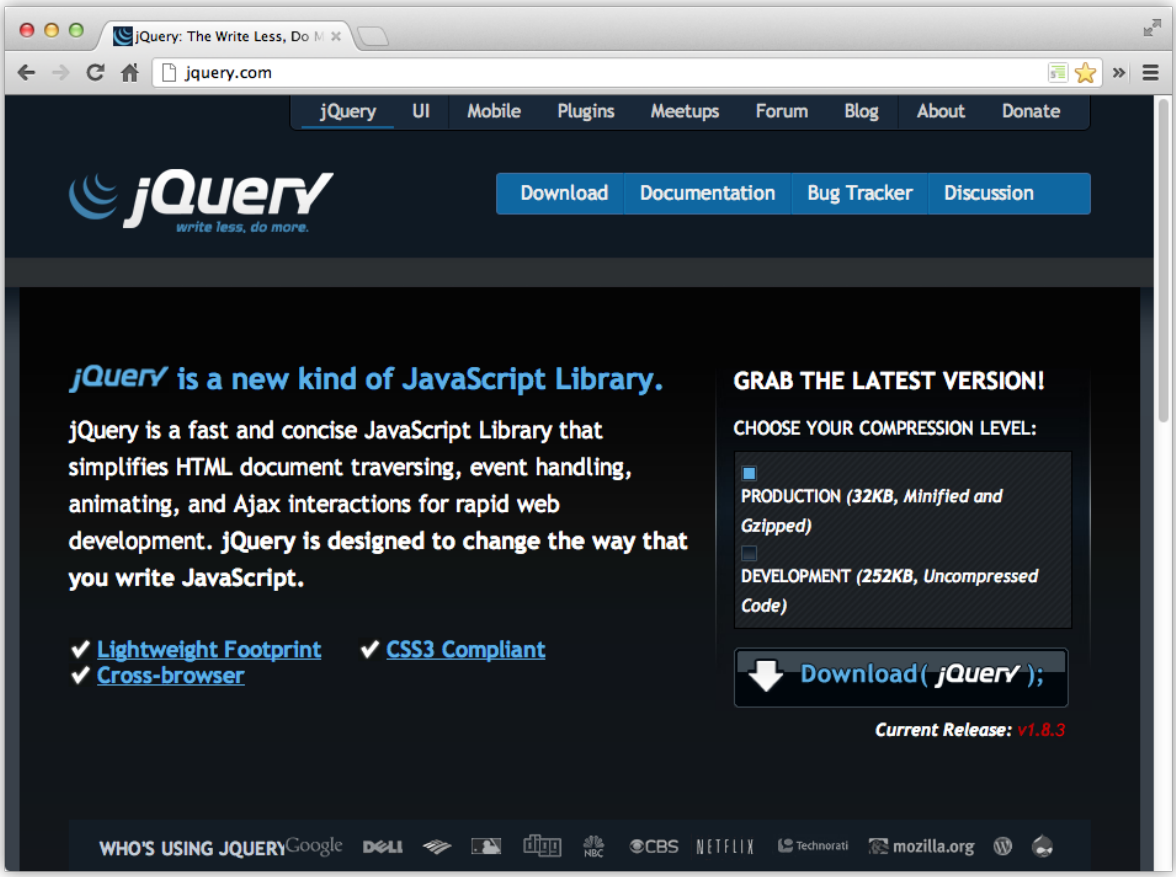
You'll get to try out many of the methods that jQuery offers through this course, but there are many many more methods that we won't have time to cover. So as part of the quizzes and projects, you might

occasionally be asked to go to the jQuery.com website, look at the documentation to get information on a method we haven't used yet, and try it out. Once you get the hang of jQuery, this will help you learn new methods and trying new things on your own.

The jQuery.com Web Site

Familiarize yourself with—and probably bookmark— the jQuery.com website. It includes extensive reference documentation, which you'll use in this course, as well as a few tutorials and links to various articles about jQuery. You'll also want to be aware of which version of jQuery you're using in case code you write in the future depends on a specific version.

The version of jQuery that we're using in this course is 1.8.3, which is the most current version as of this writing. However, the authors of jQuery update the library frequently, so by the time you read this, the current version might be different. The jQuery.com website will have information about the current version, as well as any big changes between versions so you'll know if it is safe for you to upgrade your code.



Take some time to practice jQuery and do the project before moving on to the next lesson.

Selecting and Manipulating Elements

Lesson Objectives

You will select elements using jQuery, using a variety of selectors.

Selecting and Manipulating Elements

Now that you know some basics about jQuery, let's explore some jQuery features that will help you to select and manipulate elements in your page. We'll also take a closer look at the kind of object you get when you select an element (or group of elements) using these selection methods, and how that relates to the DOM element objects you're used to working with in JavaScript. By the end of this lesson, you'll know how to select elements from the page using a variety of different methods, and you'll also know how to convert jQuery objects to DOM objects and back again.

Selecting Elements by Tag Name

In this lesson, we'll work with a page that contains a variety of basic elements: some headings (<h1>, <h2>, <h3>, etc.), a list (<ul> and <li>), some paragraphs of text (<p>), and links (<a>). Create a new file and type in this HTML (feel free to copy and paste the text—this time—to save yourself some typing):

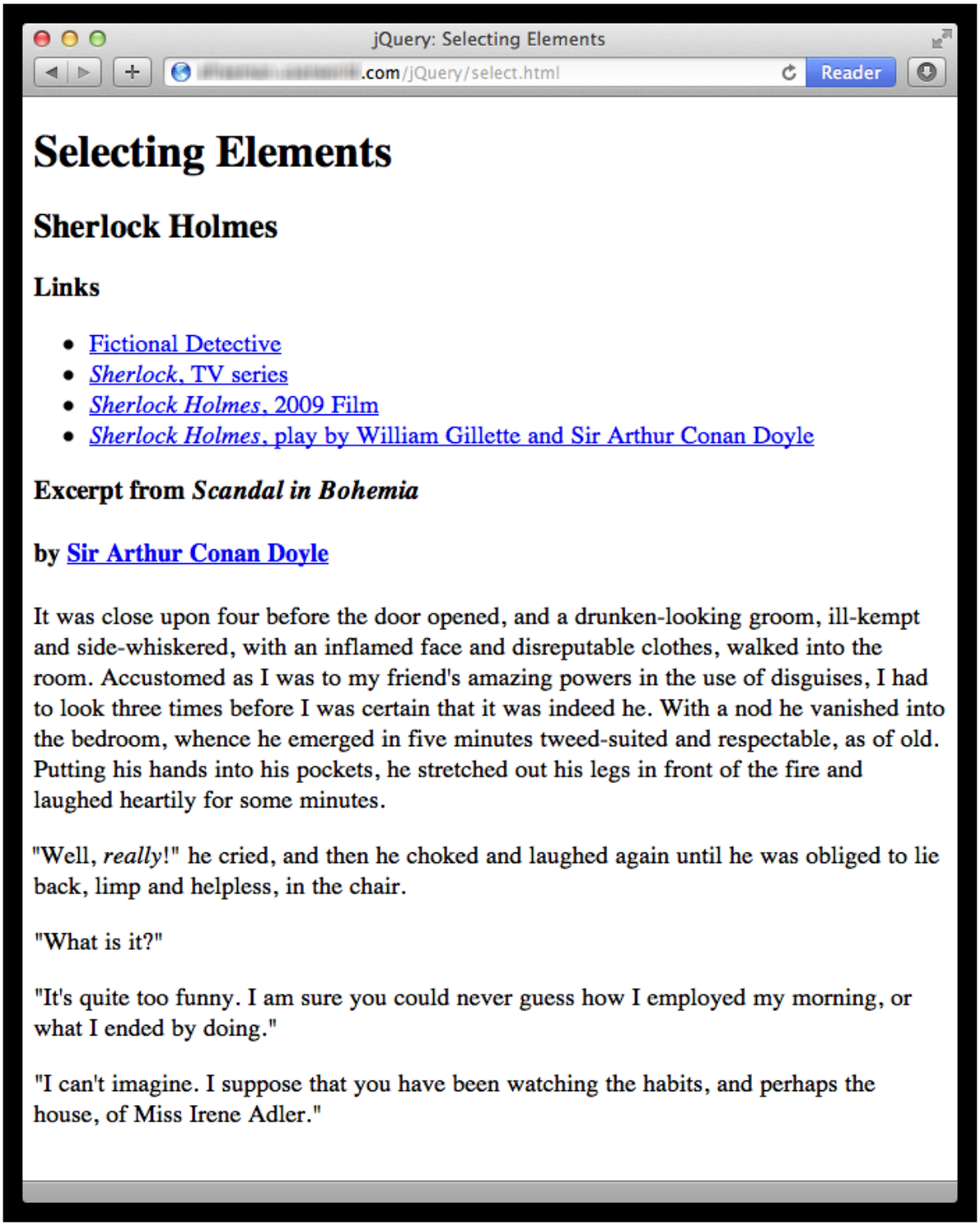
CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Selecting Elements</title>
  <meta charset="utf-8">
  <style>
    .highlight {
        background-color: yellow;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>

  </script>
</head>
<body>
  <h1>Selecting Elements</h1>
  <div>
      <h2>Sherlock Holmes</h2>
      <h3>Links</h3>
      <ul id="links">
        <li class="first"><a href="http://en.wikipedia.org/wiki/Sherlock_holmes"
>Fictional Detective</a></li>
        <li><a href="http://en.wikipedia.org/wiki/Sherlock_(TV_series)"><em>Sher
lock</em>, TV series</a></li>
        <li><a href="http://en.wikipedia.org/wiki/Sherlock_Holmes_(2009_film)"><
em>Sherlock Holmes</em>, 2009 Film</a></li>
        <li><a href="http://en.wikipedia.org/wiki/Sherlock_Holmes_(play)"><em>Sh
erlock Holmes</em>, play by William Gillette and Sir Arthur Conan Doyle</a></li>
      </ul>

      <h3>Excerpt from <em>Scandal in Bohemia</em></h3>
      <h4>by <a href="http://en.wikipedia.org/wiki/Sir_Arthur_Conan_Doyle">Sir A
rthur Conan Doyle</a></h4>
      <p class="first">
        It was close upon four before the door opened, and a drunken-looking gro
om,
        ill-kempt and side-whiskered, with an inflamed face and disreputable clo
thes,
        walked into the room. Accustomed as I was to my friend's amazing powers
in
        the use of disguises, I had to look three times before I was certain tha
t
        it was indeed he. With a nod he vanished into the bedroom, whence he eme
rged
        in five minutes tweed-suited and respectable, as of old. Putting his han
ds
        into his pockets, he stretched out his legs in front of the fire and lau
ghed
        heartily for some minutes.
      </p>
      <p>
        "Well, <em>really</em>!" he cried, and then he choked and laughed again
until he
        was obliged to lie back, limp and helpless, in the chair.
      </p>
      <p>
        "What is it?"
      </p>
      <p>
        "It's quite too funny. I am sure you could never guess how I employed
        my morning, or what I ended by doing."
      </p>
      <p>
        "I can't imagine. I suppose that you have been watching the habits,
        and perhaps the house, of Miss Irene Adler."
```

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

Save this file as **select.html** in your **/jQuery** folder.

Preview Preview the file:



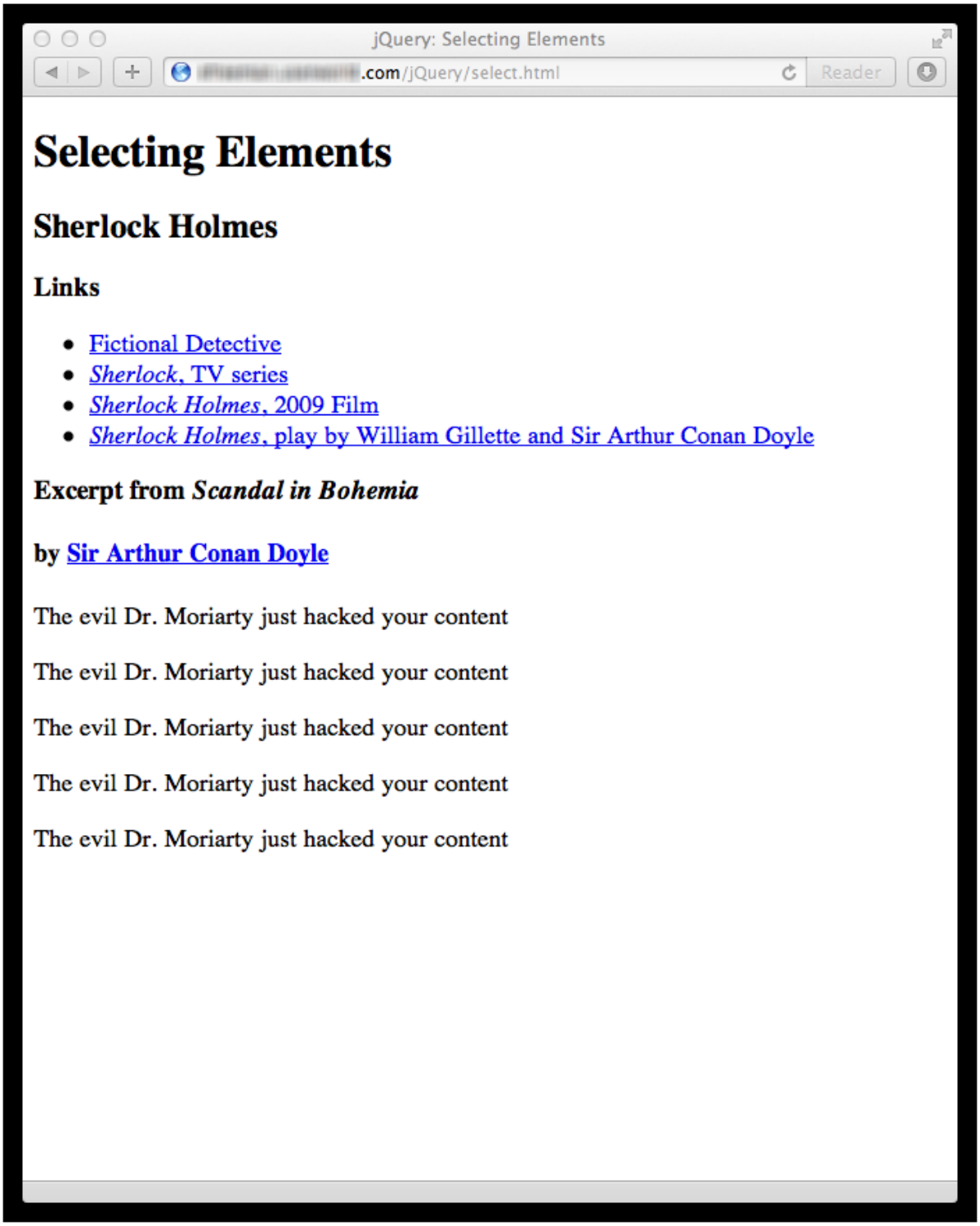
Notice the empty <script> element in the <head> of the document. That's where you'll add your jQuery. We're going to write a jQuery program to select and manipulate various elements from the page, using different kinds of queries.

Update your **select.html** to add the jQuery script below in the <script> element in the page:

CODE TO TYPE:

```
<script>
  $(document).ready(function() {
    var $result = $("p");
    $result.text("The evil Dr. Moriarty just hacked your content");
  });
</script>
```

Save the file and Preview preview again:



Just like we did in our previous example, we set up the jQuery **ready** function by calling the **ready()** method on the jQuery document object. Our ready function selects every <p> element in the page with **$("p")** and the result of selecting those elements is stored in the variable **$result**. Then we use the jQuery **text()** method to update the text content of each of those <p> elements to "The evil Dr. Moriarty just hacked your content."

We use the **text()** method here to update the *text content* of the <p> elements. This is slightly different from using the **html()** method like we did in the previous example. **html()** allows you to add content with HTML to your page, whereas **text()** allows you to manipulate only the text content of an element. We'll come back to discuss this distinction further, in a later lesson.

jQuery function **$()** can be used to select one or multiple elements in the page. Try changing your code to select the <h2> element instead (there's only one <h2> in the page):

CODE TO TYPE:

```
<script>
  $(document).ready(function() {
    var $result = $("~~p~~h2");
    $result.text("The evil Dr. Moriarty just hacked your content");
  });
</script>
```

 Save the file and Preview preview. Now the second-level heading has changed. Here, we selected only one element, but the rest of the code is exactly the same!

When you select elements with **$()**, you get back a jQuery object that acts just like an array of results, whether there are zero, one, or many results. *For our purposes in this course, we'll always treat the results as a special kind of array*, so our **$result** variable always contains this special array, whether we select one or many elements.

The jQuery method **text()** works whether that array contains one or many elements. jQuery functions like **text()**, **html()**, and **css()** work with an array of elements, so if you use **text()** to change the text content of those elements, it will change the content of *all* of them. Now, try this:

CODE TO TYPE:

```
<script>
  $(document).ready(function() {
    var $result = $("~~h2~~p");
    ~~$result.text("The evil Dr. Moriarty just hacked your content");~~
    $result.addClass("highlight");
  });
</script>
```

 Save the file and Preview preview. We selected all the <p> elements in the page and added the CSS "highlight" class to those elements (recall from the HTML that we defined this class in our <style> element):

OBSERVE:

```
<style>
  .highlight {
      background-color: yellow;
  }
</style>
```

You'll find that many jQuery methods, like **addClass()**, work on one or multiple elements, so you don't have to worry about how many elements are selected by **$()**.

When you use **$()** to select elements, you get back a jQuery object that acts like a special kind of array, and contains all of the elements that match the selector. You get this array of results by selecting with **$()** even if there is just a single match or even no matches at all. When you call a method like **addClass()** or **text()** on the result, jQuery makes sure that the method applies to every element in that array. This functionality makes selecting and manipulating elements more straightforward than it is in pure JavaScript. If you used only Javascript, you'd have to consider which method to use to select elements (which would depend on how many elements you expect in the results), then handle the results based on the number of elements that

match. Using jQuery, even when no elements are selected, you can call these methods without getting an error message, so you don't have to worry about making sure there's at least one result.

You always get an array when you select elements using jQuery, so you can always use the **length** property to find out how many elements match your selection. Modify your jQuery script code:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("p");
  alert($result.length + " elements matched the selection");
  $result.addClass("highlight");
});
```

 Save the file and  preview. You see an alert indicating how many elements matched the selector "p".

Descendant Selectors

When you use **$()** to select elements, the string you pass into this function is called the *selector*, just like it is in CSS. Also, just like in CSS, you have a wide variety of ways to select elements.

Let's experiment with some other kinds of selectors you can use in jQuery. Modify your jQuery script as shown:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("p h4 a");
  alert($result.length + " elements matched the selection");
  $result.addClass("highlight");
});
```

 Save the file and  preview. (Make sure you reload the page, to remove the previous highlight from the <p> elements.) You can read the selector "h4 a" just like you would in CSS; you're selecting <a> elements that are descendants of <h4> elements. In our document, there's just one. Now select just those <a> elements that are within <li> elements:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("h4 li a");
  $result.addClass("highlight");
});
```

 Save the file and  preview. (Make sure you reload again to clear any previous highlights).

Now, only the "a" elements nested within <li> elements are highlighted. What other selector could you use to select only the elements within the list? (Hint: you can use the <ul> element in the selector).

You can select direct descendants too, just like in CSS. Let's say you want to select only elements that are direct descendants (children) of <p> elements:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("p > em li a");
  $result.addClass("highlight");
});
```

 Save the file and  preview. The word "really" is highlighted in the second paragraph. How would you select only elements that are direct descendants of <a> elements? Give it a try!

Class Selectors

You've already seen how to select elements by id using jQuery; we used the selector "div#hello" to select a element with the id "hello" in the first lesson.

You can also use classes to select elements. In our HTML, we added the class "first" to the first <li> element in the list, and the first <p> element of the five paragraphs. Try this:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("p > em .first");
  $result.addClass("highlight");
});
```

 Save the file and  preview. The first list item and the first paragraph are now highlighted; we selected all elements with the class "first." Notice that we use the same syntax that we do in CSS to select classes, using a period before the class name (compare with using a # in front of an id name, to select an element by id).

So, what if you want to select only paragraphs with the class "first"?

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("p.first");
  $result.addClass("highlight");
});
```

 Save the file and  preview. Now, only the first paragraph is highlighted. Can you think of how would you select only the first list item using a class selector?

jQuery selectors: Going Beyond the Basics

As you can probably tell, many of the jQuery selectors are exactly like the CSS selectors you've used before.

There are also many jQuery selectors that do more than we typically do with CSS (although most of them are available in CSS too). For instance, try this:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("p.first p[class='first']");
  $result.addClass("highlight");
});
```

 Save the file and  preview. The first paragraph is again highlighted. (Be careful to use double quotation marks around the entire selector, and single quotation marks around the string 'first'—we have to do this because we have one string nested inside another.) Select all <p> elements with an attribute named "class" which is equal to the string "first." This selector does the everything that our previous class selector did, and more. For instance, check this out:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("p[class!='first']");
  $result.addClass("highlight");
});
```

 Save the file and  preview. Now all paragraphs *except* the one with the class "first" are highlighted. You can read "!=" as "NOT equal to."

So the syntax **[name="value"]** selects elements with a **name** attribute (like class), with a value that's equal to **value**; and **[name!="value"]** selects elements that *don't* have a **name** attribute with the value **value**.

This works on any attribute, so you can use it for id, or other attributes too, like href. Try this:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("p[class!='first']a[href*='TV']");
  $result.addClass("highlight");
});
```

 Save the file and  preview. We selected all <a> elements with "href" attributes that contain the string "TV". Could you select only <a> elements with "href" attributes that contain the string "Doyle"? Give that a try.

Now try adding another jQuery function:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("a[href*='TV']");
  var $result = $("a[href*='wiki']").not("a[href*='TV']");
  $result.addClass("highlight");
});
```

 Save the file and  preview. What do you think this selects?

First, we select all <a> elements with "href" attributes that contain the string "wiki" (which in our case is all of them). Then we take that result and refine it. We use the jQuery function **not()** to find all results that are *not* <a> elements whose "href" attribute contains the string "TV". So all the <a> elements except the one that links to the Sherlock TV series are highlighted.

jQuery provides many convenient ways to select elements: this is the "Query" in "jQuery." It's an incredibly powerful part of the library, but just one of many ways you can select elements with jQuery. We'll encounter many of these and other selectors, throughout the course. Once you've completed this lesson, take some time to look through the selectors at the jQuery.com website.

jQuery Objects and JavaScript Objects

As we've worked through some different selectors, you might have been wondering why we used the variable name **$result** rather than **result**. We do this as a convention, to remind ourselves that what's stored in this variable is the result of using a jQuery function or method, rather than pure JavaScript code.

So what's the difference between a jQuery object and a JavaScript object? A jQuery object *wraps* a JavaScript object with extra functionality, providing access to all the convenience methods we're using to do things like set the text content of an element (**text()**), add a class to an element (**addClass()**), change the CSS of an element (**css()**), and more.

Let's take a closer look at what it means to "wrap" an object. Suppose you want to get the "links" element in JavaScript; you could write this:

OBSERVE:

```
var result = document.getElementById("links");
```

The JavaScript object you get in **result** is a single HTML Element object that has various methods and properties, like **innerHTML**, **tagName**, **appendChild()**, **insertBefore()**, and more.

If you want to get the same object in jQuery, you'd write **var $result = $("ul#links");** to select the "links" element. In this case, **$result** is an array that contains one element (since the "links" element is unique in the document). You can then call a variety of jQuery methods on that array, like **html()**, **text()**, **css()**, methods you haven't seen yet, such as **append()**, **prepend()**, and **replaceWith()**, and properties like **length** (to find out how many elements are in the **$result** array), and many more. Remember that any jQuery method you call on this array will apply to *all* the elements in the array; in this case, we know we have

just one. However, if your **$result** is an array containing multiple elements, and you use the **css()** method to style the results, that method will apply the style to every element in that array.

If you want to *unwrap* the HTML element object from inside the jQuery object to get the pure JavaScript object (the same one you'd get if you wrote the JavaScript above to get the document element by id), you can do that:

OBSERVE:

```
var ul = $result.get(0);
```

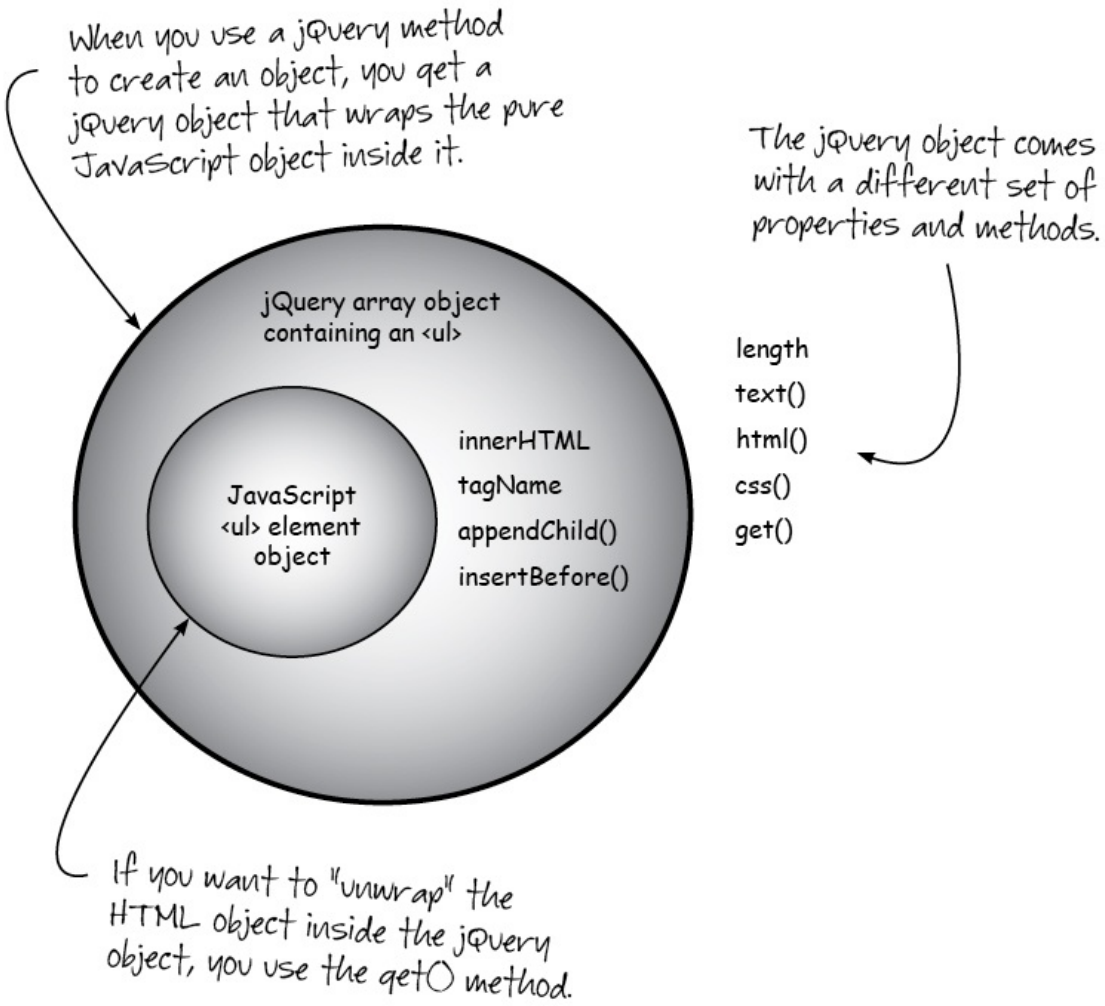
We use the **get()** method to get the object at array index 0 (it's the only object in our array, so we know it's the one we want). **get()** returns the HTML object that's wrapped up in the jQuery object. This **ul** object is the same HTML element object in **result** above, but now that it's converted to an HTML element object from a jQuery object, you can no longer use any of those jQuery methods we just mentioned, because it's unwrapped. You won't often find a need to unwrap jQuery objects, but every once in a while it comes up (we'll see an example of that in the next lesson).

You may want to go the other direction more often though; you may want to take a pure JavaScript object and wrap it so you have access to all those jQuery methods and properties that make working with elements more convenient. Here's how you'd take the **ul** object we just created and turn it back into a jQuery object again:

OBSERVE:

```
var ul = $result.get(0);
var $ul = $(ul);
```

Let's go over this carefully. We take the HTML Element object, **ul**, that we created by unwrapping the jQuery object, and pass it to the jQuery function, **$**, which wraps it back up in a jQuery object. We store that jQuery object in the new variable, **$ul**. Be careful not to confuse the *function*, **$()**, we use to create jQuery objects, with the *convention* we use to create variable names, like **$ul**, to store jQuery objects.



We'll encounter a situation where we need to wrap a JavaScript object fairly soon, so watch for it!

In this course, we'll adhere to the convention of using **$** in the names of variables that we create with jQuery, so we can keep track of which variables are jQuery objects and which are pure JavaScript objects. To reiterate, any variable with **$** in the name is a jQuery variable (that is, the result of a jQuery function or method), and any variable without **$** is a pure JavaScript variable. This is a common convention among jQuery developers, so get in the habit of doing it yourself.

jQuery Methods and Chaining

Frequently, we use *chaining* to combine method calls in jQuery. We can simplify our code by combining the two lines. Modify your jQuery script code as shown:

CODE TO TYPE:

```
$(document).ready(function() {
  var $result = $("a[href*='wiki']").not("a[href*='TV']");
  $result.addClass("highlight");
  $("a[href*='wiki']").not("a[href*='TV']").addClass("highlight");
});
```

Save the file and Preview preview. This version of the code does exactly the same thing as the previous version, but we got rid of the intermediate variable, **$result**, and did all of the operations on one line by chaining the method calls together.

This works so well in jQuery because each jQuery method call returns a jQuery object: the array of elements that match the selector. So, here's how to read this line:

OBSERVE:

```
$("a[href*='wiki']").not("a[href*='TV']").addClass("highlight");
```

1. Select all the **<a> elements with "href" attributes that contain the string "wiki."**
2. Take that set of elements (in an array), and **call the jQuery not() method, which looks at each element in the array and removes any <a>element with "href" attributes that contain the string "TV."**
3. On the resulting, slightly smaller, array of elements, **call the addClass() method, which adds the "highlight" class to every element in the array**.

Every jQuery method returns the selected set of elements in this manner, which makes chaining the methods together more straightforward. Practice reading these lines of chained methods so you can look at three or four chained methods and quickly understand how they work together to select and manipulate one or more elements.

Manipulating Element Style with css(), addClass(), and removeClass()

Before we end the lesson, let's take a closer look at the **addClass()** method we use to add a CSS class to our elements, and compare this way of styling elements to the **css()** method that we used in the previous lesson. While you might think these are two ways of doing essentially the same thing, there is a subtle difference.

Look again at our code to add the "highlight" class to a set of elements:

OBSERVE:

```
$(document).ready(function() {
    $("a[href*='wiki']").not("a[href*='TV']").addClass("highlight");
});
```

Preview Preview the file again and look at the HTML source code of the resulting page. The selected <a> elements have been changed so they have a **new class attribute**:

OBSERVE:

```
<ul id="links">
  <li class="first"><a href="http://en.wikipedia.org/wiki/Sherlock_holmes" class
="highlight">Fictional Detective</a></li>
  <li><a href="http://en.wikipedia.org/wiki/Sherlock_(TV_series)"><em>Sherlock</
em>, TV series</a></li>
  <li><a href="http://en.wikipedia.org/wiki/Sherlock_Holmes_(2009_film)" class="
highlight"><em>Sherlock Holmes</em>, 2009 Film</a></li>
  <li><a href="http://en.wikipedia.org/wiki/Sherlock_Holmes_(play)" class="highl
ight"><em>Sherlock Holmes</em>, play by William Gillette and Sir Arthur Conan Do
yle</a></li>
</ul>
```

As soon as this class is added to the elements, the style is applied, and the yellow highlighting appears on the elements.

Now let's compare this to what happens when we use the **css()** method to apply a yellow background to these same elements:

CODE TO TYPE:

```
    $(document).ready(function() {
      $("a[href*='wiki']").not("a[href*='TV']").addClass("highlight")css("backgr
ound-color", "yellow");
    });
```

Save the file and Preview, and look at the HTML source code:

OBSERVE:

```
<ul id="links">
  <li class="first"><a href="http://en.wikipedia.org/wiki/Sherlock_holmes" style
="background-color: yellow;">Fictional Detective</a></li>
  <li><a href="http://en.wikipedia.org/wiki/Sherlock_(TV_series)"><em>Sherlock</
em>, TV series</a></li>
  <li><a href="http://en.wikipedia.org/wiki/Sherlock_Holmes_(2009_film)" style="
background-color: yellow;"><em>Sherlock Holmes</em>, 2009 Film</a></li>
  <li><a href="http://en.wikipedia.org/wiki/Sherlock_Holmes_(play)" style="backg
round-color: yellow;"><em>Sherlock Holmes</em>, play by William Gillette and Sir
 Arthur Conan Doyle</a></li>
</ul>
```

This is setting the style directly on the element, rather than through a class. You can see how these two methods of styling an element are slightly different if you try this:

CODE TO TYPE:

```
    $(document).ready(function() {
      $("a[href*='wiki']").not("a[href*='TV']").addClass("highlight");
      $("a[href*='wiki']").not("a[href*='TV']").css("background-color", "yellowr
ed");
    });
```

 Save the file and Preview preview. What background color do the select <a> elements have, red or yellow?

Now, switch the two lines:

CODE TO TYPE:

```
    $(document).ready(function() {
      $("a[href*='wiki']").not("a[href*='TV']").addClass("highlight");
      $("a[href*='wiki']").not("a[href*='TV']").css("background-color", "red");
      $("a[href*='wiki']").not("a[href*='TV']").css("background-color", "red");
      $("a[href*='wiki']").not("a[href*='TV']").addClass("highlight");
    });
```

 Save the file and Preview preview. Did you expect your selected <a> elements to have a yellow background this time? If you did, remember that a style you apply directly to an element takes precedence over style you apply through a class. Think of it this way: first the browser applies all the style that's defined in the class of the element; *then* it applies the style defined in the "style" attribute of the element. In our case, first the background color is set to yellow, and then it is set to red, so what you see is red.

In some situations, you're going to want to choose **addClass()**, and in others, **css()**, to add style to your elements.

To remove a class you've set on an element, use the **removeClass()** method, like this:

CODE TO TYPE:

```
    $(document).ready(function() {
      $("a[href*='wiki']").not("a[href*='TV']").css("background-color", "red");
      $("a[href*='wiki']").not("a[href*='TV']").addClass("highlight");
      $("a[href*='play']").removeClass("highlight");
    });
```

 Save the file and Preview preview. We add the "highlight" class to all <a> elements that don't contain the string "TV" in their "href" attributes, and then we remove it from all <a> elements that have the string "play" in their "href" attribute (which is just the last link in the list).

So far you've learned how to select elements using jQuery, using a variety of selectors, some of which look a lot like the CSS

selectors you already know, and some which are unique to jQuery. jQuery selectors give you lots of flexibility.

jQuery selectors can also get quite complex! It's a good idea to keep your selectors as simple as possible to make them easier to read, and to keep your code efficient. The more complex your selectors, the more computation the browser needs to do to search for and create that set of elements. We'll come back to this again later as we learn tricks for optimizing your code.

Put all your new jQuery skills to use doing the quiz and the project. In the next lesson, you'll learn more ways to select and manipulate content, as we take your jQuery skills to the next level, building a playlist application!

Filtering Elements

Lesson Objectives

- You will create a Playlist builder application using jQuery.
- You will use new select and filter elements.
- You will use the method **each()** to iterate over a jQuery object, and **attr()** to both retrieve and set the value of an attribute of an element.

Filtering Elements

In the previous lesson, we learned how to select elements in a variety of different ways using jQuery selectors. Selecting elements to manipulate (remove, style, modify, and such) is the primary function of jQuery library. The library provides several selectors and methods to use to create a set of elements to manipulate. In this lesson, we'll look at ways to select and filter elements, select elements in a form, and select elements by content. We'll also look at a couple of jQuery methods you haven't seen yet: **attr()** and **each()**. We'll do all of that while building an application that lets you select songs from an album to add to a playlist. That sounds fun, right?

Filtering jQuery Selector Results

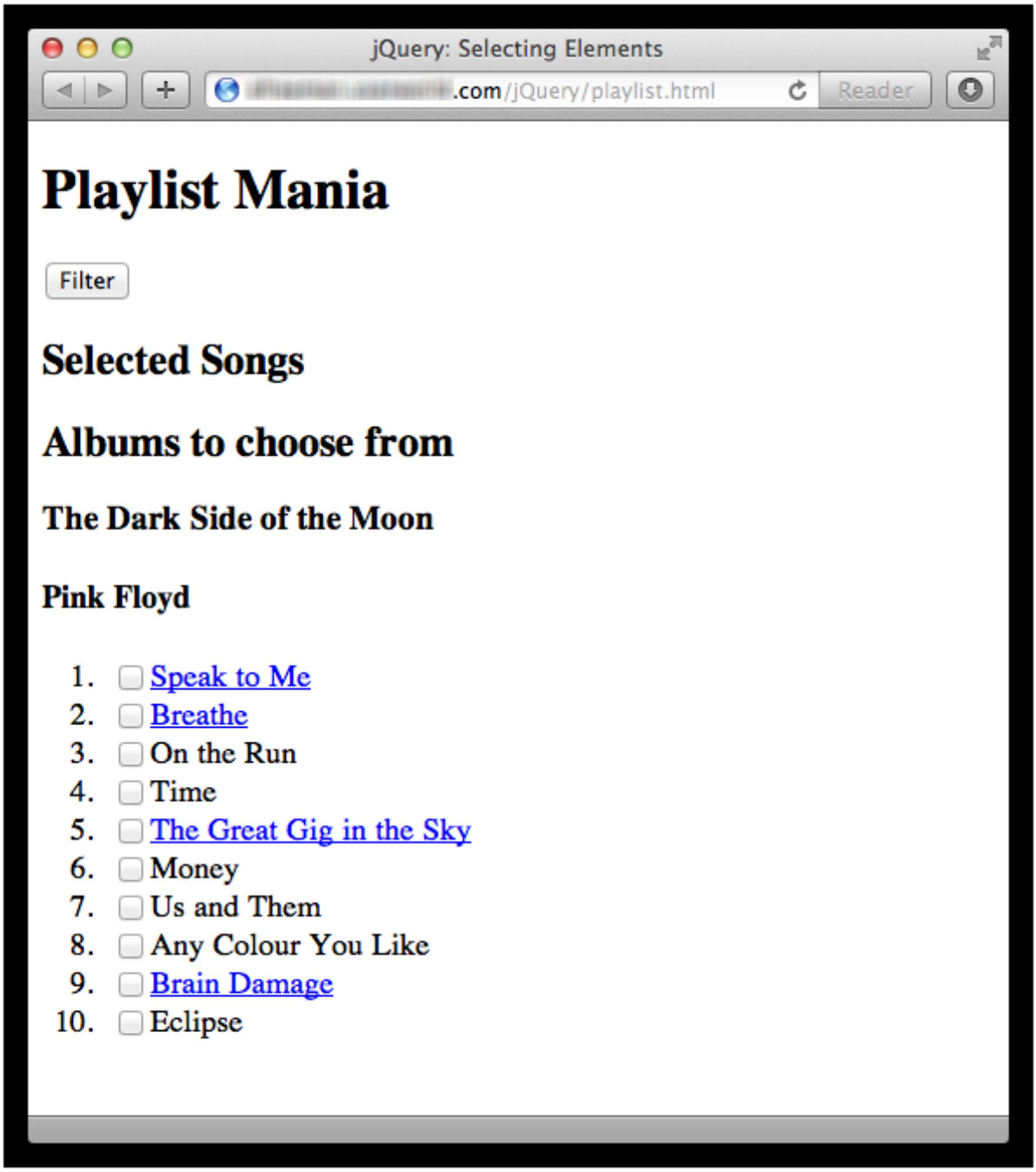
Before we try out any more jQuery selectors, we need a web page. We're building a Playlist application, so let's start by creating a new file and adding this HTML:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Selecting Elements</title>
  <meta charset="utf-8">
  <style>
    .highlight {
        background-color: yellow;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>

  </script>
</head>
<body>
  <h1>Playlist Mania</h1>
  <form>
    <input type="button" value="Filter" id="filterButton">
    <h2>Selected Songs</h2>
    <ul id="playlist">
    </ul>

    <h2>Albums to choose from</h2>

    <div class="album">
      <h3>The Dark Side of the Moon</h3>
      <h4>Pink Floyd</h4>
      <ol>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Speak_to_Me" data-length="1:30">Speak to Me</a></li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Breathe_(Pink_Floyd_song)" data-length="2:43">Breathe</a></li>
        <li><input type="checkbox"><a data-length="3:36">On the Run</a></li>
        <li><input type="checkbox"><a data-length="7:01">Time</a></li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/The_Great_Gig_in_the_Sky" data-length="4:36">The Great Gig in the Sky</a></li>
        <li><input type="checkbox"><a data-length="6:22">Money</a></li>
        <li><input type="checkbox"><a data-length="7:46">Us and Them</a></li>
        <li><input type="checkbox"><a data-length="3:25">Any Colour You Like</a></li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Brain_Damage_(song)" data-length="3:48">Brain Damage</a></li>
        <li><input type="checkbox"><a data-length="2:03">Eclipse</a></li>
      <ol>
    </div>
  </form>
</body>
</html>
```

 Save this file in your **jQuery/** folder as **playlist.html**.

Preview the file; you'll see this web page:



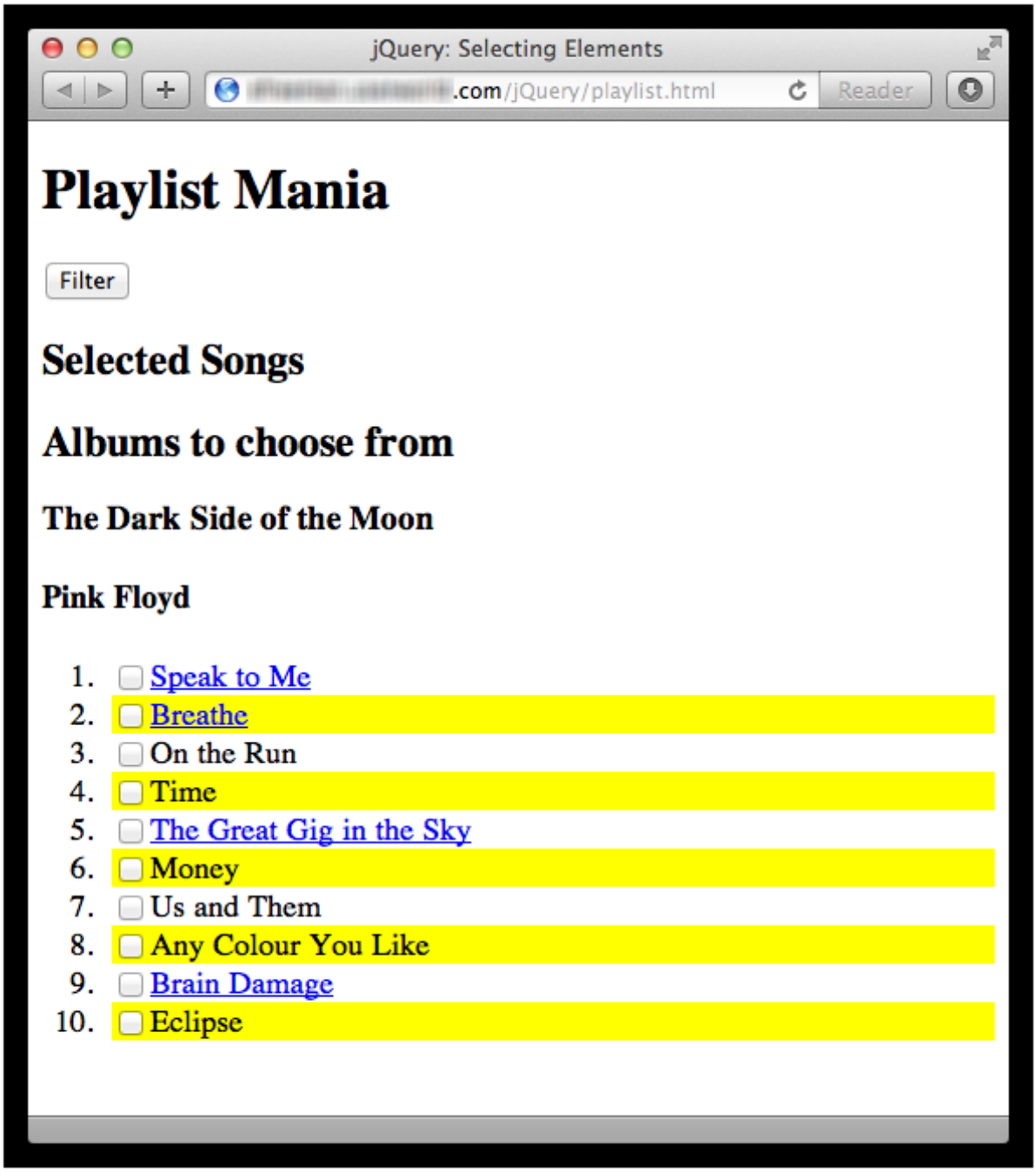
Note Because we're not submitting the form in this page to a server, we don't need an action attribute on the form or the name or the value attributes on the <input> elements like we would if we intended to process the form with a server-side program.

So now we have a <script> element in the <head> of the document and we're ready to add some jQuery:

CODE TO TYPE:

```
<script>
  $(document).ready(function() {
    $("input#filterButton").click(filterOdd);
  });
  var filterOdd = function() {
    $("li:odd").addClass("highlight");
  };
</script>
```

Save the file and Preview preview again. Click the **Filter** button, and you'll see this:



In this code, we set up the usual ready function, and in that function we set up a click handler for the **Filter** button, so that when you click that button, the **filterOdd()** function is called. To add the click handler, we select the button with the **input#filterButton** selector.

OBSERVE:

```
var filterOdd = function() {
    $("li:odd").addClass("highlight");
};
```

The **filterOdd()** function is called when you click the "Filter" button, and it **selects all <li> elements** to **highlight**. Once we've selected the <li> elements, we **filter them to pull out only the *odd* elements**. The **:odd** filter refines the set of elements initially selected (all <li> elements in the page) and filters them by some criteria; in this case, by whether the <li> element has an odd-numbered index in the array of <li> elements returned by the "li" selector (remember that all selectors return a jQuery object that acts just like an

array of matching elements).

So, if we're selecting *odd* elements, then why are the *even*-numbered items from the list highlighted? Welp, an array is indexed beginning at 0, while our numbered list begins at 1, so the first item in the array is at index 0, but has the number 1. The second item in the array is at index 1, but has the number 2, and so on.

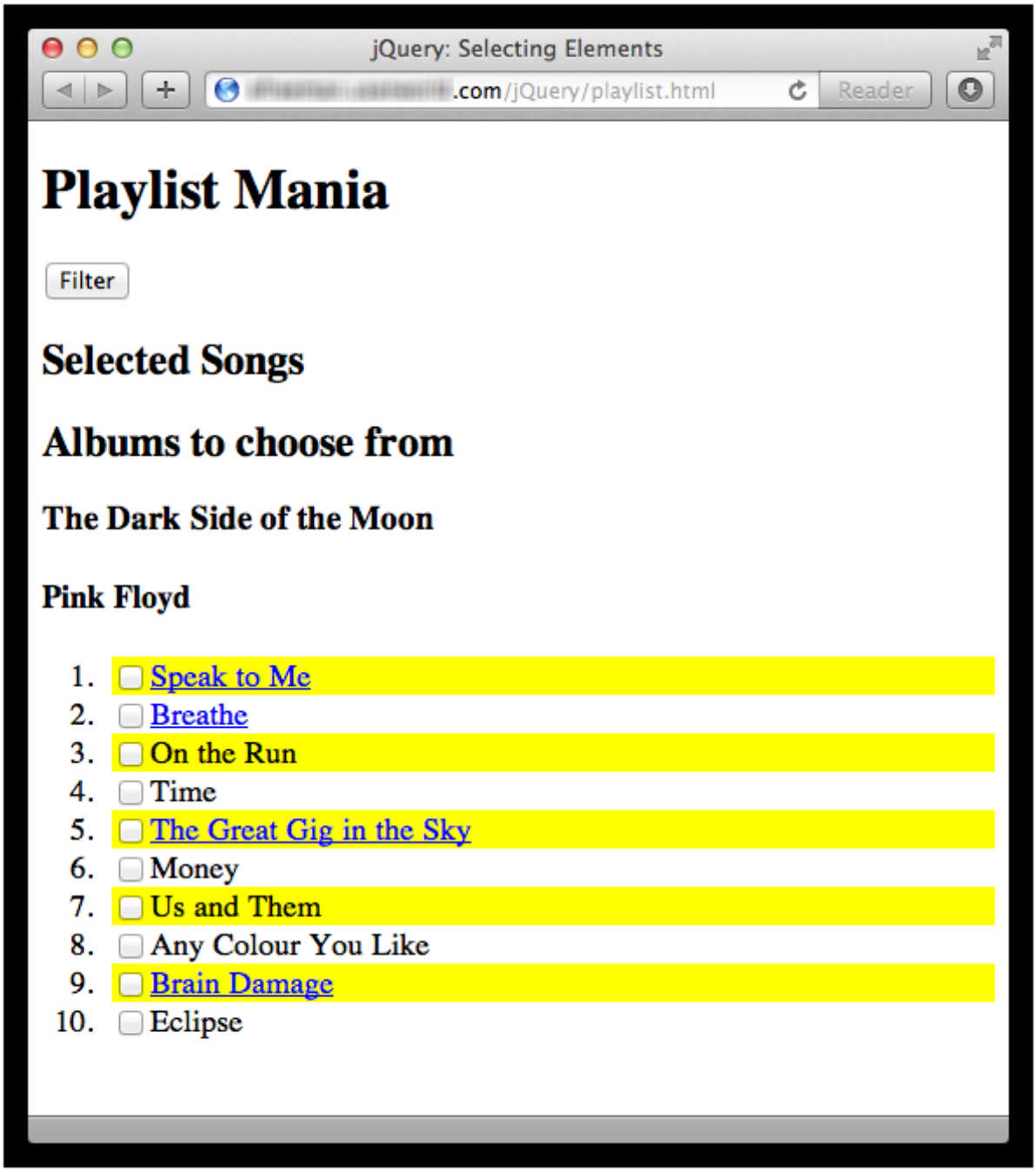
Let's try changing the filter from **:odd** to **:even**:

CODE TO TYPE:

```
$(document).ready(function() {
  $("input#filterButton").click(filterOddfilterEven);
});
var filterOdd = function() {
  $("li:odd").addClass("highlight");
};
var filterEven = function() {
  $("li:even").addClass("highlight");
};
```

 Save the file, 

preview, and click **Filter** again. Now the even-indexed songs are highlighted:



Keep in mind that song 0 has the number 1 in our list of songs! So the *even* songs are displayed with odd numbers.

There are many ways to refine your selector results by filtering the selected elements with criteria like **:odd** and **:even**; take a minute to look through these basic filters at jQuery.com.

Note We're deleting each of the previous filter functions as we go to keep the source code short, but you can leave them in your file if you like; we use a different name for each function so they won't clash with one another. Just remember to update the name of the function you're calling in the filter button click handler.

The each() Method

So far, we've let jQuery filter selected elements for us using its built-in filtering methods like **:odd**, **p > em**, **p[class='first']**, and **not()**. But what if you want to look through each of the elements selected with a jQuery selector using your own code? Maybe you want to create a filter that's more complex than the basic filters provided by jQuery. How can you look at each element selected by a query?

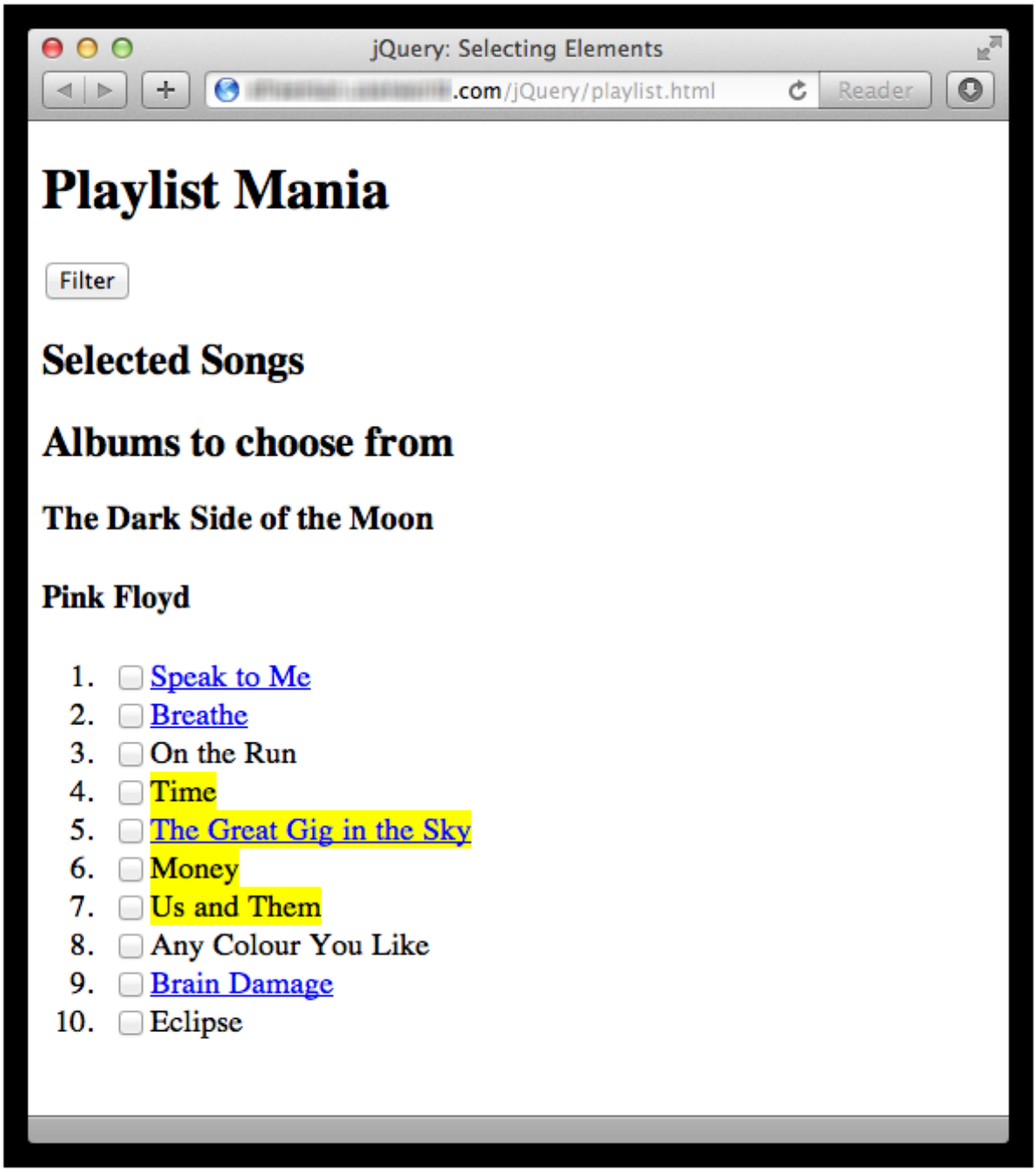
jQuery provides the handy **each()** method to let you examine the result of your query. Let's give it a try:

CODE TO TYPE:

```
$(document).ready(function() {
  $("input#filterButton").click(filterEvenfilterLength);
});
var filterEven = function() {
  $("li:even").addClass("highlight");
};
var filterLength = function() {
  $("a[data-length]").each(function() {
    var length = $(this).attr("data-length");
    var min = (length.split(":"))[0];
    var min = parseInt(min);
    if (min > 3) {
      $(this).addClass("highlight");
    }
  });
};
```

 Save the file and 

preview. When you click **Filter**, only the songs that are at least 4 minutes long are highlighted:



There's quite a bit going on in this code. Let's go through it piece by piece.

OBSERVE:

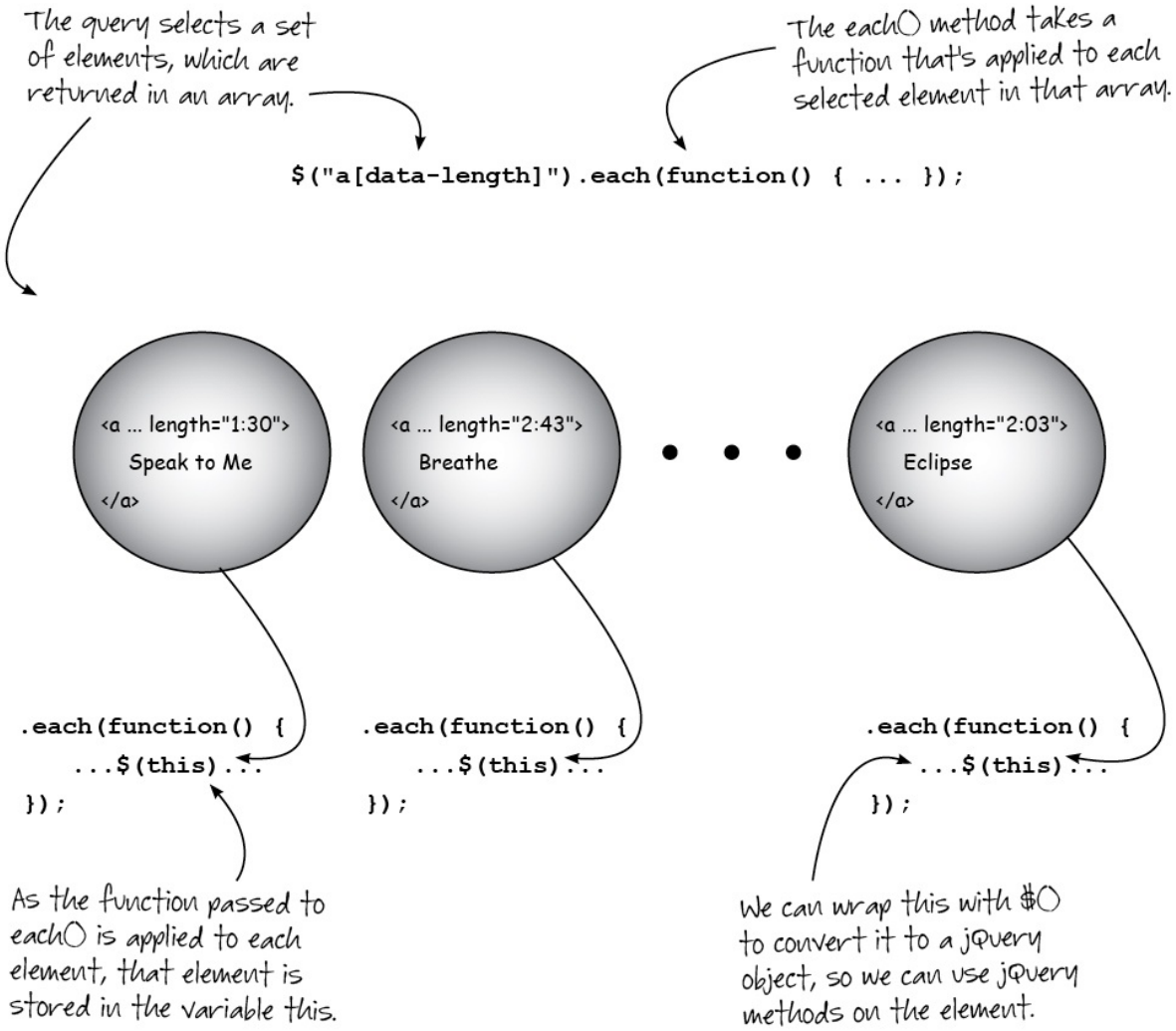
```
var filterLength = function() {
    $("a[data-length]").each(function() {
        var length = $(this).attr("data-length");
        var min = (length.split(":"))[0];
        var min = parseInt(min);
        if (min > 3) {
            $(this).addClass("highlight");
        }
    });
};
```

In the **filterLength()** function, we **select all <a> elements that have a "data-length" attribute**. Back up in the HTML; you'll see that each of the list items making up the songs in the album contains an <a> element with the attribute **data-length**:

OBSERVE:
`<li><input type="checkbox"><a data-length="7:01">Time</a></li>`

In case you haven't run into **data attributes** in HTML before (these are new in HTML5), data attributes allow you to add arbitrary attributes to your HTML elements, as long as the attribute name is prefixed with "data-". This allows you to add metadata to your elements beyond the built-in attributes, like **class** and **title**. It's basically a custom attribute for your element. In this case, we've added a "data-length" attribute containing the length of the song in minutes and seconds.

Our selector returns an array of all the <a> elements that have one of these "data-length" attributes (in this case, that's all the <a> elements in the list), and then we **call the method each() on that array**. The **each()** method takes a **function()** that is applied to each element in the array, one at a time. So, if five elements are selected by the **selector**, the **function** will be called five times, once for each element. Within that function, the selected element corresponding to that application of the function is accessible via the **this** variable. You probably remember using **this** in JavaScript, often to represent the object with a method that's being invoked. Here we have a similar situation. Think of the selected element as the object and the **function passed into each()** as the method.



Let's take a closer look at our **function**. We want to filter the selected elements by the length of the song, and highlight only songs that are greater than or equal to 4 minutes in length.

OBSERVE:

```
var filterLength = function() {
    $("a[data-length]").each(function() {
        var length = $(this).attr("data-length");
        var min = (length.split(":"))[0];
        var min = parseInt(min);
        if (min > 3) {
            $(this).addClass("highlight");
        }
    });
};
```

This is a short function, but it's got a lot packed into it. It selects all of the <a> links with a "data-length" attribute, and then, for each of those elements, compares the minutes value stored in the "data-length" attribute with 3; if the value is greater than 3, our function highlights the <a> element by adding the "highlight" class to it.

First we **wrap this with the $() function** to convert **this** to a jQuery object, so we'll have access to jQuery methods we want to use like **addClass()**. Now, remember that **this** represents the selected element, which is one of the <a> elements with a "data-length" attribute, so **$(this)** represents that same element (only now it's a jQuery object).

We can use the method **attr()** to get the value of the "data-length" attribute for the song. The value of the attribute is returned as a string, which we store in the **length** variable; it might be "1:30" if the selected element is the link for "Speak to Me" or "7:01" if the selected element is the link for "Time." **length** is just an ordinary string, so we don't use $ in the name. We then **split()** the string at the character ":", which returns an array with two values (the minutes and seconds), and we get the first element in the array (the minutes) and store that in the variable **min**. Then we convert that to an integer, and if **min** is greater than 3, we add the "highlight" class to the selected element in **$(this)** using the **addClass()** method.

We'll use **each()** often in this course to work on *each* element in a selected set of elements.

Filtering Form Elements

Each of our playlist items is a "checkbox" input element in a form. Now let's say you want to highlight all of the items you've checked. jQuery provides form selectors that allow you to select form elements by type, as well as by other attributes, for instance, whether they are checked, selected, or even hidden. Let's try using one of these selectors. We'll write a filter function to select all checked input elements and highlight them:

CODE TO TYPE:

```
    $(document).ready(function() {
      $("input#filterButton").click(filterLengthfilterChecked);
    });
    var filterLength = function() {
      $("a[data-length]").each(function() {
        var length = $(this).attr("data-length");
        var min = (length.split(":"))[0];
        var min = parseInt(min);
        if (min > 3) {
          $(this).addClass("highlight");
        }
      });
    };
    var filterChecked = function() {
      $("li input:checked").parent().addClass("highlight");
    };
```

 Save the file and  preview again. Check the checkboxes next to some of the song names. Click **Filter**. The checked items are highlighted.

The selector we use here, "li input:checked", selects all checked <input> elements that are nested within <li> elements. For our application we could leave off the "li" and it would do the same thing, but we wanted to demonstrate that you can combine form selectors with descendant selectors. Go ahead and try removing "li"

from the selector so that it's just "input:checked," and make sure your code works the same way.

We used the **parent()** method in the function **filterChecked()**. Let's go over the code and see why we did that:

OBSERVE:

```
var filterChecked = function() {
    $("li input:checked").parent().addClass("highlight");
}
```

We **select <input> elements that are checked and nested inside <li> elements**. If we add the class highlight to the selected <input> elements only, the highlight will not be visible, because the class will be applied only to the checkbox and not on the <a> element (as it has been previously), or the entire <li> element, so instead we highlight the entire <li> element.

We can use the **parent()** method to ask for the *parent* element of the selected <input> element, which is the <li> element that contains the checkbox and the name of the song. So, if you check the song "On the Run," the parent element of the checkbox next to this song name is the <li> element that contains the "On the Run" song. Then we add the highlight class to this <li> element so the whole list item is highlighted.

Combining Filters

You can also combine form filters like **:checked** with other filters. Just for fun, let's change the **filterChecked()** function so it selects all <input> that is *not* checked, by adding the **:not()** filter function:

CODE TO TYPE:

```
$(document).ready(function() {
  $("input#filterButton").click(filterChecked);
});
var filterChecked = function() {
  $("li input:not(:checked)").parent().addClass("highlight");
};
```

 Save the file and 

preview again. Check the boxes next to some of the song names. Click **Filter**. Now all the list items *except* those that are checked are highlighted.

There are lots of form selectors in jQuery; here are just a few:

Selector	What it selects
:text	<input type="text"> elements
:radio	<input type="radio"> elements
:checkbox	<input type="checkbox"> elements
:submit	<input type="submit"> elements
:button	<input type="button"> elements
:hidden	<input type="hidden"> elements
:checked	All input elements that are currently checked (works on checkbox and radio)

Before you move on, take look at the full list of form selectors at the jQuery.com website.

We talked briefly about navigating documents using methods like **parent()**; we'll go over this topic in greater detail later in the course.

Selecting Elements by Content

In the previous lesson, we selected elements based on content in an attribute, with selectors like **a[href*='TV']**. We can also select elements based on the element's content. For example, we might want to select all the songs that contain the word "Time." In our case, that's just one song, but you get the idea. Let's add a way to enter some content to filter songs. Update your **playlist.html** file. Add a new "text" <input> element where the user can enter content. Also, add a "Clear" button (you'll see why we need that shortly):

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Selecting Elements</title>
  <meta charset="utf-8">
  <style>
    .highlight {
        background-color: yellow;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>

  </script>
</head>
<body>
  <h1>Playlist Mania</h1>
  <form>
    <input type="text" id="containsContent">
    <input type="button" value="Filter" id="filterButton">
    <input type="button" value="Clear" id="clearButton">
    <h2>Selected Songs</h2>
    <ul id="playlist">
    </ul>

    <h2>Albums to choose from</h2>

    <div class="album">
      <h3>The Dark Side of the Moon</h3>
      <h4>Pink Floyd</h4>
      <ol>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Speak_to_Me" data-length="1:30">Speak to Me</a></li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Breathe_(Pink_Floyd_song)" data-length="2:43">Breathe</a></li>
        <li><input type="checkbox"><a data-length="3:36">On the Run</a></li>
        <li><input type="checkbox"><a data-length="7:01">Time</a></li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/The_Great_Gig_in_the_Sky" data-length="4:36">The Great Gig in the Sky</a></li>
        <li><input type="checkbox"><a data-length="6:22">Money</a></li>
        <li><input type="checkbox"><a data-length="7:46">Us and Them</a></li>
        <li><input type="checkbox"><a data-length="3:25">Any Colour You Like</a></li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Brain_Damage_(song)" data-length="3:48">Brain Damage</a></li>
        <li><input type="checkbox"><a data-length="2:03">Eclipse</a></li>
      <ol>
    </div>
  </form>
</body>
</html>
```

 Save the file and  preview. Now you have an area where you can enter content, and a new **Clear** button. Let's get these elements hooked up to some actions with jQuery:

CODE TO TYPE:

```
$(document).ready(function() {
    $("input#filterButton").click(filterCheckedfilterContent);
});
var filterChecked = function() {
    $("li input:not(:checked)").parent().addClass("highlight");
};
var filterContent = function() {
    var lookFor = $("input#containsContent").val();
    $("li a:contains('" + lookFor + "')").addClass("highlight");
};
```

 Save the file and 

preview again. Now, try entering a word that appears in at least one song name into the text input, like "the." Click **Filter**; songs containing that word are now highlighted. If you try "the," the song "Breathe" is highlighted, because "the" appears in that word—but "Us and Them" is *not* highlighted because of the "T" in "Them" is capitalized.

Let's go over our code:

OBSERVE:

```
var filterContent = function() {
    var lookFor = $("input#containsContent").val();
    $("li a:contains('" + lookFor + "')").addClass("highlight");
};
```

First we **select the "containsContent" <input> element**, using its id (so we know we have only one of them, since that id is unique). Then we use **val()** to get the value of that text <input>, which is the text that the user has entered. Note that if our selector matches multiple <input> elements instead of just one, **val()** will return the value of the *first one* that matches.

Once we establish the **content to look for**, we select any songs that match, by looking at the content of the **<a> element** in each <li> element; that content is the song name. We use the **:contains** filter to specify the word for which we're looking. We use single quotes around the word to differentiate from the selector, which is already enclosed in double quotes. Here's how this selector looks if the variable **lookFor** contains the word "the":

OBSERVE:

```
"li a:contains('the')"
```

With this selector, we're saying "select all <a> elements that are nested within <li> elements, and that contain the letters 'the'." Then we use the **addClass()** method to add the highlight to those <a> elements. We highlight the <a> element rather than the <li> element, like in the previous example (by using **parent()**). The difference in the way the songs look when they are highlighted is minor. How would you highlight the entire list item instead of just the <a> element?

Clearing the Previous Filter Results

You have at least one song highlighted from your first filter. Enter another word, and click **Filter** again. Notice that the previous items stay highlighted. That's because we don't reload the page, and we don't remove the highlight from the first filter before we add the highlight from the second (and subsequent) filters. Now, you see why we need a way to clear the previous results!

CODE TO TYPE:

```
$(document).ready(function() {
  $("input#filterButton").click(filterContent);
  $("input#clearButton").click(clearAll);
});
var clearAll = function() {
  $("*").removeClass("highlight");
  $("form").get(0).reset();
};
var filterContent = function() {
  var lookFor = $("input#containsContent").val();
  clearAll();
  $("li a:contains('" + lookFor + "')").addClass("highlight");
};
```

 Save the file and 

preview. Enter a word to apply filter to and click **Filter**. Now, enter another word, and click **Filter** again. The results from your first filter disappear, and only songs that match the new filter are highlighted. The word you typed in the text <input> disappears too, so you have an empty text input ready to go for your next filter. Now, try entering a word, so you have some highlighted songs, and click the **Clear** button. The highlights disappear. Try checking some of the songs by clicking the checkboxes, and then click "Clear" again. The checkboxes are reset.

The **clearAll()** function is called when we click the "clearButton" <input>. We added a click handler for this button to call **clearAll()** in the ready function. We also added a call to **clearAll()** in the **filterContent()** function, that way the form is cleared automatically when you filter again. We added the call to **clearAll()** *after* we got the value (the text the user entered) from the "containsContent" text <input>. Why do you think this is? Ponder and we'll come back to this topic in just a moment.

Let's take a closer look at the **clearAll()** function:

OBSERVE:

```
var clearAll = function() {
    $("*").removeClass("highlight");
    $("form").get(0).reset();
};
```

First, we **select all the elements** in the page (with the selector **"*"**), and **remove the "highlight" class** from those elements. Of course, most of the elements that match won't have that class, but that's ok; if you try to remove a class that doesn't exist from an element, it doesn't do any harm.

Once we've removed the "highlight" class (which turns off the highlight effect), we reset the form. We do this by selecting all <form> elements in the page. We just have one now, but the jQuery function **$()** returns an array of elements that match. So, given that we know that we just have one <form> in the page, we can get the first one using the method **get(0)**, which returns the first matching form in the matching results. Then we call **reset()** on that form to reset it to its default state (which in our case means unchecking any checked items) and set the "containsContent" text input back to empty.

Unwrapping a jQuery Object

So, why can't we just call **reset()** directly on the selector results like this?:

OBSERVE:

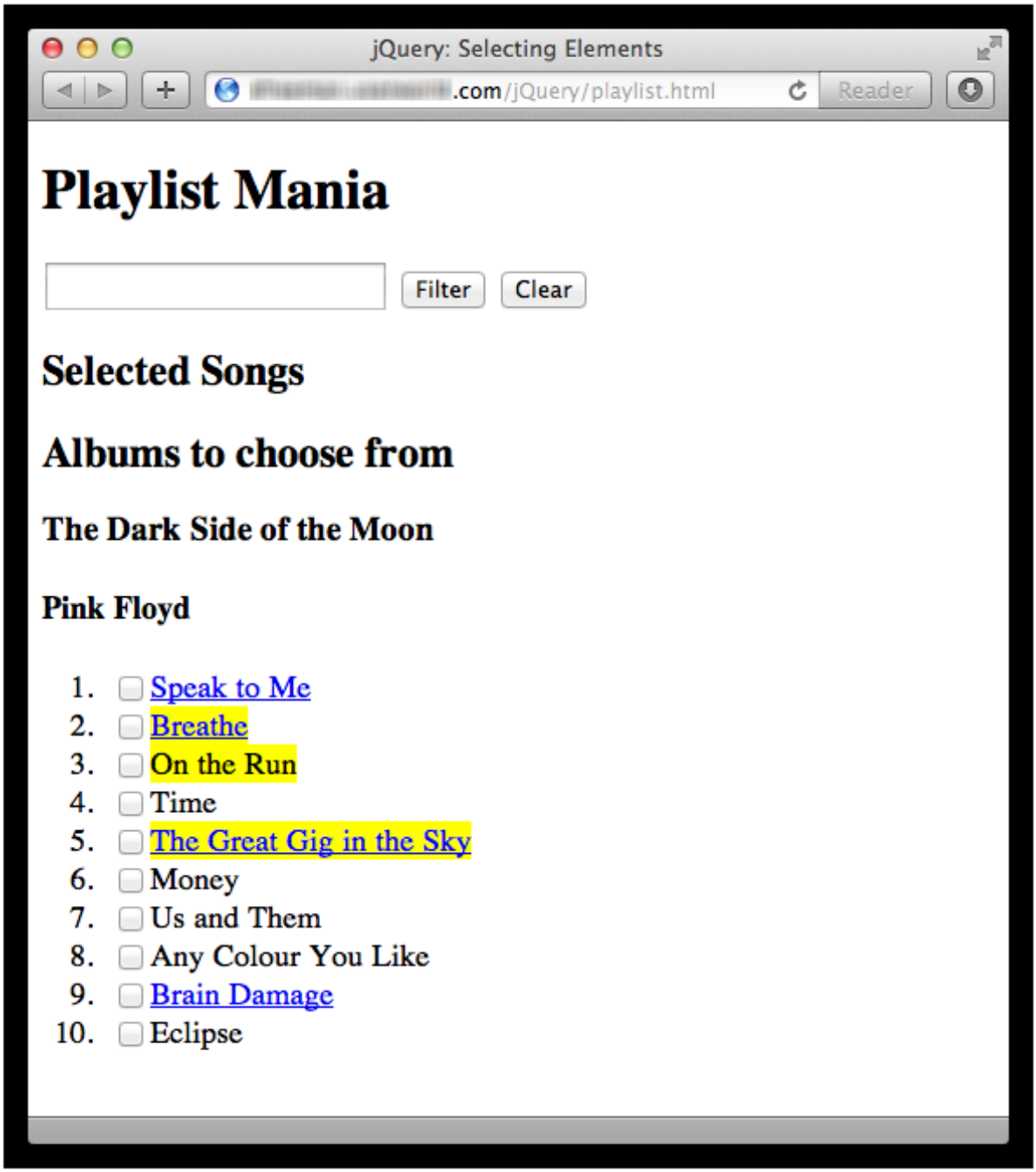
```
$("form").reset();
```

We can't do this because, the result of **$("form")** is a jQuery object, and it just so happens that there is no **reset()** jQuery method to reset a form! We have to unwrap the DOM form element from the jQuery object in order to call **reset()** (which is a plain old JavaScript method).

We call **clearAll()** *after* we get the value the user entered in the "containsContent" text input, because the "containsContent" text input is reset in the function **clearAll()**; if we call **clearAll()** *before* we get the value, there will be no value to get! We have to get the new value the user entered first, and then reset the form.

Now, enter "the" into the text input and click **Filter**. You get "the" as the value entered into the form. Clear any

previous highlights and reset the form, before highlighting the items that match "the":



Checking and Unchecking Form Elements with jQuery

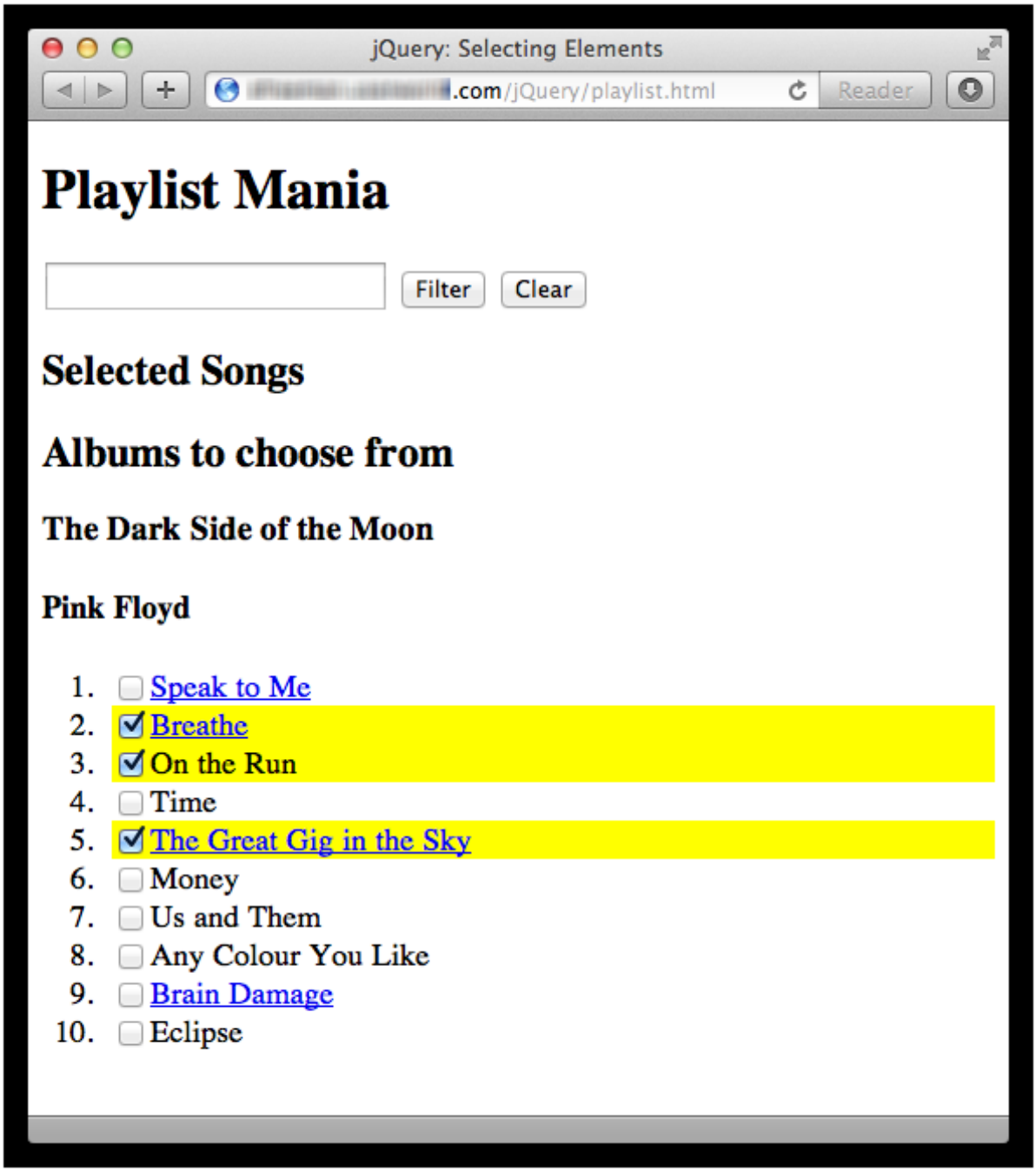
It would be nice if we could not highlight only songs that match a word you enter into the form, and also check the checkbox corresponding to that song, wouldn't it? Let's make it happen!

CODE TO TYPE:

```
$(document).ready(function() {
  $("input#filterButton").click(filterContentfilterContentAndCheck);
  $("input#clearButton").click(clearAll);
});
var clearAll = function() {
  $("*").removeClass("highlight");
  $("form").get(0).reset();
};
var filterContent = function() {
  var lookFor = $("input#containsContent").val();
  clearAll();
  $("li a:contains('" + lookFor + "')").addClass("highlight");
};
var filterContentAndCheck = function() {
  var lookFor = $("input#containsContent").val();
  clearAll();
  $("li a:contains('" + lookFor + "')").parent().addClass("highlight");
  $("li.highlight input:checkbox").attr("checked", true);
};
```

 Save the file and 

preview. Enter a word and click **Filter**. Now, songs that match the word you enter are highlighted and checked:



The first two lines of **filterContentAndCheck()** are the same as our previous **filterContent()** function, but we do something slightly different in the third line:

OBSERVE:
`$("li a:contains('" + lookFor + "')").parent().addClass("highlight");`

We select all the <a> elements whose content contains the text typed into the text input, and then, for each of those <a> elements, we get the **parent** element. In this case, the parent element of a matching <a> element is an <li> element, so we add the "highlight" class on the <li> instead of the <a> element. This is an important point to consider for the next line of code.

OBSERVE:
`$("li.highlight input:checkbox").attr("checked", true);`

Here, we select all the ** elements we just highlighted** by using the class as a selector. **This selector**

says "select all the <li> elements with the highlight class." We just added that class to the matching <li> elements in the line before, so we're using this class not only to change the appearance of the elements, but also as a marker for subsequent selection.

The other part of the selector **selects all the checkbox <input>** elements that are nested within ** elements with the "highlight" class**. Then we **set the "checked" attribute of those input elements to true**, which causes the checkboxes to become checked. Setting this attribute is like adding the "checked" attribute to the checkbox <input> element in HTML. Notice that we use the **attr()** method like we used earlier to *get* the value of an attribute. but now, we pass both the name of the attribute and a value for that attribute. If you pass **attr()** only the name of an attribute, it retrieves the value; if you pass it both the name of the attribute and a value, it sets the value of that attribute.

Go ahead and enter a word to filter. Note which songs are highlighted, and then click the "Clear" button. The checkboxes next to the songs that were highlighted are still checked. What's going on? Why isn't the form reset in the **clearAll()** function working?

It turns out that when you set an attribute using this method, to the form, it looks like the checkboxes should be checked by default. So, when you reset the form using the **reset()** method (which is just like clicking a reset button on a form, if you added one in HTML), the form thinks those checkboxes should stay checked. We can remove those checks by setting the "checked" attribute back to false:

CODE TO TYPE:

```
var clearAll = function() {
  $("*").removeClass("highlight");
  $("form").get(0).reset();
  $("form input:checked").attr("checked", false);
};
```

 Save the file and 

preview. Now, when you enter a word to filter, the checkboxes corresponding to the matching songs are checked; and when you click **Clear**, those checkboxes go back to unchecked.

Here, we use the selector "form input:checked" to select all checked input elements in the form, and then use the **attr()** method to set the "checked" attribute back to false.

Adding New Elements to Your Page

As a final feature in our Playlist builder application, we'll actually add the songs you select by filtering or manually checking to a playlist. We've already got a heading, "Selected Songs," and a list with the id "playlist" ready to go. We just need a way to add the selected songs to the list. First, edit your **playlist.html** HTML to add another button:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Selecting Elements</title>
  <meta charset="utf-8">
  <style>
    .highlight {
        background-color: yellow;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>

  </script>
</head>
<body>
  <h1>Playlist Mania</h1>
  <form>
    <input type="text" id="containsContent">
    <input type="button" value="Filter" id="filterButton">
    <input type="button" value="Add to Playlist" id="addButton">
    <input type="button" value="Clear" id="clearButton">
    <h2>Selected Songs</h2>
    <ul id="playlist">
    </ul>

    <h2>Albums to choose from</h2>

    <div class="album">
      <h3>The Dark Side of the Moon</h3>
      <h4>Pink Floyd</h4>
      <ol>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Speak_t
o_Me" data-length="1:30">Speak to Me</a></li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Breathe
_(Pink_Floyd_song)" data-length="2:43">Breathe</a></li>
        <li><input type="checkbox"><a data-length="3:36">On the Run</a></li>
        <li><input type="checkbox"><a data-length="7:01">Time</a></li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/The_Gre
at_Gig_in_the_Sky" data-length="4:36">The Great Gig in the Sky</a></li>
        <li><input type="checkbox"><a data-length="6:22">Money</a></li>
        <li><input type="checkbox"><a data-length="7:46">Us and Them</a></li>
        <li><input type="checkbox"><a data-length="3:25">Any Colour You Like</a>
</li>
        <li><input type="checkbox"><a href="http://en.wikipedia.org/wiki/Brain_D
amage_(song)" data-length="3:48">Brain Damage</a></li>
        <li><input type="checkbox"><a data-length="2:03">Eclipse</a></li>
      <ol>
    </div>
  </form>
</body>
</html>
```

Next, update the jQuery to add a click handler, **addToPlaylist()** for the "addButton":

CODE TO TYPE:

```
$(document).ready(function() {
  $("input#filterButton").click(filterContentAndCheck);
  $("input#clearButton").click(clearAll);
  $("input#addButton").click(addToPlaylist);
});
var clearAll = function() {
  $("*").removeClass("highlight");
  $("form").get(0).reset();
  $("form input:checked").attr("checked", false);
};
var filterContentAndCheck = function() {
  var lookFor = $("input#containsContent").val();
  clearAll();
  $("li a:contains('" + lookFor + "')").parent().addClass("highlight");
  $("li.highlight input:checkbox").attr("checked", true);
};
var addToPlaylist = function() {
  $("form input:checked").parent().each(function() {
    var song = $(this).text();
    $("ul#playlist").append("<li>" + song + "</li>");
  });
};
```

 Save the file and 

preview. Enter a word in the text input and click **Filter**. Then click **Add to Playlist**, and the songs will appear under the "Selected Songs" heading. Try checking some songs and clicking **Add to Playlist**. Those songs appear in the Selected Songs as well.

As you can see, we've added a new **addToPlaylist()** function, where we add the songs to the "playlist" element in the page. Let's take a closer look at how this function works:

OBSERVE:

```
var addToPlaylist = function() {
    $("form input:checked").parent().each(function() {
        var song = $(this).text();
        $("ul#playlist").append("<li>" + song + "</li>");
    });
};
```

This function is only three lines long, but there's a lot going on! First, we select **all checked <input>**

elements. This will select all the songs that match the word you enter into the text input, as well as any songs you've checked yourself. Then we get the **parent element of the input, which is the <li> element containing the selected <input> element**. We could have multiple <input> elements that match, and like other jQuery methods, **parent()** returns a jQuery object that acts just like an array of parent elements: the parent elements of all those that matched the selector. We call a function on **each of those parent <li> elements**.

Just like before, **each()** is used to apply a function to each element in an array of matching jQuery objects. Also like before, inside this function, we can refer to each selected element as **this**. So, if your selector matches say, two songs, "Breathe" and "On the Run," for the first match, **this** will be the <li> containing the song "Breathe," and for the second match, **this** will be the <li> containing the song "On the Run."

We want to use the jQuery **text()** method to get the text content of the <li> element, so we wrap **this** with **$()** to convert it to a jQuery object, and then call **text()**, and store the result in the variable **song**. Now you might be thinking, "but the only text content (that is, the song name) in each list item is actually text content in the nested <a> element!" Well, let's take a look at the description of the **text()** method from the jQuery.com site:

***text()**: Get the combined text contents of each element in the set of matched elements, including their descendants.*

So the result of calling **text()** on the selected <li> element is the text content of all of <li> element's nested elements (as well as any text that is in the <li> itself, but we don't have any). In our case, that content is the song name from the nested <a> element.

Finally, we **select the "playlist" element**, and **add a new <li> element** to it, with the name of the song, using the **append()** method. **append()** takes HTML, and appends it to the selected element, in this case, the "playlist" . The HTML we pass to **append()** is a list item containing the name of the song. So now you see the song in the playlist!

Congratulations—you've built a Playlist builder application with jQuery! Try adding another album of songs to your page. As long as you use the correct format (list items containing checkbox inputs, and songs in <a> elements with a "data-length" attribute), it will work with those new songs as well. In the process of building this application, you've learned new ways to select and filter elements with jQuery, as well as how to use the method **each()** to iterate over a jQuery object, and **attr()** to both retrieve and set the value of an attribute of an element. Nicely done!

Manipulating Element Style

Lesson Objectives

- You will manipulate elements in jQuery.
- You will be able to change an element's size and alter its appearance, using various jQuery methods.

Using jQuery to Style your Elements

Style plays an important role in the design and behavior of web pages. The style of your page gives it an identity. We can use JavaScript and jQuery to change the style of web pages dynamically, to include all kinds of cool stuff, like menus, animation, content updates, interactive games, and more. In this lesson, we'll explore jQuery methods that can update the style of elements in your pages to produce a variety of different effects.

Getting Started with the css() Method

We'll start with a simple web page that has just a single element:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Selecting Elements</title>
  <meta charset="utf-8">
  <style>
    div {
      text-align: center;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="style.js"> </script>
</head>
<body>
  <div>I'm a div in need of some style!</div>
</body>
</html>
```

Save this file in your **/jQuery** folder as **style.html**. Go ahead and Preview the file; you'll see a plain page with a bit of text. The only style we apply to the with CSS here is center alignment. We'll apply more style with jQuery in a bit.

We added a link from the HTML to a JavaScript file, **style.js**. Let's create that file now:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css("border-width", "2px");
    $("div").css("border-style", "solid");
    $("div").css("border-color", "black");
};
```

Save the file in your **/jQuery** folder as **style.js**. Preview Preview the HTML file; you'll see a black border around the element:

We're using the jQuery **css()** method to add the border to the . We've already used this method a couple of times before to add style to an element. Here, we use it to the set the border property of the to be 2px in width, solid in style, and black in color.

You can also use the **css()** method to *get* the value of a CSS property of an element. Try this:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css("border-width", "2px");
    $("div").css("border-style", "solid");
    $("div").css("border-color", "black");
    $("div").mouseover(function() {
        var borderWidth = $("div").css("border-width");
        $("div").html("My border width is: " + borderWidth);
    });
};
```

Save your **style.js** file and Preview preview your **style.html** file. Make sure you reload the page if you're using the same tab or window. Now, mouse over the ; its contents change to display the value of the border-width property: 2px. Notice that the value is a string, "2px", not a number, "2". We'll return to this point shortly.

Note In some browsers (such as Firefox), you may need to specify *which* border's width you want the css() function to retrieve. If the mouseover doesn't work as shown above, try changing **var borderWidth = $("div").css("border-width")** to **var borderWidth = $("div").css("border-top-width")**. You can still *set* the border-width with one command.

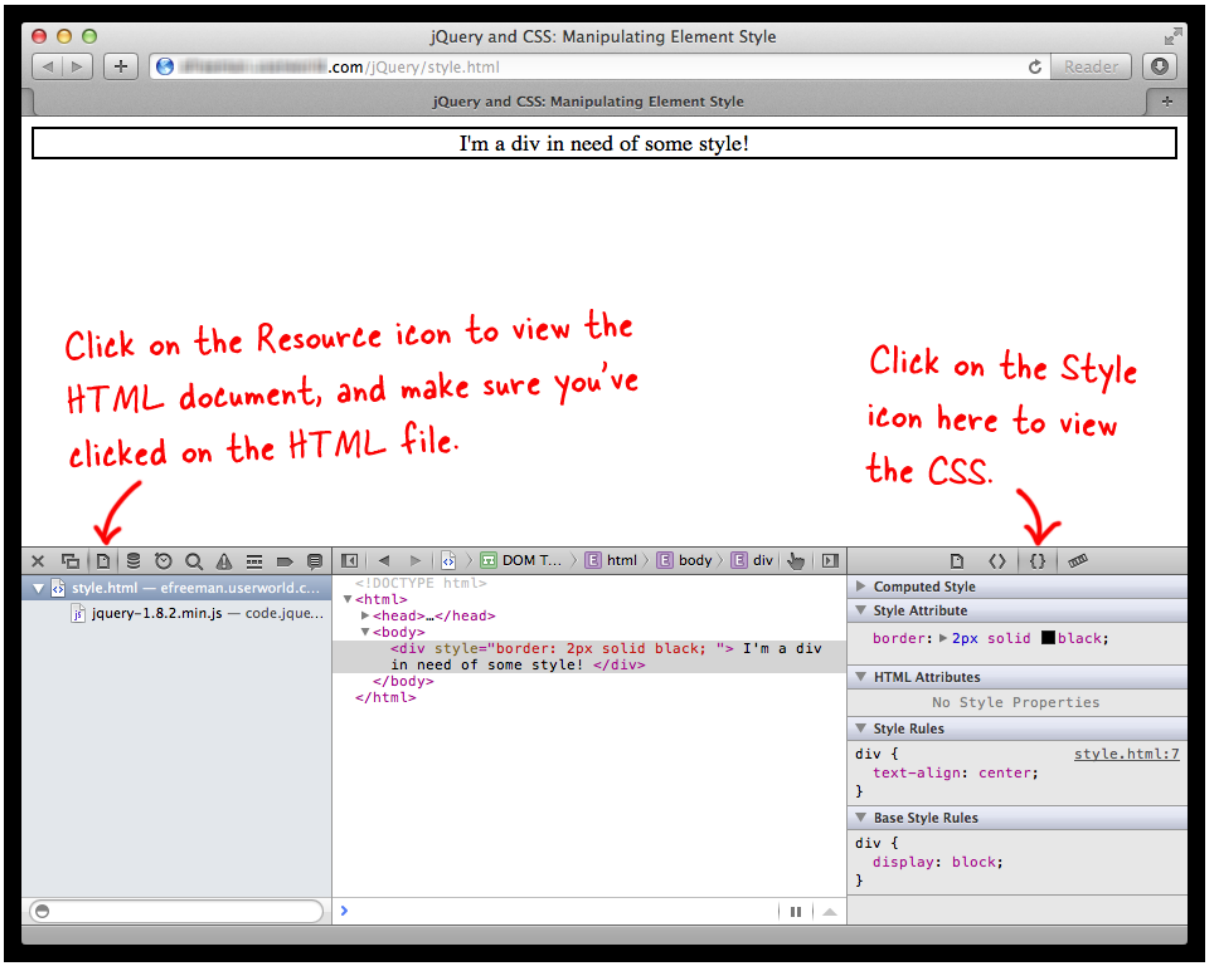
Here, we created a handler function for mouse-over events using the jQUery **mouseover()** method (similar to how we used **click()** before), and we pass the handler function directly into the method (just like we've been doing with **ready()**). In the handler function, which gets called whenever you place your mouse over the , we get the value of the CSS **border-width** property, which is 2px (because we set it to that in the first line of the **init()** function). We then change the content of the element from "I'm a div in need of some style" to a string containing the value of the border width.

Viewing Your Style with the Web Inspector Developer Tool

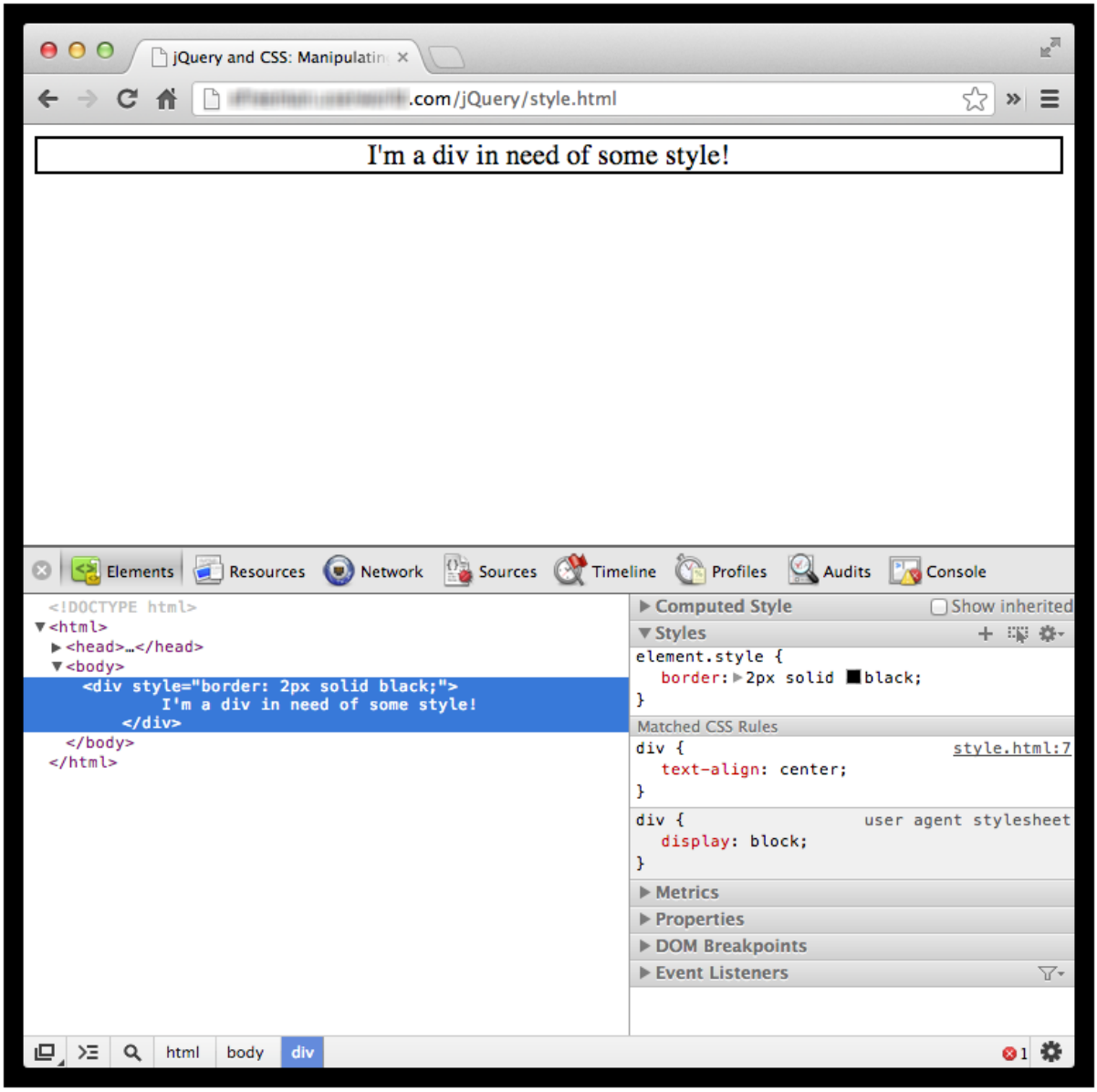
When you're learning how to manipulate the style of elements dynamically through JavaScript and jQuery, it's

helpful to know how to use the Element Inspector that modern browsers include in the Developer tools. You'll also want to test your web applications in all the major browsers, so you want to become familiar with the Web Inspector in each major browser. The Web Inspector is a great debugging tool, and it allows you to verify that your code is working as you expected.

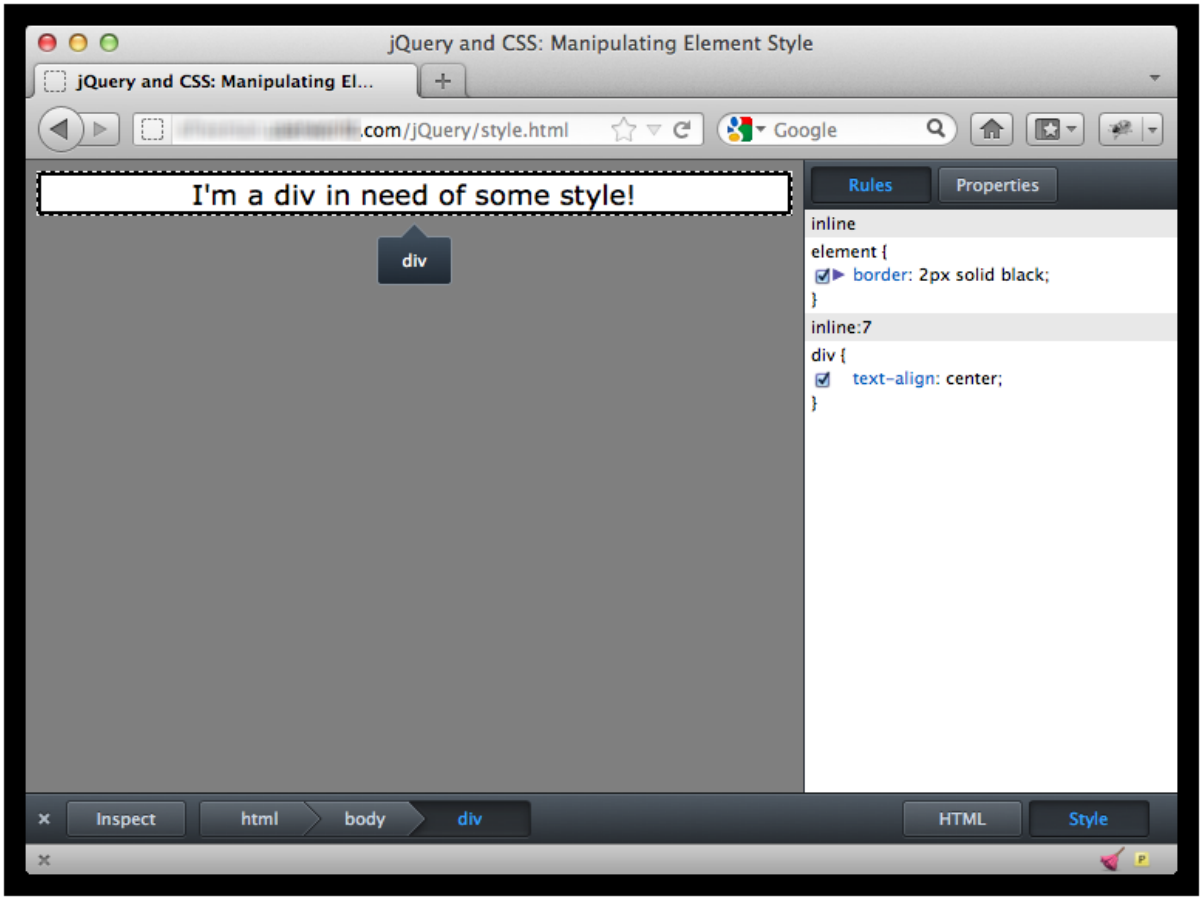
To enable the Web Inspector in Safari 6, select **Develop | Show Web Inspector**. The Developer Tools window appears. Make sure you've selected the Resource icon (on the left tab menu) and the Style icon (on the right tab menu) so you see the correct options. Select the file **style.html** under Resources, and the select the in the area where you see the HTML for your page (just click anywhere on the), then you can view the style under "Style Attribute" in the right third of the Web Inspector window.



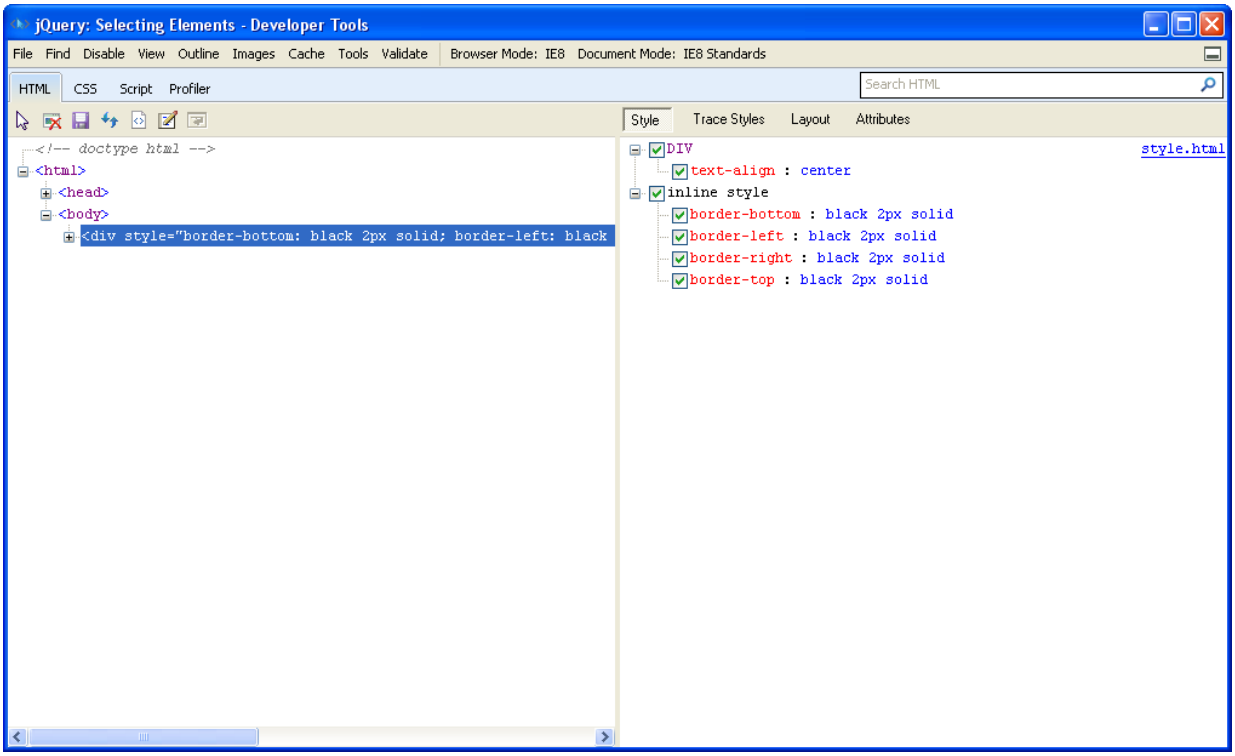
The Web Inspector in Chrome is similar to that in Safari 6. To access the developer tools, select **View | Developer | JavaScript Console**, and click on the **Elements** tab. Make sure you're looking at the Styles section on the right side of the Inspector tool.



In Firefox, choose **Tools | Web Developer | Inspect**. Then, click on the element, and make sure the **Rules** tab is selected on the right side of the Inspector tool.



In Microsoft Internet Explorer, select **Tools | Developer Tools**, and click on the element.



Note The menu options and the web inspector tools in the browsers change frequently, often with each new release of the browser, so the menus and tools may look different by the time you read this, but the basics should be the same.

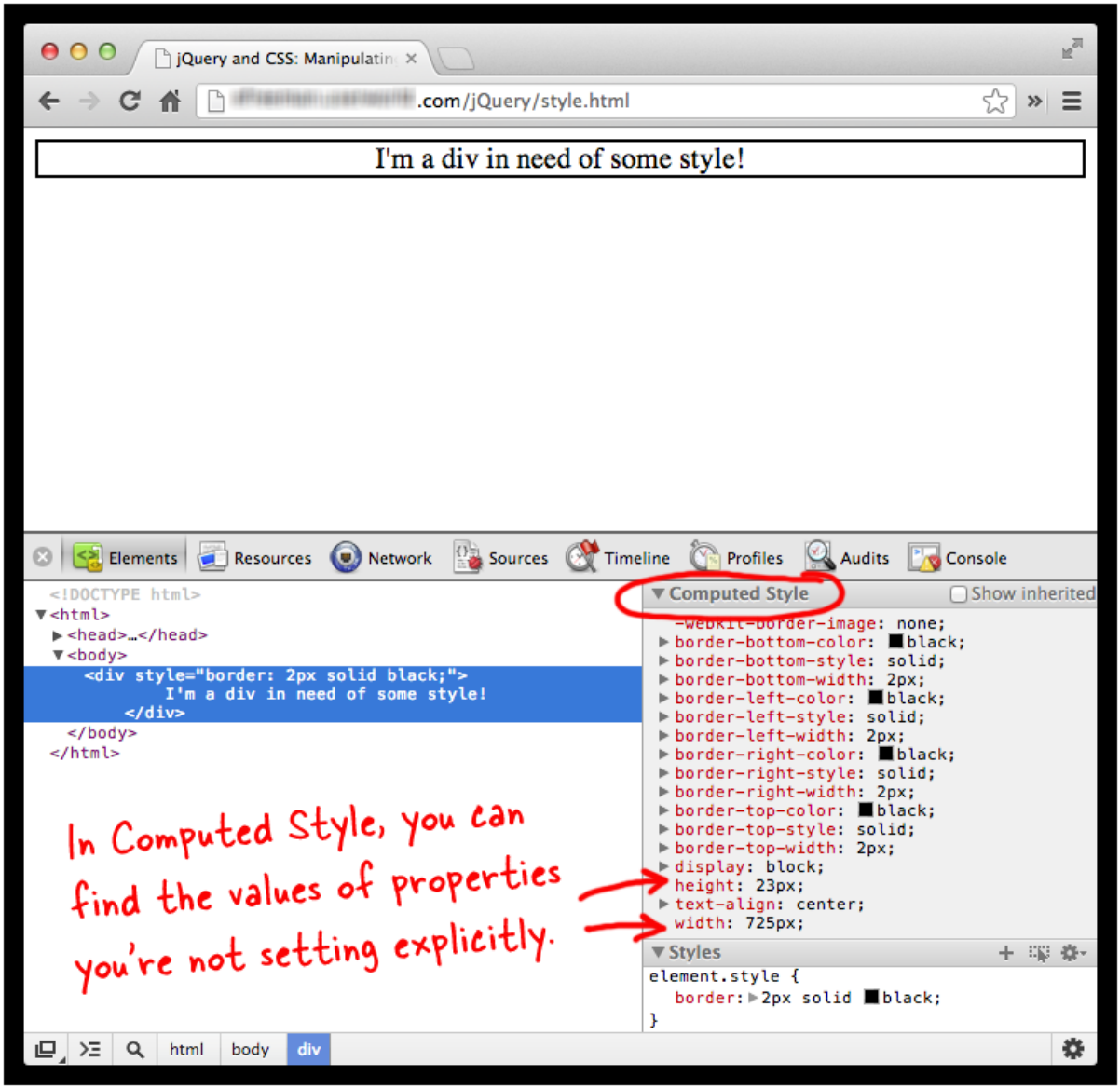
The Box Model

Let's do a quick review of the CSS box model. The CSS box model describes how elements are formatted in a web page, and is key to understanding how style is applied to elements.

The box model describes an element as set of nested boxes composed of the element's content, padding, border, and margin.

Some elements, like <p> and <h1>, have default values set for padding and margin, while for other elements, like , those values are not usually set or are set to 0.

When you compute the width of the element, you add together the left margin, border, and padding, the right margin, border, and padding, and the width of the content itself. The same goes for the height (except of course, you're adding the top and bottom property values). Often times you won't set the width or height of an element explicitly, but if you need to, you can with CSS or with jQuery (we'll see how later in this lesson). If you ever need to know what those values are, you can use the Web Inspector Tool, and look at the "Computed Style" section on the right side of the tool. For example, in Chrome, you can view the values for the width and height of our in "Computed Style". (Each of the other browsers has a way to view "Computed Style" as well).



The Box Model determines how elements look and behave in the page. For example, if you add a background image or background color to an element, that image or color shows behind the content and the padding, but not the margin. You'll get the hang of how the Box Model works in different situations as you become more familiar with HTML, CSS, and jQuery.

Units of Measurement and jQuery css()

When we write CSS to set the padding, border, and margin properties of an element, we can use one of several different units of measurements to define the values of these properties, including em, px, and %.

In our previous example, we used the jQuery **css()** method to get the width of the border of a like this:

OBSERVE:

```
var borderWidth = $("div").css("border-width"); // or "border-top-width"
```

The result (the value in **borderWidth**) is a string, "2px." So, what if you want to convert that value to a number? One way to do that is with the JavaScript function, **parseInt()**. Let's use this function to change the width of the border of our element each time we mouse over it:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css("border-width", "2px");
    $("div").css("border-style", "solid");
    $("div").css("border-color", "black");
    $("div").mouseover(function() {
        var borderWidth = $("div").css("border-width"); // or "border-top-width"
        var borderWidthValue = parseInt(borderWidth);
        borderWidthValue = borderWidthValue + 2;
        $("div").css("border-width", borderWidthValue);
        borderWidth = $("div").css("border-width");  // or "border-top-width"
        $("div").html("My border width is: " + borderWidth);
    });
};
```

 Save **style.js**, and 

preview your **style.html** file. Now, mouse over the a few times. The border (in FireFox, the top border) gets 2px wider, each time.

When we use **parseInt()** on a string like "2px," the function parses the numbers in the string until it reaches the first non-numerical character, in this case "p," and returns the numbers up to that point as an integer. This is a quick and direct way to convert our string to a number; it works even if your browser returns "2em" or "2%" or even "200px"!

Once we have a numerical value for our border-width, we can add 2 to it, and then set the value of the border-width property to that new value. Unlike in the first line of the **init()** function, where we set the border-width to "2px", here, we just use the number in the variable **borderWidthValue** for the value of the border-width property:

OBSERVE:

```
$("div").css("border-width", borderWidthValue);
```

Because the border-width was previously set using "px," the numerical value we provide in **borderWidthValue** is converted automatically to "px."

Setting Multiple CSS Properties with css()

So far, we've used the **css()** method to get the value of a CSS property, and set the value of a few CSS properties, one at a time. You can also use this method to set multiple CSS properties at once. Let's use the **css()** method to set all three of the properties we were setting before, border-style, border-width, and border-color, at once.

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css("border-width", "2px");
    $("div").css("border-style", "solid");
    $("div").css("border-color", "black");
    $("div").mouseover(function() {
        var borderWidth = $("div").css("border-width");
        var borderWidthValue = parseInt(borderWidth);
        borderWidthValue = borderWidthValue + 2;
        $("div").css("border-width", borderWidthValue);
        borderWidth = $("div").css("border-width"); // or border-top-width, for some browsers
        $("div").html("My border width is: " + borderWidth);
    });
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black"});
};
```

 Save, and 

preview your **style.html** file. We removed the code that increases the border width when you mouse over the . The should look exactly the same as before (that is, it has a black 2 pixel solid border), but now you're setting all three properties with one method call.

To set multiple CSS properties with the **css()** method, you create a literal object. Just like any other literal JavaScript object; you use the curly brackets, **{** and **}**, to delimit the object, and create one or more properties and values, with each property/value pair separated by a comma.

OBSERVE:

```
{"border-width": "2px",
 "border-style": "solid",
 "border-color": "black"}
```

Try using the **css()** method to set some other properties on the like the padding (for example, "padding": "10px"), color, background-color, and others. Make sure you understand how to create this object correctly. If you omit a comma (or put one in the wrong place), it can cause the entire **css()** method to fail.If you run into trouble, check your syntax.

Sizing Elements with jQuery

We've been changing the style of a element in our web page using the **css()** method. You might think that in order to manipulate the size of elements, we'd use this same method and add properties like "width" and "height" to the CSS with **css()**—but jQuery offers a few methods that make it even easier to manipulate the width and height of elements. Let's experiment a bit with the size of our .

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black",
                  "padding": "10px",
                  "margin": "10px"});
    $("div").width("300px");
};
```

 Save the file, and  preview your **style.html** file. We added some padding and a margin (using the **css()** method, so there's a bit more space between the content and the border (created by the padding), as well as between the border and the side of the web page (created by the margin). Then we used the **width()** method to set the width of the to 300 pixels. The should take up only a portion of the width of the page, rather than stretching across the entire page (remember that, by default, block elements like take up the entire width of the page unless you change the width specifically).

We can improve this code by chaining the method calls like this:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black",
                  "padding": "10px",
                  "margin": "10px"}).width("300px");
    $("div").width("300px");
};
```

 Save the file and  preview your **style.html** file. The result is exactly the same as the last preview, but this code is better, because now we retrieve the object once instead of twice, which makes our code just a tiny bit more efficient.

Notice that we passed a string, "300px," to the **width()** method. Try passing a number, 300, instead. Does it still work?

What happens if you pass "30%" instead? Let's try it:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black",
                  "padding": "10px",
                  "margin": "10px"}).width("300px""30%");
};
```

 Save the file and  preview your **style.html** file. Try resizing the browser window; now, instead of keeping the width fixed at 300px, the changes width relative to the size of the window: the will always be 30% the width of the page.

Let's try changing the height of the too:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black",
                  "padding": "10px",
                  "margin": "10px"}).width("30%").height(300);
};
```

 Save the file and  preview your **style.html** file. Now your is 300 pixels high. Now change the size of the browser window to see how behaves. Experiment with different values for the height and width.

Note

What happens if you change the height to "30%"? Does the height of the change? Probably not, because when you specify height as a percentage, that percentage is relative to the height of the parent element. In our case, that parent element is the <body> element. By default the body takes up the full width of the page, but its height is determined by its content unless you specify a height in pixels.

We can also use the **width()** and **height()** methods to *get* the width and height of an element:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black",
                  "padding": "10px",
                  "margin": "10px"}).width("30%").height(300);
     alert("The width, height of the div is: " + $("div").width() + ", " + $("di
v").height());
};
```

 Save the file and  preview your **style.html** file. An alert indicates the width and height of your element. Because the height is specified as 300 pixels, the alert shows that the height is 300, but because the width is specified as a percentage, the alert gives a different value depending on the width of your browser window. Try resizing the browser window and then reloading the page, and you'll see a different value for the width. Remember, when you specify the width using a percentage, the width of the element is a percentage of the parent element, in our case, the <body>, which takes up the entire width of the page by default.

Notice that using the **width()** and **height()** methods to get the values for width and height returns a number, not a string with the units (that is, you get 300 back from **height()** rather than "300px"). This comes in handy when you use jQuery to change the size of your elements, because you don't have to worry about parsing the value from a string like we did before for the border width.

Other Size Methods

Along with **width()** and **height()**, jQuery offers a few other methods to manipulate the size of your elements:

Method	Description
width()	Get the current computed width for the first element in the set of matched elements. This is the width of the content only.

height()	Get the current computed height for the first element in the set of matched elements. This is the height of the content only.
innerWidth()	Get the current computed width for the first element in the set of matched elements, *including padding, but not border*.
innerHeight()	Get the current computed height for the first element in the set of matched elements, *including padding, but not border*.
outerWidth([includeMargin?])	Get the current computed width for the first element in the set of matched elements, *including padding and border, and optionally margin*.
outerHeight([includeMargin?])	Get the current computed height for the first element in the set of matched elements, *including padding, border, and optionally margin*.

Note that **outerWidth()** and **outerHeight()** can include the margin in the computation of the width and height respectively, if you pass **true** to the method.

Take a look at the documentation for each of these methods and make sure you understand the differences. Try using them in your code, and see the different values you get back depending on which method you use. Use the Element Inspector in your browser's console to verify the values you are seeing with jQuery.

Adding, Removing, Querying, and Toggling Classes

You've already used the **addClass()** and **removeClass()** methods to add and remove a class from an element (we used these methods when we created the Playlist application). Let's create a highlight class for our . Update the CSS in **style.html** to include the new class:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Selecting Elements</title>
  <meta charset="utf-8">
  <style>
    div {
      text-align: center;
    }
    .highlight {
      background-color: lightyellow;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="style.js"> </script>
</head>
<body>
  <div>I'm a div in need of some style!</div>
</body>
</html>
```

Now, let's use jQuery to add that class to the . In addition, we'll write a click handler function to test to see if the has a class. If it does, we'll remove the highlight; if the doesn't have the class, we'll add the highlight.

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black",
                  "padding": "10px",
                  "margin": "10px"}).width("30%").height(300).addClass("highligh
t");
     alert("The width, height of the div is: " + $("div").width() + ", " + $("di
v").height());
     $("div").click(function() {
        if ($(this).hasClass("highlight")) {
            $(this).removeClass("highlight");
        } else {
            $(this).addClass("highlight");
        }
    });
};
```

 Save the file and 

preview your **style.html** file. When you first load the page, you see the highlight (which is just a light yellow background color) on the . Then, if you click on the , the highlight goes away. Click again, and it reappears (like a light switch!).

Notice that we use **$(this)** in the click handler function; **$(this)** is the we clicked on. (We know it's the right one because there's only one in our page! If you have more than one, you'll need to be a bit craftier and use an id on your elements).

In the click handler function, use the **hasClass()** method to check to see if the has the class "highlight." If the element already has the class, then **hasClass()** returns true; otherwise it returns false. So, if our has the class "highlight," we remove it using **removeClass()**; if it doesn't have the class, we add it using **addClass()**. This creates the toggle effect.

A Better Way to Toggle

It turns out that this toggling effect is so useful that jQuery added a method **toggleClass()**. Let's change our code to use this method instead:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black",
                  "padding": "10px",
                  "margin": "10px"}).width("30%").height(300).addClass("highligh
t");
    $("div").click(function() {
        if ($(this).hasClass("highlight")) {
            $(this).removeClass("highlight");
        } else {
            $(this).addClass("highlight");
        }
        $(this).toggleClass("highlight");
    });
};
```

 Save the file and  preview your **style.html** file. The page looks and behaves the same as before, but we replaced four lines of code with one using **toggleClass()**. As you can probably guess, **toggleClass()** adds a class to an element if it's not already there, and removes the class if it is.

Showing and Hiding Elements with jQuery

Another common use of CSS is to either hide or show elements. Once again, jQuery has methods to make this convenient! We'll add two links above the in our page that we'll use to show and hide the . First, update **style.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Selecting Elements</title>
  <meta charset="utf-8">
  <style>
    div {
      text-align: center;
    }
    .highlight {
      background-color: lightyellow;
    }
    p {
      margin: 20px 10px 20px 10px;
    }
    span {
      cursor: pointer;
      color: blue;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="style.js"> </script>
</head>
<body>
  <p><span class="show">Show</span> | <span class="hide">Hide</span></p>
  <div>I'm a div in need of some style!</div>
</body>
</html>
```

 Save and  preview the file (**style.html**). Two blue links are added at the top of the page. Notice that when you move your mouse over the links, the pointer turns into the pointing-hand icon. This is good feedback for the user because it lets them know that they can click on these words. However, if you click on them now they don't do anything just yet. Go ahead and update your **style.js**:

CODE TO TYPE:

```
$(document).ready(function() {
    init();
});

var init = function() {
    $("div").css({"border-width": "2px",
                  "border-style": "solid",
                  "border-color": "black",
                  "padding": "10px",
                  "margin": "10px"}).width("30%").height(300).addClass("highligh
t");
    $("div").click(function() {
        $(this).toggleClass("highlight");
    });
    $("span").click(function() {
        $div = $("div");
        if ($(this).hasClass("show")) {
            $div.show();
        } else {
            $div.hide();
        }
    });
};
```

 Save the file and  preview your **style.html** file again. Now when you click **Hide**, the disappears, and when you click **Show**, it reappears! Magic? No, not really, just jQuery.

Let's go over the code:

OBSERVE:

```
$("span").click(function() {
    if ($(this).hasClass("show")) {
        $("div").show();
    } else {
        $("div").hide();
    }
});
```

We added a click handler function to *both* of our elements. Take a quick look back at the HTML and you'll see we've got two elements, one each for Show and Hide. We can add the function to both elements at the same time! Remember that when you **select an element** in jQuery, jQuery returns an array of elements. Here we get back the two elements from the selector **$("span")**. When we use the **click()** method to add a click handler, the same handler is added to *both* elements.

Now in this click handler function, **$(this)** will be set to whichever element the user clicks on. We can **test to see if the element has the class "show"**; if it does, we know the user clicked on the **Show** . In that case, we use the **show() method to show the **. If the ** element** doesn't have the "show" class, then we know the user clicked on **Hide**. In that case we **hide the with the hide() method**.

Show, Hide, and Display: none

When you use the jQuery **show()** and **hide()** helper methods, be aware that the way they show and hide an element is by setting the display property of the element in CSS. You can hide an element by setting its display property to "none," like this:

OBSERVE:

```
div {
    display: none;
}
```

As you may know, when you set the value of the display property to "none," this effectively removes the element from the layout flow of the page (so it no longer takes up space in the page). This hides the element, but it may also affect how the other elements in the page are laid out, which might not be what you want!

If you want the element to be hidden, but remain in the flow of the layout (and take up space), rather than using **hide()** and **show()**, you'll need to set the CSS property **visibility** instead. To hide an element using the **visibility** property, set the value to "hidden":

OBSERVE:

```
$("div").css("visibility", "hidden");
```

and to show the element, set the value to "visible":

OBSERVE:

```
$("div").css("visibility", "visible");
```

We'll see an example of using visibility instead of **show()** and **hide()** later in the course.

Well done! You've learned lots about manipulating elements and their CSS with jQuery in this lesson. You can start to build some really interesting applications with what you've learned: change elements' size, and alter their visibility and other properties, using the methods you've learned about. Maybe you already have some ideas for games or fun effects you'd like to try out! Keep practicing and work on any homework assignments. I'll see you in the next lesson!

Looping, Functions, and This

Lesson Objectives

- You will use the **this** variable with **each()**.
- You will use functions with **each()**

.

Looping, Functions, and This

You already know that you can add the "highlight" class to every <a> element in a page, like this:

OBSERVE:

```
$("a").addClass("highlight");
```

As you've seen, the way this works is that the selector, **$("a")** selects all the <a> elements and returns them as a jQuery object that acts like a special array. Then, when we call the **addClass()** method, jQuery applies that method to every element in the array, adding the class "highlight" to each of the selected <a> elements.

There are many of these jQuery methods we can use to manipulate elements in one way or another by chaining method calls together like this. For instance, you might then call another method to show those <a> elements (if they they were hidden):

OBSERVE:

```
$("a").addClass("highlight").show();
```

But sometimes we need to do something a little more complicated to the elements selected by a selector, and that's where **each()** comes in. For instance, in our earlier Playlist application, we used **each()** to check the value of each data-length attribute of an array of selected <a> elements to see if the value was greater than 3 minutes:

OBSERVE:

```
var filterLength = function() {
    $("a[data-length]").each(function() {
        var length = $(this).attr("data-length");
        var min = (length.split(":"))[0];
        var min = parseInt(min);
        if (min > 3) {
            $(this).addClass("highlight");
        }
    });
});
```

In this lesson, we'll explore **each()** a little more deeply. We'll take a look at how the **this** variable behaves with **each()** and also how to use functions with **each()**.

A Deeper Dive into each()

To get started, create a new HTML file using code:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Looping, Functions and This</title>
  <meta charset="utf-8">
  <style>
    .highlight {
      background-color: lightyellow;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="looping.js"> </script>
</head>
<body>
  <header>
    <h1>Computer Programming</h1>
    <h2>From: Wikipedia entry on
      "<a href="http://en.wikipedia.org/wiki/Programming">Programming</a>"</h2>
  </header>
  <section>
    <article>
      Computer programming (often shortened to programming or coding) is the
      process of
      <a href="http://en.wikipedia.org/wiki/Software_design">designing</a>,
      writing,
      <a href="http://en.wikipedia.org/wiki/Software_testing">testing</a>,
      <a href="http://en.wikipedia.org/wiki/Debugging">debugging</a>,
      and maintaining the
      <a href="http://en.wikipedia.org/wiki/Source_code">source code</a> of
      <a href="http://en.wikipedia.org/wiki/Computer_program">computer programs<
/a>.
      This source code is written in one or more
      <a href="http://en.wikipedia.org/wiki/Programming_language">programming la
nguages</a>
      (such as
      <a href="http://en.wikipedia.org/wiki/Java_(programming_language)">Java</a
>,
      <a href="http://en.wikipedia.org/wiki/C%2B%2B">C++</a>,
      <a href="http://en.wikipedia.org/wiki/C">C#</a>,
      <a href="http://en.wikipedia.org/wiki/Python_(programming_language)">Pytho
n</a>,
      etc.). The purpose of programming is to create a set of instructions
      that computers use to perform specific operations or to exhibit desired
      behaviors.  The process of writing source code often requires expertise
      in many different subjects, including knowledge of the application domain,
      specialized
      <a href="http://en.wikipedia.org/wiki/Algorithm">algorithms</a> and
      <a href="http://en.wikipedia.org/wiki/Logic">formal logic</a>.
    </article>
  </section>
  <section id="links">
    <h1>Links</h1>
    <ul>
    </ul>
  </section>
</body>
</html>
```

 Save this file in your **/jQuery** folder as **looping.html**. Preview the file.

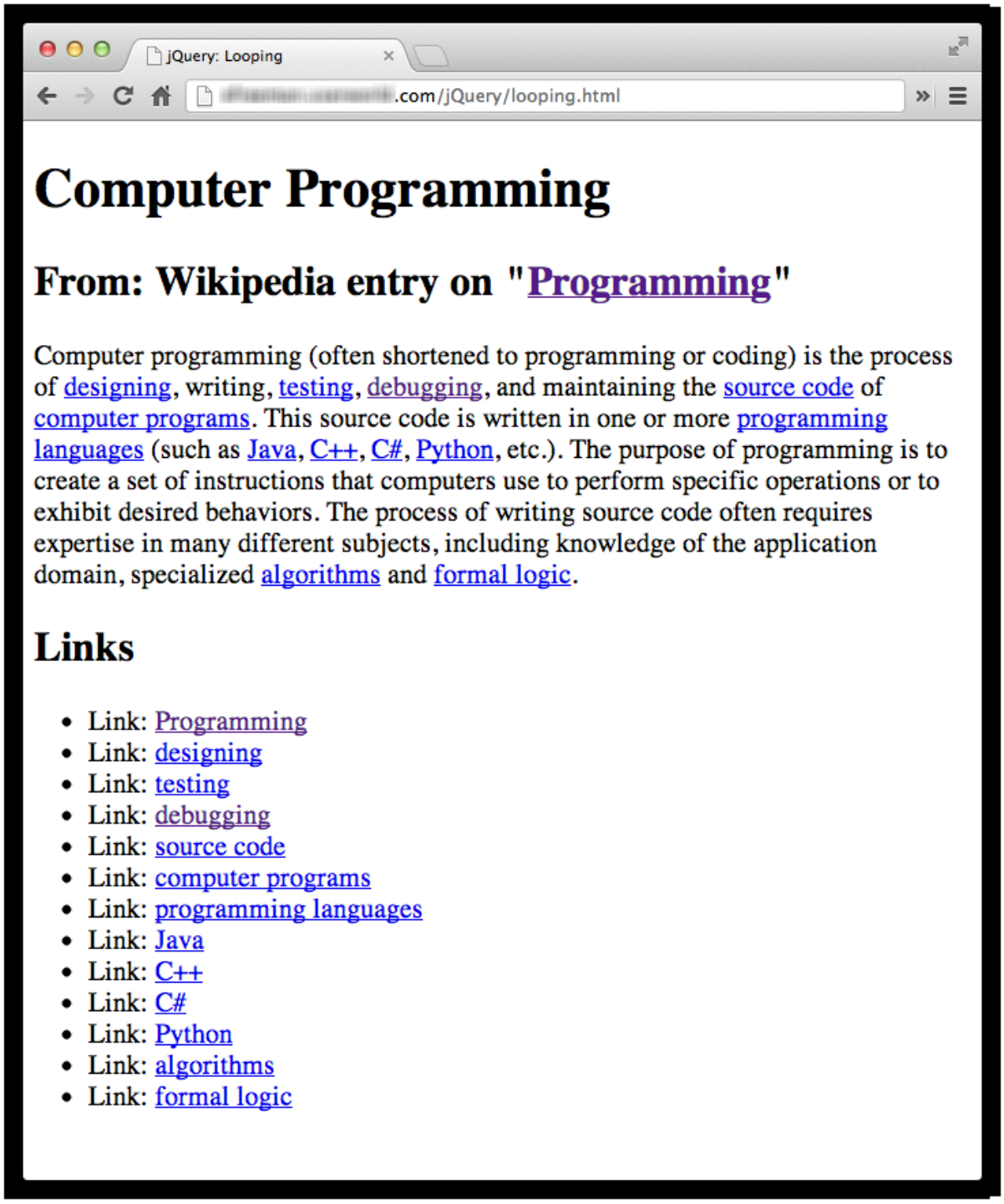
Our goal in this example is to create a list of all the links in the page using jQuery. We've got an empty list ready to go, a <ul> element nested in a <section> with the id "links." Now we just need to write a little code to get each link from the page and add it to the list. Create a new JavaScript file in the editor as shown:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function() {
        $("section#links ul").append("<li>Link: <a href=\"" +
            $(this).attr("href") + "\">" + $(this).text() +
            "</a></li>");
    });
};
```

 Save the file in your **/jQuery** folder as **looping.js**. 

Preview your **looping.html** file, and you'll see that each link in the page has been added to the list in the "links" <section>.



Let's go over the **createLinks()** function:

OBSERVE:

```
var createLinks = function() {
    $("a").each(function() {
        $("section#links ul").append("<li>Link: <a href=\"" +
            $(this).attr("href") + "\">" + $(this).text() +
            "</a></li>");
    });
};
```

First, let's examine our use of **each()** here. We might describe this function in English as, "For **each selected <a> element** in the page, **append a new <li> element** containing a link with the same **"href" attribute** and **text content** to the ** element in the "links" <section>**." In the **each() function**,

we change the <ul> element, using the values of the currently selected <a> element's "href" attribute and text content.

Notice that the **<a> element** we use in the **each() function** is represented by **$(this)** inside the function. We use **$(this)** and not **this** because we want to use the jQuery functions **attr()** and **text()**. **this** represents the DOM <a> element, while **$(this)** is a wrapped version of that DOM element that gives you access to all the jQuery methods you can use on an element object. If you tried to use **this** instead of **$(this)**, you'd get an error. Try it and see!

Finally, recall that **attr()** gets the value of an attribute, in this case the "href" attribute, and **text()** gets the text content of an element, in this case the <a> element we're acting on currently. We use those values to construct a whole new <a> element to add to the "links" list, so the href value and text content of the new links will match those from the rest of the page.

Comparing each() to a JavaScript Loop

each() is the preferred way to loop with jQuery. It's a pattern you'll get used to as you continue to program with jQuery and soon you won't have to think twice about it. It's a little tricky to get used to at first though, so don't worry if it takes a while for now. Let's compare how we'd write this code with a regular JavaScript loop:

OBSERVE:

```
$selectedElements = $("a");
for (var i = 0; i < $selectedElements.length; i++) {
    $("section#links ul").append("<li>Link: <a href=\"" +
        $($selectedElements.get(i)).attr("href") + "\">" + $($selectedElements.g
et(i)).text() +
        "</a></li>");
}
```

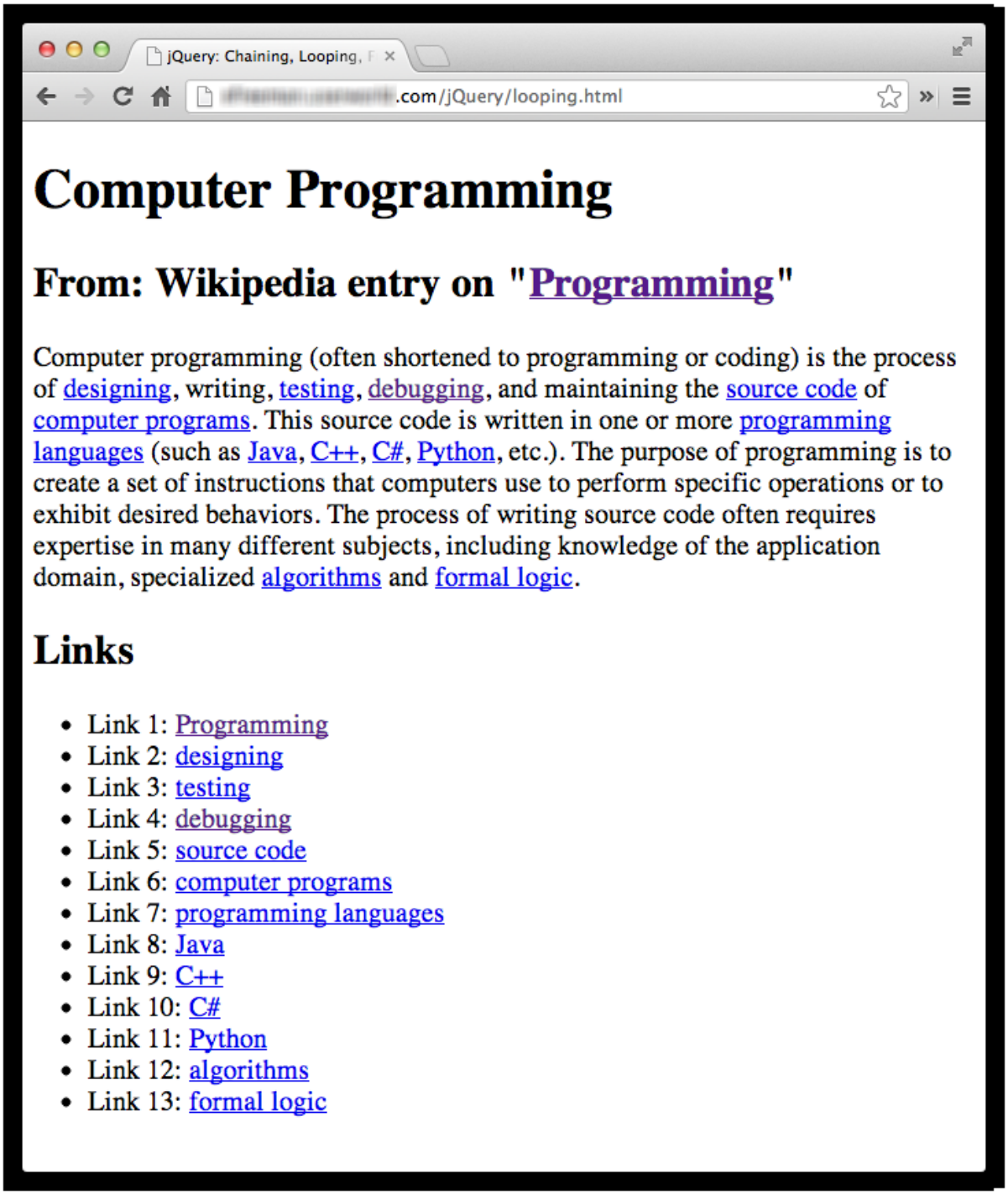
The function that we pass into the **each()** is like the **body of the loop**. The **each()** hides some of the mechanics of the **for** looping variable, but the basic idea is the same in both jQuery and Javascript: each time you iterate through the loop, you execute the statements in the body of the loop (or **each()** function).

For this **for** loop version, first we save the selected elements we get from the selector in a variable named **$selectedElements**. We use a **$** in the name of the variable because it's a jQuery object: an object that acts just like an array of elements. Because this object acts just like an array, we can loop through it like we'd loop through a normal array, and use the **length** property to determine how many elements are in the array (and also when to stop the loop).

Previously we said that if you needed to get an element from the selected elements array, you could use **get()**. We can use that method here too, passing in the index of the element in the array. **get()** returns the DOM element from the array (we used it earlier to unwrap a DOM element), so to use the **attr()** and **text()** methods, we have to re-wrap the DOM element with **$()**. That's why you see the syntax $(**$selectedElements.get(i)**).

It can be tricky to keep track of which is a DOM element and which is a jQuery object at times. In this particular case, you can avoid confusion by using **each()** to loop instead. Still, you want to understand how all of this works behind the scenes so you can maneuver around both types of objects: DOM objects (like elements) and jQuery objects.

We may want to number the links in our "links" list, so the output looks more like this:



That **i** in the JavaScript loop comes in handy for this task. As it turns out, **each()** has a convenient way for us to the get the index of the currently selected element we're using inside the function passed into **each()**. We'll see how to do that next.

Using the each() Function Parameters

So far, we've used **each()** to loop over each element in an array of selected elements by passing a function with no parameters to **each()**, and using **$(this)** to access the currently selected element.

The function we pass to **each()** actually has two parameters we can use: the index of the element from the array of selected elements, and the element itself. Let's see how to use them:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function(i, el) {
        $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
            $(this)$(el).attr("href") + "\">" + $(this)$(el).text() +
            "</a></li>");
    });
};
```

 Save the file and  preview your **looping.html** file. Each link now has a number, beginning with 1, as in the previous screenshot.

Now the **each()** function has two parameters: the index, **i**, and the element, **el**, that's currently being acted on in the loop. If we have fourteen <a> elements in the page, then the **each()** function will be called fourteen times. The first time it's called, **i** will be set to 0, and **el** to the first <a> element in the page; the second time, **i** will be set to 1, and **el** to the second <a> element in the page, and so on.

Because the index starts at 0 rather than 1, we added 1 to the value of **i** to display the link numbers in the page.

We replaced **$(this)** with **$(el)** to get the value of the <a> element being acted upon, but notice that in this case, **$(this)** and **$(el)** are the same thing, so we could use either! However, the value of **el** is a DOM element object, not a jQuery object, so before we use **el**, we wrap it, **$(el)**. That way we can make use of the **attr()** and **text()** methods, just like we do when we're using **this**.

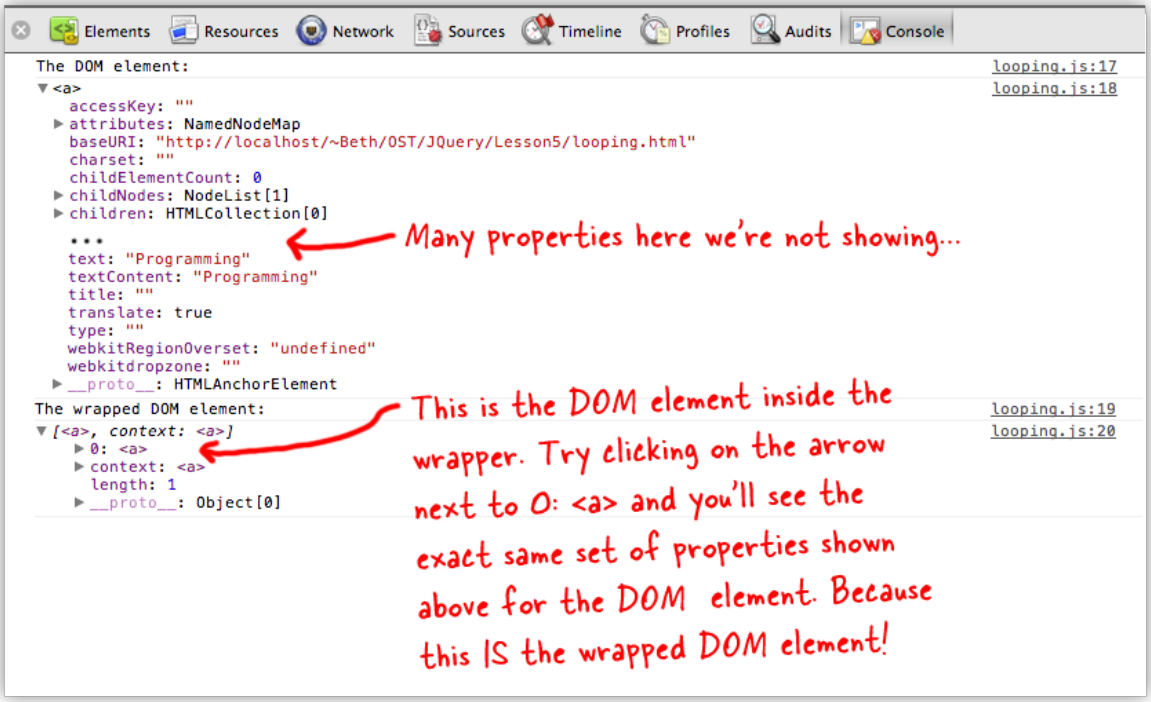
Comparing Wrapped and Unwrapped Objects

To get a real sense of the difference between an unwrapped DOM element object and a wrapped DOM element object (a jQuery object), try adding some log messages to your code like this:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function(i, el) {
        if (i == 0) {
            console.log("The DOM element: ");
            console.log(el);
            console.log("The wrapped DOM element: ");
            console.log($(el));
        }
        $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
            $(el).attr("href") + "\">" + $(el).text() +
            "</a></li>");
    });
};
```

 Save the file and  preview your **looping.html** file. Open the JavaScript console on your browser to look at the console messages (reload the page if you don't see any messages). Here's what you'll see in Chrome (it will be similar in other browsers):



Click the arrow next to the **<a>** element in the top part of the console. You'll see all the properties and methods associated with the <a> element (the DOM element object). Next, click the arrow next to the wrapped object, and then click the arrow next to the **0: <a>**. Again, you'll see all those properties. That's because this is the DOM element inside the jQuery object; that is, it's the wrapped DOM object! Don't worry about the details of what all that stuff is; the important thing to notice is that the DOM element is inside that wrapped object, and also observe the high-level differences between the DOM element version and the wrapped version—the jQuery object.

Go ahead and remove the console messages from your code before we move on to the next section:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function(i, el) {
        if (i == 0) {
            console.log("The DOM element: ");
            console.log(el);
            console.log("The wrapped DOM element: ");
            console.log($(el));
        }
        $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
            $(el).attr("href") + "\">" + $(el).text() +
            "</a></li>");
    });
};
```

Pay Attention to This

In the **each()** function in our example, **$(el)** and **$(this)** are the same: they are both the jQuery object that wraps the <a> DOM element object currently being acted upon in the loop. Change your code to use **$(this)** again, so we can see how **$(this)** behaves when we change things later:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function(i, el) {
        $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
            $(el)$(this).attr("href") + "\">" + $(el)$(this).text() +
            "</a></li>");
    });
};
```

 Save the file and  preview your **looping.html** file, just to make sure your code still works and you see the links in the list.

Now, suppose you decide to move the code in the body of the **each()** function into a separate function. You might do this if you have a lot of code in the **each()** function, or if you need to call a helper function that you've already written to do part of the computation:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function(i) {
        $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
            $(this).attr("href") + "\">" + $(this).text() +
            "</a></li>");
        addLink(i);
    });
};
var addLink = function(i) {
    $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
        $(this).attr("href") + "\">" + $(this).text() +
            "</a></li>");
};
```

Here we move the code that was in the **each()** function to a new function **addLink()**, and add a call to **addLink()** to the **each()** function. We pass the index, (**i**) into **addLink()** so we can use it to display the number of the link.

 Save the file and  preview your **looping.html** file. Does your code work? Do you see the links in the list? Probably not. So, why isn't it working?

Here's a clue: **this** might not be what you think it is in the function **addLink()**! Let's add a console log message to see what **this** is when we are in the **addLink()** function:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function(i) {
        addLink(i);
    });
};
var addLink = function(i) {
    console.log(this);
    $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
        $(this).attr("href") + "\">" + $(this).text() +
            "</a></li>");
};
```

 Save the file and Preview preview your **looping.html** file again. Check your browser's console window (and reload the page if you need to in order to see the message). What's displayed as the value of **this**? Instead of the <a> element object, you'll see either a "Window" object or "undefined." What happened? How did **this** get reset from the <a> element object it was when we were in the **each()** function, to the "Window" object (or "undefined")?

When you use jQuery's **each()** method to loop through a set of selected results, jQuery makes sure that the **this** variable is set to the right object in the body of the **each()** function. It works much like JavaScript does when it makes sure that **this** is set to the right object in the body of an object's method call. However, when you call a separate function yourself, that doesn't happen. In JavaScript, by default, when you call a function from the body of another function, **this** is set to the global object**window**, automatically (or, under some circumstances, **undefined**).

Note Beginning with ECMAScript 5 (the standard that JavaScript is based on) **this** is set to **undefined** rather than the global **window** object for function calls (not method calls).

We can fix this. We can pass the current object to **addLink()** along with **i**:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function(i) {
        addLink(i, this);
    });
};
var addLink = function(i, el) {
    console.log(this);
    $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
        $(this)$(el).attr("href") + "\">" + $(this)$(el).text() +
            "</a></li>");
};
```

 Save the file and Preview preview your **looping.html** file again. The links appear in the list.

We pass **this** (not **$(this)**) to **addLink()**. We certainly could pass **$(this)** instead, but then you wouldn't need to wrap **el** again. Make sure you understand why.

We named the parameter **el** in **addLink()**, rather than **this**. It's illegal in JavaScript to use **this** as a name for a variable you create yourself! This is good, because you might need the real **this** for something else, and it helps avoid confusion for other people who might read your code.

Replacing an Anonymous Function with a Named Function

Now, we're going to change the way we call **addLink()**:

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(function(i, el) {
        addLink(i, this);
    });
    $("a").each(addLink);
};
var addLink = function(i, el) {
    console.log(this);
    $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
        $(el).attr("href") + "\">" + $(el).text() +
            "</a></li>");
};
```

 Save the file and  preview your **looping.html** file. Everything still works as it did before. Let's go through these changes to understand what we did and the reasons behind it.

Rather than calling **addLink()** from the body of the **each()** function, we *replace* the previous **each()** function completely with the name **addLink**. If this seems odd, remember that both the anonymous function we were passing to **each()** previously, and **addLink** are function *values*. That is, they are just like any other variable value, like the number 3, or the string value "I need coffee," except that the value is a function rather than a number or a string. So in both cases, the value that we pass to **each()** is a function value. We pass in one of those values—the anonymous function—directly (like passing 3 to a function that expects a number) and in the other, we pass a variable name—**addLink**—that holds a function value (like passing the variable x containing the value 3 to a function that expects a number).

We don't *call* the function **addLink()** when we pass it to **each()**; we just give **each()** the name of the function to call when it's ready with each element object as it loops through the array of selected objects.

Just like the anonymous function expected two arguments, **addLink()** expects two arguments. Also, just like those arguments were the index of the element and the element itself when we were using an anonymous function, the arguments are those same values when we use **addLink()** as the **each** function.

Which is Better?

Why would we use a named function rather than an anonymous function? Which is better? Or are they equivalent?

Most of the time, you'll see jQuery programmers using anonymous functions rather than named functions. The benefit of using an anonymous function is that the code for the **each()** function is typically short, and by defining it right there, you can usually see what the **each()** is going to do to each of the selected elements; you don't have to go hunting around to find the function definition to see what's going to happen.

In some circumstances though, using a named function instead a bit more efficient, because JavaScript only has to create the named function once, and then it can use that same function each time through the **each()** loop. If you use an anonymous function instead, then JavaScript has to recreate the function each time through the loop. For small programs like this one, it won't make a difference in execution speed, but if you're building a game with thousands of elements, for instance, then small changes like this can add up to impact execution speed signficantly. For this course, we'll typically use anonymous functions for our **each()** functions, but now you know when and why you might want to use a named function instead.

A Little More of this

Before we leave this topic, open your console window and reload the page. Notice what the **console.log()** in the **addLink()** function displays as the value of **this**. Recall from the previous section that **this** was set to the global **window** object or undefined. Now, you see the <a> element as the value of **this** each time you call **addLink()** (fourteen times, since we have fourteen links). What changed?

Now, because we're passing the function **addLink** directly to **each()** as the value to use for the **each()**

function, jQuery is setting up the value of **this** correctly for us. When you *call* a function from the body of an **each()** function, **this** does *not* get set up for you; but when you *pass* a function value to **each()** to use for the **each()** function, **this** does get set up correctly for you to use in that function. Keep this in mind as you continue to explore **each()** and jQuery.

Optimizing jQuery (Just a Bit More)

Using a named function, like **addLink()**, as the **each()** function can optimize your code just a bit. Another way to optimize this code is by looking for variables that we need to wrap up as jQuery objects more than once, and use a temporary variable to store the wrapped value instead (the same logic applies to the reverse: unwrapping objects multiple times):

CODE TO TYPE:

```
$(document).ready(function() {
    createLinks();
});

var createLinks = function() {
    $("a").each(addLink);
};
var addLink = function(i, el) {
    console.log(this);
    var $el = $(el);
    $("section#links ul").append("<li>Link " + (i+1) + ": <a href=\"" +
        $(el)$el.attr("href") + "\">" + $(el)$el.text() +
            "</a></li>");
};
```

 Save the file and 

preview your **looping.html** file. Your code will behave the same way it did before, but now it will be a bit more efficient (not enough to notice in this small example though).

Before we wrapped the **el** variable twice to create the new <li> element: once to use the **attr()** method to get the value of the "href" attribute, and again to use the **text()** method to get the content of the element. Now, we create a variable **$el** and set it to the wrapped **$(el)** jQuery object, and then use **$el** in our code instead of **$(el)**. Make sure you understand the difference between **el**, **$(el)**, and **$el**.

Don't Use each() Unless It's Absolutely Necessary

When you begin using jQuery, and learn about **each()**, you might try to use **each()** in the wrong place. For instance, look at this code:

OBSERVE:

```
$("a").each(function() {
$(this).addClass("highlight").css("font-style", "italic");
});
```

Do you really need the **each()** here? No. You can write that code like this:

OBSERVE:

```
$("a").addClass("highlight").css("font-style", "italic");
```

To avoid making this mistake, keep the selected elements you're working on in mind. If the next action you want to take is on the selected elements returned by the previous action, and you can perform that action with a built-in jQuery method like **addClass()** or **css()**, you probably don't need an **each()**.

You're on the right track. Practice using these new functions in the homework and I'll see you in the next lesson!

Events

Lesson Objectives

- You will use **bind()** to bind events to event handler functions.
- You will use some shortcut methods, like **click()** and **mouseover()**.
- You will use the event object.

We've used event handlers in many of the examples we've created so far. Events are the main way a user communicates with a web application. The user loads a web page, clicks a mouse, checks a checkbox, or presses a key. All of these actions cause an event to happen in your web page, and if you want, you can watch for these events and do something with your program in response.

We've seen examples of handling both *user events*, like clicking a mouse, and *browser events*, like loading a page. To respond to a user event like a mouse click, we used the jQuery **click()** method, and to respond to a browser event like a page load, we used the jQuery **ready()** method.

jQuery's event API helps you to handle events better than plain JavaScript, because the methods are more flexible, and they work across all browsers consistently. Let's jump in and take a closer look at jQuery events and see how they work.

Setting Up Event Handlers with bind()

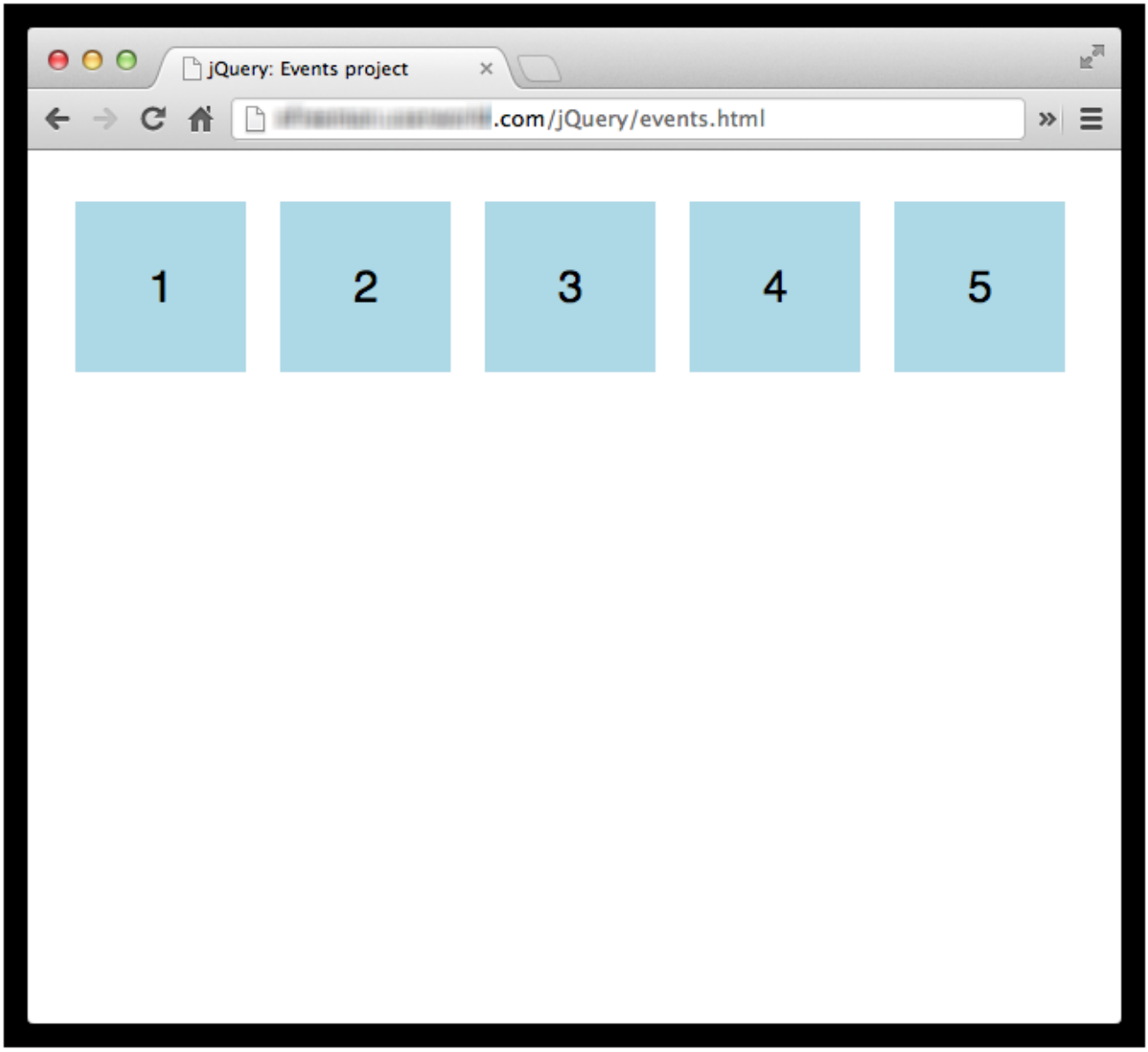
As always, we need an example to play with! Create a new HTML file and add the code below:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Events</title>
  <meta charset="utf-8">
  <style>
    div.container {
      display: table;
      border-spacing: 20px;
      margin-top: 10px;
    }
    div.row {
      display: table-row;
    }
    .box {
      display: table-cell;
      width: 100px;
      height: 100px;
      background-color: lightblue;
      text-align: center;
      vertical-align: middle;
      font-size: 130%;
      font-family: Helvetica, sans-serif;
    }
    .highlight {
      background-color: yellow;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="events.js"> </script>
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="box">1</div>
      <div class="box">2</div>
      <div class="box">3</div>
      <div class="box">4</div>
      <div class="box">5</div>
    </div>
  </div>
</body>
</html>
```

 Save this file in your **/jQuery** folder as **events.html**. It's basic; we just create five elements with some style to make them all line up nicely. Go ahead and 

the file:

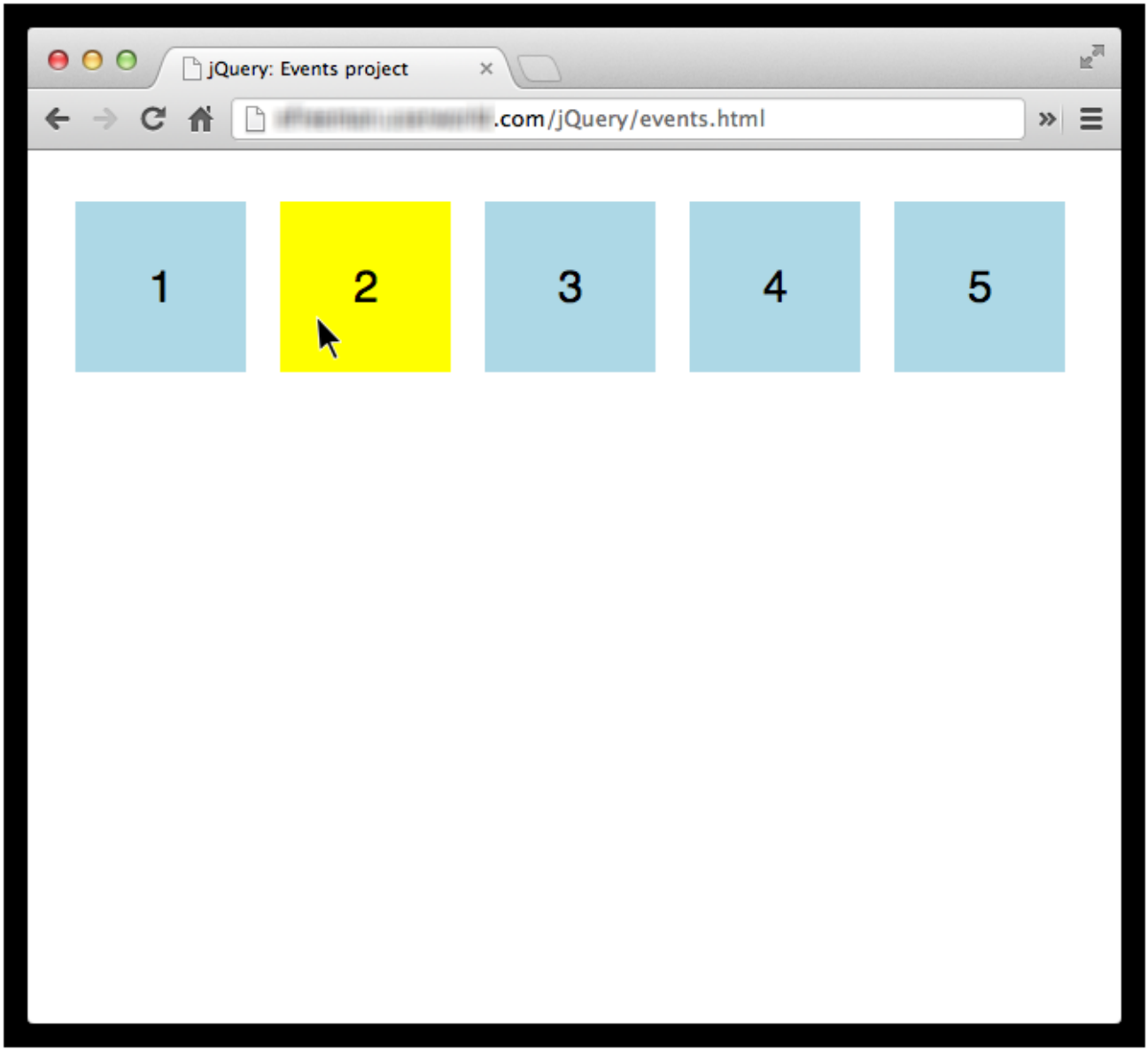


Now, create a new jQuery file and add this code:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").bind("mouseover", changeColor);
    $("div.box").bind("mouseout", changeColor);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
```

Save the file in your **/jQuery** folder as **events.js**. Preview your **events.html** file, and move your mouse over the elements. The background color of each element turns yellow as you move your mouse over it, and turns back to blue as you move your mouse away from it.



In this code, we have three different kinds of event handlers: the **ready()** event handler, which is called when the DOM is ready to be manipulated after the page has loaded; and two user event handlers, one for the **mouseover** event, and one for the **mouseout** event.

OBSERVE:

```
$(document).ready(function() {
    $("div.box").bind("mouseover", changeColor);
    $("div.box").bind("mouseout", changeColor);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
```

We use the jQuery **bind()** method to bind a function, **changeColor()**, to an event. **bind()** is a general-purpose method for setting up event handlers, similar to the JavaScript **addEventListener()** (or **attachEvent()** in IE version 8 and earlier) function. You specify the name of the event you're handling in the first argument, and pass a function as the event handler in the second argument to **bind()**, then that function is called when the event occurs. The event handler is bound to a specific element (or set of elements). In this case, it's bound to all the elements with the "box" class. The selector **$("div.box")** returns an array of elements that match; **bind()** then binds the **changeColor()** function to each one of those matching elements. This is a handy way to bind an event handler to many elements with one line of code.

We could have used an anonymous function in the call to **bind()**, but instead we used a named function,

changeColor(), because we want to use the same function as the handler for both the "mouseover" and "mouseout" events.

The **changeColor()** function uses the **toggleClass()** method, which handles adding or removing the class for us, depending on whether the has the class "highlight." Using **toggleClass()** means we can use the same event handler function for both the "mouseover" and "mouseout" events. In **changeColor()**, **$(this)** is set to the <div> element for which the event is being handled; so, if you move your mouse over the first <div> element, **$(this)** is set to that element; the same goes for each of the other elements. When you pass a function name to **bind()**, jQuery handles setting up **$(this)** so that it points to the right element in the handler function.

Using Event Shortcuts

jQuery provides shortcut methods for most of the common events you'll want to handle. You've already used one: the **click()** shortcut method. Let's see how we'd replace **bind()** for the "mouseover" and "mouseout" events with the equivalent shortcuts:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").bind("mouseover", changeColor);
    $("div.box").bind("mouseout", changeColor);
    $("div.box").mouseover(changeColor);
    $("div.box").mouseout(changeColor);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
```

 Save **events.js** and 

preview your **events.html** file. It works just like it did before, except now we're using the **mouseover()** and **mouseout()** methods instead of **bind()**.

jQuery provides many types of events you can use with **bind()** and equivalent shortcut methods. You'll use these most often:

Event name	Shortcut method	Purpose
blur	blur()	Event is triggered on an element when that element loses focus.
focus	focus()	Event is triggered on an element when that element gains focus.
click	click()	Event is triggered on an element when that element is clicked.
keydown	keydown()	Event is triggered on the focused element when the browser registers keyboard input.
mouseover	mouseover()	Event is triggered on an element when the mouse moves over that element.
mouseout	mouseout()	Event is triggered on an element when the mouse moves off that element.
resize	resize()	Event is triggered on the window element when the brower window is resized.
submit	submit()	Event is triggered on a form element when the form is submitted.

For a complete list of event handling methods, see the jQuery documentation for Events.

Binding Multiple Events to One Handler with One bind()

One of the advantanges of using **bind()** is that we can specify multiple events with one line of code:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").mouseover(changeColor);
    $("div.box").mouseout(changeColor);
    $("div.box").bind("mouseover mouseout", changeColor);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
```

 Save **events.js** and  preview your **events.html** file; again, the page behaves as before. Notice that we specify both the "mouseover" and "mouseout" events in one string as the first argument to the **bind()** method. Make sure you separate the event names with a space, not a comma, or it won't work! (It's an easy mistake to make.)

You may not find too many occasions to bind multiple events to one handler in the same **bind()** method call, but in the case where you use the same handler for multiple events, like we did here, it does come in handy.

Using hover()

Most events require only one event handler. However, **hover** is an event you'll use fairly often and it requires *two* event handlers: one to handle when the mouse is hovering over an element and one for when it's not. Modify **events.js** as shown:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").bind("mouseover mouseout", changeColor);
    $("div.box").hover(function() {
        $(this).addClass("highlight");
    },
    function() {
        $(this).removeClass("highlight");
    });
});
function changeColor() {
    $(this).toggleClass("highlight");
}
```

 Save the file and  preview your **events.html** file. When you hover your mouse over one of the elements, it highlights just like before, but now we're using the **"hover"** event rather than the "mouseover" and "mouseout" events. It accomplishes basically the same task, but in a slightly different way.

The **hover()** method takes two event handlers. In this case, we pass two anonymous functions to **hover()**; the first adds the "highlight" class to the selected , and the second removes that class.

We could use our **changeColor()** function for both handler functions instead, which is slightly more efficient because we're reusing the same function twice:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").hover(function() {
        $(this).addClass("highlight");
    },
    function() {
        $(this).removeClass("highlight");
    });
    $("div.box").hover(changeColor, changeColor);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
```

 Save **events.js** and  preview your **events.html** file; the behavior is exactly the same.

Using Events to Hide and Show Elements

You can use events in jQuery programs to hide and show elements. Let's look at two ways we can hide and show a "box" element:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").hover(changeColor, changeColor);
    $("div.box").bind("click", hideElement);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
function hideElement() {
    $(this).hide();
}
```

 Save **events.js** and  preview your **events.html** file. Now try clicking on one of the elements. It disappears! The only way to get it back is to reload the page. Let's add a better way to retrieve the elements. Modify **events.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Events</title>
  <meta charset="utf-8">
  <style>
    div.container {
      display: table;
      border-spacing: 20px;
      margin-top: 10px;
    }
    div.row {
      display: table-row;
    }
    .box {
      display: table-cell;
      width: 100px;
      height: 100px;
      background-color: lightblue;
      text-align: center;
      vertical-align: middle;
      font-size: 130%;
      font-family: Helvetica, sans-serif;
    }
    .highlight {
      background-color: yellow;
    }
    p {
      cursor: pointer;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="events.js"> </script>
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="box">1</div>
      <div class="box">2</div>
      <div class="box">3</div>
      <div class="box">4</div>
      <div class="box">5</div>
    </div>
    <p>Show all</p>
  </div>
</body>
</html>
```

 Save **events.html** and 

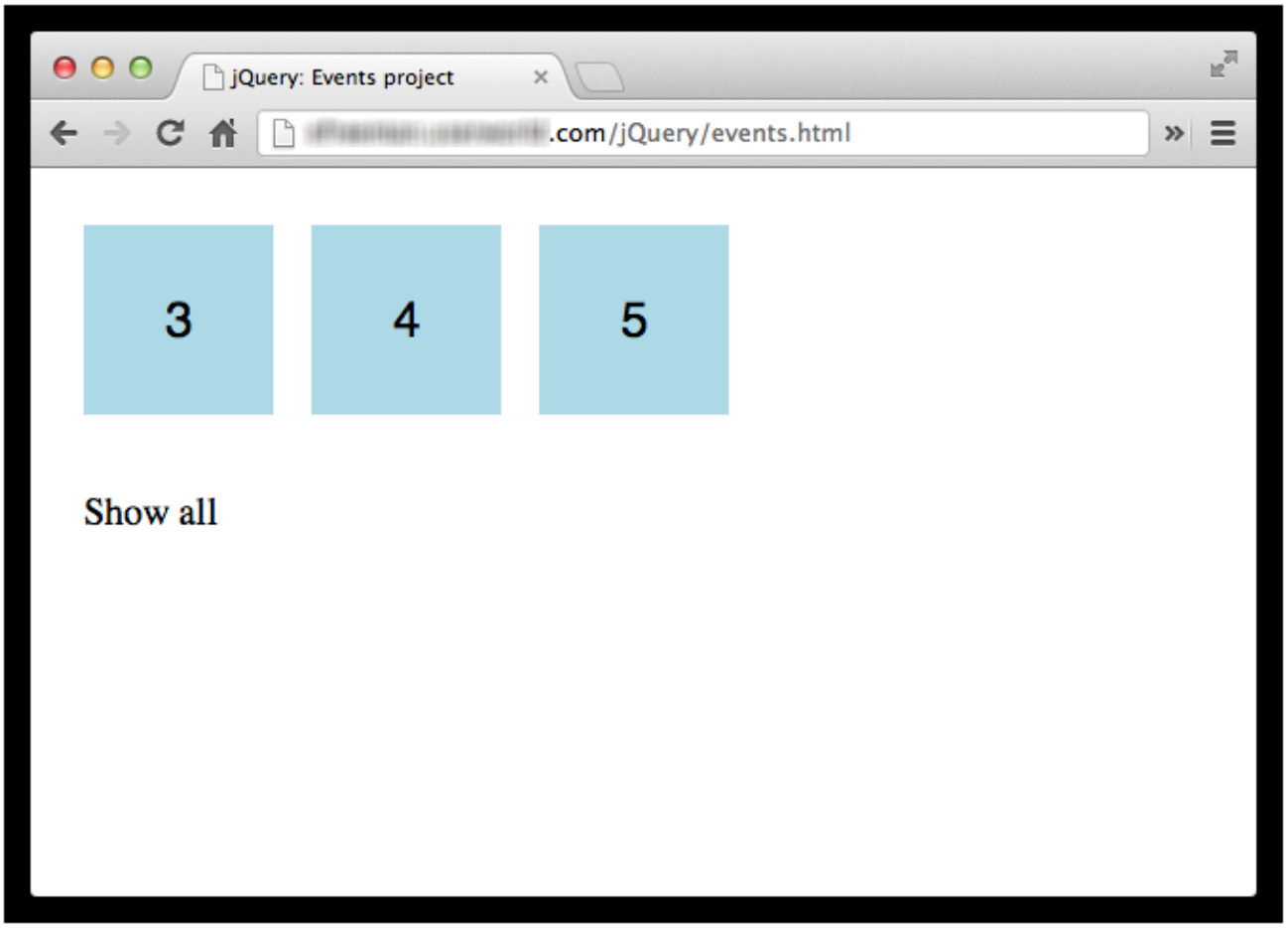
preview. Now you see the "Show all" text below the elements, and when you mouse over the text, the mouse turns into a pointer. Now, we can hook up some code to our jQuery to show the hidden elements when we click the "Show all" text:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").hover(changeColor, changeColor);
    $("div.box").bind("click", hideElement);
    $("p").click(function() {
        $("div.box:hidden").show();
    });
});
function changeColor() {
    $(this).toggleClass("highlight");
}
function hideElement() {
    $(this).hide();
}
```

Save **events.js** and Preview preview your **events.html** file. Now, when you click on a and hide it, you can click **Show all** to make it visible again. We added a click handler on the <p> element using an anonymous function, and call the **show()** method on the "box" elements to show any hidden elements. Notice that we're calling **show()** on only the elements that are hidden. The filter **:hidden** selects all elements that have been hidden with the **hide()** method.

When you click on a element to hide it, all the other elements shift over to the left:



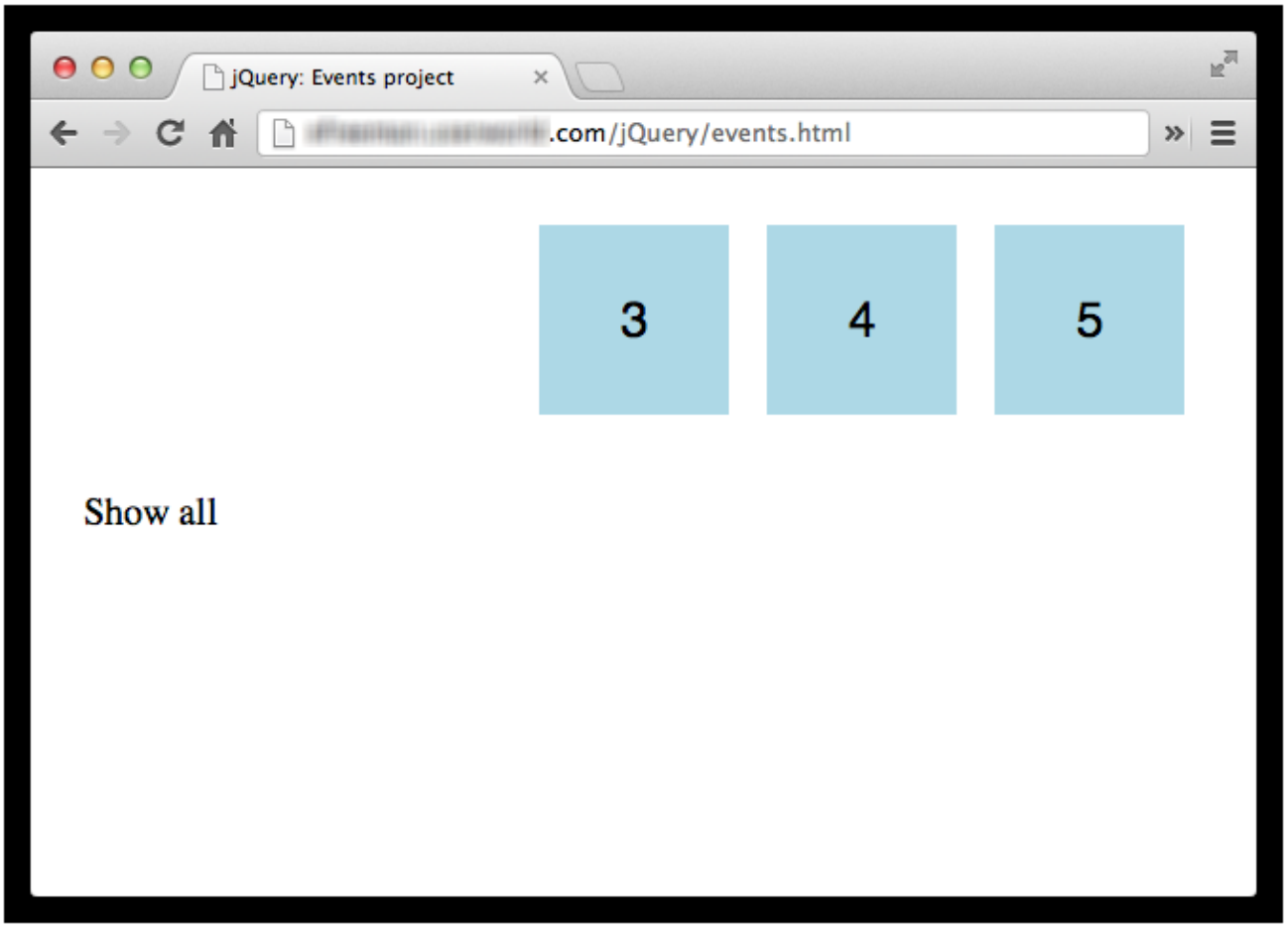
That's because when you **hide()** an element, **hide()** sets the CSS **display** property to "none," which takes the element out of the normal layout flow (the element's still there, it just doesn't take up any space in the page).

If you want the other elements to stay in their normal spots when you hide an element, you need a different way to hide the element. Use the CSS **visibility** property instead:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").hover(changeColor, changeColor);
    $("div.box").bind("click", hideElement);
    $("p").click(function() {
        $("div.box:hidden").show();
        $("div.box").css("visibility", "visible");
    });
});
function changeColor() {
    $(this).toggleClass("highlight");
}
function hideElement() {
    $(this).hide();
    $(this).css("visibility", "hidden");
}
```

Save **events.js** and Preview preview your **events.html** file. Now when you click on a to hide it, it becomes invisible, but it still takes up space in the layout of the page. When you click **Show all**, we make the invisible elements visible again by setting the **visibility** property back to "visible."



To show the elements, we can no longer use the **:hidden** filter to select only the hidden elements. That's because the **:hidden** filter works only on elements hidden with the **hide()** method. So here, we just set the CSS **visibility** property on *all* of the "box" elements. That's okay; it doesn't do any harm to set this property on elements that are already visible.

The Event Object

When an event occurs, there might be some details about it that you want to know. For example, you might want to know where the mouse clicked, or which key was pressed, depending on the type of the event. We can get this

information by accessing the *event object*, which is an object passed into every event handler function.

Let's add a element to each "box" to display the coordinates of the mouse click when the user clicks on one of the "box" elements in the page. Update your HTML and CSS as shown:

CODE TO TYPE:

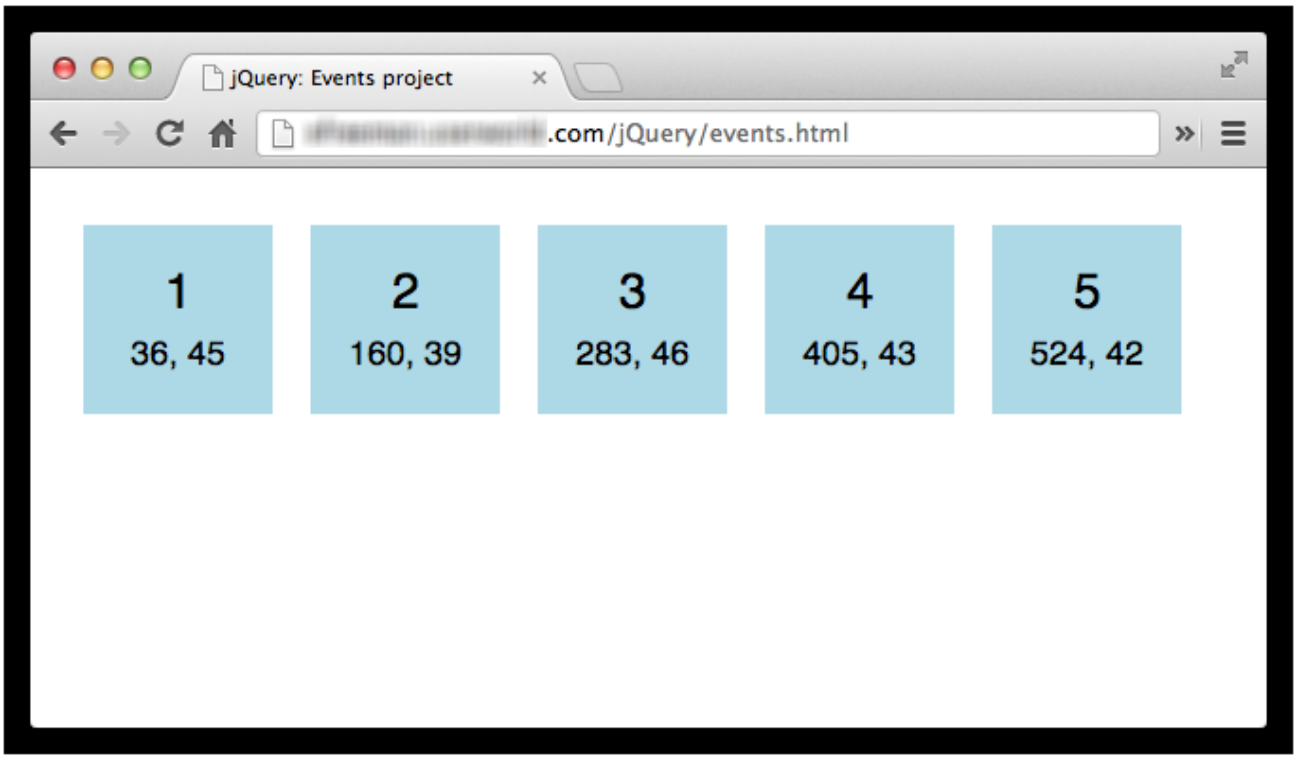
```
<!doctype html>
<html>
<head>
  <title>jQuery: Events</title>
  <meta charset="utf-8">
  <style>
    div.container {
      display: table;
      border-spacing: 20px;
      margin-top: 10px;
    }
    div.row {
      display: table-row;
    }
    .box {
      display: table-cell;
      width: 100px;
      height: 100px;
      background-color: lightblue;
      text-align: center;
      vertical-align: middle;
      font-size: 130%;
      font-family: Helvetica, sans-serif;
    }
    .highlight {
      background-color: yellow;
    }
    div.box span {
      font-size: 70%;
    }
    p {
      cursor: pointer;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="events.js"> </script>
</head>
<body>
  <div class="container">
    <div class="row">
      <div class="box">1<br><span></span></div>
      <div class="box">2<br><span></span></div>
      <div class="box">3<br><span></span></div>
      <div class="box">4<br><span></span></div>
      <div class="box">5<br><span></span></div>
    </div>
    <p>Show all</p>
  </div>
</body>
</html>
```

 Save **events.html** and update your jQuery:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").hover(changeColor, changeColor);
    $("div.box").bind("click", hideElement);
    $("p").click(function() {
        $("div.box").css("visibility", "visible");
    });
    $("div.box").click(showPosition);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
function hideElement() {
    $(this).css("visibility", "hidden");
}
function showPosition(e) {
    $("span", this).text(e.pageX + ", " + e.pageY);
}
```

Save **events.js** and Preview preview your **events.html** file. Click on one of the elements in the page; the mouse coordinates of the click appear in the . The first number, **e.pageX** is the number of pixels from the left of the web page, and the second number, **e.pageY** is the number of pixels from the top of the web page. The variable **e** is the **event object** that's passed into the click handler function, **showPosition()**. It has several properties with information about the click event. The **pageX** and **pageY** properties allow you to determine the coordinates of a mouse click.



Selector Context

Before we move on to the next section and use the **event object** to find out which key is pressed on the keyboard, we need to take a closer look at the **showPosition()** handler function:

OBSERVE:

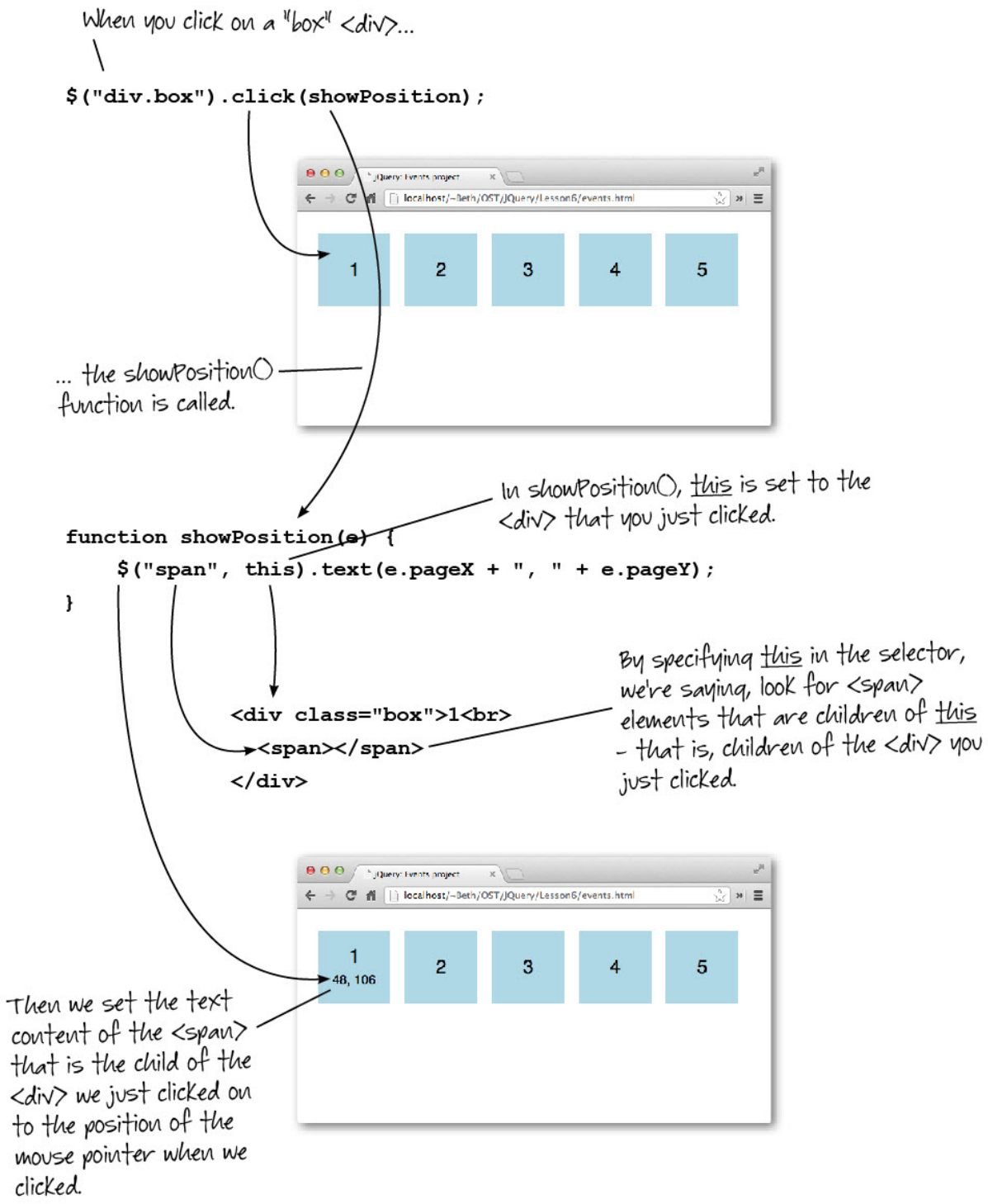
```
function showPosition(e) {
    $("span", this).text(e.pageX + ", " + e.pageY);
}
```

The selector, **$("span", this)** looks a little different from the selectors we've used so far. What's going on here?

this is the "box" we just clicked on, because those are the elements that are bound to the click handler **showPosition()**. By specifying **this** as the second argument to the **$** function, we specify a **context** for our selector. This context acts as a partial DOM tree in which we search for elements that match "span." So rather than searching the entire document for elements, we only search *within* the recently clicked on "box" element. None of the other elements are selected; just one is selected. (Recall from the HTML that each "box" has only one child element):

OBSERVE:
`<div class="box">1 <span></span></div>`

Specifying a context for a selector gives us a way to match elements within a portion of the page, and provides a convenient way to find elements related to an element that just had an event happen, like a click.



Keyboard Events

Let's have some fun with keyboard events. We can check for the "keydown" event, and bind this event to a new function, **move**. So, what happens when you press a key on the keyboard? How do you know what key was pressed? You can find out using the **which** property of the **event** object:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").hover(changeColor, changeColor);
    $("div.box").click(showPosition);
    $("body").bind("keydown", move);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
function showPosition(e) {
    $("span", this).text(e.pageX + ", " + e.pageY);
}
function move(e) {
    console.log(e.which);
}
```

 Save **events.js** and  preview your **events.html** file. Open your JavaScript console and reload the page. Now click on the background of the web page (not in the console, but in the main part of the page), and then press the up arrow key. The number 38 appears in the console. Now try the down arrow key. The number 40 appears. These numbers represent the keys on the keyboard. Each key has a different number. Try some other keys and see what happens.

Let's update the **move()** function to detect the up and down arrows, and move the elements up or down in the page when the user presses one of these two keys. (We'll just ignore any other keyboard input):

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.box").hover(changeColor, changeColor);
    $("div.box").click(showPosition);
    $("body").bind("keydown", move);
});
function changeColor() {
    $(this).toggleClass("highlight");
}
function move(e) {
    console.log(e.which);
    var margin = $("div.container").css("margin-top"); // returns NUMpx
    margin = parseInt(margin);
    if (e.which == 40) { // move down
        margin = margin+20;
        $("div.container").css("margin-top", margin);
    } else if (e.which == 38) { // move up
        if (margin > 10) {
            margin = margin-20;
            if (margin < 10) {
                margin = 10;
            }
            $("div.container").css("margin-top", margin);
        }
    }
}
```

 Save **events.js** and  preview your **events.html** file. Press the down arrow key. The elements move down the page, a few pixels each time you press the down arrow. (If you don't see them moving, click on the background of the page, and try again). Now, press the up arrow. The elements will move up the page.

In the **move()** event handler, we get the top margin, in pixels, of the "container" (the that contains all the "box" elements). This will be a value like "10px", so we can get the numerical value by using **parseInt()**. Then we check the **which** property of the **event object** to see if the user pressed the down arrow or the up arrow. If the down arrow was pressed, we add 20 to the margin, and update the "margin-top" property; if the up arrow was pressed, we reduce the margin by 20, but only if the margin is bigger than 10px (we want a minimum value of 10px for the top margin).

Building a Popover with jQuery

Before we leave events behind, let's build one more web application: a web page with a popover. You've probably been on those web pages that contain links that when you mouseover, you get a (sometimes annoying!) popover, right? Well, these popovers are relatively straightforward to build with jQuery once you understand events. Start by creating a new HTML file:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Company Info Popover</title>
  <meta charset="utf-8">
  <style>
.companyRollover {
    cursor: pointer;
    background-color: lightgray;
    border: 1px solid #333333;
}
.companyData {
    position: absolute;
    background-color: white;
    border: 2px solid gray;
    padding: 10px;
    box-shadow: 2px 1px 2px gray;
    z-index: 1;
}
.companyDataTriangle {
    position: absolute;
    width: 24px;
    height: 24px;
    background-color: white;
    border-left: 2px solid gray;
    border-top: 2px solid gray;
    transform: rotate(45deg);
    -ms-transform: rotate(45deg);
    -webkit-transform: rotate(45deg);
    -o-transform: rotate(45deg);
    -moz-transform: rotate(45deg);
    z-index: 2;
}
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="popover.js"> </script>
</head>
<body>
  <div id="main">
    <h1>Online Courses from O'Reilly Media</h1>
    <p>
The <span class="companyRollover" id="OST">O'Reilly School of Technology</span>
was built as a place to give aspiring programmers and administrators
like you a ladder into the Information Technology and Systems
industry. With unique, online, hands-on courses leading to
Certificates of Professional Development, OST will help you
gain an edge in your career -- on your own time, at your own
pace. When you have completed our courses, you will not only
have a Certificate, but you will also have a portfolio of
completed projects to show for your effort.
    </p>
  </div>
  <div class="companyData" data-ticker="OST" data-price="100">
    <h2>O'Reilly School of Technology (OST)</h2>
    <h3>U.S.: Fake Nasdaq</h3>
    <p>Stock Price: $100</p>
    <h3>Recent Articles</h3>
    <ul>
      <li><a href="http://blog.oreillyschool.com/2012/09/ost-author-has-a-close-encount
er-on-mars-with-the-rover-curiosity.html">Close Encounter with Mars Rover</a></li>
      <li><a href="http://blog.oreillyschool.com/2012/04/automatic-grading-misses-the-p
oint.html">Automatic grading misses the mark</a></li>
    </ul>
  </div>
  <div class="companyDataTriangle">
  </div>
```

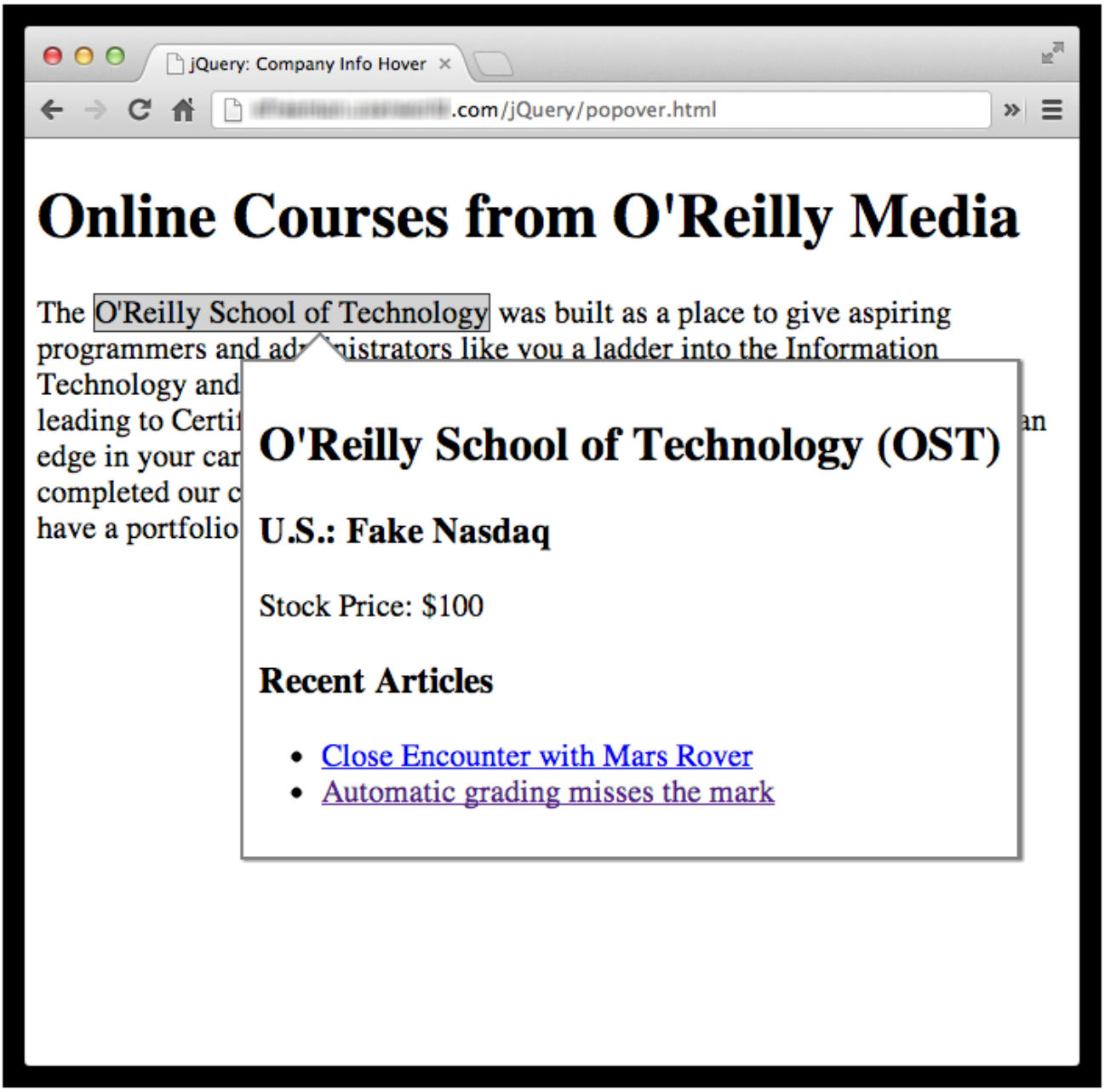
```
</body>
</html>
```

 Save the file in your **/jQuery** folder as **popover.html**, and  preview. The text, "O'Reilly School of Technology" is highlighted in gray. When you mouseover that text, nothing happens—yet. Create a new file for your jQuery named **popover.js**:

CODE TO TYPE:

```
$(document).ready(function() {
    $("div.companyData").hide();
    $("div.companyDataTriangle").hide();
    $("span.companyRollover").mouseover(function(evt) {
        var id = $(this).attr("id");
        var $companyData = $("div.companyData[data-ticker='" + id + "']");
        $companyData
                .css({ top: evt.pageY + 30, left: evt.pageX-50 })
                .toggle(100);
        $("div.companyDataTriangle")
                .css({ top: evt.pageY + 18, left: evt.pageX-12 })
                .toggle(0);
    });
});
```

 Save the file in your **/jQuery** folder as **popover.js**, and  preview your **popover.html** file. Now, when you mouseover the highlighted text, a popover appears! Notice the behavior of the popover; it stays up if you mouse off the highlighted text (so you can access the links to the articles in the popover if you want), and goes away when you mouseover the text again. So it's "toggling" on and off.



We made this popover happen with just a few lines of code! Let's go over it in detail:

OBSERVE:

```
$(document).ready(function() {
    $("div.companyData").hide();
    $("div.companyDataTriangle").hide();
    $("span.companyRollover").mouseover(function(evt) {
        var id = $(this).attr("id");
        var $companyData = $("div.companyData[data-ticker='" + id + "']");
        $companyData
                .css({ top: evt.pageY + 30, left: evt.pageX-50 })
                .toggle(100);
        $("div.companyDataTriangle")
                .css({ top: evt.pageY + 18, left: evt.pageX-12 })
                .toggle(0);
    });
});
```

The two pieces of the popover are created with two elements, the "companyData" , and the "companyDataTriangle" . (To see how we made the triangle from a , take a look at the CSS). First, we **hide**

those elements because we don't want to see them unless the mouse is over the highlighted text.

Next, we add a **mouseover event handler function** to the "companyRollover" containing the highlighted text. When you put your mouse over this text, this handler function is run. We get the id of the and use that to create an attribute value for the "data-ticker" attribute, so we can select the correct "companyData" . In this case, we only have one "companyData" , but this gives you an idea how you can use the id of an element you mouse over in a selector to find another element; this can come in handy!

Then we **set the CSS of the "companyData" ** so that its **top and left positions are based on the position of the mouse**. We modified the position values a bit to make room for the "companyDataTriangle" that we're going to place above the "companyData" in the next step.

After we set the position for the "companyData" in CSS, we call the function **toggle()**. This is *not* the same as the **toggleClass()** function we've seen before. Don't get them mixed up (it's easy to do)! **toggle()** toggles the display of an element between "none," and "block" or "inline," depending on what the default display is for that element. So, effectively, **toggle()** shows or hides the element, depending on whether it's currently hidden.

The **argument we pass to toggle(), 100,** tells jQuery how long to take to show the element (if it's hidden), so it happens over a period of time, in this case 100 milliseconds. This gives you a slight animation effect, which looks like the popover is growing out of the highlighted text. The same thing happens in reverse when you mouse out of the highlighted text, and then mouse over again to make the popover go away.

Finally, we **set the CSS position of the "companyDataTriangle" ** to position the triangle between the mouse and the "companyData" , and **toggle() the visibility of the triangle**. We toggle the display of the triangle immediately (0 milliseconds), so it appears as soon as the "companyData" has completed its animation and is fully visible.

Experiment with **hover()** instead of **mouseover** to see which you like better. When you use **hover()**, you can't access the links in the popover, but you may like the behavior of the popover (that is, how it appears and disappears) a bit better. Also, experiment by changing how far away from the mouse position the popover appears; if the popover is too far away, it feels disconnected, but if it's too close, the triangle of the popover can get in the way of the mouse event.

In this lesson you learned how to use **bind()** to bind events to event handler functions. You also learned about some shortcut methods, like **click()** and **mouseover()**, that you can use instead of **bind()**. You can use either way to create event handlers in your jQuery code; one is not necessarily better than the other in most situations.

You also learned about the event object and how to use it to get information about the event that caused a click handler function to be called. There are several properties you can get from the event object other than the ones we explored in this lesson; some of them depend on what event happened (for instance, if you press a key rather than click a mouse, it's not going to make sense to get the mouse coordinates of a click).

For a lot more information about the event object, check out the jQuery documentation. Keep going! You're doing really well. Work on the homework and we'll reconvene when you're done.

Manipulating and Traversing Elements

Lesson Objectives

- You will use jQuery methods to add or remove content, find content, and test content.

Adding New Content to a Page

So far we've used three methods to add new content to a page: **html()**, **text()**, and **append()**. It's time we took a closer look at these (and other) methods of adding content to a page. Let's begin with **html()**. Create a new file and add the following HTML:

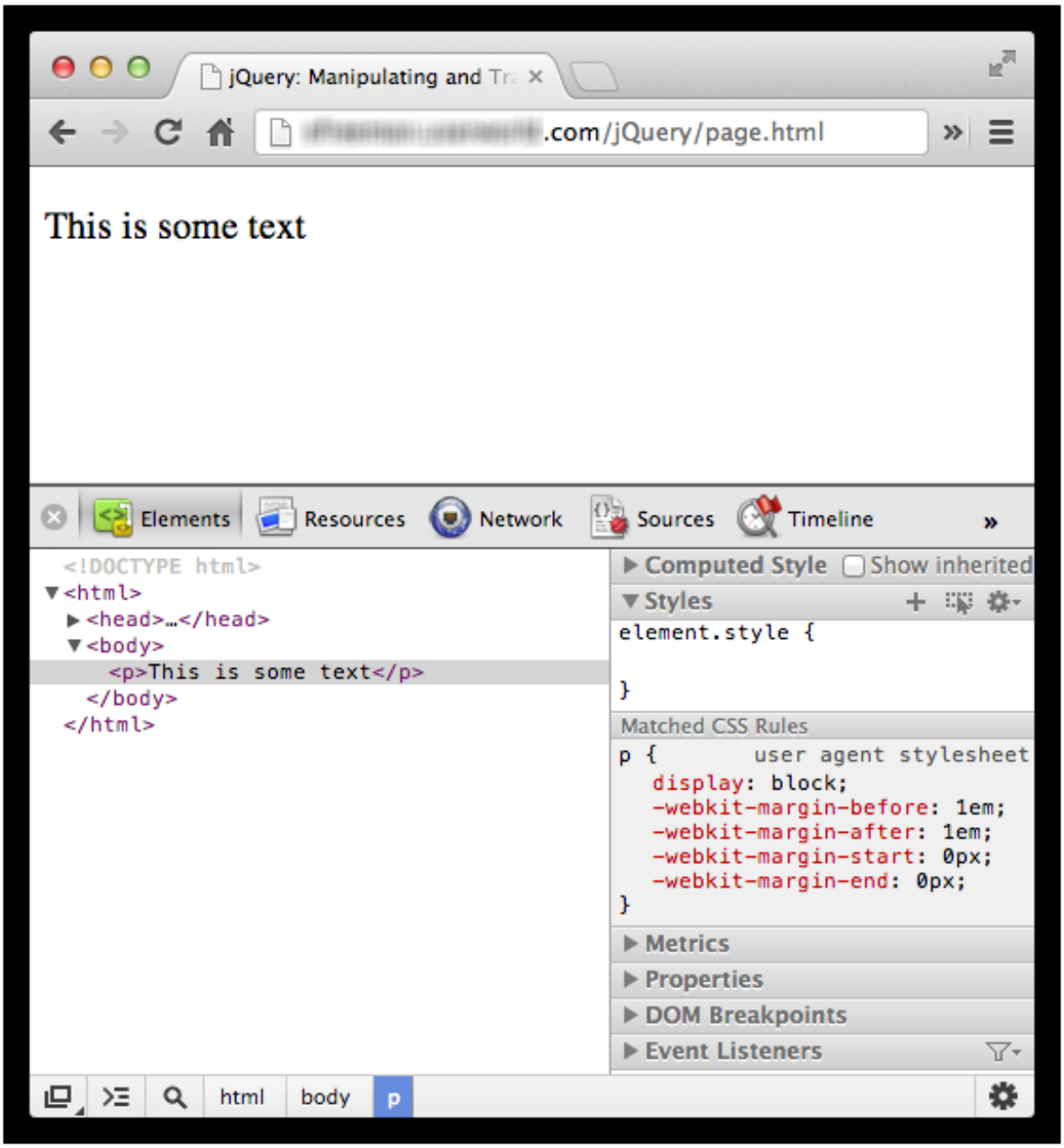
CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Manipulating and Traversing Elements</title>
  <meta charset="utf-8">
  <style>
    .highlight {
      background-color: lightyellow;
    }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>
    $(document).ready(function() {
        $("body").html("<p>This is some text</p>");
    });
  </script>
</head>
<body>
</body>
</html>
```

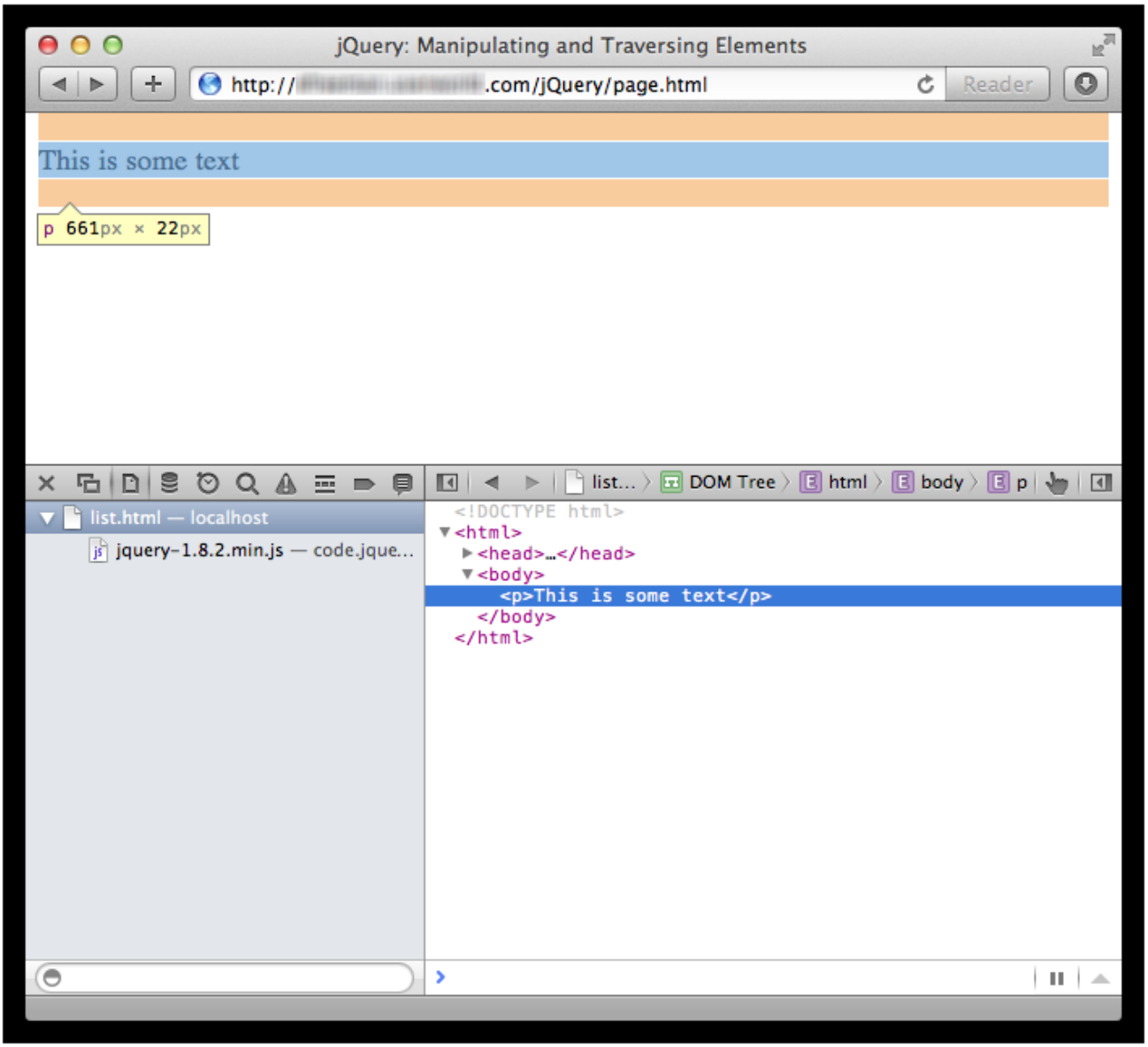
Save this file in your **/jQuery** folder as **page.html**. Go ahead and Preview preview the file, open the developer tools in your browser, and bring up the DOM tree view.

Note The DOM tree view is the "Resources" tab in Safari, the "Elements" tab in Chrome, the "Inspect" option under **Tools | Web Developer** in Firefox, and the **HTML** tab under **Tools | Developer Tools** in Microsoft Internet Explorer.

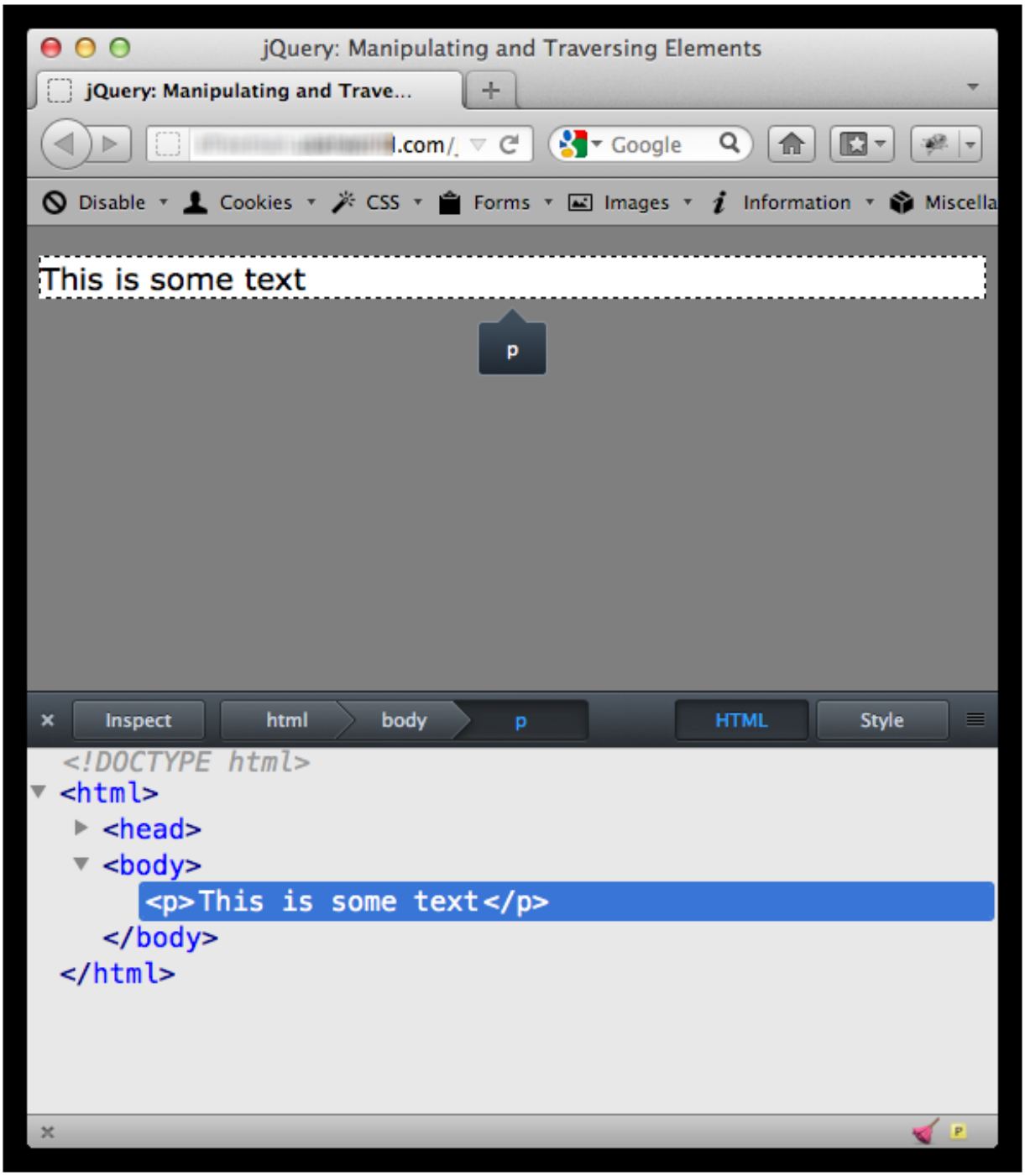
Here is the DOM tree view in Chrome:


jQuery: Manipulating and Tr
.com/jQuery/page.html
This is some text
Elements
Resources
Network
Sources
Timeline
<!DOCTYPE html>
<html>
<head>…</head>
<body>
This is some text
</body>
</html>
Computed Style
Show inherited
Styles
element.style {
}
Matched CSS Rules
p { user agent stylesheet
display: block;
-webkit-margin-before: 1em;
-webkit-margin-after: 1em;
-webkit-margin-start: 0px;
-webkit-margin-end: 0px;
}
Metrics
Properties
DOM Breakpoints
Event Listeners
html
body
p

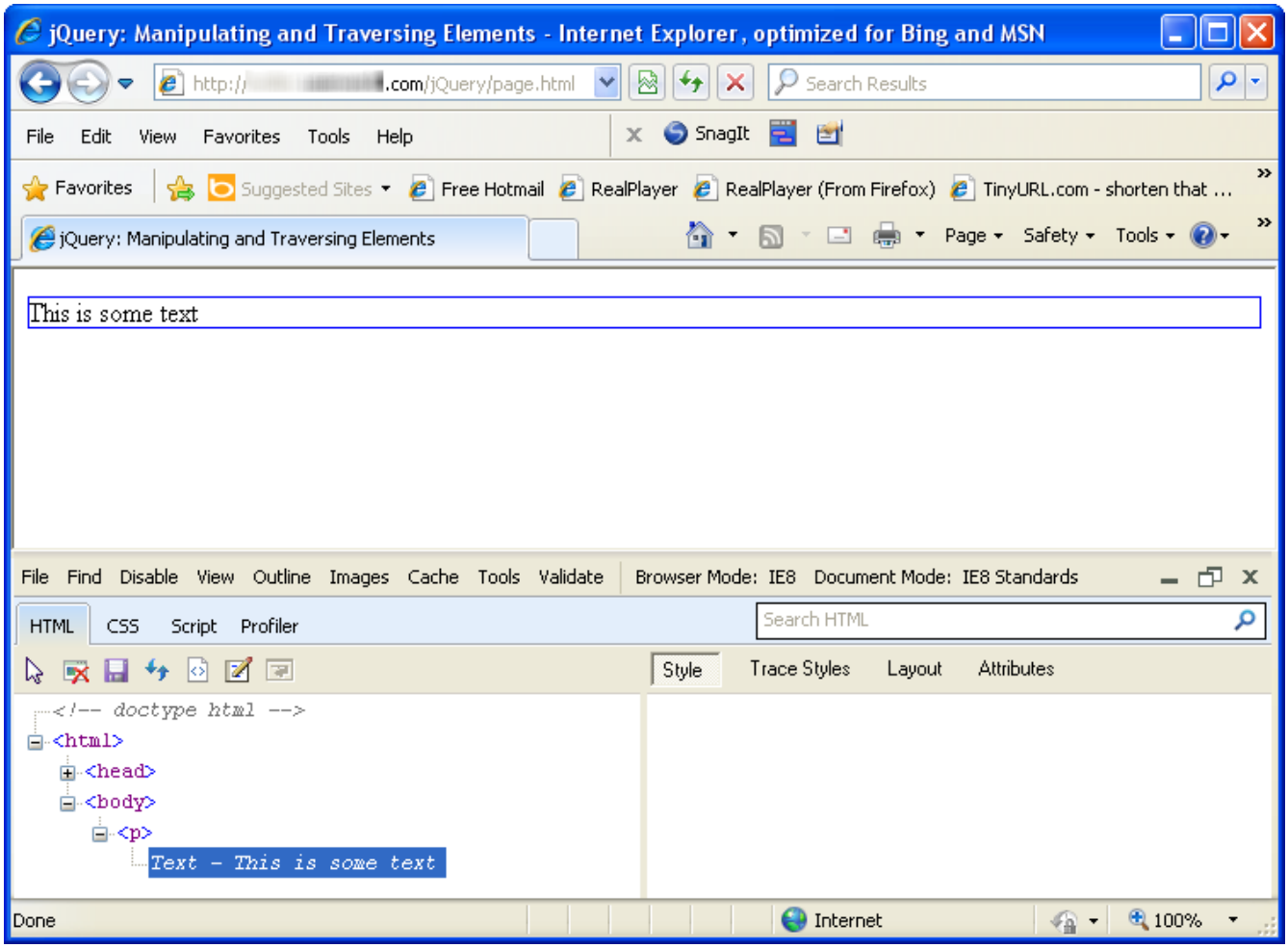
in Safari:


jQuery: Manipulating and Traversing Elements
http://.com/jQuery/page.html
Reader
This is some text
p 661px × 22px
list...
DOM Tree
html
body
p
list.html — localhost
jquery-1.8.2.min.js — code.jque...
<!DOCTYPE html>
<html>
<head>…</head>
<body>
This is some text
</body>
</html>

in Firefox:



and in Microsoft Internet Explorer:



The HTML for the page is just an empty body when the page loads (because that's what we wrote in the page), and in the jQuery, we add a <p> element to the page using the **html()** method:

OBSERVE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
});
```

This code sets the content of the <body> element to the HTML "<p>This is some text<p>".

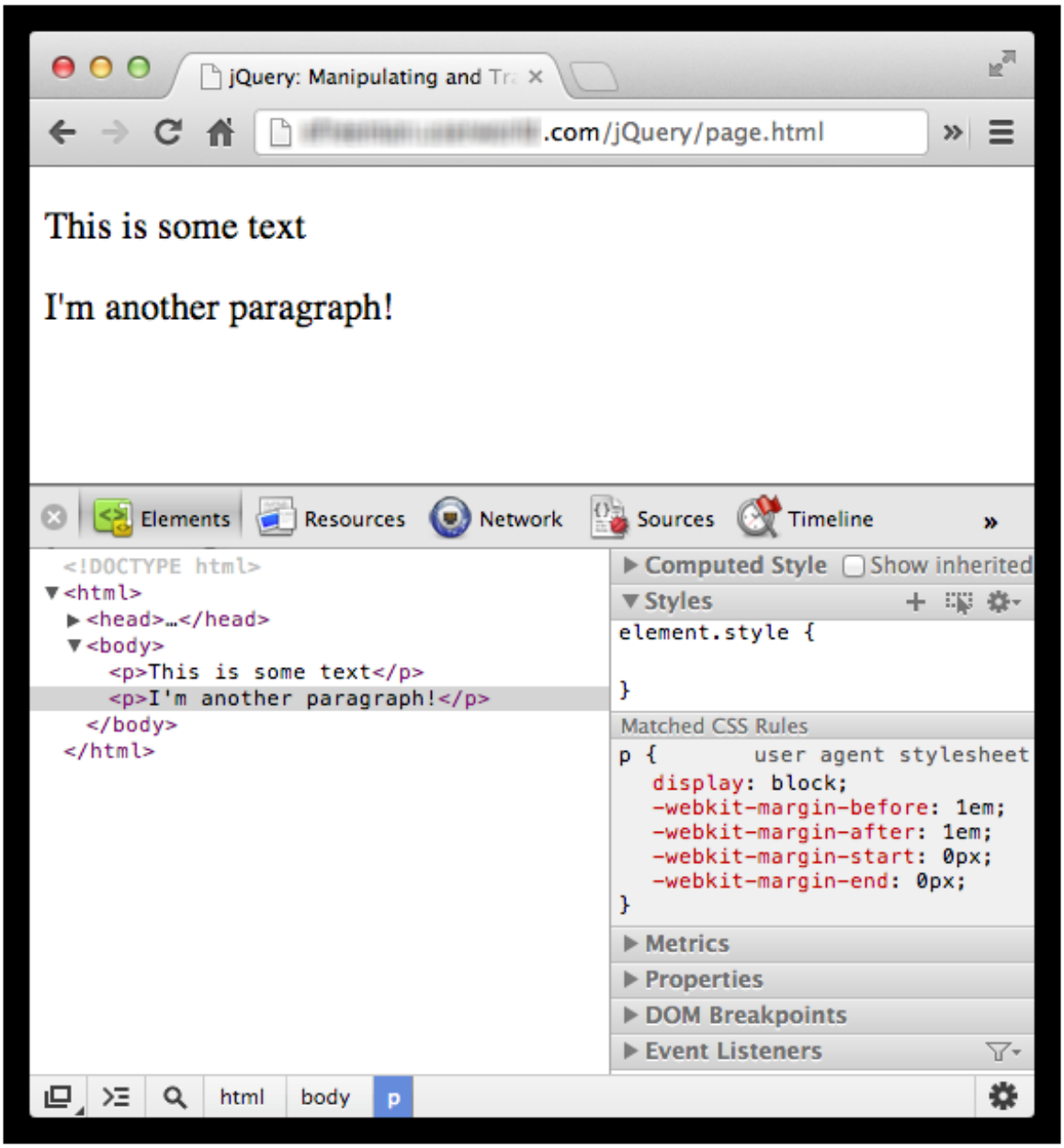
Adding Content with append()

How does the **html()** method compare to the **append()** method? Modify the jQuery as shown:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
});
```

Save and Preview preview. You see two paragraphs in your page, like this:



append() added the new paragraph below the existing paragraph in the page. Specifically, **append()** inserts content as the *last child* of the selected element (or elements, if there are more than one). In our case, we select one element, <body>, and insert the new paragraph below all the other content in the body (which is just one paragraph, so far).

What happens if you change **append()** to **html()** in the code above? Remember that **html()** *sets* the content of the selected element (in this case, the <body> element), while **append()** inserts the content below all the other content in the body. Give it a try and see!

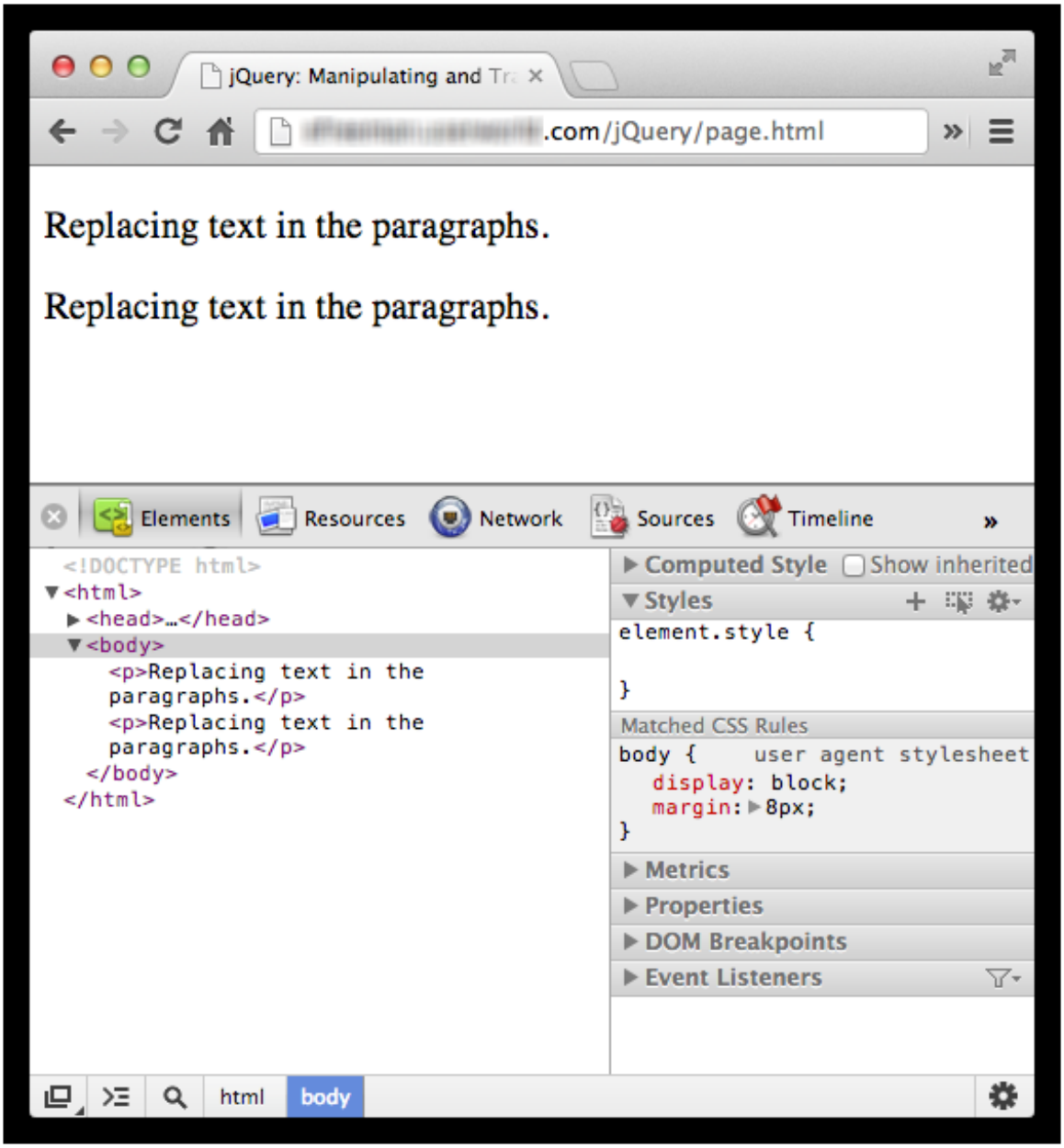
Comparing text() and html()

Now that we have a couple of paragraphs, let's try changing the content:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").text("Replacing text in the paragraphs.");
});
```

Save and Preview preview; the content of both paragraphs is replaced with the text "Replacing text in the paragraphs.":



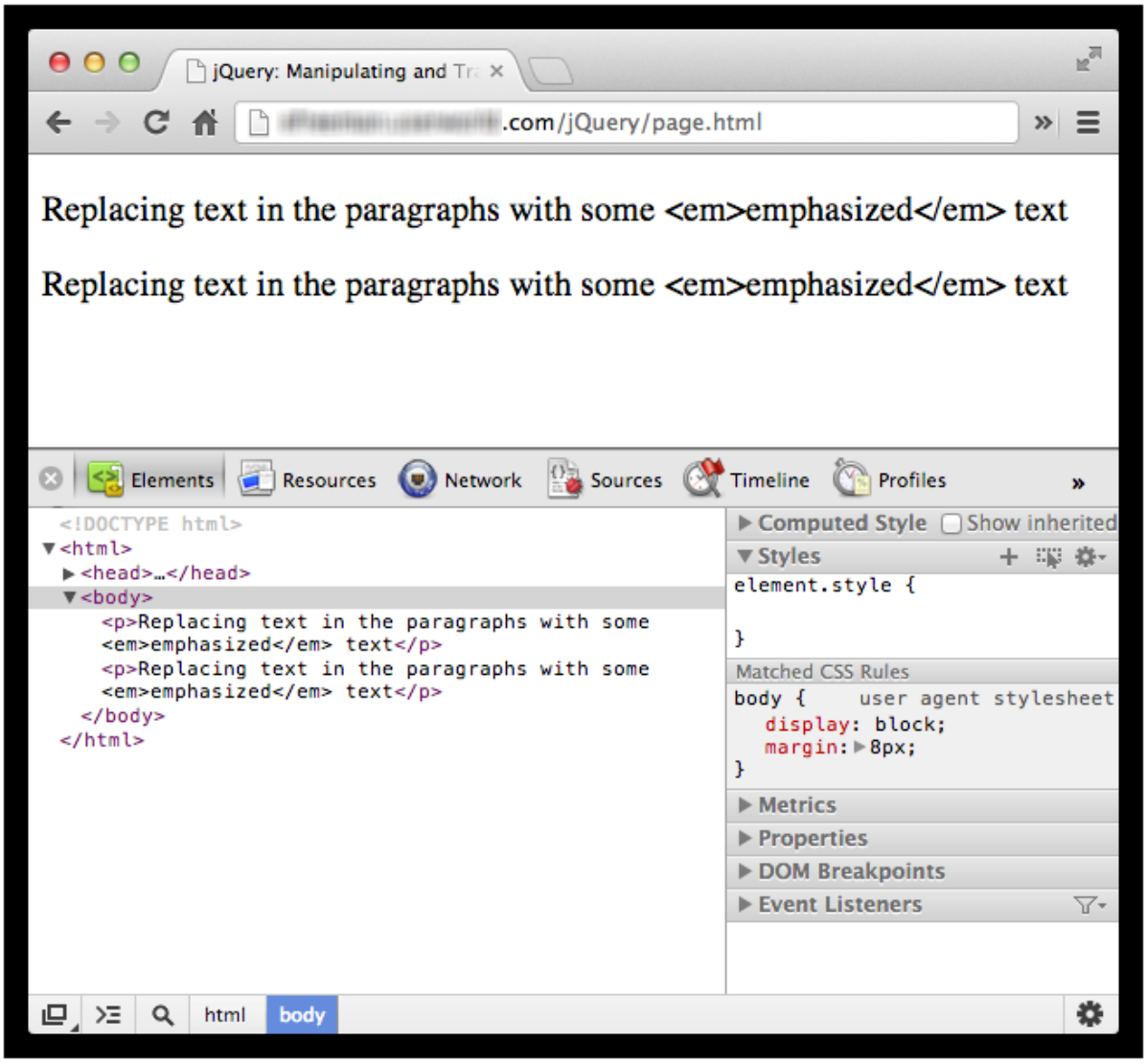
Remember that these methods are acting on *all* of the selected elements, and since we now have *two* paragraphs in the page, *both* paragraphs are selected with the "p" selector, and have their content changed by this code.

We use the **text()** method here to set the content of the <p> elements. That works because we're updating only the text content of the paragraphs; that is, we're not adding any new HTML elements to the paragraphs. Let's see what happens if we try to use the **text()** method to add HTML:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    ~~$("p").text("Replacing text in the paragraphs.");~~
    $("p").text("Replacing text in the paragraphs with some <em>emphasized</em>
text.");
});
```

Save and Preview preview. The page looks like this:

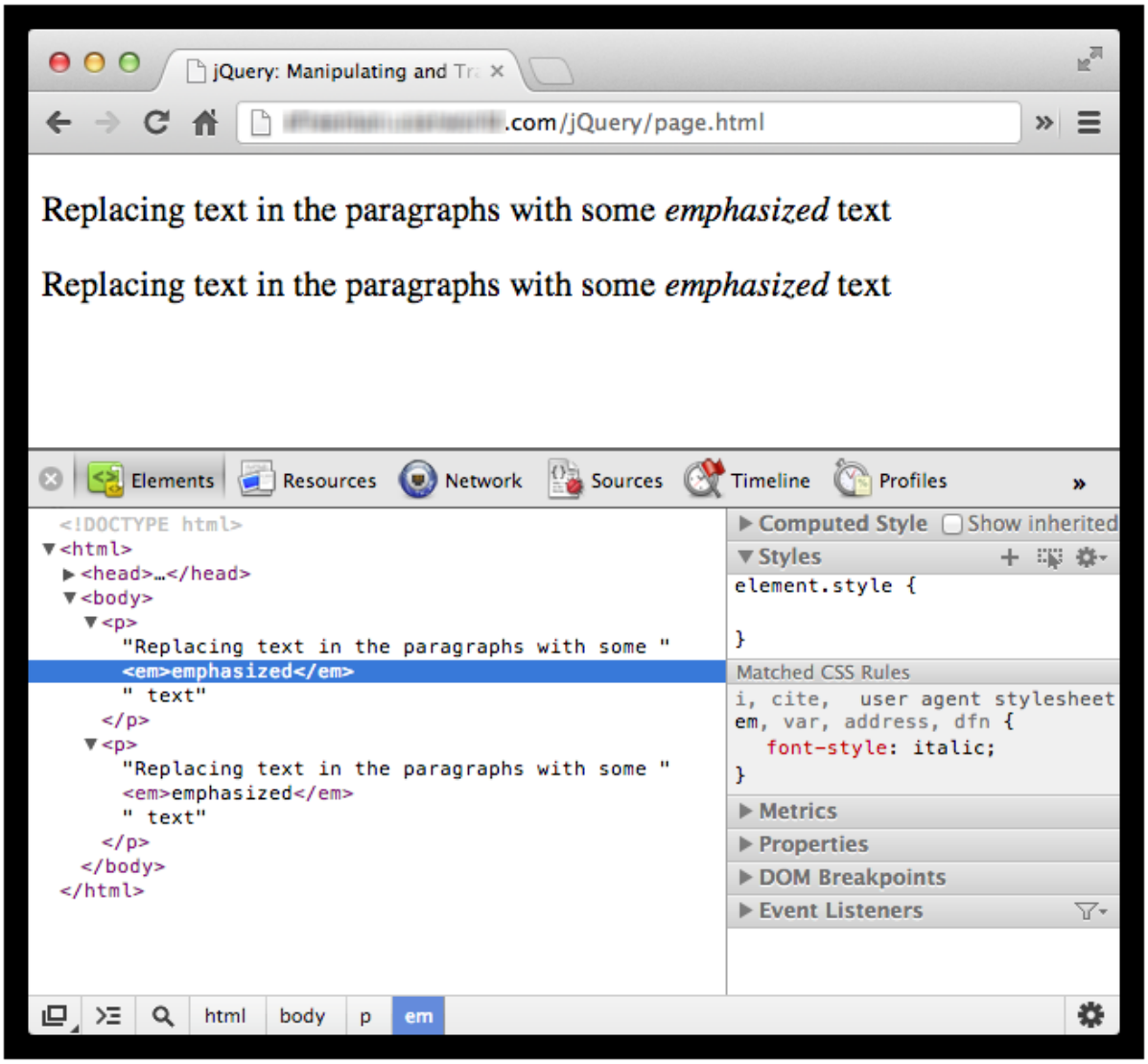


The HTML was added to the page, but not as an HTML element; instead, it was added as text, which probably isn't what we want. Change the **text()** to **html()** and try again:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").~~text~~html("Replacing text in the paragraphs with some <em>emphasized</
em> text.");
});
```

Save and Preview preview again. Now we have emphasized text in the paragraphs, and if you look at the DOM tree in the browser, you'll see that the element is being treated as an element, not just text.



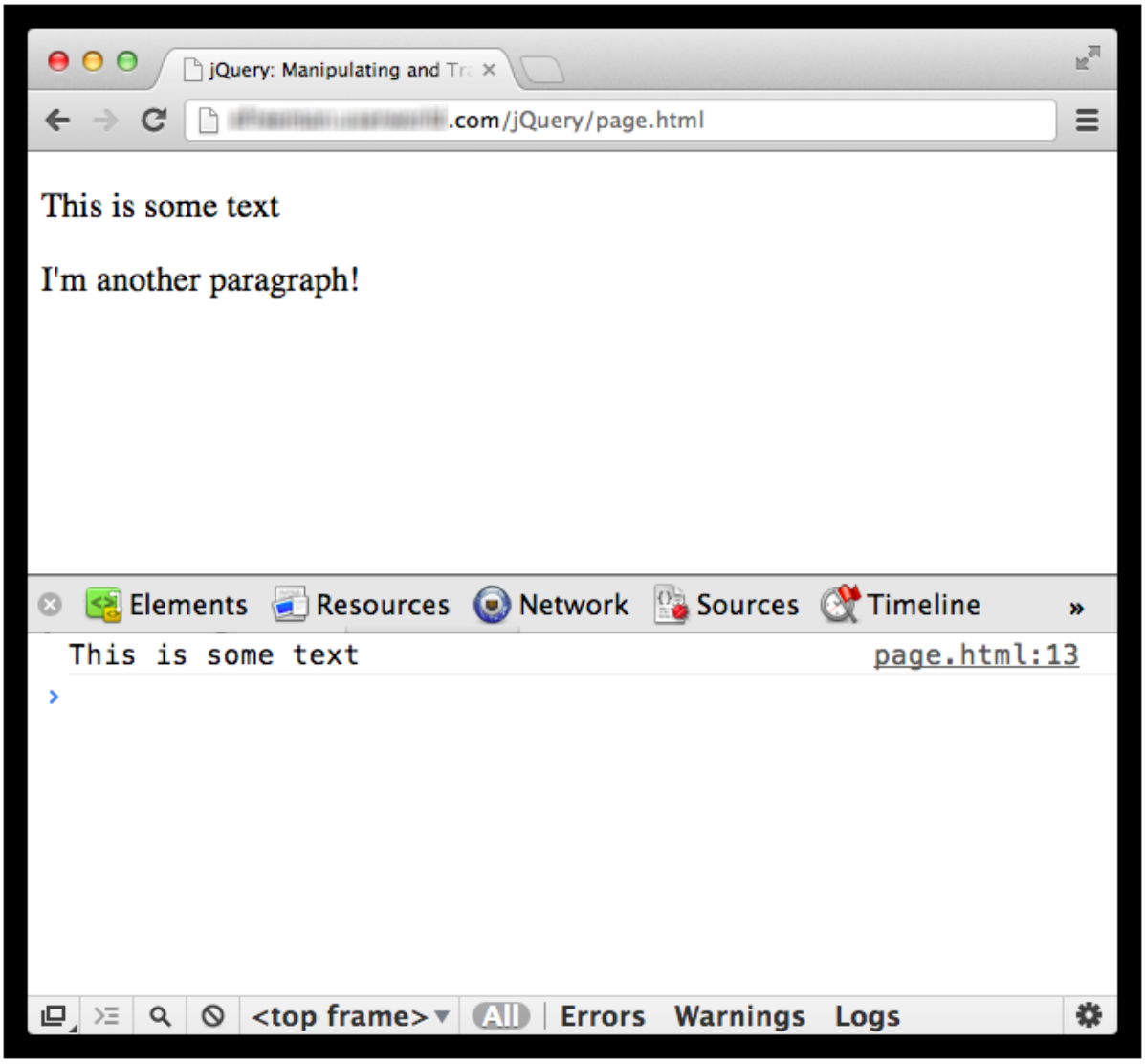
So, you use **html()** when you want to set the HTML content of an element and you use **text()** when you just want to set the text content.

You can use **html()** and **text()** to *get* the HTML content and text content of an element, respectively. However, there is an important difference in how these two methods work when you're *getting* content from an element:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").html("Replacing text in the paragraphs with some <em>emphasized</em> text.");
    console.log($("p").html());
});
```

We use the **html()** method to get the HTML content of all elements selected by **$("p")**, which in this case means both of the paragraphs we've added. So, what would you expect back from **html()**? You might expect to get an array of HTML content, or perhaps all the HTML content from both paragraphs concatenated together. Save and Preview preview, and then open the JavaScript console to see the result (reload the page if you don't see the output from **console.log()** there):



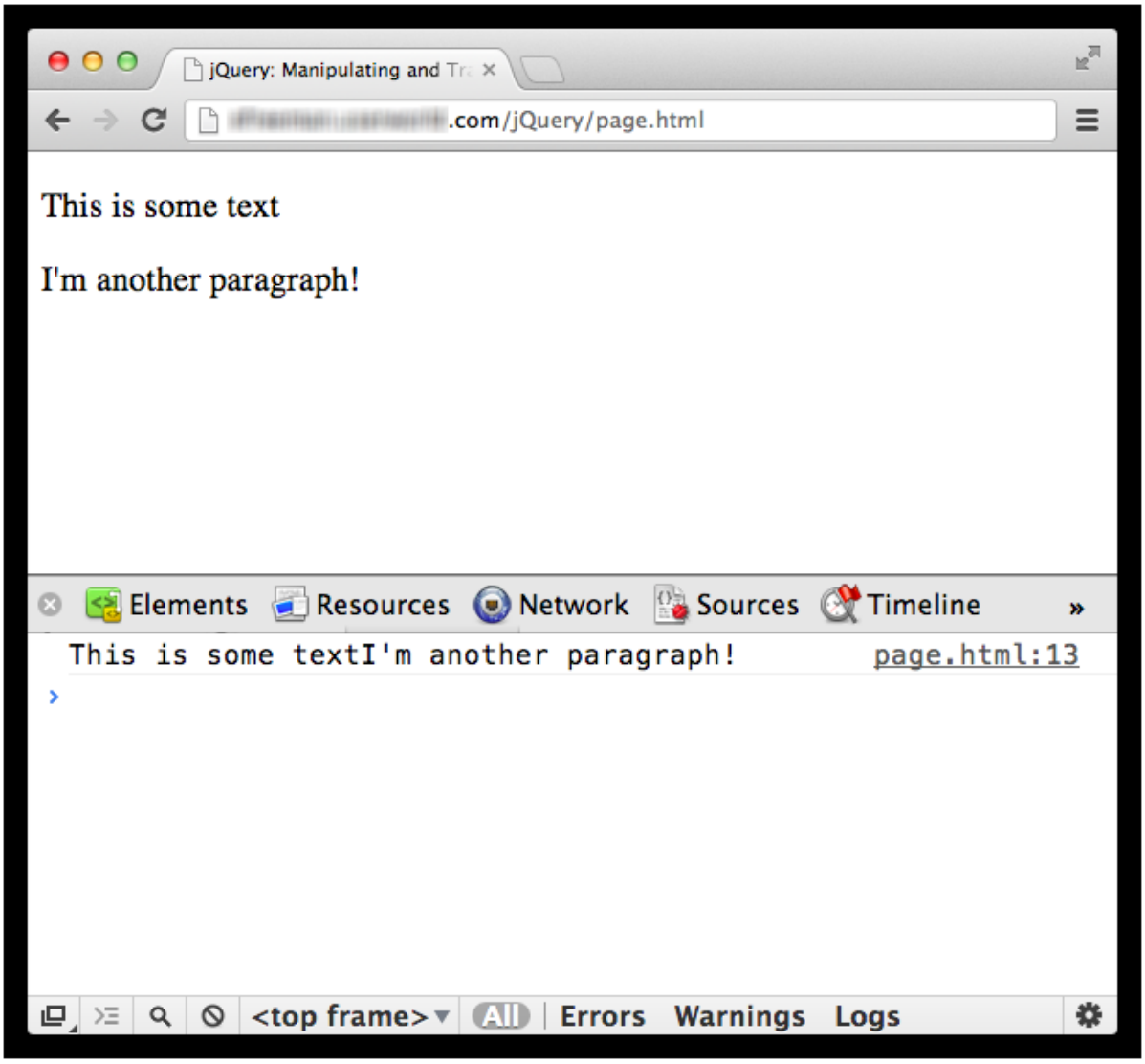
The **html()** returns only the HTML from the *first* element selected by the selector, which in this case is the paragraph with the text, "This is some text."

Now try changing **html()** to **text()**:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    console.log($("p").~~html~~text());
});
```

Save and Preview preview, and open the JavaScript console to see the result.



$("p").text() gives you a string containing *all* the text content from *both* paragraphs, concatenated together!

Keep this difference in mind when you're using **html()** and **text()** to get content.

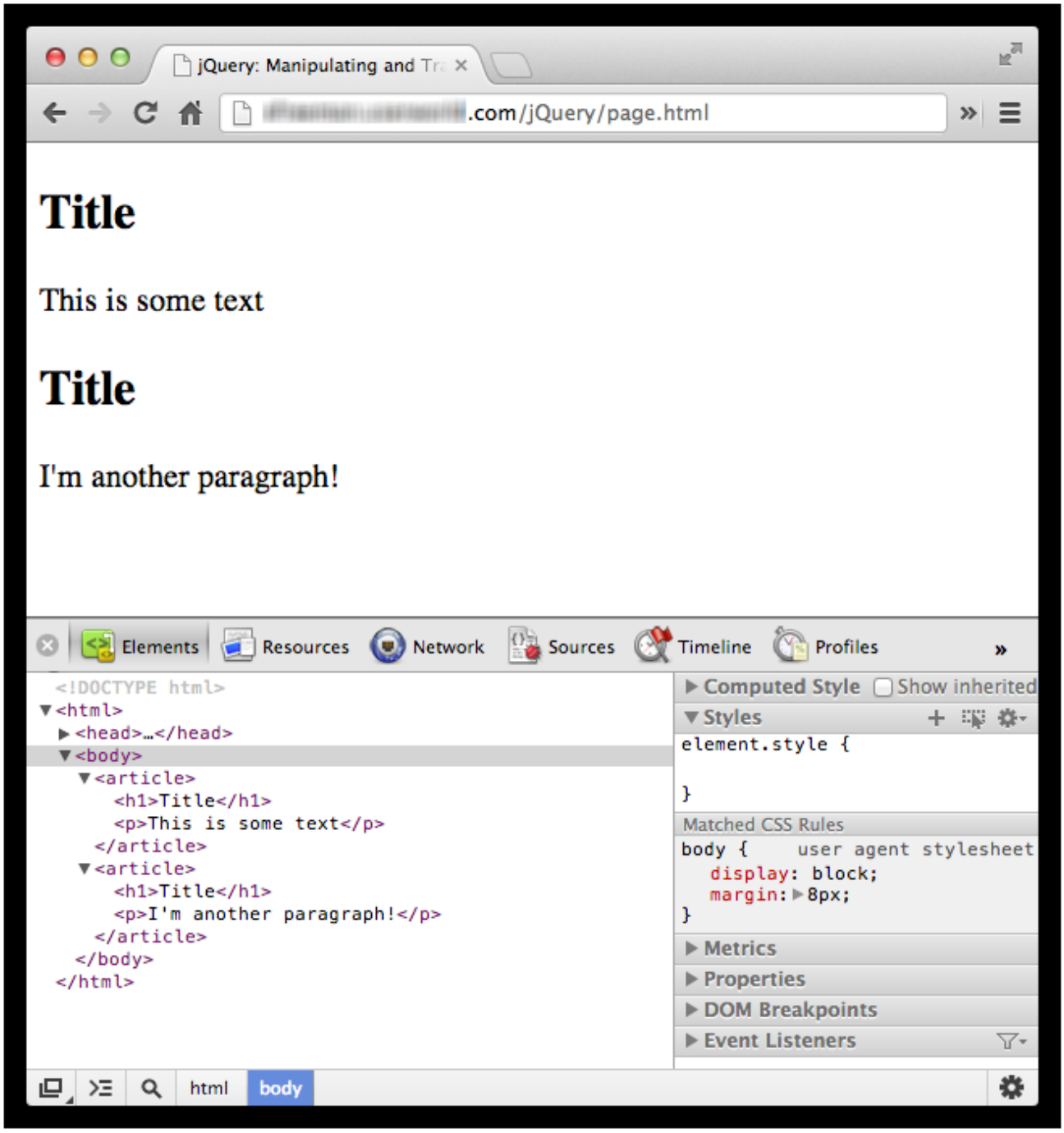
More Ways to Add Content

While you'll probably use **html()**, **text()**, and **append()** most often in your programs, jQuery offers a variety of methods to add content to your page. Let's take a look at a few others:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    console.log($("p").text());
    $("p").wrap("<article></article>");
    $("p").before("<h1>Title</h1>");
});
```

Save and Preview preview:



OBSERVE:

```
$("p").wrap("<article></article>");
$("p").before("<h1>Title</h1>");
```

wrap() wraps *each* <p> element with an **<article>** element, and **before()** adds content *before* the selected elements. In this case, we add a Title heading before each paragraph, nested within the article. Look carefully at the HTML that is generated in the DOM Tree view in your console and make sure you understand exactly what these two methods do.

Removing Content

jQuery provides lots of methods for adding content; but we'll need a way to remove content too. We can use the **remove()** method for that. First, let's add some content:

CODE TO TYPE:

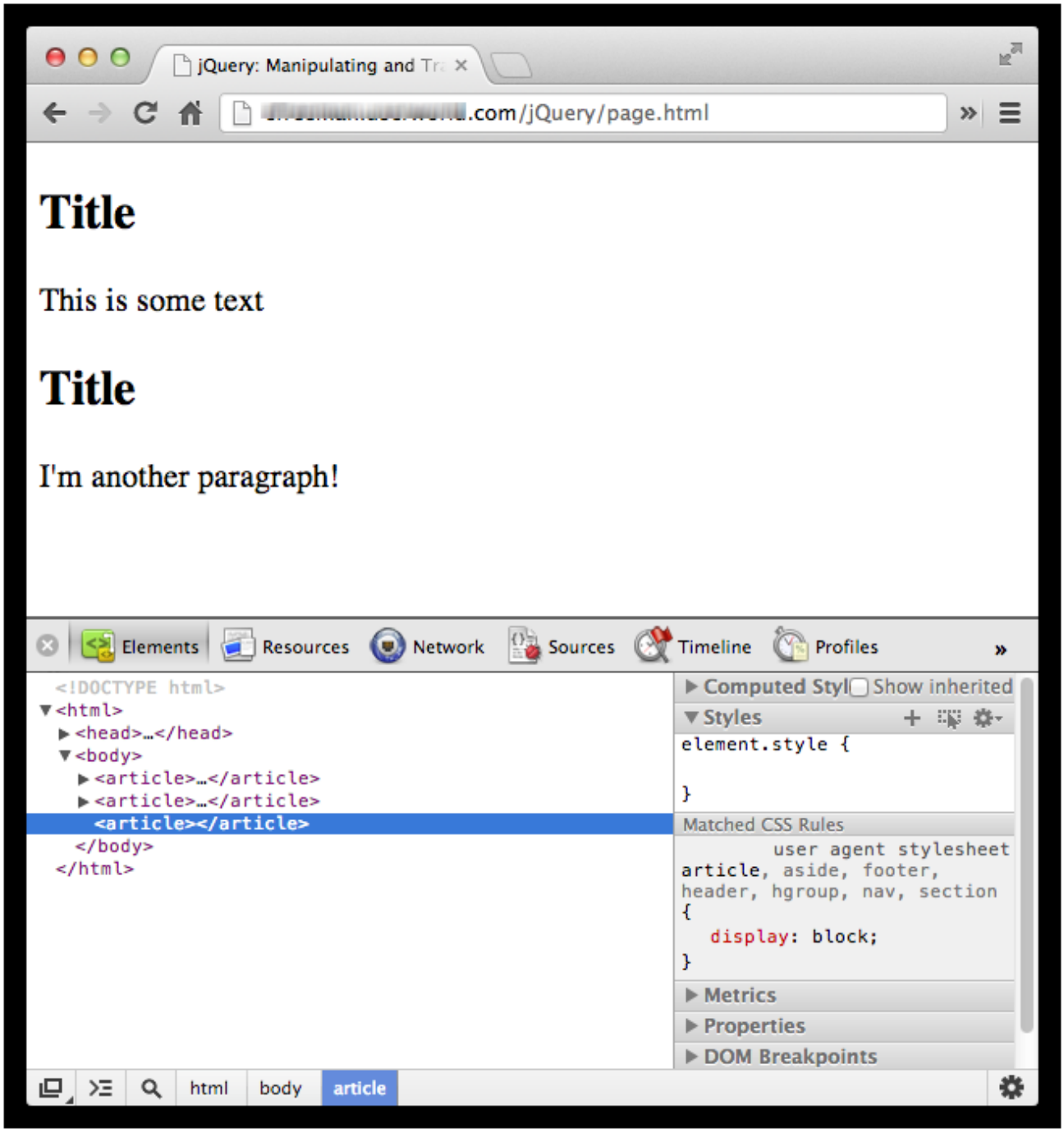
```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").wrap("<article></article>");
    $("p").before("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
});
```

 Save and  preview. The new content is added to the page. Now let's try removing that content:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").wrap("<article></article>");
    $("p").before("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("div").remove();
});
```

 Save and  preview. Now you don't see the text content that was in the article because we've removed the element that contained that text:



The <article> element is still there (it just doesn't have any content). The **remove()** method removes all the selected elements; in this case, there was only one element in the page, so only one element was removed. If you wrote **$("article").remove()** instead, all the <article> elements would be removed from the page, which would remove all the content within the <body>!

Try changing **remove()** to **empty()**:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").wrap("<article></article>");
    $("p").before("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("div").remove()empty();
});
```

 Save and  preview. The element is still there. **empty()** removes all the content within the selected element, but not the element itself. Compare that to **remove()**, which removes the selected element and all the content within it.

Traversing the DOM

Getting an Element's Children

We can do a lot with selectors to find the element we want from the DOM, but there are some methods that can help us find elements in the page. Update your code to add a title to the page (outside the <article> elements), by prepending it to the <body>. Note that **prepend()** works like **append()** except that the content is added to the selected element *above* all the other content in that element:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").wrap("<article></article>");
    $("p").before("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("div").empy();
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("body").children().addClass("highlight");
});
```

After we add the heading to the top of the page, we use the **children()** method to select all the child elements of the <body> element. This selects *all* the elements within the <body>, no matter what kind of elements they are.  Save and  preview. The heading and all the articles are highlighted:

Now, try changing the selector to highlight the elements to "article" instead, and notice the difference:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text</p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("bodyarticle").children().addClass("highlight");
});
```

Save and Preview preview, and you'll see that the heading is no longer highlighted, because now we're selecting only children of the <article> elements (the headings and paragraphs).

Getting an Element's Parent

You can get all of an element's children with the **children()** method; you can also get an element's parent

with the **parent()** method. Let's take another quick look at **parent()**:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("article").children().addClass("highlight");
    $("a").click(function() {
        $(this).parent().addClass("highlight");
    });
});
```

We add an <a> element to the first paragraph, and then add a click handler for that <a> element. In the click handler, we get the <a> element you clicked on (in the $(this) variable), then get that <a> element's parent (which is the <p> element that contains it), and then add the "highlight" class to it, so the whole paragraph is highlighted.  Save and 

preview to see how it works.

Getting an Element's Next Sibling

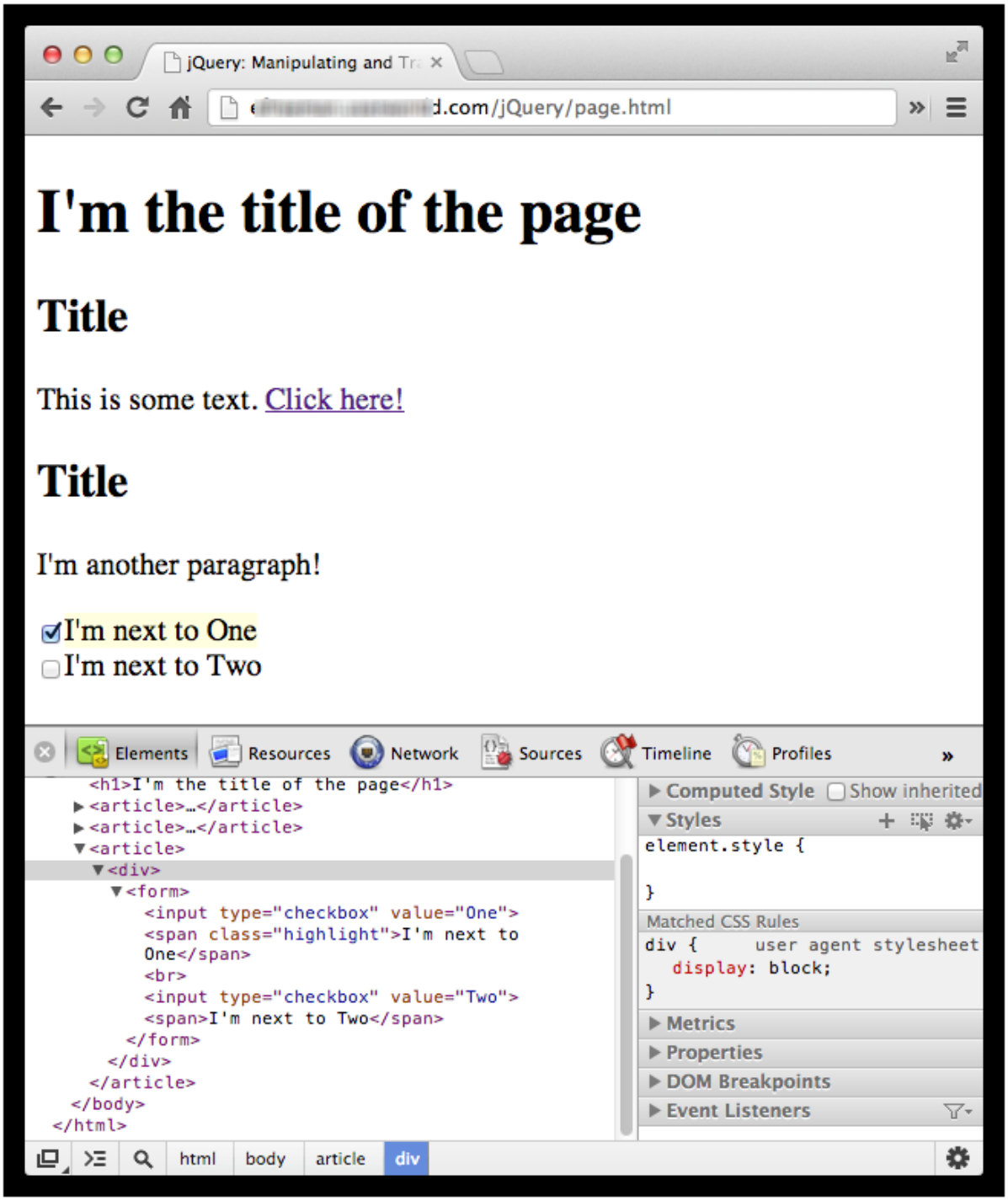
SO, you can get an element's parent, an element's children, but what about its *siblings*? This is a real family affair! If you think about the DOM tree, an element's siblings are all of the elements at the same level in the tree. Let's create some siblings; change the content in the element that we added to the third article in the page:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("a").click(function() {
        $(this).parent().addClass("highlight");
    });
    $("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'
m next to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to
 Two</span></form>");
    $("input:checkbox").click(function() {
        $(this).next().toggleClass("highlight");
    });
});
```

 Save and 

preview and look at the generated HTML. Try checking one of the checkboxes. The text next to it is highlighted. Think for a moment about how you'd select the element next to the checked item using a selector. You could always get the checked <input> element's parent, and then select the element from there, but there is a more efficient way.



OBSERVE:

```
$("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'm ne
xt to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to Two
</span></form>");
$("input:checkbox").click(function() {
    $(this).next().toggleClass("highlight");
});
```

We can get the element next to the checked <input> using the **next()** method. This method finds the next sibling of the selected element. In our case, the selected element is the checkbox that you just clicked (in **$(this)**). Because the checkbox <input> element and the next to it are at the same level in the DOM (they are both children of the <form> element), we can use **next()** to get the as a *sibling* of the checkbox <input> element.

Getting the First or Last Element in a Result Set

What if you want to get the first <input> element in the <form>? If you write **$("input:checkbox")**, you'll select both <input> elements. You could use **get(0)**, save the resulting <input> element in a variable, and then wrap it, and use it to do whatever it is you were going to do, but there's a better way:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph!</p>");
    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("a").click(function() {
        $(this).parent().addClass("highlight");
    });
    $("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'
m next to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to
 Two</span></form>");
    $("input:checkbox").click(function() {
        $(this).next().toggleClass("highlight");
    });
    $("input:checkbox").first().next().text("I am still next to One!!");
});
```

 Save and  preview; the text in the next to the first <input> element has changed. We use the **first()** method to reduce the set of selected elements (in this case, that's two <input> elements) to just the first one. Then we use **next()** to get the element, and use **text()** to change the text content of that . Whew!

Can you guess which method you'd use to get the *last* element in a set of selected elements? You got it—the **last()** method. Experiment with using **last()** in your code.

Finding Elements

In an earlier lesson, we learned how to select elements within a *context*: usually a selected element. Let's update our program to include another link in a paragraph (so both of our paragraphs contain links), and then create a click handler for the paragraphs. Then we can select the <a> element inside the clicked paragraph like this:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph! <a href=\"#\">Another link!</a></p>");

    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("a").click(function() {
        $(this).parent().addClass("highlight");
    });
    $("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'm next
to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to Two</span></f
orm>");
    $("input:checkbox").click(function() {
        $(this).next().toggleClass("highlight");
    });
    $("input:checkbox").first().next().text("I am still next to One!!");
    $("p").click(function() {
        $("a", this).addClass("highlight");
    });
});
```

Save and Preview preview, then click anywhere on one of the paragraphs. The link in that paragraph becomes highlighted. Try the other paragraph too.

We added a click handler to both paragraphs, and then selected the <a> element in the paragraph you just clicked on, using:

OBSERVE:

```
$("a", this).addClass("highlight");
```

This says, "Find the <a> elements that are nested inside the element that's in the **this** variable." In our case, the context is the element in **this**, which is the <p> element we clicked on, so the selector finds the <a> element within that paragraph (and not the other <a> element in the page).

Note You can use either a DOM element like **this**, or a jQuery object like **$(this)**, as the context in a selector.

Another way of doing the same thing is to use the **find()** method:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph! <a href=\"#\">Another link!</a></p>");

    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'm next
to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to Two</span></f
orm>");
    $("input:checkbox").click(function() {
        $(this).next().toggleClass("highlight");
    });
    $("p").click(function() {
        $("a", this).addClass("highlight");
        $(this).find("a").addClass("highlight");
    });
});
```

Save and Preview preview. The program behaves the same way as before, except now we use the **find()** method to find all <a> elements in the selected <p> element.

Sometimes you'll want to find out if an element contains, or *has*, another element inside of it before actually looking for that nested element:

CODE TO TYPE:

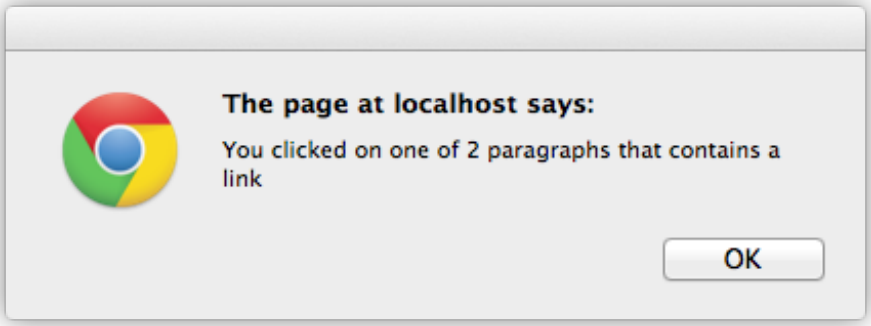
```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph! <a href=\"#\">Another link!</a><</p>");

    $("body").append("<p>I don't have a link!</p>");
    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'm next
to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to Two</span></f
orm>");
    $("input:checkbox").click(function() {
        $(this).next().toggleClass("highlight");
    });
    $("p").click(function() {
        $(this).has("a").find("a").addClass("highlight");
    });
});
```

 Save and  preview, then click on each of the three paragraphs (we added a third paragraph without a link in it for testing). Your code will work the same way as before, but now we test to see if the paragraph contains an <a> element with **has()** before we **find()** the element. In our case, this code doesn't change the behavior, so **has()** doesn't really add anything, but you can use **has()** to do things like find out how many paragraph elements contain links, like this:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph! <a href=\"#\">Another link!</a><</p>");

    $("body").append("<p>I don't have a link!</p>");
    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'm next
to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to Two</span></f
orm>");
    $("input:checkbox").click(function() {
        $(this).next().toggleClass("highlight");
    });
    $("p").click(function() {
        $(this).has("a").find("a").addClass("highlight");
        var numParasWithLinks = $("p").has("a").length;
        alert("You clicked on one of " + numParasWithLinks + " paragraphs that contains
 a link");
    });
});
```

 Save and  preview again, then click on a paragraph. You see an alert message:



We use the **has()** method to find all paragraphs that "have" an <a> element. That creates an array of elements (in our case, two elements, since we have two paragraphs that contain an <a> element). We can then find out how many paragraphs contain at least one link by using the **length** property of the array.

Testing for Elements

Once you've found an element, we have a couple of useful methods for to find out if you've got the right one: **eq()** and **is()**. Let's check them out:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph! <a href=\"#\">Another link!</a><</p>");

    $("body").append("<p>I don't have a link!</p>");
    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'm next
to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to Two</span></f
orm>");
    $("input:checkbox").click(function() {
        $(this).next().toggleClass("highlight");
    });
    $("p").click(function() {
        $(this).find("a").addClass("highlight");
        var numParasWithLinks = $("p").has("a").length;
        alert("You clicked on one of " + numParasWithLinks + " paragraphs that contains
 a link");
    });
    var item = Math.floor(Math.random() * 2);
    console.log(item);
    $("input:checkbox").eq(item).attr("checked", "checked");
});
```

 Save and  preview. One of the checkboxes is checked. Reload the page a few times and observe which box is checked. It changes randomly between the first and second box. Look at the JavaScript console to see the number that's being randomly generated (a 0 or 1); it corresponds to which checkbox is checked each time you reload the page.

We use the **eq()** function to do this:

OBSERVE:

```
var item = Math.floor(Math.random() * 2);
console.log(item);
$("input:checkbox").eq(item).attr("checked", "checked");
```

First we **generate a random number, 0 or 1**, and store that in the **item** variable. Then we select **all the checkbox <input> elements** in the page. Then we use **eq()** to refine the selection to the element at the index in the **item** variable. Remember that the object returned from a jQuery selector, like **$("input:checkbox")**, is an array of elements, so **eq()** finds the element at a specific index in that array. If **item** is 0, we select the first checkbox in the page; if **item** is 1, we select the second one (remember that JavaScript arrays start at index 0).

Once we select a specific element at the index in **item**, we check it by **setting its checked attribute to "checked"**.

You can use **eq()** when you need to find an element at a specific location. This comes in handy when targeting, say, an element at a specific row or column in a grid of elements.

Another useful jQuery method for testing elements is the **is()** method:

CODE TO TYPE:

```
$(document).ready(function() {
    $("body").html("<p>This is some text. <a href=\"#\">Click here!</a></p>");
    $("body").append("<p>I'm another paragraph! <a href=\"#\">Another link!</a><</p>");

    $("body").append("<p>I don't have a link!</p>");
    $("p").wrap("<article></article>");
    $("p").prepend("<h1>Title</h1>");
    $("body").append("<article><div>I'm a div in an article</div></article>");
    $("body").prepend("<h1>I'm the title of the page</h1>");
    $("article div").html("<form><input type=\"checkbox\" value=\"One\"><span>I'm next
to One</span><br><input type=\"checkbox\" value=\"Two\"><span>I'm next to Two</span></f
orm>");
    $("input:checkbox").click(function() {
        $(this).next().toggleClass("highlight");
        var $firstInput = $("input:checkbox").eq(0);
        if ($(this).is($firstInput)) {
            alert("You clicked the first checkbox");
        } else {
            alert("You clicked a checkbox that isn't the first one");
        }
    });
    $("p").click(function() {
        $(this).find("a").addClass("highlight");
    });
    var item = Math.floor(Math.random() * 2);
    console.log(item);
    $("input:checkbox").eq(item).attr("checked", "checked");
});
```

 Save and 

preview. Click one of the checkboxes, and you see an alert indicating whether you clicked the first checkbox or not. Let's take a closer look at how we did this:

OBSERVE:

```
$("input:checkbox").click(function() {
    $(this).next().toggleClass("highlight");
    var $firstInput = $("input:checkbox").eq(0);
    if ($(this).is($firstInput)) {
        alert("You clicked the first checkbox");
    } else {
        alert("You clicked a checkbox that isn't the first one");
    }
});
```

First, we **select all the checkbox <input> elements**, and then use the **eq(0)** method to get just the first one (exactly like doing **$("input:checkbox").first()**). Then we **compare the checkbox element we just clicked on (in $(this)) with the first checkbox** using the **is()** method. This checks to determine whether the two elements are the same. If they are, the result is true, and you see an alert indicating you clicked on the first checkbox, otherwise, you must have clicked on the second one and you'll see a message indicating that.

You might be wondering if you can write:

OBSERVE:

```
if ($(this) == $firstInput){
```

Nope, this won't work (but give it a try if you want). The JavaScript operator **==** won't work to compare two objects (it just works with primitives, like numbers and strings, as well as values like null).

So use **is()** instead when comparing two objects like we're doing here.

In this lesson you learned about jQuery methods that allow you to add or remove content, find content, and test content. We covered quite a few methods, but there are more. There's usually more than one way to accomplish a task in jQuery; jQuery provides a variety of methods that you can use in combination to achieve your goals.

Take some time to look through the jQuery API documentation on both traversing and manipulating content. Check out all the different methods and understand their differences, so you can choose the best one for your program. You'll probably discover even more new and useful methods as you work through your project.

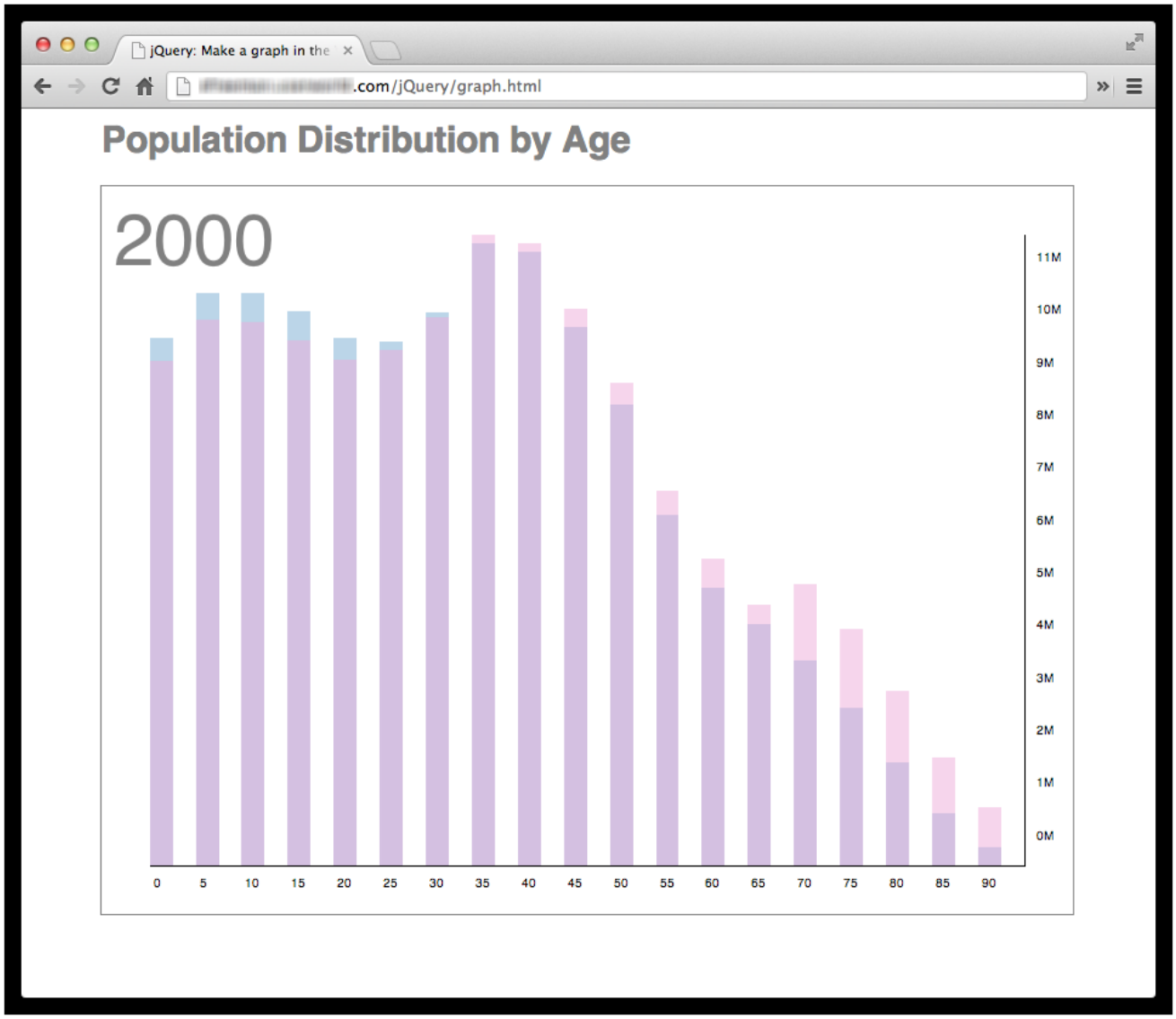
Positioning Elements in the Page

Lesson Objectives

- You will use jQuery to get and set the width, height, and positions of elements.
- You will use the **appendTo()** method.

Element Metrics

These days, you can create some impressive data graphics, games, and animations in the browser using the <canvas> element, SVG, or one of the many JavaScript libraries and jQuery plugins that are available online. Even without these tools, it's possible to do a lot with plain old DOM elements and a little jQuery. In this lesson, you'll build a population distribution bar chart, like this:



In order to create graphics like this, it's important to understand *element metrics* and how elements are positioned. Element metrics are properties of an element, like its width and height, and its position within the browser window, or **viewport** (the visible portion of the page in the browser window). Before we start building the graph, let's go over the basics of element metrics.

Getting Some Elements in Place

To see how you can access the various element properties like width, height, and position, create this simple HTML page to play with:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Playing with the Viewport</title>
  <meta charset="utf-8">
  <style>
body {
    position: relative;
}
div {
    position: relative;
    width: 200px;
    height: 200px;
    background-color: rgb(158, 190, 166);
    border: 1px solid black;
}
div#two {
    width: 600px;
}
div#four {
    position: absolute;
    bottom: 5px;
    right: 5px;
    width: 70px;
    height: 70px;
    background-color: lightyellow;
}
ul {
    list-style-type: none;
    padding: 0px;
    margin: 0px;
}
div#popover {
    position: absolute;
    background-color: white;
    border: 1px solid black;
    box-shadow: 3px 3px 5px black;
}
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="viewport.js"></script>
</head>
<body>
  <div id="one">One</div>
  <div id="two">Two</div>
  <div id="three">
    Three
    <div id="four">Four</div>
  </div>
  <div id="popover"> </div>
</body>
</html>
```

 Save this file in your **/jQuery** folder as **viewport.html**. Some important features of this file are:

- The arrangement and nesting of the elements: the "four" is nested within the "three" , but the "one," "two," and "three" s are all at the same level in the DOM.
- The positioning of the elements: the "one," "two," and "three" s are positioned "relative" (which means they are positioned in their normal position in the flow, and we're not modifying their positions with the "top" and "left" properties), while the "four" is positioned absolute (relative to the position of "three").
- The widths and heights of each of the elements: the "one" and "three" s are 200px by 200px; the "two" is 200px x 600px, and the "four" is 70px x 70px.

Once you load the page, you'll see how all this looks, but before you do, create a new file with jQuery to hide the "popover" :

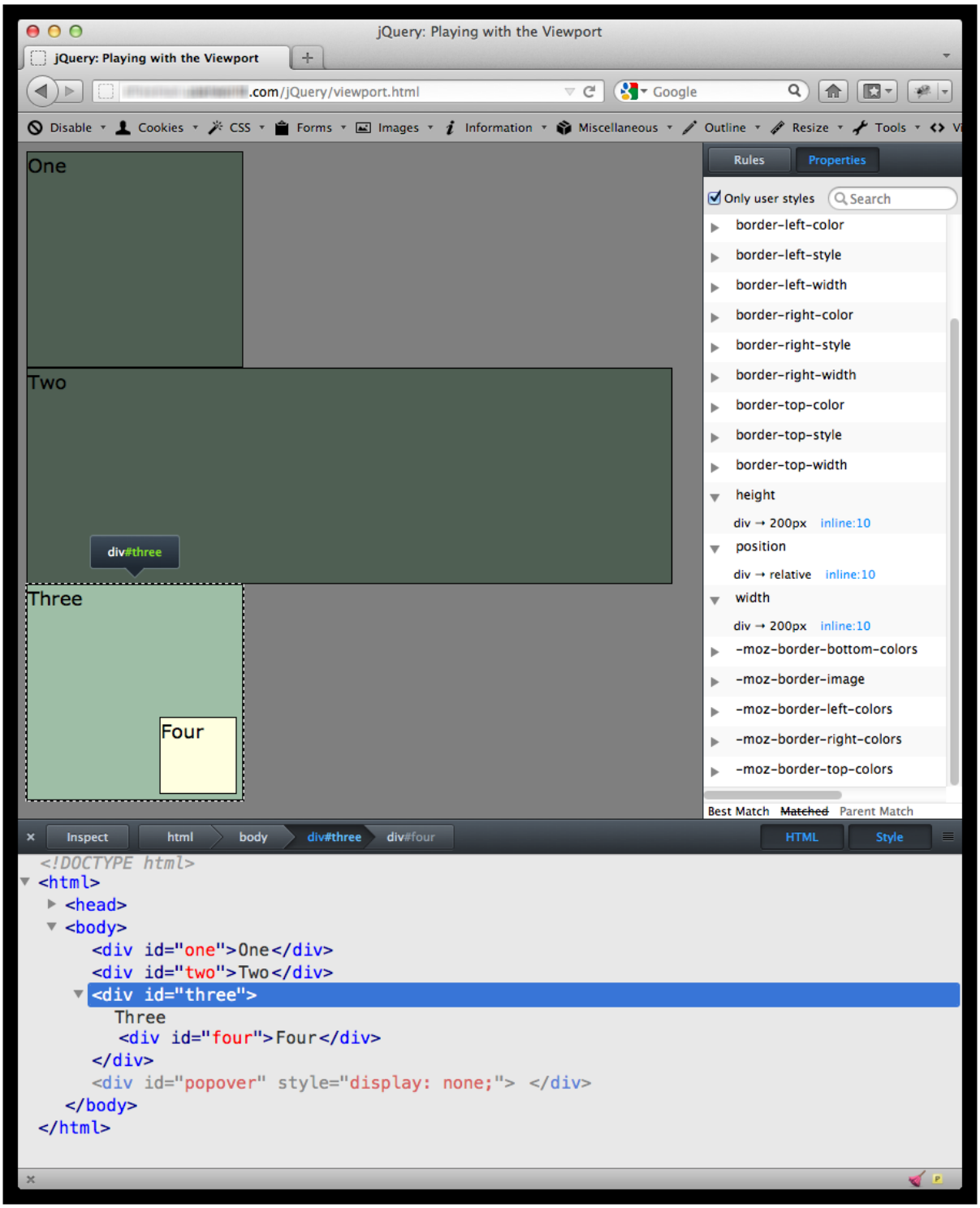
CODE TO TYPE:

```
$(document).ready(function() {
    var $popover = $("div#popover");
    $popover.hide();
});
```

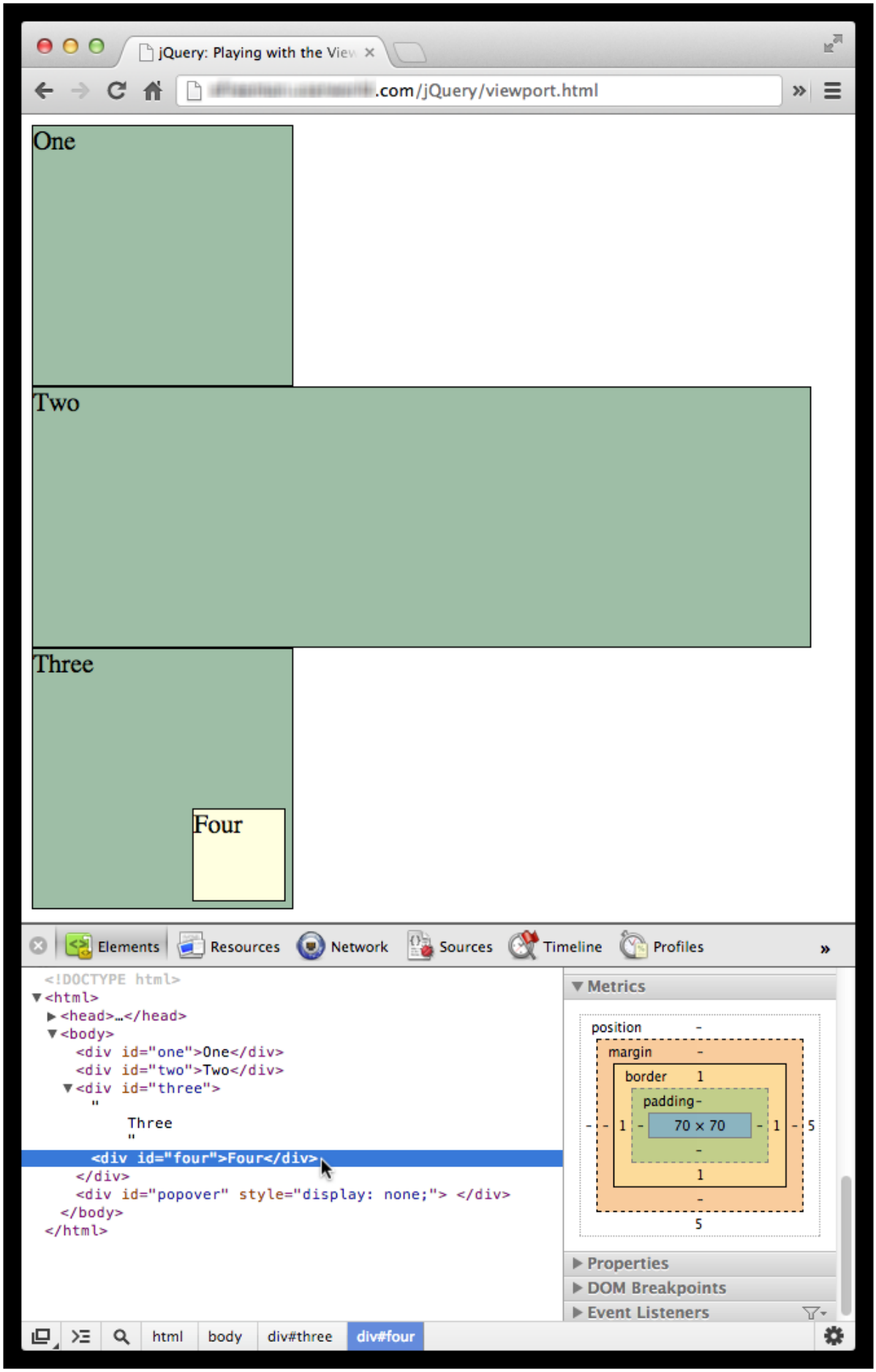
 Save this file in your **/jQuery** folder as **viewport.js**. 

Preview your **viewport.html** file, open the developer tools in your browser, and bring up the DOM tree view.

The positions and widths and heights of each of the elements are as you'd expect. Now, click on a in the DOM tree view. Click on one of the "one," "two," "three," or "four" s. On the right side of the console window in Chrome and Safari, look at the **Computed Style** for the selected element to see the various properties computed by the browser for the position, the width, the height, and so on. If you're using Firefox, you can also view this information by choosing the **CSS** tab in the Developer Tools window:



If you're using Chrome, you'll find a **Metrics** section at the lower right. Open this up and you'll see a nice graphic showing you details about the selected element, like the width and height, padding, border, and margin:



This is a handy view of the element, so if you have Chrome available on your computer, give it a try.

Accessing Element Metrics with jQuery

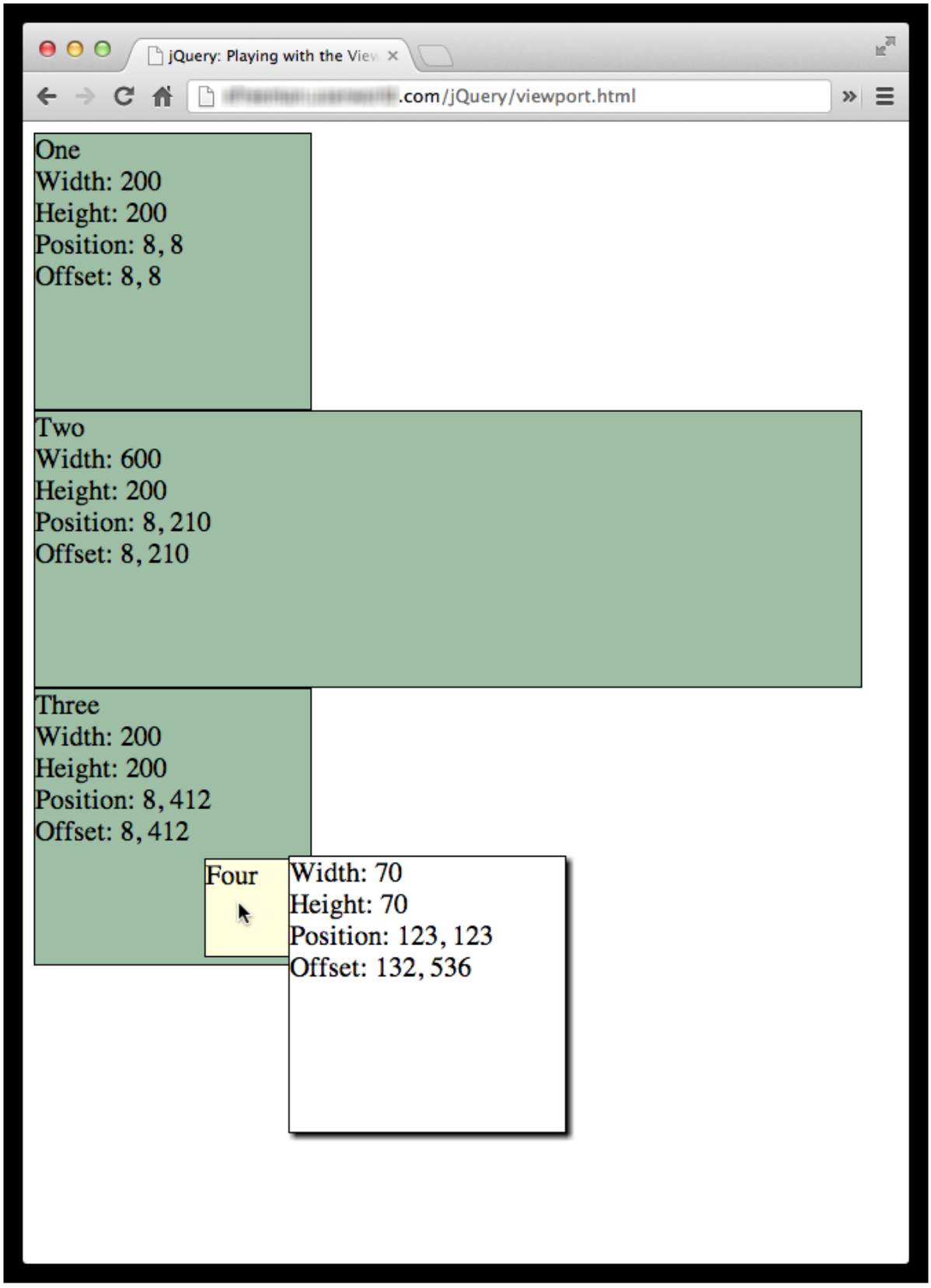
Now that we have some elements in place, let's take a look at how to access some of these element metrics using jQuery:

CODE TO TYPE:

```
$(document).ready(function() {
    var $popover = $("div#popover");
    $popover.hide();
    $("div").not("#popover").not("#four").each(function() {
        $(this).append("<ul></ul>");
        $("ul", this).append("<li>Width: " + $(this).width() + "</li>");
        $("ul", this).append("<li>Height: " + $(this).height() + "</li>");
        $("ul", this).append("<li>Position: " + $(this).position().left + ", " +
 $(this).position().top + "</li>");
        $("ul", this).append("<li>Offset: " + $(this).offset().left + ", " + $(t
his).offset().top + "</li>");
    });

    $("div#four").hover(function(evt) {
        $popover.empty();
        $popover.append("<ul></ul>");
        $("ul", $popover).append("<li>Width: " + $(this).width() + "</li>");
        $("ul", $popover).append("<li>Height: " + $(this).height() + "</li>");
        $("ul", $popover).append("<li>Position: " + $(this).position().left + ",
 " + $(this).position().top + "</li>");
        $("ul", $popover).append("<li>Offset: " + $(this).offset().left + ", " +
 $(this).offset().top + "</li>");

        $popover.css({ top: evt.pageY - 10, left: evt.pageX + 10 }).toggle();
    });
});
```

 Save this file and 

preview your **viewport.html** file. Some of the element properties are displayed in the elements "one," "two," and "three." To see the properties for the "four" , hover your mouse over it:



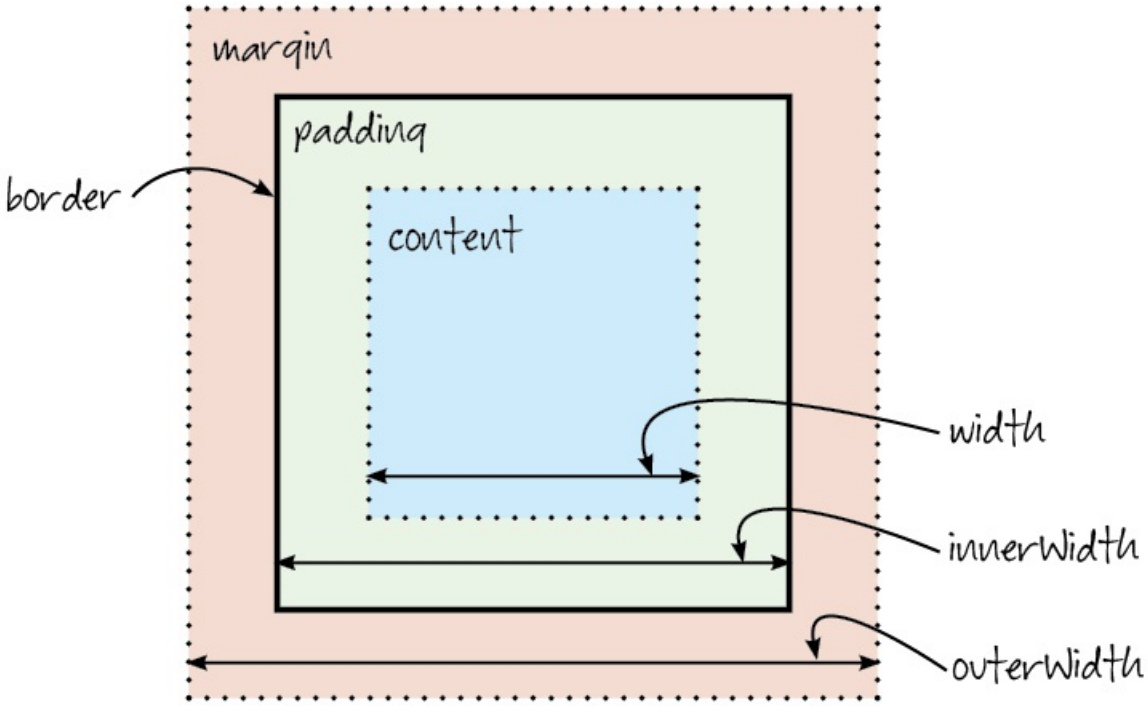
Let's step through how we get and show these properties in the page, and also what each of them means. Here's the portion of the code that shows the properties for the "one," "two," and "three" elements:

OBSERVE:

```
$("div").not("#popover").not("#four").each(function() {
    $(this).append("<ul></ul>");
    $("ul", this).append("<li>Width: " + $(this).width() + "</li>");
    $("ul", this).append("<li>Height: " + $(this).height() + "</li>");
    $("ul", this).append("<li>Position: " + $(this).position().left + ", " + $(t
his).position().top + "</li>");
    $("ul", this).append("<li>Offset: " + $(this).offset().left + ", " + $(this)
.offset().top + "</li>");
});
```

First, we create a function to run on **each of the "one," "two," and "three" elements** (we select them by selecting all the elements and then filtering out both the "popover" and the "four"). Then, we **add a <ul> element to the **, and then the rest of the code appends <li> items to this ** element**, each containing information about the element's metrics.

We add the **element's width** using the **width()** method. The width of an element refers to the *content*, not including any padding, border, and margin the element might have. In this case, our elements don't have any padding or margin, but they do have a border. When you specify an element's width in CSS, you're specifying the width of the *content*, and any padding, border, and margin you add on to that gets added to the total width of the element when it's displayed in the browser:



The **width()** method returns only the width of the *content*. What happens if you change this method to the **outerWidth()** method?:

CODE TO TYPE:

```
$(document).ready(function() {
    var $popover = $("div#popover");
    $popover.hide();
    $("div").not("#popover").not("#four").each(function() {
        $(this).append("<ul></ul>");
        $("ul", this).append("<li>Width: " + $(this).width()outerWidth() + "</li
>");
        $("ul", this).append("<li>Height: " + $(this).height() + "</li>");
        $("ul", this).append("<li>Position: " + $(this).position().left + ", " +
 $(this).position().top + "</li>");
        $("ul", this).append("<li>Offset: " + $(this).offset().left + ", " + $(t
his).offset().top + "</li>");
    });

    $("div#four").hover(function(evt) {
        $popover.empty();
        $popover.append("<ul></ul>");
        $("ul", $popover).append("<li>Width: " + $(this).width() + "</li>");
        $("ul", $popover).append("<li>Height: " + $(this).height() + "</li>");
        $("ul", $popover).append("<li>Position: " + $(this).position().left + ",
 " + $(this).position().top + "</li>");
        $("ul", $popover).append("<li>Offset: " + $(this).offset().left + ", " +
 $(this).offset().top + "</li>");

        $popover.css({ top: evt.pageY - 10, left: evt.pageX + 10 }).toggle();
    });
});
```

 Save this file and 

preview your **viewport.html** file. Now, the width of each element is 2 pixels bigger. That's because the **outerWidth()** method includes the border in the width it returns; because our border is 1px wide, when you compute the total width you add the width of the content to the widths of the left and right borders. For example, the "one" is 200px wide, plus 1px of left border and 1px of right border, so the **outerWidth()** is 202px.

There are several methods you can use to get metrics about your elements:

- **width()** returns the width of the *content*.
- **innerWidth()** returns the width of the *content* plus the *padding*.
- **outerWidth()** returns the width of the *content* plus the *padding* and *border*.
- **outerWidth(true)** returns the width of the *content* plus the *padding*, *border*, and *margin*.

As you'd expect, there are analogous methods for height: **height()**, **innerHeight()**, **outerHeight()**, and **outerHeight(true)**.

Go ahead and change **outerWidth()** back to **width()** before proceeding:

Update your jQuery:

```
$("ul", this).append("<li>Width: " + $(this).width()outerWidth() + "</li>");
```

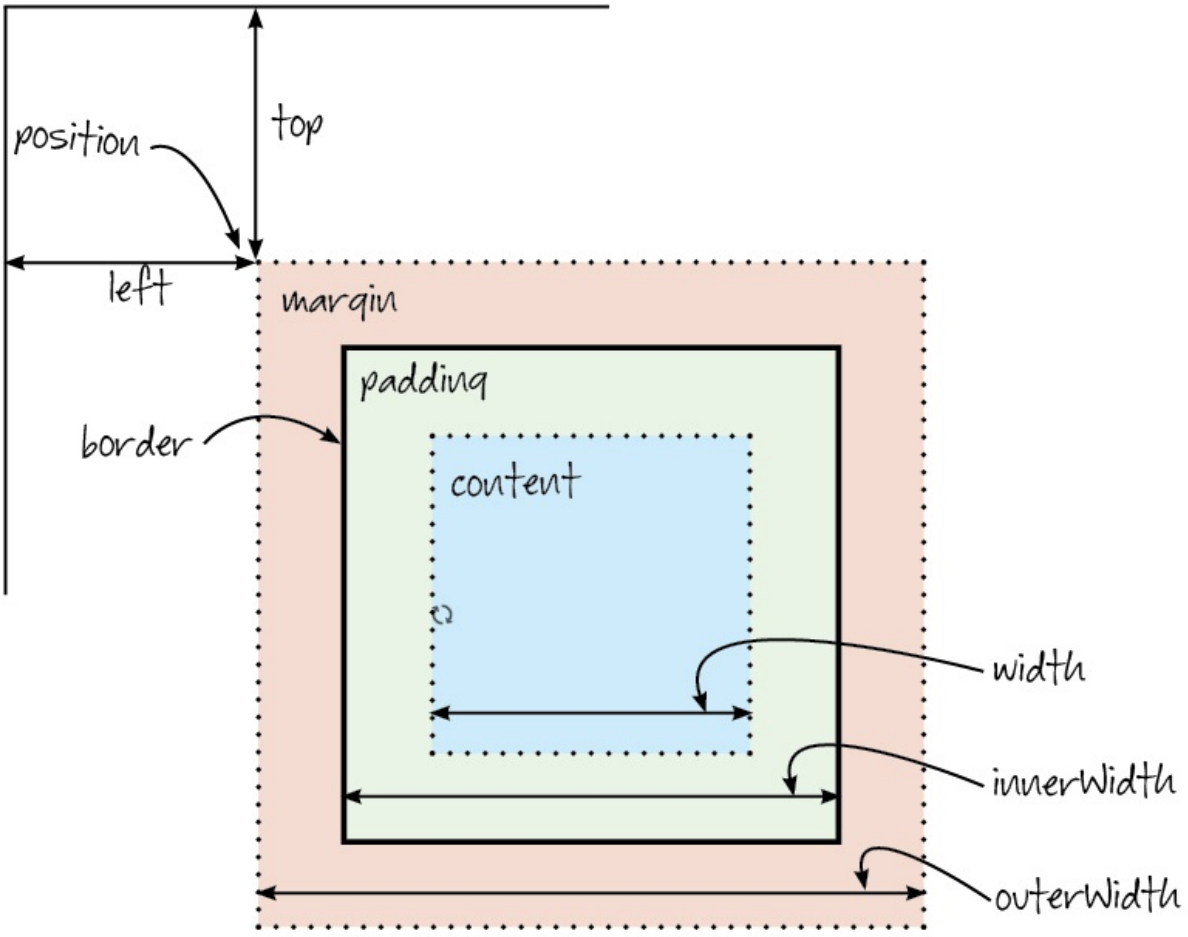
Let's continue going over the details of our code:

OBSERVE:

```
$("div").not("#popover").not("#four").each(function() {
    $(this).append("<ul></ul>");
    $("ul", this).append("<li>Width: " + $(this).width() + "</li>");
    $("ul", this).append("<li>Height: " + $(this).height() + "</li>");
    $("ul", this).append("<li>Position: " + $(this).position().left + ", " + $(t
his).position().top + "</li>");
    $("ul", this).append("<li>Offset: " + $(this).offset().left + ", " + $(this)
.offset().top + "</li>");
});
```

After **getting the width()** of the element, we get the**height()**, which works like you'd expect.

What about the next properties, **position().left**, **position().top**, **offset().left**, and **offset().top**? Both **position()** and **offset()** return information about an element's position in the page, but in two slightly different ways. Remember that when you position an element in the page, you usually position it a certain distance from the top left corner of the window, or of the element in which it's nested:



An element's position has two values: left and top. So does its offset. What's the difference? Look at the values for position and offset for the "one," "two," and "three" elements. The position and offset values are the *same* for these three s, but for "four," they are different. **position()** returns an object containing the number of pixels from the **left** and **top** of the *positioned parent*, while **offset()** returns an object containing the number of pixels from the **left** and **top** of the *document*.

For s "one," "two," and "three," the top left position is based on the <body> (the closest positioned parent), so the offset and position are the same. However, for "four," the closest positioned parent is "three," so the values you see for **position are relative to "three"**, while the values you see for **offset are relative to the <body>**.

For "four," we show exactly the same properties as for "one," "two," and "three," except that we show them in a "popover" , because the values won't fit inside the itself.

The position for the left is 123, and it's the same for the top (123); the offset left is 132, and for the top it's 536

(these measurements are all in pixels). If you look back at the CSS, you'll see that we positioned "four" 5px from both the bottom and right of "three":

OBSERVE:

```
div#four {
    position: absolute;
    bottom: 5px;
    right: 5px;
    width: 70px;
    height: 70px;
    background-color: lightyellow;
}
```

We know the width of the content in "three" is 200 x 200, and that the width of the content in "four" is 70 x 70. To compute the left position of "four" (relative to "three") we:

- add the width of the border to the width of the content to compute the total width of "four": 72px.
- subtract the total width of "four" from the content width of "three": 200 - 72 = 128px.
- subtract the distance of "four" from the right edge of "three": 128 - 5 = 123px (which is the number you see for the left position of "four").

You can perform a similar computation for the top position property of "four."

The offset values are the number of pixels from the top and left of the document (that is, the body), which depend on where the element sits in the flow of the page.

Using Element Metrics to Create a Bar Chart

Now that we know how to get an element's metrics using jQuery, let's build the bar chart data visualization from the beginning of the lesson. We'll do it in steps, beginning with the HTML and CSS. Create a new HTML file as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Data Visualization - Bar Chart</title>
  <meta charset="utf-8">
  <style>
html, body {
    width: 100%;
    height: 100%;
}
h1 {
    margin: auto;
    margin-bottom: 20px;
    width: 800px;
    font-size: 150%;
    font-family: Helvetica, Arial, sans-serif;
    color: gray;
}
div.container {
    margin: auto;
    position: relative;
    width: 800px;
    height: 600px;
    border: 1px solid gray;
}
div.year {
    font-size: 300%;
    font-family: Helvetica, Arial, sans-serif;
    color: gray;
}
div.dataContainer {
    border-bottom: 1px solid black;
    border-right: 1px solid black;
}
span.X {
    font-size: 50%;
    font-family: Helvetica, Arial, sans-serif;
}
span.Y {
    font-size: 50%;
    font-family: Helvetica, Arial, sans-serif;
}
div.F {
    background-color: rgba(238, 173, 218, .5);
}
div.M {
    background-color: rgba(121, 173, 210, .5);
}
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="graph.js"></script>
</head>
<body>
</body>
</html>
```

 Save this file in your **/jQuery** folder as **graph.html**. You can 

preview, but you won't see anything yet because we need jQuery to make the magic happen! The HTML currently consists of just an empty <body>; we'll add all the other HTML we need using jQuery. Get started on the jQuery by creating a new file, and adding the **ready()** function and the data we want to visualize (population distribution for males and females in the year 2000 in the United States):

CODE TO TYPE:

```
$(document).ready(function() {

    var data = [
        [0,1,9735380],
        [0,2,9310714],
        [5,1,10552146],
        [5,2,10069564],
        [10,1,10563233],
        [10,2,10022524],
        [15,1,10237419],
        [15,2,9692669],
        [20,1,9731315],
        [20,2,9324244],
        [25,1,9659493],
        [25,2,9518507],
        [30,1,10205879],
        [30,2,10119296],
        [35,1,11475182],
        [35,2,11635647],
        [40,1,11320252],
        [40,2,11488578],
        [45,1,9925006],
        [45,2,10261253],
        [50,1,8507934],
        [50,2,8911133],
        [55,1,6459082],
        [55,2,6921268],
        [60,1,5123399],
        [60,2,5668961],
        [65,1,4453623],
        [65,2,4804784],
        [70,1,3792145],
        [70,2,5184855],
        [75,1,2912655],
        [75,2,4355644],
        [80,1,1902638],
        [80,2,3221898],
        [85,1,970357],
        [85,2,1981156],
        [90,1,336303],
        [90,2,1064581]
    ];

    $("body").append("<h1>Population Distribution by Age</h1>");
    $("body").append("<div class=\"container\"></div>");
    $("<div>2000</div>").appendTo("div.container").addClass("year").css({
        position: "relative",
        top: 10,
        left: 10
    });
    $("<div></div>").appendTo("div.container").addClass("dataContainer").css({
        position: "absolute",
        top: 40,
        left: 40,
        width: $("div.container").width() - 80,
        height: $("div.container").height() - 80
    });

});
```

 Save this file in your **/jQuery** folder as **graph.js** and 

preview your **graph.html** file. You see the title for the page, the year (2000), and the x and y axes for the graph.

Take a moment to look through the data in the **data** array. There are three elements in each array, representing the beginning age for the group, the gender (male=1 and female=2), and the number of people. In

the first two entries, we have 9,735,380 males aged 0-4, and 9,310,714 females aged 0-4:

OBSERVE:

```
[0,1,9735380],
[0,2,9310714],
```

In the bar chart, we'll display the data for males and females for each age group with one bar (not two): pink for females, blue for males, and purple for both. Look at the graph at the beginning to see how this works: you'll see a little blue at the top if there were more males in an age group, pink if there were more females, and the purple represents the same number of males and females.

In the code, we have four jQuery statements that add new content to the page. Most of this probably looks familiar, except for **appendTo()**. This method comes in really handy when you want to create new content, and then immediately execute some jQuery methods on the object created by the new content. Let's take, for example, the element we're creating and adding to hold the year, 2000:

OBSERVE:

```
$("<div>2000</div>").appendTo("div.container").addClass("year").css({
    position: "relative",
    top: 10,
    left: 10
});
```

If we wrote the code like we usually would, we'd write something like this:

OBSERVE:

```
$("div.container").append("<div>2000</div>");
```

This appends a new with the content 2000 to the with the class "container," but the object returned by this statement is the "container" . In order to do any more to the *new* (the one we just created and added), we'd have to select that new with a separate selector! We could certainly do that, but using **appendTo()** instead allows us to create, add, and then modify the new all in one chain. When you write:

OBSERVE:

```
$("<div>2000</div>").appendTo("div.container");
```

...you're using the jQuery function **$()** to *create* new content. **$()** can both select content and create new content! **$(" 2000 ")** returns the jQuery object representing this new , then we can call **appendTo()** to append that new object to an existing object by specifying the existing object as a **selector in the argument to appendTo()**. Here, we append the new to the existing "container" . Just like with other jQuery chained methods, the new is again returned from the **appendTo()** method, so we can keep calling methods on it.

OBSERVE:

```
$("<div>2000</div>").appendTo("div.container").addClass("year").css({
    position: "relative",
    top: 10,
    left: 10
});
```

We **give it a class, "year,"** and **then some CSS, positioning it 10 from the top and left of the "container" in which it's nested**.

Finally, take note of the structure we're building in the page. We have a heading at the top, followed by a "container" that contains the "year" and the "dataContainer" . We'll add the bar chart data to the "dataContainer" , and we'll add the labels for the axes to the "container" and position the labels along the bottom and right borders of the "dataContainer" . In the CSS for the "dataContainer" , we set the bottom and right borders to 1px solid black to create the x and y axes of the graph:

OBSERVE:

```
div.dataContainer {
    border-bottom: 1px solid black;
    border-right: 1px solid black;
}
```

Adding Labels to the X and Y Axes

Next we'll add labels to the x and y axes for our bar chart graph. The trick is to add all the data for the graph (the labels, as well as the data points for the bars in the chart) in such a way that it will work even if we change our data, so the program will work for population distribution data for *any* year (not just 2000):

CODE TO TYPE:

```
$(document).ready(function() {

    var data = [
        [0,1,9735380],
        [0,2,9310714],
        [5,1,10552146],
        [5,2,10069564],
        [10,1,10563233],
        [10,2,10022524],
        [15,1,10237419],
        [15,2,9692669],
        [20,1,9731315],
        [20,2,9324244],
        [25,1,9659493],
        [25,2,9518507],
        [30,1,10205879],
        [30,2,10119296],
        [35,1,11475182],
        [35,2,11635647],
        [40,1,11320252],
        [40,2,11488578],
        [45,1,9925006],
        [45,2,10261253],
        [50,1,8507934],
        [50,2,8911133],
        [55,1,6459082],
        [55,2,6921268],
        [60,1,5123399],
        [60,2,5668961],
        [65,1,4453623],
        [65,2,4804784],
        [70,1,3792145],
        [70,2,5184855],
        [75,1,2912655],
        [75,2,4355644],
        [80,1,1902638],
        [80,2,3221898],
        [85,1,970357],
        [85,2,1981156],
        [90,1,336303],
        [90,2,1064581]
    ];

    $("body").append("<h1>Population Distribution by Age</h1>");
    $("body").append("<div class=\"container\"></div>");
    $("<div>2000</div>").appendTo("div.container").addClass("year").css({
        position: "relative",
        top: 10,
        left: 10
    });
    $("<div></div>").appendTo("div.container").addClass("dataContainer").css({
        position: "absolute",
        top: 40,
        left: 40,
        width: $("div.container").width() - 80,
        height: $("div.container").height() - 80
    });
    addAxes();

    function addAxes() {
        var numPoints = data.length / 2;
        var ptWidth = $("div.dataContainer").width() / numPoints / 2;
        var dataContainerLeft = $("div.dataContainer").position().left;
        for (var i = 0; i < data.length; i += 2) {
            var dataPt = data[i];
            var left = dataContainerLeft + (ptWidth * i);
```

```
            $("<span>" + dataPt[0] + "</span>").appendTo("div.container").addCla
ss("X").css({
                position: "absolute",
                bottom: 20,
                left: left + $("span.X").width()
            });
        }
        var max = findMax();
        var magMax = findMag(max);
        var ptHeight = $("div.dataContainer").height() / (magMax+1);
        var dataContainerTop = $("div.dataContainer").position().top;
        var left = $("div.dataContainer").width() + 50;
        for (var i = 0; i <= magMax; i++) {
            var top = dataContainerTop + (ptHeight * i);
            $("<span>" + (magMax-i) + "M</span>").appendTo("div.container").addC
lass("Y").css({
                position: "absolute",
                top: top + $("span.Y").height(),
                left: left
            });
        }
    }

    function findMax() {
        var max = 0;
        for (var i = 0; i < data.length; i++) {
            var dataPt = data[i];
            if (dataPt[2] > max) {
                max = dataPt[2];
            }
        }
        return max;
    }

    function findMag(num) {
        while (num > 1) {
            num = num / 10;
        }
        num = num * 100;
        num = Math.floor(num);
        return num;
    }

});
```

 Save **graph.js** and 

preview your **graph.html** file. Now we have labels on the x and y axes. We labeled the x axis with the age groups, and the y axis with population values (from 0 to 11 million).

We use the age group data from each data point (the first number in the sub-arrays) to create the x axis labels, and the population data from each data point (the third number in the sub-arrays) to create the y axis labels, so if these numbers change, the graph will still work. Let's see how:

The X axis labels

OBSERVE:

```
addAxes();

function addAxes() {
    // create the x axis labels
    var numPoints = data.length / 2;
    var ptWidth = $("div.dataContainer").width() / numPoints / 2;
    var dataContainerLeft = $("div.dataContainer").position().left;
    for (var i = 0; i < data.length; i += 2) {
        var dataPt = data[i];
        var left = dataContainerLeft + (ptWidth * i);
        $("<span>" + dataPt[0] + "</span>").appendTo("div.container").addClass("
X").css({
            position: "absolute",
            bottom: 20,
            left: left + $("span.X").width()
        });
    }
    // create the y axis labels
    var max = findMax();
    var magMax = findMag(max);
    var ptHeight = $("div.dataContainer").height() / (magMax+1);
    var dataContainerTop = $("div.dataContainer").position().top;
    var left = $("div.dataContainer").width() + 50;
    for (var i = 0; i <= magMax; i++) {
        var top = dataContainerTop + (ptHeight * i);
        $("<span>" + (magMax-i) + "M</span>").appendTo("div.container").addClass
("Y").css({
            position: "absolute",
            top: top + $("span.Y").height(),
            left: left
        });
    }
}
```

In **addAxes()**, the function we use to create the labels for the x and y axes, we first compute the **width of the labels on the x axis** by getting the number of data points and dividing by two (because we're going to display the male and female data for each age group as one bar, rather than two), and then dividing the width of the "dataContainer" by the number of data points, and then by 2 again (to provide space between the bars).

Next we get the **left position of the "dataContainer" and store it in a variable**. We'll position the labels for the x axis based on the left position of this . We use this variable in a **for** loop, but rather than computing the same value each time through the loop, it's more efficient to just compute it once before we enter the loop.

Once we know where the labels should be placed on the left, we **loop over all the data points.** Notice that we increase **i** by two each time, to create just one label per age group (for both males and females).

Inside the **for** loop that is looping over the **data[]** array, we first get the data point (which is also an array, with three values: the age group, the M/F flag, and the population for that sex and age group). Then we compute the **left position for the label** for this age group by multiplying the **ptWidth**, which is the width of an individual label, by **i**, and then adding to the left position of the "dataContainer" , so the leftmost label aligns perfectly beneath the bottom border of the "dataContainer" . So, if we're on the first data point, i = 0, and the left position is the same as the "dataContainer" ; if we're on the next data point, i = 2, and the left position is 2 times the width of an individual label (so the label takes up half the space, and there is also space between the labels).

Then we create a **new element to hold the label**. The content of the is the age group value (in **dataPt[0]**). We append the new to the "container" , add the "X" class to the and position it 20 pixels from the bottom of the "container" (halfway between the bottom of the "container" and the bottom of the "dataContainer"), and **set the left position to the left value** that we computed (relative to the "dataContainer"). We **add the width of an "X" element to the left position here**; this is to push the label just a bit further to the right, so it will line up better with the middle of the bar representing the data that we'll be adding in the next section.

The Y axis labels

The code for adding the labels to the Y axis is similar to what we just did for the X axis, except of course, the Y axis labels are based on the population from each the data points, rather than the age groups:

OBSERVE:

```
addAxes();

function addAxes() {
    var numPoints = data.length / 2;
    var ptWidth = $("div.dataContainer").width() / numPoints / 2;
    var dataContainerLeft = $("div.dataContainer").position().left;
    for (var i = 0; i < data.length; i += 2) {
        var dataPt = data[i];
        var left = dataContainerLeft + (ptWidth * i);
        $("<span>" + dataPt[0] + "</span>").appendTo("div.container").addClass("
X").css({
            position: "absolute",
            bottom: 20,
            left: left + $("span.X").width()
        });
    }
    var max = findMax();
    var magMax = findMag(max);
    var ptHeight = $("div.dataContainer").height() / (magMax+1);
    var dataContainerTop = $("div.dataContainer").position().top;
    var left = $("div.dataContainer").width() + 50;
    for (var i = 0; i <= magMax; i++) {
        var top = dataContainerTop + (ptHeight * i);
        $("<span>" + (magMax-i) + "M</span>").appendTo("div.container").addClass
("Y").css({
            position: "absolute",
            top: top + $("span.Y").height(),
            left: left
        });
    }
}
```

We don't want to display *all* the population numbers from the data; that would be way too many for the graph. We just want to display the main chunks of population from 0 to whatever the maximum population for an age group is. To do that, we first **find the maximum population for all the age groups (just over 11 million), and then get the magnitude of this number (11) using the findMax() and findMag() helper functions**.

Then we **compute how tall each label should be** by taking the total height of the "dataContainer" and dividing by the magnitude of the maximum population point (11) plus one (to account for 0).

We also compute the **top and left** position points to create our labels. We use the top position of the "dataContainer" to compute the top position of the topmost label. To compute the left position is just a bit trickier. Remember that the "dataContainer" is positioned 40 pixels from the left of the "container" (we did this when we created the "dataContainer"):

OBSERVE:

```
$("<div></div>").appendTo("div.container").addClass("dataContainer").css({
    position: "absolute",
    top: 40,
    left: 40,
    width: $("div.container").width() - 80,
    height: $("div.container").height() - 80
});
```

So the left position of the labels for the y axis needs to be 40 + the width of the "dataContainer" + 10 (for some space) from the left of the "container" in which they are nested, which is how we get:

OBSERVE:
`$("div.dataContainer").width() + 50;`

We want to create labels from 0 to the magnitude of our maximum population, which in this case is 11, so we loop from 0 to **magMax**, and each time through the loop, compute the **top position for the label** and create a new element to hold the label. We add the labels top down, but **i** starts at 0, so we use **(magMax - i)** to compute the value of the current label we're adding (so 0 is at the bottom of the graph and 11 is at the top). We add an "M" after each number to indicate that it's in millions.

Now, the left position of each label is the same (based on the width of the "dataContainer" as we explained above), but the top changes for each label. Just like before, we add the height of one label so the label appears in the middle of the data point rather than at the top (it looks better that way, and since the label numbers are rounded down, gives a slightly more accurate representation of the numbers in the bars).

Note

We define the **addAxes()** function (and also the **findMax()** and **findMag()** helper functions) *inside* the ready function. This is a common way of structuring jQuery programs because it hides all the functions used by the program, limiting visibility to only the code in the ready function that requires those functions. If we needed to use the **addAxes()** function outside the ready function for some reason, we would need to define **addAxes()** globally, outside the ready function.

Adding the Data

Whew! We're almost there. We just need to add the data. Adding the data isn't much different from adding the axis labels. We need to create a new element for each data point, with a height that represents the popluation for that data point. Remember that we're adding both the male and female bars at the *same* location for each age group (and if you look at the CSS, you'll see that the colors for bars, specified in the "M" and "F" classes, have an opacity of .5, so the colors blend together where they overlap, creating the purple color where the male and female values are the same):

CODE TO TYPE:

```
$(document).ready(function() {

    var data = [
        [0,1,9735380],
        [0,2,9310714],
        [5,1,10552146],
        [5,2,10069564],
        [10,1,10563233],
        [10,2,10022524],
        [15,1,10237419],
        [15,2,9692669],
        [20,1,9731315],
        [20,2,9324244],
        [25,1,9659493],
        [25,2,9518507],
        [30,1,10205879],
        [30,2,10119296],
        [35,1,11475182],
        [35,2,11635647],
        [40,1,11320252],
        [40,2,11488578],
        [45,1,9925006],
        [45,2,10261253],
        [50,1,8507934],
        [50,2,8911133],
        [55,1,6459082],
        [55,2,6921268],
        [60,1,5123399],
        [60,2,5668961],
        [65,1,4453623],
        [65,2,4804784],
        [70,1,3792145],
        [70,2,5184855],
        [75,1,2912655],
        [75,2,4355644],
        [80,1,1902638],
        [80,2,3221898],
        [85,1,970357],
        [85,2,1981156],
        [90,1,336303],
        [90,2,1064581]
    ];

    $("body").append("<h1>Population Distribution by Age</h1>");
    $("body").append("<div class=\"container\"></div>");
    $("<div>2000</div>").appendTo("div.container").addClass("year").css({
        position: "relative",
        top: 10,
        left: 10
    });
    $("<div></div>").appendTo("div.container").addClass("dataContainer").css({
        position: "absolute",
        top: 40,
        left: 40,
        width: $("div.container").width() - 80,
        height: $("div.container").height() - 80
    });
    addAxes();
    addData();

    function addAxes() {
        var numPoints = data.length / 2;
        var ptWidth = $("div.dataContainer").width() / numPoints / 2;
        var dataContainerLeft = $("div.dataContainer").position().left;
        for (var i = 0; i < data.length; i += 2) {
            var dataPt = data[i];
```

```
            var left = dataContainerLeft + (ptWidth * i);
            $("<span>" + dataPt[0] + "</span>").appendTo("div.container").addCla
ss("X").css({
                position: "absolute",
                bottom: 20,
                left: left + $("span.X").width()
            });
        }
        var max = findMax();
        var magMax = findMag(max);
        var ptHeight = $("div.dataContainer").height() / (magMax+1);
        var dataContainerTop = $("div.dataContainer").position().top;
        var left = $("div.dataContainer").width() + 50;
        for (var i = 0; i <= magMax; i++) {
            var top = dataContainerTop + (ptHeight * i);
            $("<span>" + (magMax-i) + "M</span>").appendTo("div.container").addC
lass("Y").css({
                position: "absolute",
                top: top + $("span.Y").height(),
                left: left
            });
        }
    }

    function addData() {
        var numPoints = data.length / 2;
        var ptWidth = $("div.dataContainer").width() / numPoints / 2;
        var max = findMax();
        var dataContainerLeft = $("div.dataContainer").position().left - 40;
        for (var i = 0; i < data.length; i++) {
            var dataPt = data[i];
            var sexClass;
            var pos;
            if (dataPt[1] == 1) {
                sexClass = "M";
                pos = i;
            } else {
                sexClass = "F";
                pos = i-1;
            }
            var left = dataContainerLeft + (ptWidth * pos);
            var barHeight = (dataPt[2] * $("div.dataContainer").height()) / max;
            $("<div></div>").appendTo("div.dataContainer").addClass("dataPoint")
.addClass(sexClass).css({
                position: "absolute",
                width: ptWidth,
                height: barHeight,
                bottom: 0,
                left: left
            });
        }
    }

    function findMax() {
        var max = 0;
        for (var i = 0; i < data.length; i++) {
            var dataPt = data[i];
            if (dataPt[2] > max) {
                max = dataPt[2];
            }
        }
        return max;
    }

    function findMag(num) {
        while (num > 1) {
            num = num / 10;
        }
```

```
        num = num * 100;
        num = Math.floor(num);
        return num;
    }

});
```

 Save the file and  preview your **graph.html** file. Now you see the full graph! (Make sure you don't miss the call to **addData()** we added, just below the call to **addAxes()**):

OBSERVE:

```
function addData() {
    var numPoints = data.length / 2;
    var ptWidth = $("div.dataContainer").width() / numPoints / 2;
    var max = findMax();
    for (var i = 0; i < data.length; i++) {
        var dataPt = data[i];
        var sexClass;
        var pos;
        if (dataPt[1] == 1) {
            sexClass = "M";
            pos = i;
        } else {
            sexClass = "F";
            pos = i-1;
        }
        var left = ptWidth * pos;
        var barHeight = (dataPt[2] * $("div.dataContainer").height()) / max;
        $("<div></div>").appendTo("div.dataContainer").addClass("dataPoint").add
Class(sexClass).css({
            position: "absolute",
            width: ptWidth,
            height: barHeight,
            bottom: 0,
            left: left
        });
    }
}
```

We used the same technique to find the left position for each data bar as we did before for the x axis labels. We get the **number of data points divided by 2** (because the male and female bars are in the same spot), and use that to compute the **width for each bar** by dividing the total width of the "dataContainer" by the number of points, and then again by 2 (so each bar takes up half the space, providing space between the bars).

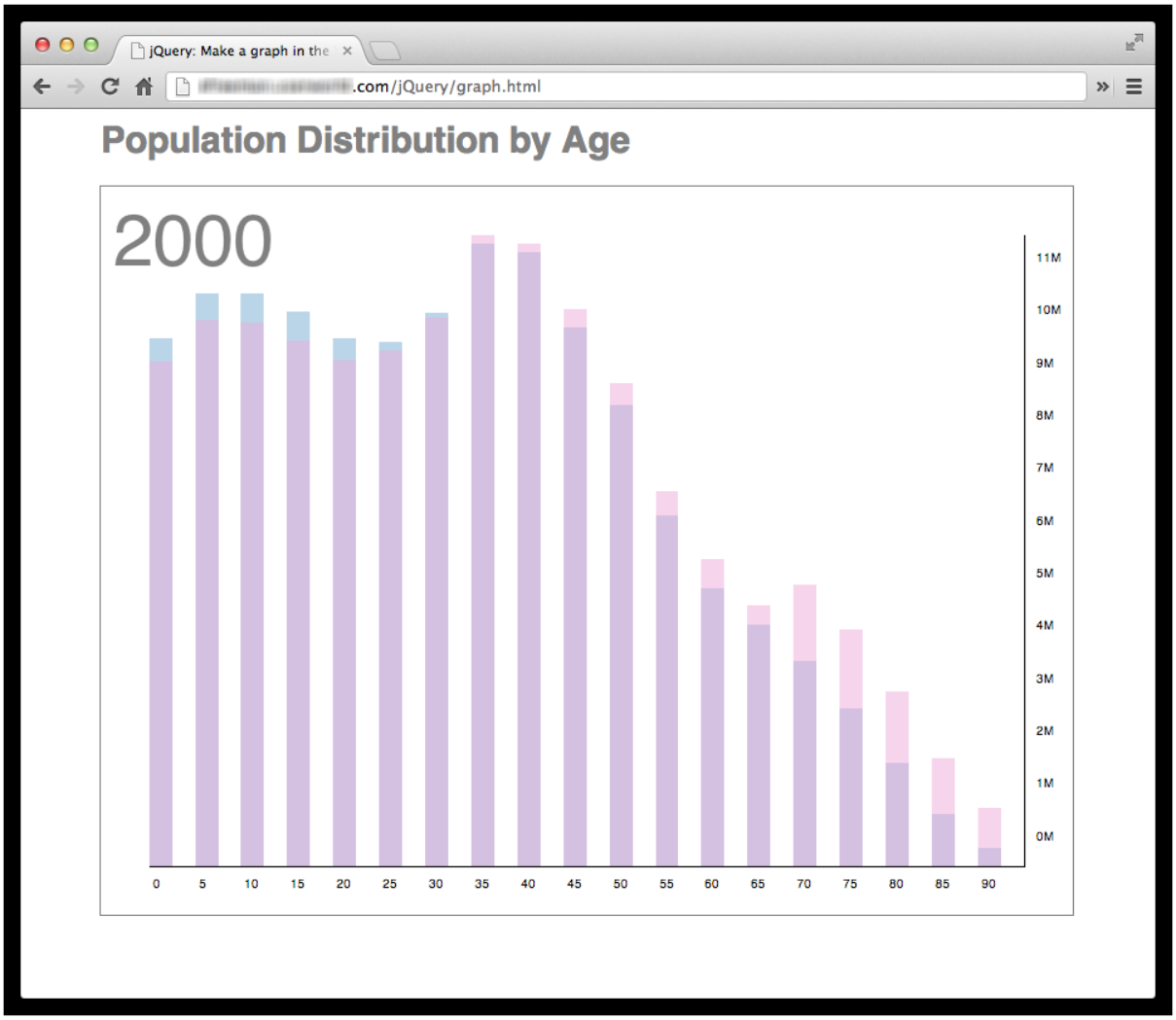
Then we find the **maximum population value**, so we can use that to compute the relative size of each bar compared to the maximum (again, we want to reduce the amount of computation we do each time through the loop, so we compute this once outside the loop, since it won't change).

Then we loop through all the data points, and **keep track of whether we are processing the male or female data point for a given age group**. The variable **pos** will always be an even number (0, 0, 2, 2, and so on) as we loop through the data points; we use that to create the **left position of both the male and female bars**. Because the bars are being added to the "dataContainer" , we position them relative to the left side of that , at position 0. The first and second times through the loop, **pos** is 0, so the first two bars will display up against the left side of the "dataContainer" , which is where we want them.

To compute the **correct height of the bar**, we multiply the population value at this data point (e.g. 9735380) by the height of the "dataContainer" and then divide by the maximum data population number (11635647). This gives us a bar height that is at most the height of the data container, scaled appropriately to represent the population value within the .

Finally, we create a new to represent the var, add it to the "dataContainer" , add the class "dataPoint," add the class "M" or "F" depending on if the bar is representing the male or female data point for this age group, and set the CSS properties, including the computed **width**, **height**, and **left position**.

The final bar chart looks like the graph you saw at the beginning of the lesson:



It appears that women still had an advantage over men in their later years, at least we did back in the year 2000! Sorry guys. It will be interesting to add more recent data to find out if that trend is changing.

Well done! This was a big lesson, with a big project. You can use jQuery to create some fairly sophisticated web applications. You learned how to use jQuery to get and set the width, height, and positions of elements in the page, which comes in handy when you're creating pages like the bar chart, where you need elements with precise widths and heights, in precise positions. You also learned about **appendTo()**, another method of adding content to a page. This is useful when you want to modify the object representing the content using jQuery methods right away.

Use the projects to practice your element metrics skills before moving on to the next lesson.

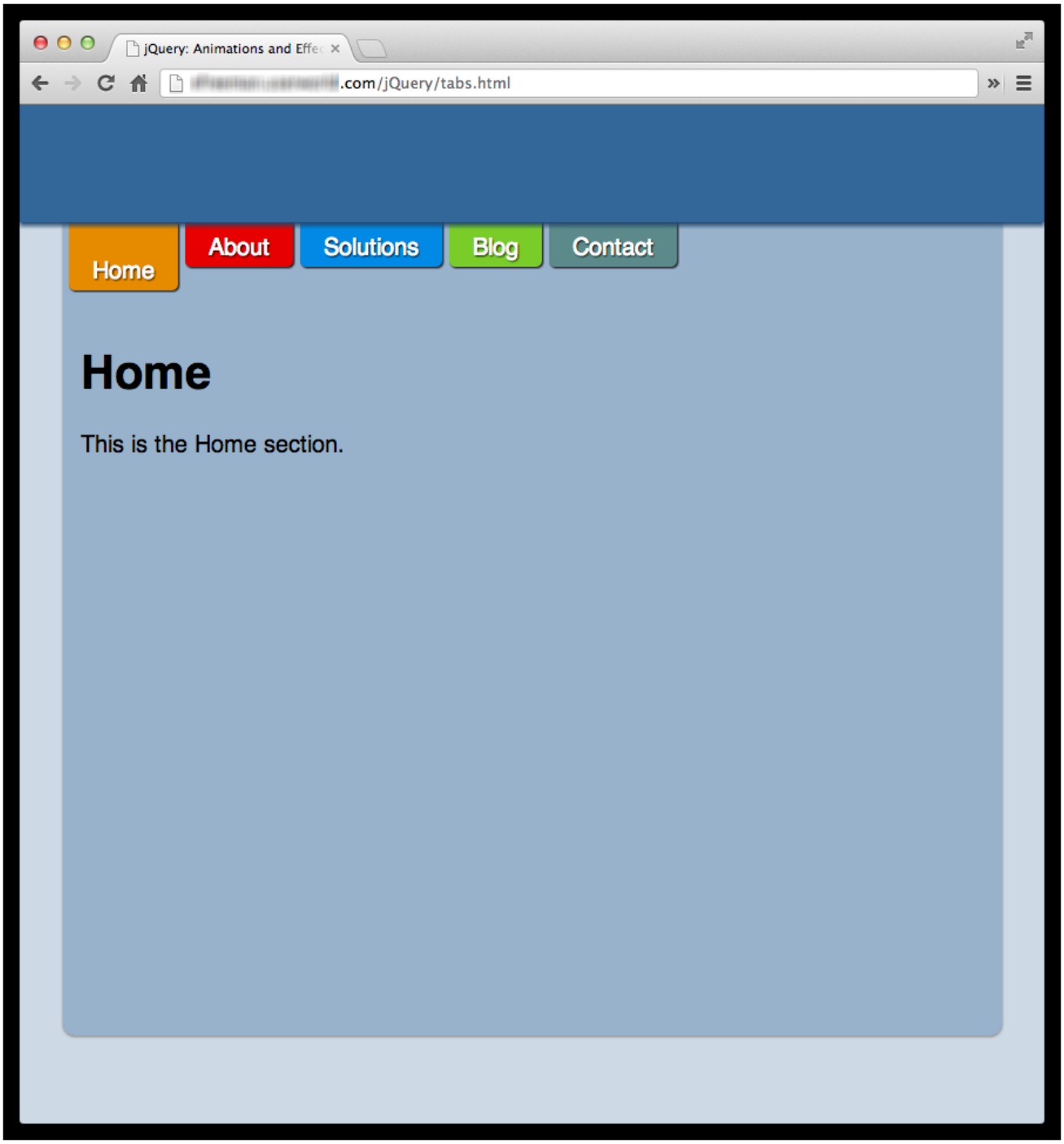
Effects

Lesson Objectives

- You will use the jQuery methods **show()** and **hide()**, **slideUp()** and **slideDown()**, **fadeIn()** and **fadeOut()**, and **animate()**.
- You will use **setTimeout()** with an effect.

Effects

One important feature that jQuery is known for is its capability for basic effects and animations of elements. You've probably used web pages with sections that expand when you click on a title or menu option, or an image slide show that slides or fades images in and out as you view the page. These effects are not difficult to achieve with the jQuery effects functions. In this lesson, we'll explore some of these methods as we build a tabbed menu that looks like this:



When you click on a menu tab, it slides down, while the previously selected tab slides up. In addition, the section corresponding to the newly selected tab slides down.

Using show() and hide() to Create a Tabbed Menu

We used the show() and hide() methods to create basic effects in web pages in earlier lessons. Let's begin with an implementation of the menu tabs that uses show() and hide() to show and hide content as you click on each tab. Create a new file and add this HTML:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Animations and Effects</title>
  <meta charset="utf-8">
  <style>
html, body {
    width: 100%;
    height: 100%;
    font-family: Helvetica, Arial, sans-serif;
}
body {
    margin: 0px;
    padding: 0px;
    background-color: #CDDAE6;
}
header {
    position: relative;
    top: 0px;
    height: 100px;
    margin: 0px;
    background-color: #336699;
    box-shadow: 0px 3px 5px #24476B;
    z-index: 3;
}
ul#tabs {
    position: relative;
    top: 10px;
    list-style-type: none;
    width: 800px;
    margin: 0px auto;
    padding: 0px 0px 0px 10px;
    z-index: 2;
}
ul#tabs li {
    position: relative;
    display: inline;
    padding: 30px 20px 5px 20px;
    border-radius: 0px 0px 5px 5px;
    box-shadow: 1px 1px 2px black;
    background-color: #E68A00;
}
ul#tabs li a {
    color: white;
    text-shadow: 1px 1px 2px black;
    text-decoration: none;
}
div#content {
    position: relative;
    top: -25px;
    margin: auto;
    width: 800px;
}
div.tab-content {
    position: absolute;
    top: 0px;
    width: 770px;
    height: 600px;
    overflow: hidden;
    margin: 0px auto;
    padding: 80px 15px 15px 15px;
    background-color: #99B2CC;
    border-radius: 0px 0px 10px 10px;
    box-shadow: 0px 1px 3px grey;
    z-index: 0;
}
```

```
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="tabs.js"></script>
</head>
<body>
  <header></header>
  <ul id="tabs">
    <li class="selected"><a href="home">Home</a></li>
    <li><a href="about">About</a></li>
    <li><a href="solutions">Solutions</a></li>
    <li><a href="blog">Blog</a></li>
    <li><a href="contact">Contact</a></li>
  </ul>
  <div id="content">
    <div id="home" class="tab-content">
        <h1>Home</h1>
        <p>This is the Home section.</p>
    </div>
    <div id="about" class="tab-content">
        <h1>About</h1>
        <p>This is the About section.</p>
    </div>
    <div id="solutions" class="tab-content">
        <h1>Solutions</h1>
        <p>This is the Solutions section.</p>
    </div>
    <div id="blog" class="tab-content">
        <h1>Blog</h1>
        <p>This is the Blog section.</p>
    </div>
    <div id="contact" class="tab-content">
        <h1>Contact</h1>
        <p>This is the Contact section.</p>
    </div>
  </div>
</body>
</html>
```

 Save this file in your **/jQuery** folder as **tabs.html**. 

Preview the file. Notice that all the tabs are orange—we'll change that soon with jQuery. Next, click on the link in one of the tabs. Because each tab's link is an <a> element, the browser tries to load the page in the **href** attribute of the link, but those pages don't exist. Instead, the content we're going to load is defined in the various elements that are already in the page. We need to fix those links so they show or hide the s instead of attempting to load a page. All the elements are positioned absolutely with CSS, so they are in the same position (sitting on top of one another). So, if we hide all the s except the selected one, we'll see the one we're showing in the correct position. Finally, notice that the **href** attribute of each tab is the *same* as the **id** for the that corresponds with that tab. This will become extremely useful in our code!

Let's write some jQuery to create this application:

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the content divs except the home content div
    $("div#content div").hide();
    $("div#content div#home").show();

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);
        // hide the current content
        $selectedDiv.hide();
        // show the new content
        $div.show();
        // update the currently selected tab
        $("ul#tabs li.selected").removeClass("selected");
        $(this).parent().addClass("selected");
    });
});
```

 Save this file in your **/jQuery** folder as **tabs.js**. 

Preview your **tabs.html** file, and try clicking on each of the tabs. The appropriate appears as you click on each tab.

Let's take a closer look at how this code works. Most of it is probably familiar to you already; we're not doing anything new so far, with one exception.

First, we set each of the tabs to a different color (except the home tab; we leave that one orange):

OBSERVE:

```
$("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
$("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE6");
$("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
$("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A");
```

We could have done this using CSS classes, but it's fun doing it with jQuery instead, and it gives you a chance to practice using **attribute selectors**. We use the content of the **href** attribute of each <a> element to select each tab.

Next, we **hide all of the content divs, and then show just the home content div**:

OBSERVE:

```
// hide all the content divs except the home content div
$("div#content div").hide();
$("div#content div#home").show();

// click handler for all tabs
$("ul#tabs li a").click(function(e) {
...
});
```

We also set up a **click handler for all the tabs** (on the <a> element inside the tabs, containing the tab name).

Notice that we use the **event object** in the click handler:

OBSERVE:

```
$("ul#tabs li a").click(function(e) {
    // prevent the a link from causing a page load
    e.preventDefault();
    ....
});
```

We do that so we can call a method, **preventDefault()**, on that event. Then when you click on a link, the default behavior is to load the page in the **href** attribute into the browser window. In our case, the **href** attribute doesn't have the name of a page; it's a reference to a containing the content we want to display in the page when you click on the link in the tab.

We want to prevent this default behavior, so we call **e.preventDefault()** on the event object associated with the "click" event. This means that no request to load a page is sent to the browser, which is what we want!

Instead, we implement the behavior we want to happen on the click event ourselves, in the rest of the click handler:

OBSERVE:

```
$("ul#tabs li a").click(function(e) {
    // prevent the a link from causing a page load
    e.preventDefault();
    // get the currently selected tab, and associated div
    var selectedTabName = $("ul#tabs li.selected a").attr("href");
    var $selectedDiv = $("div#" + selectedTabName);
    // get the newly selected tab, and associated div
    var tabName = $(this).attr("href");
    var $div = $("div#" + tabName);
    // hide the current content
    $selectedDiv.hide();
    // show the new content
    $div.show();
    // update the currently selected tab
    $("ul#tabs li.selected").removeClass("selected");
    $(this).parent().addClass("selected");
});
```

We get the value of the **href** attribute of the tab that's **currently selected** (that is, the one we're already on before we click a different tab). We can get this tab using the "selected" class (this class is always assigned to the currently selected tab). When the page first loads, that's the "home" tab. So, the value in **selectedTabName** will be "home." Then we can use that to **select the element associated with the tab**, in this case, the with the id "home."

Then we get the tab and its associated for the tab we've just clicked. If you click "About," **$(this)** will be the link in the "About" tab, with the **href** "about." Again, we can use that to select the ** with the id "about."**

Then we **hide the we're currently on ($selectedDiv)**, **show the for the tab we just clicked on ($div)**. Finally, we **update the two tabs to remove the "selected" class from the**

previously selected tab (Home) and add it to the tab on which we clicked (About). Now the "About" tab has the "selected" class, so the next time you click, we do the whole thing over again, except that "About" is the currently selected tab.

Simple Effects with show() and hide()

So far so good—we have the basic functionality we want from this interface, but we can spiff things up just a bit by making use of optional arguments to **show()** and **hide()**. Update your jQuery as shown:

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the content divs except the home content div
    $("div#content div").hide();
    $("div#content div#home").show();

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);
        // hide the current content
        $selectedDiv.hide("slow");
        // show the new content
        $div.show("slow");
        // update the currently selected tab
        $("ul#tabs li.selected").removeClass("selected");
        $(this).parent().addClass("selected");
    });
});
```

 Save **tabs.js** and 

preview your **tabs.html** file. Try clicking on each of the tabs. The appropriate appears as you click on each tab, but now the previous tab shrinks (to the top left), and the new tab grows (from the top left). We specify a **duration** for how fast an element should show or hide itself by passing "slow" as the argument to the **show()** and **hide()** functions. Both of these functions take either one of two strings, "slow" or "fast," or a number between 200 and 600 (the number of milliseconds for the animation). Try experimenting with different values and see which ones you like best.

We call **hide("slow")** first, to hide the currently selected , and then call **show("slow")** to show the newly selected , but the animation for the show begins before the animation for the hide is complete. That's because JavaScript doesn't wait for the animation to complete before executing the next line of code, so the animations happen together. Sometimes this might be exactly the effect you want; but sometimes you might want for one effect to finish before the next one begins.

To do that, you specify a second argument: the **complete** function. This function gets executed once the animation for **show()** or **hide()** is complete. So, in our case, we can begin the animation to **hide()** the currently visible , and wait to begin the animation to **show()** the new until the hiding animation has completed:

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the content divs except the home content div
    $("div#content div").hide();
    $("div#content div#home").show();

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        var $link = $(this);
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);
        // hide the current content
        $selectedDiv.hide("slow", function() {
            // show the new content
            $div.show("slow");
            // update the currently selected tab
            $("ul#tabs li.selected").removeClass("selected");
            $(this)$link.parent().addClass("selected");
        });
    });
});
```

 Save the file and 

preview your **tabs.html** file. Now, when you click on a new tab, the two animations for hiding one and showing the new happen separately. Experiment with different values for the duration to see which speed you prefer.

We moved the code to show the newly selected within the complete function that we pass to the **hide()** method. We had to save the value of **$(this)** (at the top of the click handler function), and then use this saved value, in the variable **$link**, to reference the link (and its parent, the <li> element representing the tab). Why do you think we had to do this?

Sliding Up and Down Effects

We can create a slightly different effect by using the methods **slideUp()** and **slideDown()** instead of **hide()** and **show()**:

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the content divs except the home content div
    $("div#content div").hide();
    $("div#content div#home").show();

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        var $link = $(this);
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);
        // hide the current content
        $selectedDiv.hideslideUp("slow", function() {
            // show the new content
            $div.showslideDown("slow");
            // update the currently selected tab
            $("ul#tabs li.selected").removeClass("selected");
            $link.parent().addClass("selected");
        });
    });
});
```

 Save the file and 

preview your **tabs.html** file. Once again, try clicking on the different tabs. Now, the currently visible is hidden with a sliding up effect, and once that animation is complete, the new is shown by sliding down. These effects work by manipulating the height of the element. Just like **show()** and **hide()**, both **slideUp()** and **slideDown()** take two optional arguments: the duration for the animation and a complete function.

Customizing Effects with animate()

Sometimes you'll want an effect that's from those you achieve using **show()**, **hide()**, **slideUp()**, and **slideDown()**. jQuery provides another versatile method, **animate()**, that allows you to create an effect using CSS properties.

You can use **animate()** to work with a variety of CSS properties. In our example, we'll use it with the **top** (position) property to manipulate the top position of the selected tab, so as you select a tab, it slides down a bit to indicate that it's the selected tab, like this:

We'll also modify the way the s are displayed just a bit so that instead of hiding the previous before showing the new one, we'll just slide the new one down over the previous one (so the effect will be like a series of layers sliding down as you click on each tab). We can accomplish this by manipulating the **z-index** property of the s to make sure the corresponding to the tab you just clicked on is always on top of all the other s. The **z-index** is a number representing how close an element is to you, if you imagine elements as layers in 3D space on the screen. The bigger the number, the closer an element is to you, so it will obscure elements with smaller **z-index**es (so they appear to be behind the elements with bigger **z-index** values).

Instead of having me tell you more about it, let's get to work! Go ahead and update your jQuery:

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the content divs except the home content div
    $("div#content div").hide();
    $("div#content div#home").show();
    $("ul#tabs li.selected").css("top", 20);

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        var $link = $(this);
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);
        // hide the current content
        $selectedDiv.slideUp("slow", function() {
            // show the new content
            $div.slideDown("slow");
            // update the currently selected tab
            $("ul#tabs li.selected").removeClass("selected");
            $link.parent().addClass("selected");
        });
        $div.css("z-index", 1);
        $div.slideDown("slow", function() {
            $selectedDiv.hide();
            $div.css("z-index", 0);
        });

        $("ul#tabs li.selected").animate({
            top: 0
        });
        $("ul#tabs li.selected").removeClass("selected");

        $(this).parent().animate({
            top: 20
        });
        $(this).parent().addClass("selected");
    });
});
```

 Save the jQuery and  preview your **tabs.html** file. When you first load the page, the "Home" tab is displayed and appears longer than the other tabs; when you click on a different tab, the "Home" tab slides up to match the other tabs, and the tab you just clicked on slides down to look "selected." When you select a tab, the corresponding to that tab slides down from the top (like before, except that the previous doesn't hide first, so it looks like the new is sliding down on top of the previous).

Let's go over the code. First, we added a line above the click handler function to set the **top** position of the selected tab to 20 pixels:

OBSERVE:

```
$("ul#tabs li.selected").css("top", 20);
```

Back at the CSS, you can see that the <li> elements representing the tabs are positioned "relative" to the parent element, which is also a positioned element, so we can manipulate the position of the tab relative to the <ul>. By default, the **top** position property of the <li> elements is 0px, so changing the selected tab to a top position of 20px makes it look longer. Each of the tabs is actually the same height; we just shift down the selected one. The tops of the other tabs are obscured by the header, which has a z-index one greater than the z-index of the menu <ul>, so the tops of all those tabs are tucked in behind the header unless they're "selected."

Next, inside the click handler for the tab links, we change the code to handle the animation of the s and add code to handle the animation of the tabs:

OBSERVE:

```
$div.css("z-index", 1);
$div.slideDown("slow", function() {
    $selectedDiv.hide();
    $div.css("z-index", 0);
});
```

We **set the z-index of the associated with the tab you just clicked on to 1**. This ensures that the newly selected will be on top of the other s. The is still hidden; we show it using the **slideDown()** method, just like before, except now in the complete function, we **hide the previous ** (so it's invisible, along with all the other non-selected s), and then **set the z-index of the newly selected back to 0**. (It doesn't need to be 1 once the other is hidden again.)

Once we've handled the s, we can make sure the tabs behave correctly too:

OBSERVE:

```
$("ul#tabs li.selected").animate({
    top: 0
});
$("ul#tabs li.selected").removeClass("selected");

$(this).parent().animate({
    top: 20
});
$(this).parent().addClass("selected");
```

Now, we use **animate()** to manipulate a CSS property for the tabs. We can't use **slideDown()** to slide a tab down when you select it because **slideDown()** and **slideUp()** show and hide an element. We always want the tabs to show; it's only the top position we want to change. So we need to use **animate()** instead, to create a custom animation.

First, we **slide the currently selected tab up by setting its top position back to 0 pixels**. (Remember, when you first load the page, the "Home" tab has its top position set to 20 pixels). Then we remove the "selected" class from this tab. Next, we **change the top position of the newly selected tab to 20 pixels**, and then add the "selected" class to this tab.

Notice that we put the code to animate the tabs *outside* of the complete function for the **slideDown()** method that handles hiding the . Why do you think we did that? Try moving the code to animate the tabs inside the complete function (remember to change **$(this)** to **$link**) and notice the difference in how the interface behaves.

When you call **animate()** to change the top position of the tabs, the previous tab slides up, and the new tab slides down. That means that the browser is *animating* the tab between its current position and its new position. This is the advantage of using these jQuery methods: it's fairly tricky to create this smooth animation behavior yourself using only JavaScript. If you just want to update the top position from one location to another in one step, that's not so tough with JavaScript, but getting the nice smooth animation between the two states is, and that's why these jQuery functions are so cool.

In addition, as with other effects functions we've discussed, you can supply a duration and a complete function to **animate()** as well. Check out the documentation for the **animate()** method, and the other effects methods online at the jQuery API site.

Repeating Effects with Timeouts

Sometimes you'll want to repeat an effect. One way to do that is with JavaScript's **setTimeout()** or **setInterval()** function. Create a new file and add this HTML:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Animations and Effects, with setTimeout</title>
  <meta charset="utf-8">
  <style>
div {
    position: relative;
    top: 30%;
    left: 30%;
    background-color: red;
    width: 400px;
    height: 400px;
}
p {
    position: relative;
    left: 30%;
}
p a:hover {
    background-color: rgba(255, 0, 0, .2);
    cursor: pointer;
}
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>
$(document).ready(function() {
    var timer;
    $("div").click(function() {
        fadeDiv();
    });
    $("a.stop").click(function() {
        clearTimeout(timer);
    });

    function fadeDiv() {
        $("div").fadeOut("slow", function() {
            $(this).fadeIn("fast");
            timer = setTimeout(function() { fadeDiv(); }, 1000);
        });
    }
});
  </script>
</head>
<body>
  <div></div>
  <p><a class="stop">stop</a></p>
</body>
</html>
```

 Save the file in your **/jQuery** folder as **fade.html** and 

preview. Click on the red square. It fades out and then back in. It will keep doing this until you click stop. You can restart the animation by clicking on the square again.

Let's go over the code to see how we use **setTimeout()** to repeat the fading effect, and also how the **fadeIn()** and **fadeOut()** effect methods work:

OBSERVE:

```
$(document).ready(function() {
    var timer;
    $("div").click(function() {
        fadeDiv();
    });
    $("a.stop").click(function() {
        clearTimeout(timer);
    });

    function fadeDiv() {
        $("div").fadeOut("slow", function() {
            $(this).fadeIn("fast");
            timer = setTimeout(function() { fadeDiv(); }, 1000);
        });
    }
});
```

First we declare a **timer** variable. Then, we set up a **click handler function** for the In the click handler, we call the **fadeDiv()** function. You'll see in a moment why we need to define a separate function to do the fading.

In **fadeDiv()**, we first call the **fadeOut()** method on the . Just like the other effect methods we've looked at, this animates the from a fully visible state to a hidden state; this method accomplishes that by fading the element. Just like the other effect methods, this method can take an optional duration, for which we're using "slow," and a complete function.

Once the has completed fading out, the complete function is called, and we immediately fade it back in using the **fadeIn()** method.

Then we create a **timer** using the **setTimeout()** function. This function has two arguments: the first is a function that is called once a specified time period has passed, and a second function is that time period (specified in milliseconds). Here, the function is called after 1000 milliseconds have passed. The function we pass to the timer calls the function **fadeDiv()** again, so each time the fades back in (so it's visible), the **fadeDiv()** function gets called again, which fades it back out, and so on. This would continue indefinitely, except that we've created a way to stop the timer.

We create a function, **fadeDiv()**, with the logic to fade the in and out, because we need a way to call that function from the **setTimeout()** function. If we'd placed all that logic within the click handler, we'd have no function to call from **setTimeout()**, and no convenient way to get those statements to be executed repeatedly.

You can stop a timer using the **clearTimeout()** function. We added a **click handler to the "stop" link** that's below the so when you click on that link, **clearTimeout(timer)** is called. Now you can see why we declared the **timer** variable at the top of the ready() function; we need it both within the **fadeDiv()** function (where its value is set) and within the **click handler for the link** (where we clear its value).

(If you're wondering why we don't need to call **e.preventDefault()** on this link in the link's click handler, like we did in the previous example, it's because the link has no **href** attribute, so the browser doesn't try to load a page.)

You could also use the **setInterval()** function to repeat an effect; this function works a little differently, but you can use it to accomplish the same thing.

In this lesson, you learned how to use jQuery methods like **show()** and **hide()**, **slideUp()** and **slideDown()**, **fadeIn()** and **fadeOut()**, and **animate()** to create some cool effects in your web pages. You also learned how to use **setTimeout()** with an effect to repeat that effect.

We have a fun project for you to do to practice your effects skills before you go on to the next lesson. In the upcoming lesson you'll learn about even more possibilities for effects, animations, and more with the jQuery UI library!

jQuery UI Library

Lesson Objectives

- You will use the jQuery UI library.

Using the jQuery UI Library

The jQuery UI library provides many useful features, including **widgets**, **interactions**, and **effects** to use with jQuery. The jQuery UI library will save you a lot of time when you need to create menus, sortable lists, dialog boxes, and more. There's a lot to this library; we're just going to cover a few of the features it offers, but once you have the basics down, you'll grasp the other features quickly.

Let's start by checking out the web page for jQuery UI (at http://jqueryui.com):

On the left you'll see links to each type of the feature offered in the library: Interactions, Widgets, and Effects. On the right, you'll see information about the library, and a link to download the current version. The version you see is different from the version of jQuery we're using. jQuery UI requires jQuery of course (because jQuery UI is built on top of jQuery), but the library developers often make changes in jQuery UI that are independent from jQuery, so the version numbers of the two libraries are different. Also, jQuery UI is updated fairly often, so by the time you take this course, the version number might be greater than the version we're using in this lesson: 1.9.2. Don't worry if it is; most likely the basic functionality we cover in this lesson hasn't changed (most of the time, new versions mean new features have been added, or major bugs have been fixed, rather than fundamental changes to the library).

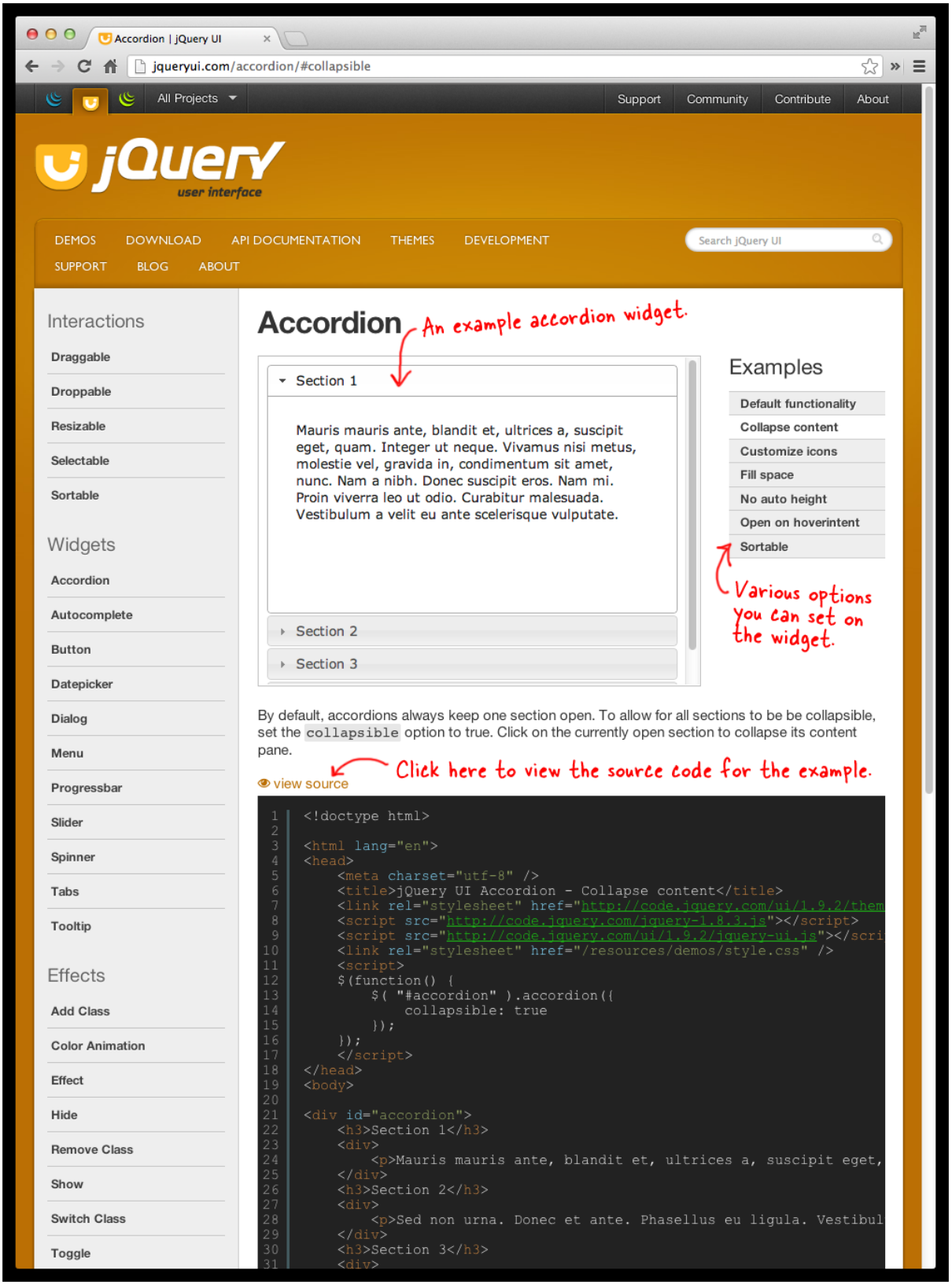
Look at the page again. Across the top you'll see links to Demos, Download, API Documentation, Themes, and more.

We'll visit some of these links shortly.

Don't click the link to download the jQuery UI library now, because we'll begin by linking directly to the library. Just like the main jQuery library, you can either download jQuery UI, or link to it. For now we'll link to it, but later you'll see how downloading it can be beneficial as well.

Using the Accordion Widget

Let's get started and use one of the widgets. On the left side of the page, look under Widgets and click **Accordion**. You see the information page for this widget. At the top center of the page is an example of an accordion widget, and on the right is a list of features you can customize for this widget. Play with the example accordion widget in the page to get a sense of how it works. Then click **Collapse content**, and compare how this accordion works to how the default accordion works. You can now collapse the open section by clicking on the section header, whereas before, you couldn't.



After you've played with the accordion widget for a bit, click **View Source** (under the example widget). Here, you can see an example of how to use the widget. If you've selected **Collapse Content** on the right, under "Examples," the source code you see in "View Source" shows you the code for that particular example of the accordion, in this case, the ability to collapse the open section.

Let's try using an accordion widget. Create a new file and add this HTML and jQuery code:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery UI: Widgets</title>
  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="http://code.jquery.com/ui/1.9.2/jquery-ui.js"></script>
  <script>
      $(document).ready(function() {
        $("#accordion").accordion({
            collapsible: true
        });
      });
  </script>
</head>
<body>
  <div id="accordion">
    <h1>Home</h1>
    <p>This is the Home section.</p>
    <h1>About</h1>
    <p>
        This is the About section. This web page uses the jQuery UI
        accordion widget. You can choose from many widgets,
        as well as effects and interactions, to make your web page
        more interesting and fun, with the jQuery UI library.
    </p>
    <h1>Solutions</h1>
    <p>This is the Solutions section.</p>
    <h1>Blog</h1>
    <p>This is the Blog section.</p>
    <h1>Contact</h1>
    <p>This is the Contact section.</p>
  </div>
</body>
</html>
```

 Save this file in your **/jQuery** folder as **accordion.html** and 

Preview it. You see all the headings and the paragraph for the "Home" section. Try clicking on another heading to see what happens.

We have the behavior of an accordion widget, with only a couple of lines of code! That's pretty cool. Let's take a closer look at both the HTML and the jQuery. First, look at the links at the top of the page:

OBSERVE:

```
<script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
<script src="http://code.jquery.com/ui/1.9.2/jquery-ui.js"></script>
```

We link to the main jQuery library, and then below that, to the jQuery UI library. It's important that the link to the main jQuery library comes first, above the link to the jQuery UI library. The browser parses JavaScript as it loads the JavaScript files, and the jQuery UI library depends on the jQuery library, so be sure to order these links correctly.

Next, let's look at how we've set up the HTML:

OBSERVE:

```
<div id="accordion">
  <h1>Home</h1>
  <p>This is the Home section.</p>
  <h1>About</h1>
  <p>
      This is the About section. This web page uses the jQuery UI
      accordion widget. You can choose from many widgets,
      as well as effects and interactions, to make your web page
      more interesting and fun, with the jQuery UI library.
  </p>
  <h1>Solutions</h1>
  <p>This is the Solutions section.</p>
  <h1>Blog</h1>
  <p>This is the Blog section.</p>
  <h1>Contact</h1>
  <p>This is the Contact section.</p>
</div>
```

In order for the accordion widget to work properly, we need to have an **enclosing element** that we can use as the container for the accordion. This is the ** with the id "accordion"**. Then, within that , we have a **series of headings and paragraphs**. The accordion requires this kind of structure in order to work properly: a heading followed by an element like <p> or that contains the content associated with the heading. The jQuery UI library code will look for this structure when it's adding code to support the accordion. So, when you look at the example code for the widgets, it's important that you look at both the HTML and how it's structured, as well as the jQuery.

Next, let's look at the jQuery:

OBSERVE:

```
$(document).ready(function() {
    $("#accordion").accordion({
        collapsible: true
    });
});
```

First, we **select the "accordion" **, and then call the **accordion()** method. This method sets up the widget based on the structure in the "accordion" . We customized the accordion just a bit by passing in an argument to the **accordion()** method: an object with the property **collapsible set to true**. This property enables the accordion feature that allows you to click on the heading of the section that's open, in order to close it.

We have the behavior of an accordion widget, but it doesn't look that great. You might think that we have to add a bunch of CSS to style the widget ourselves, but we don't. We can use jQuery UI Themes instead!

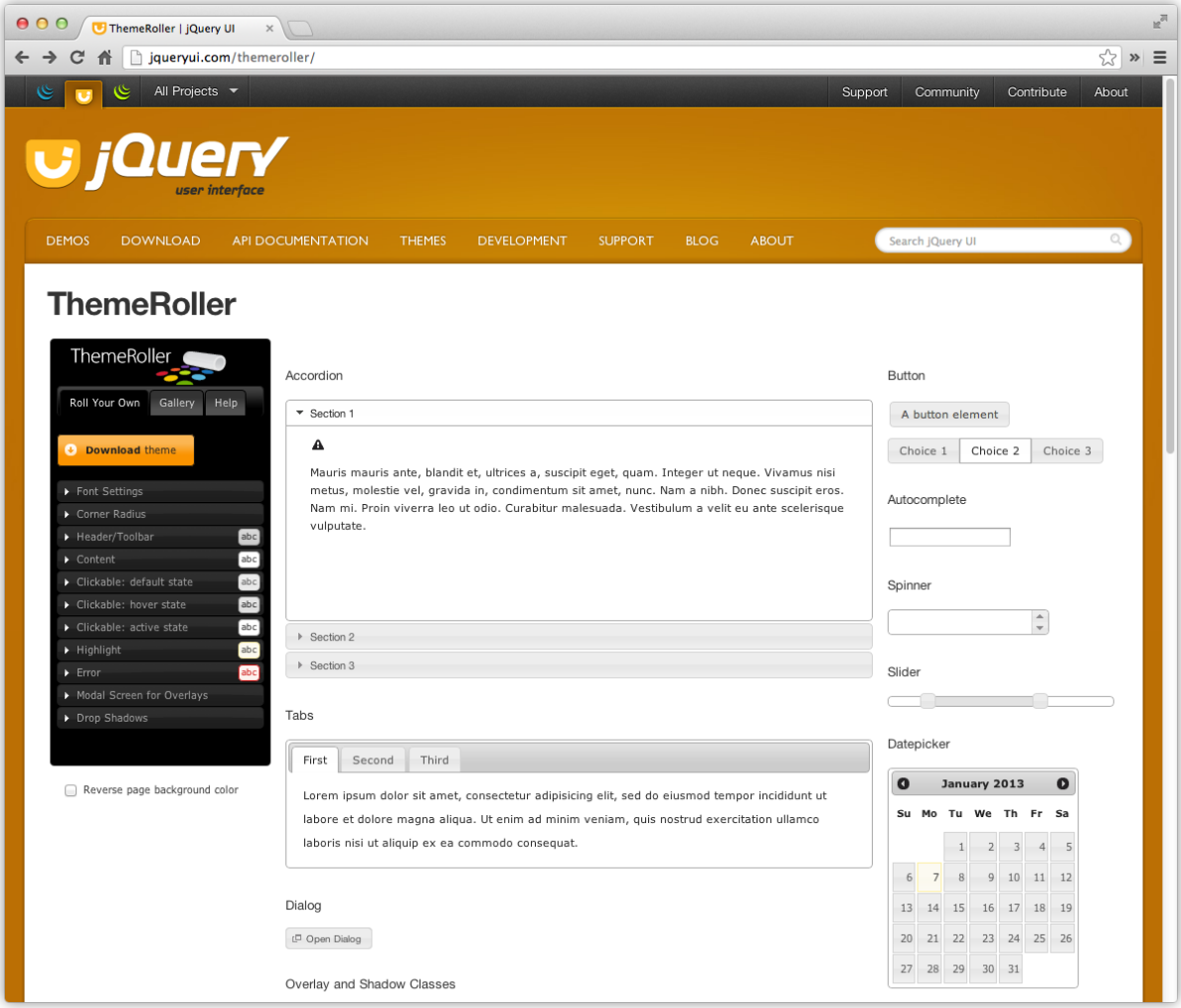
Using jQuery UI Themes

jQuery UI Themes are CSS styles that create a consistent look and feel for every jQuery UI widget you use on a web page. For example, you might use a date picker widget, a menu widget, and an accordion widget, all on one page. A theme gives all these widgets the same look and feel. Of course, you can use the CSS styles provided by the theme elsewhere in your page if you like.

The most common way to use a theme is to download it. We'll show the steps for downloading a theme and installing it, along with a customized version of the jQuery UI library. Then, we'll show you how to link directly to a theme.

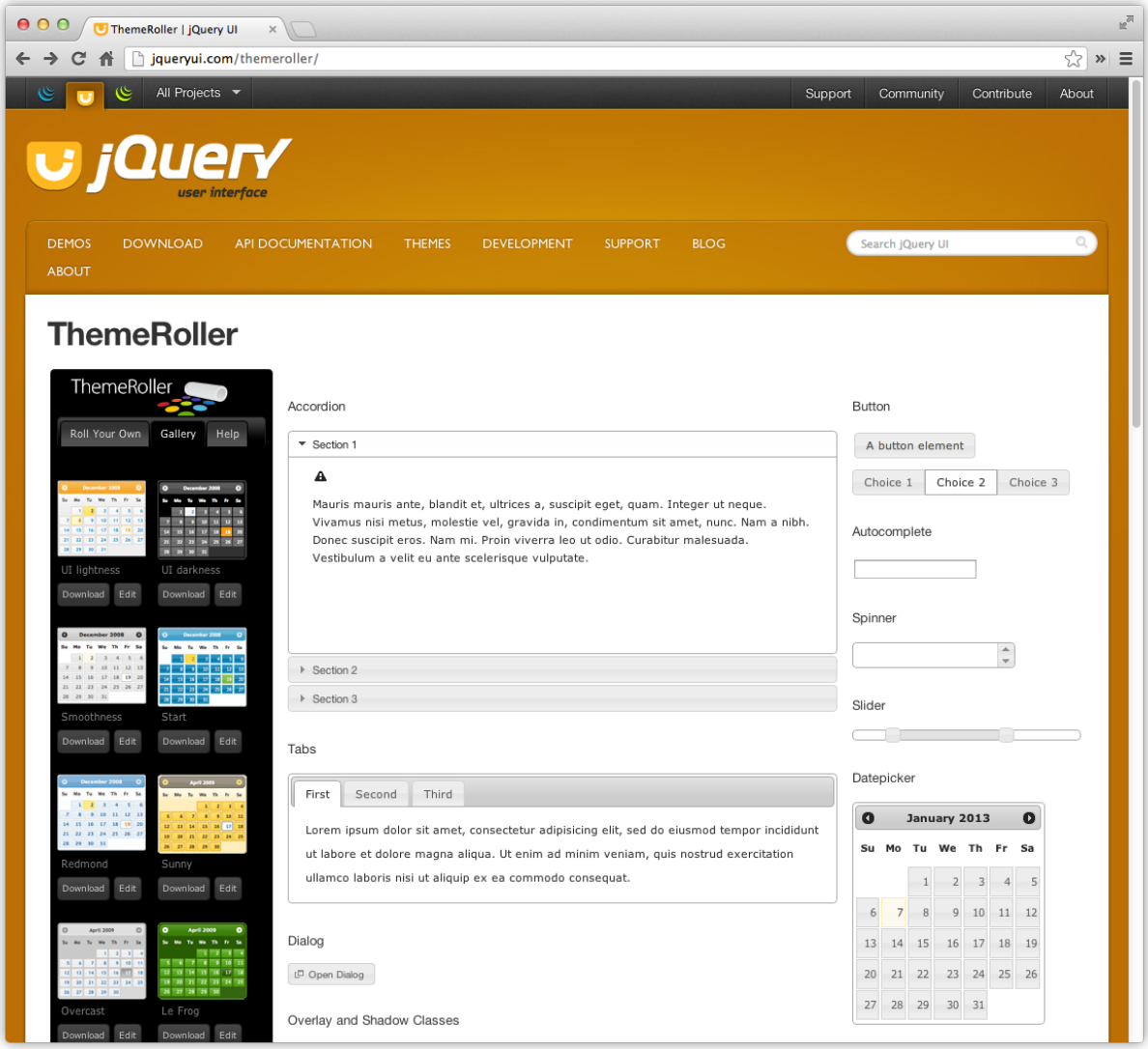
Downloading and Using a Theme

Click the **Themes** menu item on the top of the jQuery UI web page. You should see the Theme Roller page:



This page shows the default theme, and examples of how all the widgets will look using that theme.

You have many themes to choose from; click the **Gallery** tab in the Theme Roller sidebar on the left, and you'll see a wide variety of themes in different colors and styles:



If you don't like any of these themes, you can "Roll your own," which means to create your own theme by providing colors and styles. For now, we'lluse one of the built-in themes: "Cupertino." Scroll down until you see the Cupertino theme (again, in the left sidebar), and click the theme icon (a calendar). You'll see how all the widgets will look using this theme. Click the **Download** button:

This page allows you to build a customized version of the jQuery UI library that contains just the components you need for your website. Right now, we're linking to the *full library*, which contains everything, so the file size is much larger than it needs to be if we're using only the accordion widget.

By selecting only the components you need, downloading and installing the package on your site, and linking to this local version, you can reduce the amount of data that visitors to your site need to download. We'll show you how that's done step-by-step.

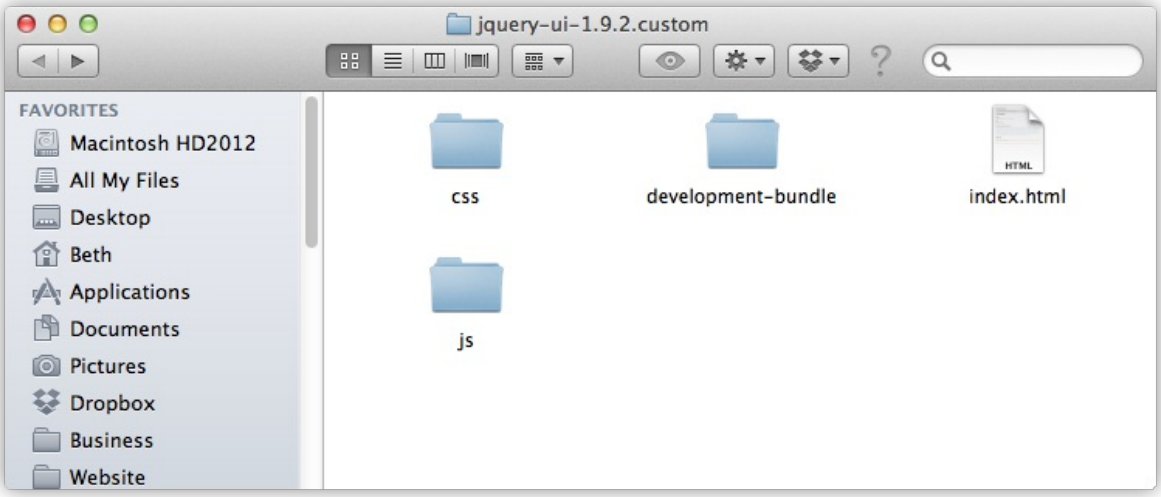
Note If you don't want to download anything, you can continue linking to the full library for the rest of the course and everything will work the same way. We'll show you how to link to an individual theme in just a moment. However, even if you're not going to actually download and install the custom jQuery UI library with your selected theme, follow along so you can get the hang of how it works in case you want to do it in the future.

Download, Install, and Link to a Customized jQuery UI Library and Theme

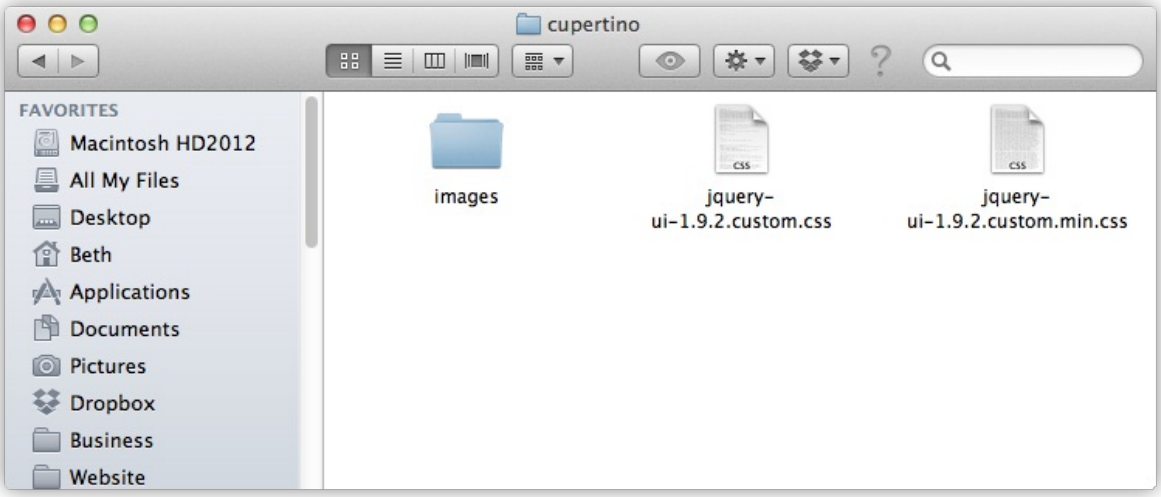
To use the accordion widget, we need only three components: "Core" and "Widget" from the UI Core part of the library, and "Accordion" from the widgets part of the library. Use the checkboxes to deselect all the other components on the page (scroll down to make sure you've deselected everything else; it's a long page with lots of components!).

Tip You can use the **Toggle All** check box in each group to select or deselect all of the items in that group.

Once you've deselected everything you don't need, scroll down to the very bottom of the page and click **Download**. This will download a zip file to your local computer, probably named something like "jquery-ui-1.9.2.custom." Locate this zip file on your computer and unzip it (if you're on a Mac, it will probably be unzipped automatically for you, in which case you'll be looking for a folder instead of a zip file; in Windows, it might be in your **/Downloads** folder). Open the folder, and you'll see:



Navigate into the **css** folder, and you'll see a folder named **/Cupertino**. All the CSS and images you need for the theme are in this folder. Navigate into the **/Cupertino** folder; you'll see:



Notice there are two versions of the CSS file: a "minified" version, and a full version. The "minified" version is compressed to take up as little space as possible. It's recommended that you link to the "minified" version of the CSS in your web page to reduce the amount of data that visitors to your website need to download.

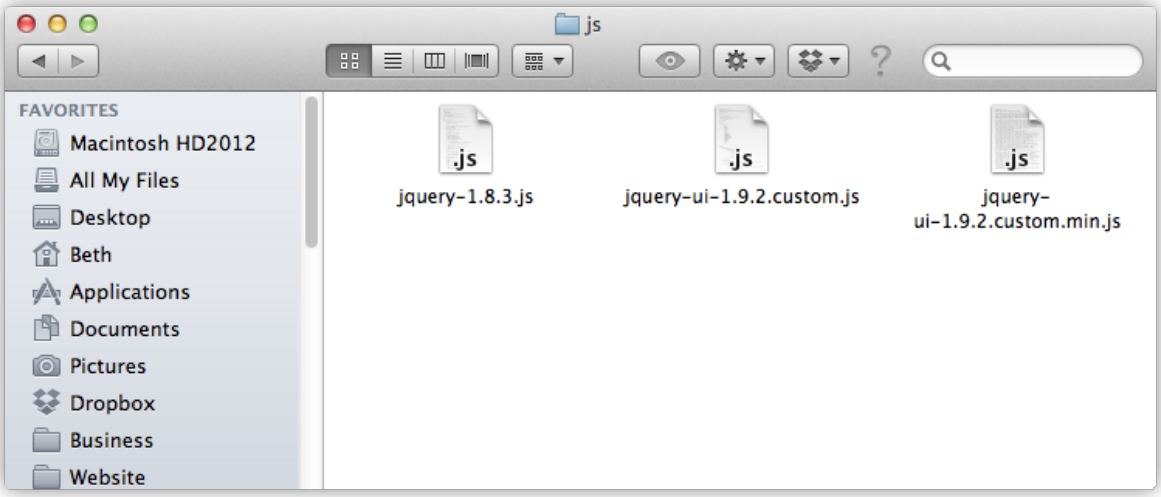
If you are experimenting with code on your own computer, you can move this folder to a location inside the folder where you're doing your local web development. You'll want to make sure it's accessible from the web page where you're experimenting via a relative path (or via a URL if you are running a web server locally), and create a link at the top of your page that looks something like this:

OBSERVE:

```
<link rel="stylesheet" href="jquery-ui-1.9.2.custom/css/cupertino/jquery-ui-1.9.
2.custom.min.css">
```

This relative path assumes that the folder containing the theme is located in the same folder where my HTML file is located.

Next, look at the unzipped folder again, and navigate back to the top level. Then, navigate into the **js** folder; you'll see three JavaScript files:



This folder contains the full jQuery UI library, your customized version, and a "minified" version of your customized version. Just like with the CSS file, it's recommended that you link to the "minified" version to reduce the amount of data your visitors need to download. Take a look at the non-minified version (and the minified version too if you want to see what "minimizing" does to JavaScript!). We certainly don't expect you to go through it or understand it all, but it might be fun for you to see how jQuery is implemented. (Also, this library is a great example of really well-written code).

To link to the customized jQuery UI library JavaScript file on your own computer, you'll create a link that looks like this:

OBSERVE:
`<script src="jquery-ui-1.9.2.custom/js/jquery-ui-1.9.2.custom.min.js"></script>`

Again, my code here assumes that I placed the unzipped folder in the same folder where my HTML file is located.

Accessing the Theme in Your Course Sandbox

If you are particularly motivated and want to upload the jQuery UI customized library and theme to your course sandbox, you can do that; however, it's not necessary! Briefly, here are the steps you'll need to follow:

First find the zip file that was downloaded from the jQuery UI theme roller site. If your computer automatically unzipped this file into a folder for you, you'll need to zip it back up.

Then, using the File Browser on the course site, use **File | Upload file** to upload the zip file and place it into your jQuery folder.

To unzip the file on the course system, open a Terminal window and log in to your student account. Then, use these commands to unzip the file:

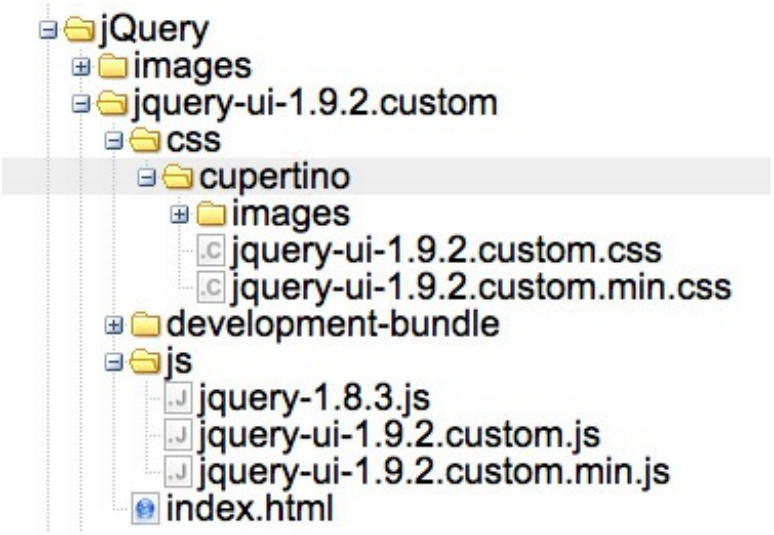
INTERACTIVE SESSION:

```
cold1:~$ cd jQuery

cold1:~/jQuery$ ls

accordion.html     images                        looping.html    select.html

first.html         jquery-ui-1.9.2.custom.zip    looping.js      style.html

firstProject.html  lightSwitch.png               playlist.html   Tilla.JPG

cold1:~/jQuery$ unzip jquery-ui-1.9.2.custom.zip
...
cold1:~/jQuery$ ls

accordion.html     images                        lightSwitch.png  __MACOSX       s
tyle.html
first.html         jquery-ui-1.9.2.custom        looping.html     playlist.html  T
illa.JPG
firstProject.html  jquery-ui-1.9.2.custom.zip    looping.js       select.html
```

Note If the version number is different for the jQuery UI library you downloaded, modify your **unzip** command appropriately.

Once you unzip the file, you see the **folder listed** in your jQuery directory.

You can now access the folder using the File Browser:



You can use the same two links we showed earlier to link to the CSS and JS files from your HTML, assuming your HTML file (with the jQuery) is located at the same directory level as where you uploaded and unzipped the jQuery UI folder:

OBSERVE:

```
<link rel="stylesheet" href="jquery-ui-1.9.2.custom/css/cupertino/jquery-ui-1.9.
2.custom.min.css">
<script src="jquery-ui-1.9.2.custom/js/jquery-ui-1.9.2.custom.min.js"></script>
```

Linking to the "Cupertino" Theme on jquery.com

If you don't want to deal with downloading and installing the custom jQuery UI library and theme locally, you

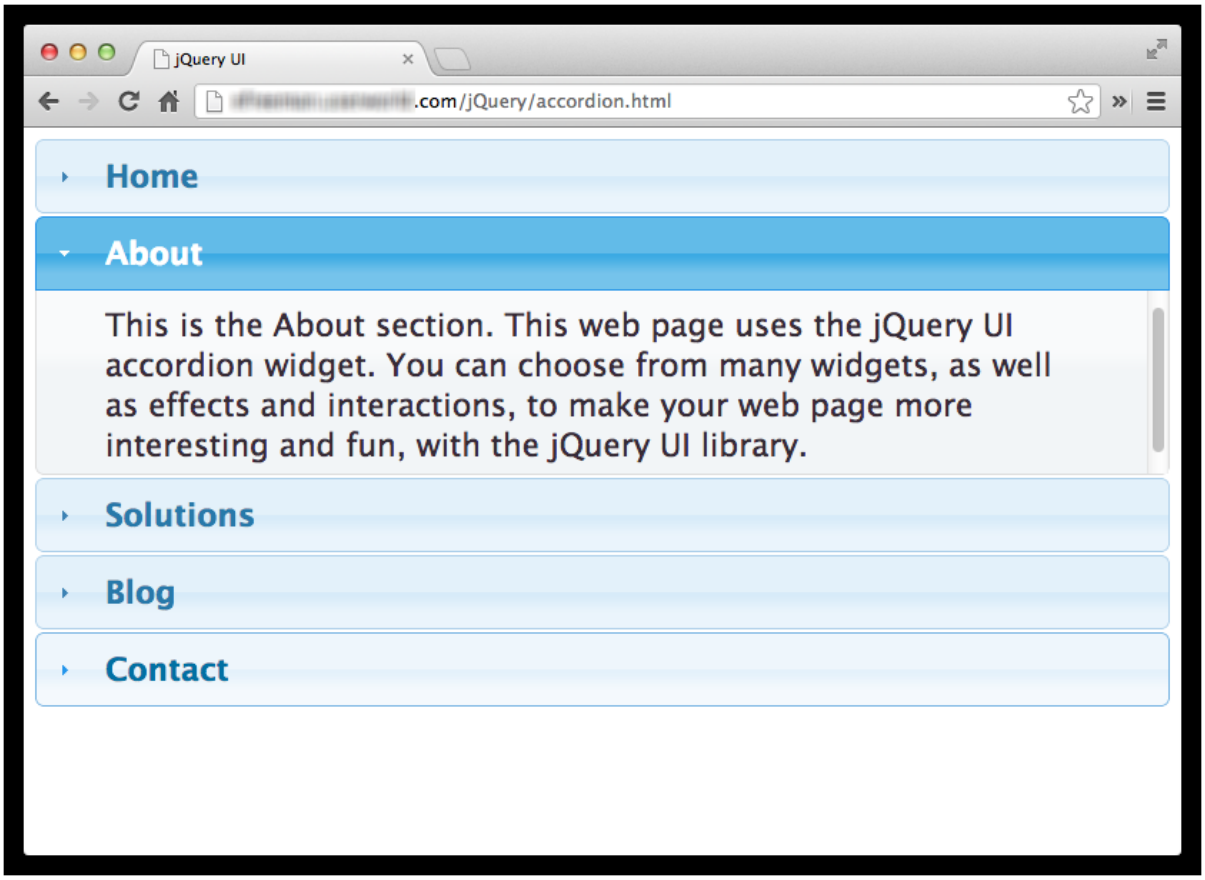
can link to the theme file for the theme we've chosen on the jQuery site. We'll leave the links to the jQuery library and the jQuery UI library the same as they were (which means that you are still linking to the *full* jQuery UI library, including components that you don't actually need, but that's okay). Update **accordion.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery UI: Widgets</title>
  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="http://code.jquery.com/ui/1.9.2/jquery-ui.js"></script>
  <link rel="stylesheet" href="http://code.jquery.com/ui/1.9.2/themes/cupertino/
jquery-ui.css">
  <script>
      $(document).ready(function() {
        $("#accordion").accordion({
            collapsible: true
        });
      });
  </script>
</head>
<body>
  <div id="accordion">
    <h1>Home</h1>
    <p>This is the Home section.</p>
    <h1>About</h1>
    <p>
        This is the About section. This web page uses the jQuery UI
        accordion widget. You can choose from many widgets,
        as well as effects and interactions, to make your web page
        more interesting and fun, with the jQuery UI library.
    </p>
    <h1>Solutions</h1>
    <p>This is the Solutions section.</p>
    <h1>Blog</h1>
    <p>This is the Blog section.</p>
    <h1>Contact</h1>
    <p>This is the Contact section.</p>
  </div>
</body>
</html>
```

Believe it or not, that's the only change you need to make to use the theme!  Save and 

preview the file:



You'll see that the accordion widget looks very different now, but the behavior is the same. You can click on a header to open a section, and you can close the open section by clicking on its header (the "collapsible" feature).

Getting your theme installed locally is a little tricky, but *using* the theme isn't.

Adding the Sortable Interaction to the Accordion

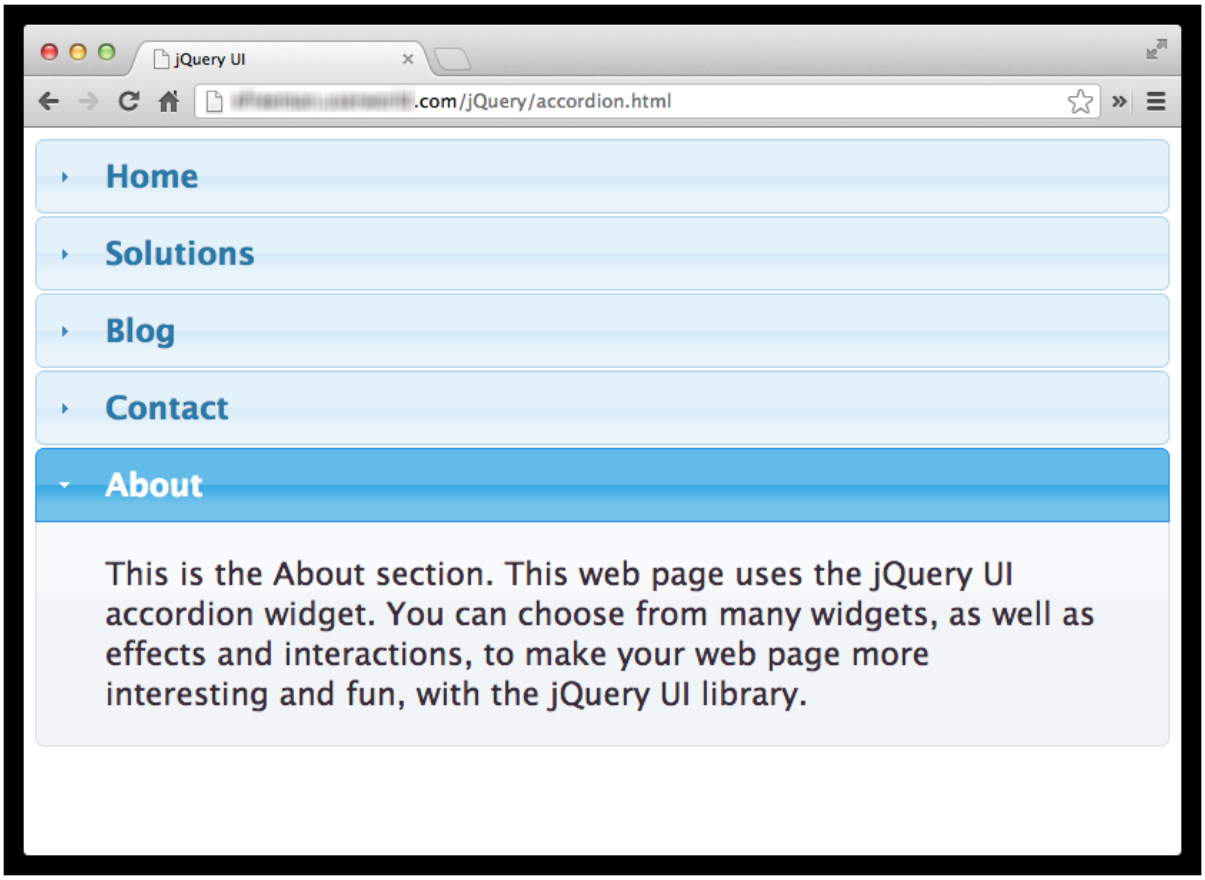
Let's customize the accordion widget just a bit more, and add the "Sortable" interaction. Go to the accordion widget page again, click **Sortable** on the right side, and play with the widget a bit. Try dragging a section, and notice that you can change the order of the sections. This is what the Sortable interaction does: it adds the capability to sort, or move around, the sections in the widget. Make sure you "View source" on this page and take a close look at the HTML and jQuery to see a few differences from the previous version of the accordion.

Update your HTML and jQuery:

```
<!doctype html>
<html>
<head>
  <title>jQuery UI: Widgets</title>
  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="http://code.jquery.com/ui/1.9.2/jquery-ui.js"></script>
  <link rel="stylesheet" href="http://code.jquery.com/ui/1.9.2/themes/cupertino/
jquery-ui.css">
  <script>
    $(document).ready(function() {
        $("#accordion").accordion({
            collapsible: true,
            header: "> div > h1"
        }).sortable({
            axis: "y",
            handle: "h1"
        });
    });
  </script>
</head>
<body>
  <div id="accordion">
    <div class="group">
      <h1>Home</h1>
      <p>This is the Home section.</p>
    </div>
    <div class="group">
      <h1>About</h1>
      <p>
        This is the About section. This web page uses the jQuery UI
        accordion widget. You can choose from many widgets,
        as well as effects and interactions, to make your web page
        more interesting and fun, with the jQuery UI library.
      </p>
    </div>
    <div class="group">
      <h1>Solutions</h1>
      <p>This is the Solutions section.</p>
    </div>
    <div class="group">
      <h1>Blog</h1>
      <p>This is the Blog section.</p>
    </div>
    <div class="group">
      <h1>Contact</h1>
      <p>This is the Contact section.</p>
    </div>
  </div>
</body>
</html>
```

 Save the file and 

preview. The accordion widget looks the same, but now you can click on a heading to move a section around. Try moving the "About" section to the bottom:



To add the "sortable" feature to the accordion widget, we had to update the HTML a bit. We added elements to group the heading and paragraph together for each section. This tells the browser how to group the items in each section, so when you click on a heading to move a section around, the browser knows which pieces of the structure to move.

Let's take a look at the changes in the jQuery:

OBSERVE:

```
$(document).ready(function() {
    $("#accordion").accordion({
        collapsible: true,
        header: "> div > h1"
    }).sortable({
        axis: "y",
        handle: "h1"
    });
});
```

First, we add a **"header" property** to the object we're passing to the **accordion()** method. This is a selector that tells the browser which item is the header for each section. We had to add this "header" property because, by default, the accordion uses the "first-child" selector on the main accordion container (in our case, that's the with the id, "accordion"), to figure out which item is acting as the header. Now that we've grouped together the header and content for each section within a , this default no longer works. So, we tell the accordion which item to use for the header with this selector, which reads "the heading level 1 which is a child of a div which is a child of the main accordion container."

Then, to add the sortable interaction to the accordion widget, we use the **sortable()** method, again passing in an object to customize the feature. First, we specify the "axis" property and set it to "y." This ensures that we can drag only the items in the accordion on the vertical axis (so we can't put an item next to another item horizontally). Then, we specify the "handle" property, which restricts which element you can click on to sort the sections to the <h1> element; that is, the heading for each section.

Spend some time exploring the jQuery UI API documentation for accordion and sortable. There is a lot of information on these pages about how to use the accordion widget and the sortable interaction, along with

plenty of examples of how to use them in different situations.

The Selectable Interaction

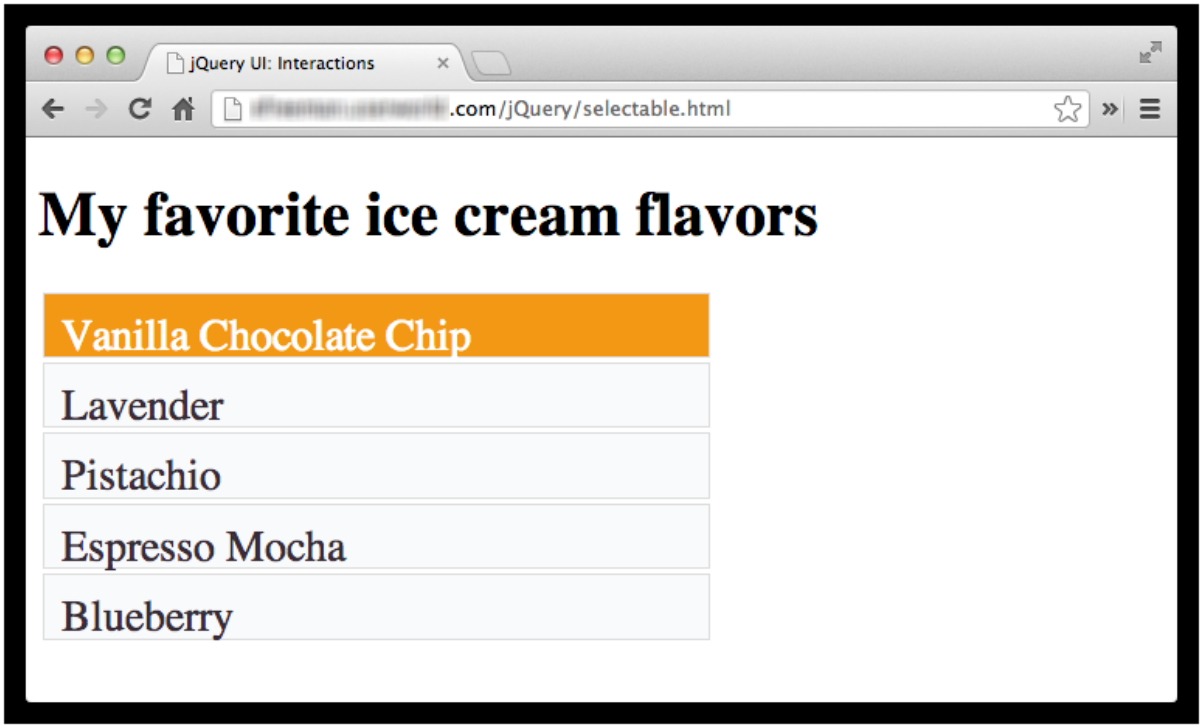
Let's explore another interaction: selectable. As you can probably tell, all jQuery UI interactions are *behavior* that you can add to widgets or elements in your page. Create a new HTML file and add this HTML and jQuery code:

CODE TO TYPE:

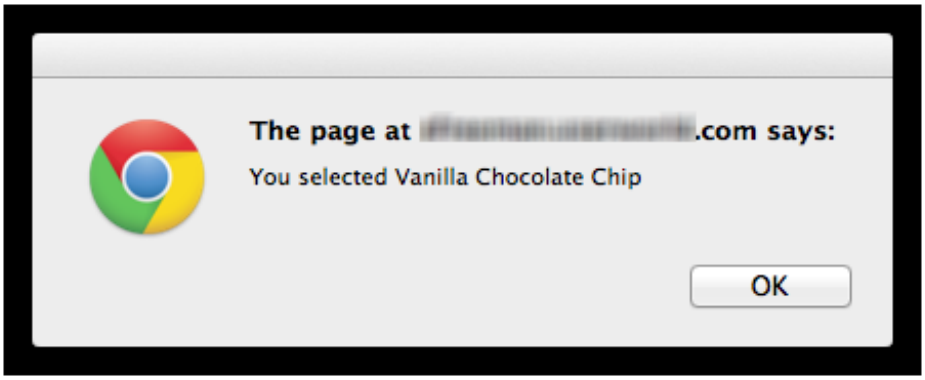
```
<!doctype html>
<html>
<head>
  <title>jQuery UI: Interactions</title>
  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="http://code.jquery.com/ui/1.9.2/jquery-ui.js"></script>
  <link rel="stylesheet" href="http://code.jquery.com/ui/1.9.2/themes/cupertino/
jquery-ui.css">
  <style>
      #selectable { list-style-type: none; margin: 0; padding: 0; width: 60%; }
      #selectable li { margin: 3px; padding: 0.4em; font-size: 1.4em; height: 18
px; }
      #selectable .ui-selected { background: #F39814; color: white; }
      #selectable .ui-selecting { background: #FECA40; }
  </style>
  <script>
    $(document).ready(function() {
        $("#selectable li").addClass("ui-widget-content");
        $("#selectable").selectable({
            selected: function(event, ui) {
                alert("You selected " + $(ui.selected).text());
            }
        });
    });
  </script>
</head>
<body>
  <h1>My favorite ice cream flavors</h1>
  <ul id="selectable">
    <li>Vanilla Chocolate Chip</li>
    <li>Lavender</li>
    <li>Pistachio</li>
    <li>Espresso Mocha</li>
    <li>Blueberry</li>
  </ul
</body>
</html>
```

 Save this file in your jQuery folder as **selectable.html**, and 

preview. Click the top item, **Vanilla Chocolate Chip**. It turns orange:



Also, an alert displays the selected item:



We've added the capability to select items in a list to a <ul> element. Now, remember, "selectable" is an **interaction**, not a **widget**, so what we've created here is a behavior, rather than a structure (with associated behavior). You can add this behavior to just about any HTML structure you want if you set it up correctly. You could also add this behavior to a widget, like we added sortable to our accordion widget earlier.

Let's take a closer look at how we created this behavior. The HTML is just a regular list element containing several list items. We added the "selectable" id to the list so we could select it in jQuery, but that's not necessary (as long as you can select a unique item to add the selectable behavior, that's enough).

Now, the jQuery:

OBSERVE:

```
$(document).ready(function() {
    $("#selectable li").addClass("ui-widget-content");
    $("#selectable").selectable({
        selected: function(event, ui) {
            alert("You selected " + $(ui.selected).text());
        }
    });
});
```

When we selected the "accordion" in the previous example, and used the **accordion()** method to turn it

into a widget, jQuery automatically added style to that widget (once we'd linked to the "Cupertino" theme). However, content you add to the page is not styled when you add a behavior to a structure. This makes sense, because typically you'll either be adding behavior to a widget (which is already styled), or to your own structures, which probably have their own style. So we need to style the "selectable" list ourselves.

We linked to the Cupertino theme in this file, so we can use the styles defined in that theme for the list, if we want. The first thing we do in our code is **add the class "ui-widget-content" to each <li> element in the "selectable" list**. The "ui-widget-content" class is a basic class for widgets that adds some nice styling to elements, so we're using it here. We're also providing some additional style of our own.

> **Note** You can look at the full CSS file for the Cupertino theme at the URL on jquery.com, or by downloading it using the theme roller, to see which styles are available in that file. You can use any of those styles on your own content; the styles aren't just for widgets!

Next, we select the "selectable" list, and call the **selectable()** method, which adds the selectable behavior to the list. We supply one option: a **selected property**, which is defined to be a **function that is called each time you select an item in the list**. This **function** has two parameters: **event** and **ui**. These get set when this function is called when you select an element. The **ui** parameter is an object that contains the selected element. You can access the selected element through the **ui.selected** property. The selected element will be one of the <li> elements, so we can display which ice cream flavor you selected by displaying the text of the selected <li> element. This element is the DOM element, not the wrapped jQuery object, so to use the **text()** method on this element, we wrap it first, using **$**.

Before we take a closer look at the CSS we added to the file, open the Element inspector development tool. Reload the page, and select one of the items. Look at this item in the Element inspector. Here's what it looks like in Chrome:

Styling Selected Elements

In the image above, the selected <li> element has three classes: "ui-widget-content," which we added in our jQuery, "ui-selectee," and "ui-selected." "ui-selectee" and "ui-selected" are added automatically to the element by the jQuery UI library. "ui-selectee" indicates that an element is eligible for selection, and "ui-selected" indicates that an element is actually selected. So, by providing styles for the "ui-selected" class, we can make an element that's been selected look different from unselected elements. That's where the CSS comes in!

OBSERVE:

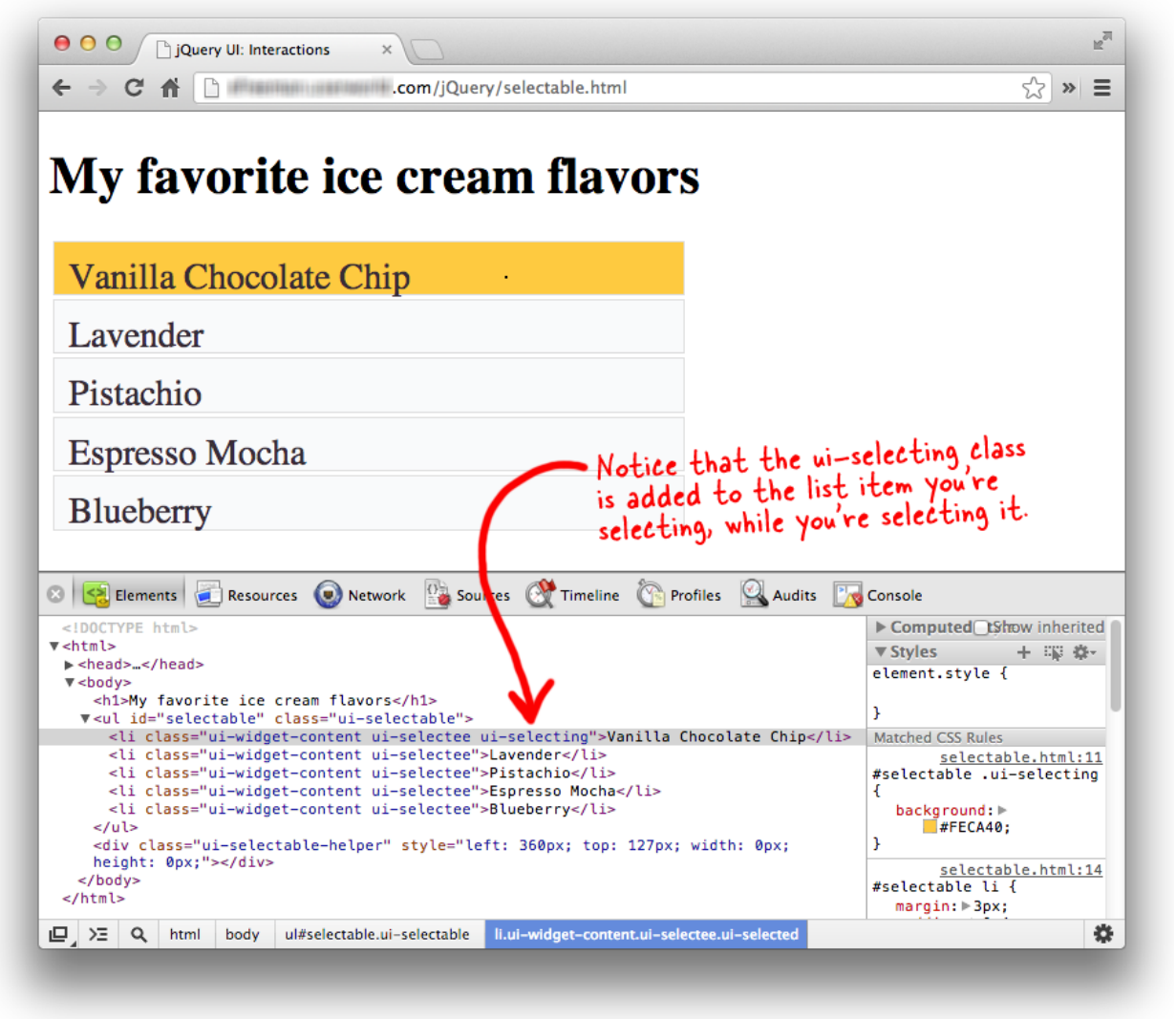
```
#selectable { list-style-type: none; margin: 0; padding: 0; width: 60%; }
#selectable li { margin: 3px; padding: 0.4em; font-size: 1.4em; height: 18px; }
#selectable .ui-selected { background: #F39814; color: white; }
#selectable .ui-selecting { background: #FECA40; }
```

The first two rules are applied to style the "selectable" list and list items; this is just additional style to that which is provided by the "ui-widget-content" class that we're using from the Cupertino theme.

Next, look at the rule to style the **.ui-selected** class. This rule sets the background color and text color of any element in the "selectable" list with the "ui-selected" class. jQuery UI is ading this class to your selected elements automatically, so they get this style.

The last rule, for the **"ui-selecting"** class, is a class that is added to an element while you're in the process

of selecting it. To check it out, go back to the web page, make sure that your element inspector is open, and click down on one of the list items— don't let go of the mouse button—just hold your mouse still over the item with the button pressed down. If you're in Chrome, you can look at the element and see the "ui-selecting" class is added to the element you're in the process of selecting.



This gives the element a slightly different look and feel while you're selecting it, versus when it's actually selected. As soon as you let up on the mouse button, this class is removed.

Using Selected Items

You're using the **selected** property in the argument to the **selectable()** method to be notified whenever you select an element, which means that each time you select an element, you'll see an alert.

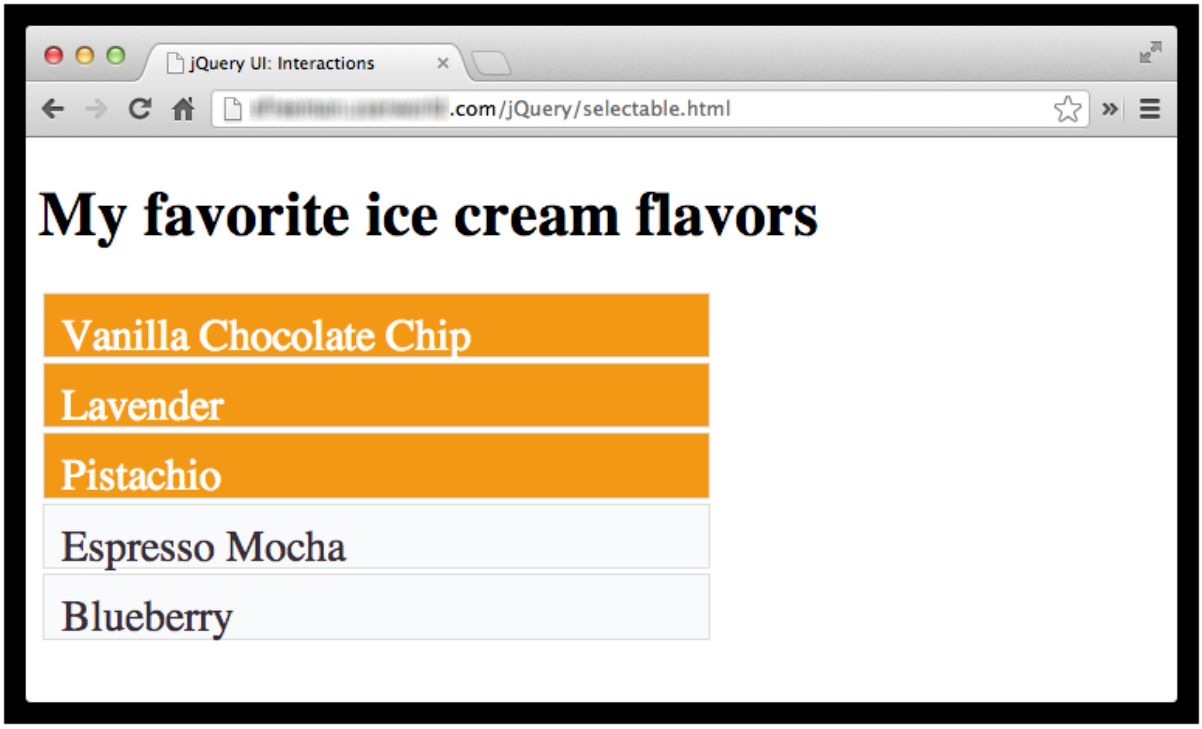
Notice that you can select multiple items from the "selectable" list. If you click on an item and then drag across other items in the list, you'll see that more than one item is selected. You'll also see that you'll get a separate alert for each selected item!

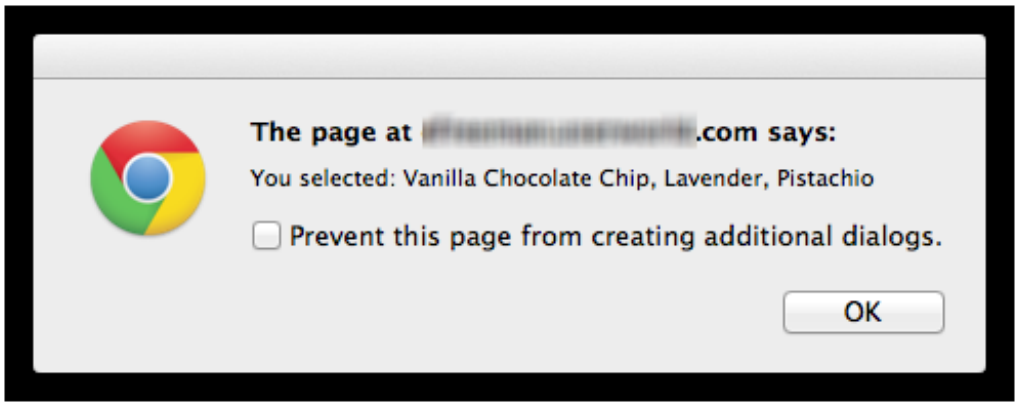
If you want to be notified only once, after the selection is complete, and do something with all those selected items at once, you can use the **stop** property:

CODE TO TYPE:

```
$(document).ready(function() {
    $("#selectable li").addClass("ui-widget-content");
    $("#selectable").selectable({
        selected: function(event, ui) {
            //alert("You selected " + $(ui.selected).text());
        },
        stop: function() {
            var flavors = "";
            $(".ui-selected").each(function() {
                flavors += $(this).text() + ", ";
            });
            flavors = flavors.substr(0, flavors.length - 2);
            alert("You selected: " + flavors);
        }
    });
});
```

We commented out the line in the selected function that alerts each time you select an item, but we've left the property in there, so you can see how to supply two properties as options for the **selectable()** method (and there may be times when you need to take action as you're selecting individual items, as well as after the selection process has completed for all items). Save the file and Preview preview. Try selecting one item, then try selecting multiple items. In either case you see just one alert that displays all the values you've selected, whether it's one or more than one.





As you saw earlier, after an element has been selected, the class "ui-selected" is added to the element. So a convenient way to select (in jQuery) all the elements that you've selected (in the page), is to **use the "ui-selected" class as your selector**:

OBSERVE:

```
stop: function() {
    var flavors = "";
    $(".ui-selected").each(function() {
        flavors += $(this).text() + ", ";
        });
    flavors = flavors.substr(0, flavors.length - 2);
    alert("You selected: " + flavors);
}
```

We know that the **stop** function is run pnly after the selection process has completed. So we **select all the selected list items** and loop through each one using **each()**. In the **each function**, we append the text of the item (the ice cream flavor) and a comma (to separate the ice cream flavors) to a string, **flavors**.

Then we **remove the extra comma** at the end of the string (because the last item in the string will have a comma after it as well), and then alert the string, which displays each ice cream flavor you selected, separated by a comma. You could do other things with the selected items depending on the functionality of your page too!

In this lesson, you learned how to use the jQuery UI library. This library has interactions, widgets, and effects you can use in your jQuery programs. You can create a consistent look and feel for the widgets (and other elements) in your page using jQuery UI themes. You can save a lot of time using the features in this library, and the good news is that you can use all these components in any of the modern browsers (and most are backward-compatible with older browsers, at least by a few versions), so you don't have to worry about differences between browsers.

Have fun with the project (building a game using a jQuery UI effect) and we'll see you in the next lesson!

jQuery and Ajax

Lesson Objectives

- You will use jQuery Ajax methods to load data dynamically into your page from external files.
- You will use **load()** to load HTML and add it to the page; and you'll use the utility methods **$.get()** and **$.getJSON()** to load JSON data for a graph.

---

jQuery and Ajax

If you've been working with web pages and JavaScript for a while, you've probably run across **Ajax**. Ajax is a set of capabilities that you use to load data and update your page from JavaScript. This allows you to create web sites that can change and respond to user input without doing a page refresh or loading a whole new page. If you've used Google mail or maps, you've seen Ajax in action. Google doesn't load a map of the entire world when you access the map application; it loads the pieces that are nearest to the location you specify. When you move the map around, new parts of the map load dynamically. That way, the amount of data loaded into the web page at any one time is fairly small, but you can access different parts of the map without loading the page again.

Ajax is also known as **XHR** because of the JavaScript object that's used to make these dyanmic requests to the server for more data: **XMLHttpRequest**. This object was named back in the day, when XML was the primary choice for data serialization; now many applications use **JSON** instead. JSON is "JavaScript Object Notation," and it's a way of storing data using a format that is expressed just like you express objects in JavaScript. The advantage to using JSON over XML for data serialization is that when you load JSON, JavaScript can turn the data into JavaScript objects that are ready for you to work with automatically. So unlike XML, you don't have to worry about parsing the data, and accessing the properties of an object is more straightforward than navigating an XML document tree.

You can also load data in formats besides XML and JSON, including text and HTML, and more recently, even binary data (like images). In this lesson, we'll work with two examples: for the first, we'll use HTML, and for the second, JSON.

The jQuery library makes working with Ajax incredibly convenient! Let's dive in and see how.

Loading Data with load()

For the first example, we'll work with the same code we used in the Effects lesson. In this example, we created menu tabs and used basic effects to slide the tabs, and the content panels, up and down. Go ahead and copy your file **tabs.html** to a new file, **tabsWithAjax.html**, and make the changes below (or you can create the file from scratch using the code below). Note that, except for the indicated changes, the code is the same as at the end of that previous lesson:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Ajax</title>
  <meta charset="utf-8">
  <style>
html, body {
    width: 100%;
    height: 100%;
    font-family: Helvetica, Arial, sans-serif;
}
body {
    margin: 0px;
    padding: 0px;
    background-color: #CDDAE6;
}
header {
    position: relative;
    top: 0px;
    height: 100px;
    margin: 0px;
    background-color: #336699;
    box-shadow: 0px 3px 5px #24476B;
    z-index: 3;
}
header h1 {
    color: white;
    margin: 0px;
    padding: 20px 0px 0px 20px;
}
ul#tabs {
    position: relative;
    top: 10px;
    list-style-type: none;
    width: 800px;
    margin: 0px auto;
    padding: 0px 0px 0px 10px;
    z-index: 2;
}
ul#tabs li {
    position: relative;
    display: inline;
    padding: 30px 20px 5px 20px;
    border-radius: 0px 0px 5px 5px;
    box-shadow: 1px 1px 2px black;
    background-color: #E68A00;
}
ul#tabs li a {
    color: white;
    text-shadow: 1px 1px 2px black;
    text-decoration: none;
}
div#content {
    position: relative;
    top: -25px;
    margin: auto;
    width: 800px;
}
div.tab-content {
    position: absolute;
    top: 0px;
    width: 770px;
    height: 600px;
    overflow: hidden;
    margin: 0px auto;
    padding: 80px 15px 15px 15px;
```

```
    background-color: #99B2CC;
    border-radius: 0px 0px 10px 10px;
    box-shadow: 0px 1px 3px grey;
    z-index: 0;
}
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="tabs.js"></script>
  <script src="tabsWithAjax.js"></script>
</head>
<body>
  <header><h1>Best Ice Cream Shop Ever</h1></header>
  <ul id="tabs">
    <li class="selected"><a href="home">Home</a></li>
    <li><a href="about">About</a></li>
    <li><a href="solutions">Solutions</a></li>
    <li><a href="blog">Blog</a></li>
    <li><a href="contact">Contact</a></li>
  </ul>
  <div id="content">
    <div id="home" class="tab-content">
        <h1>Home</h1>
        <p>This is the Home section.</p>
    </div>
    <div id="about" class="tab-content">
        <h1>About</h1>
        <p>This is the About section.</p>
    </div>
    <div id="solutions" class="tab-content">
        <h1>Solutions</h1>
        <p>This is the Solutions section.</p>
    </div>
    <div id="blog" class="tab-content">
        <h1>Blog</h1>
        <p>This is the Blog section.</p>
    </div>
    <div id="contact" class="tab-content">
        <h1>Contact</h1>
        <p>This is the Contact section.</p>
    </div>
  </div>
</body>
</html>
```

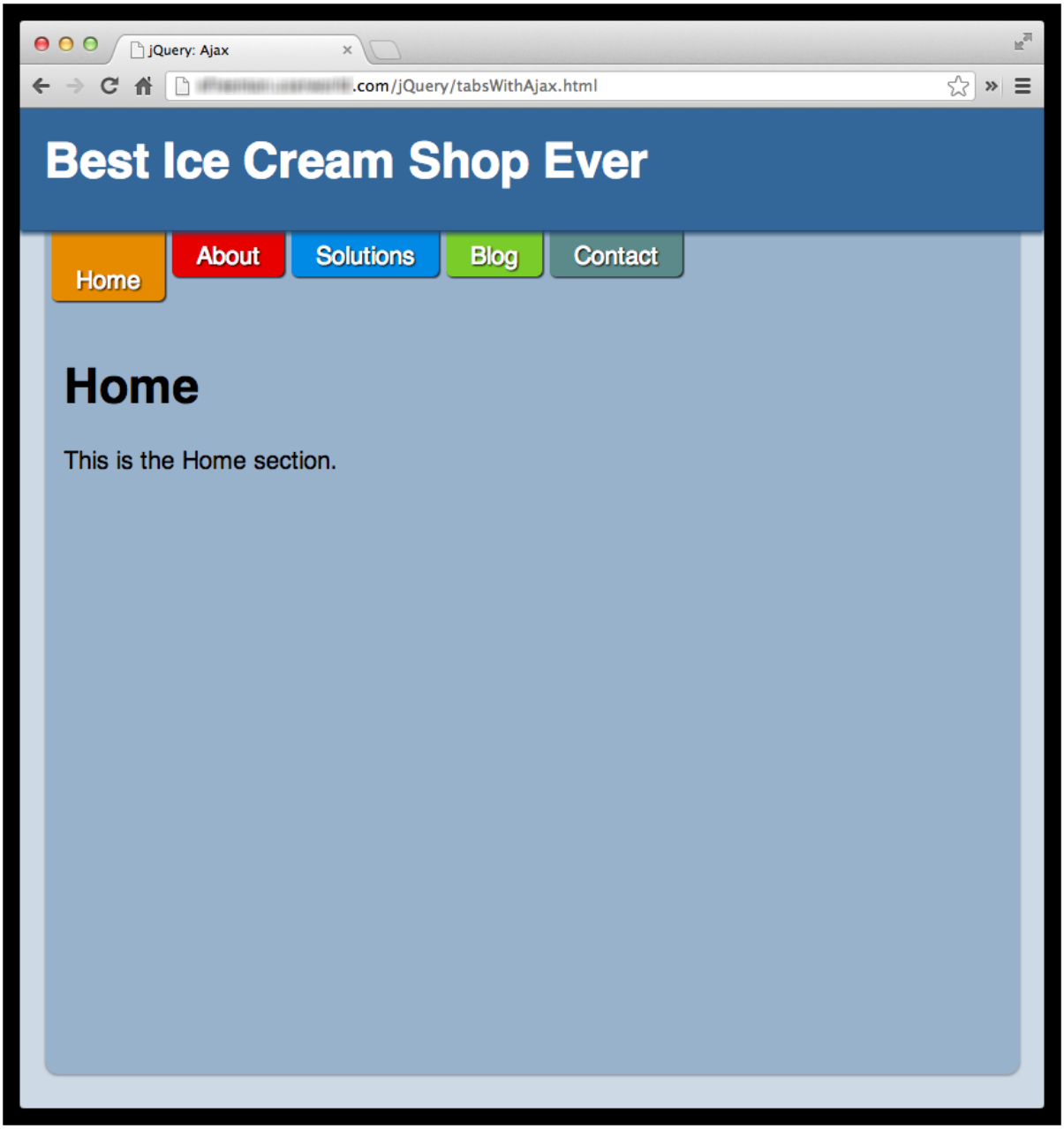
 Save this file in your **/jQuery** folder as **tabsWithAjax.html**. We added a bit of CSS and a new heading, and removed the headings from each of the content elements. Now, copy your file **tabs.js** to **tabsWithAjax.js**, or create a new file from the code below:

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the tabs except the home tab
    $("div#content div").hide();
    $("div#content div#home").show();
    $("ul#tabs li.selected").css("top", 20);

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        var $link = $(this);
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);

        $div.css("z-index", 1);
        $div.slideDown("slow", function() {
            $selectedDiv.hide();
            $div.css("z-index", 0);
        });

        $("ul#tabs li.selected").animate({
            top: 0
        });
        $("ul#tabs li.selected").removeClass("selected");

        $(this).parent().animate({
            top: 20
        });
        $(this).parent().addClass("selected");
    });
});
```

 Save this file in your **/jQuery** folder as **tabsWithAjax.js**. 

Preview your **tabsWithAjax.html** file. The page looks and behaves just like before, except that we've added a heading at the top:



Now we'll create HTML files containing content for each of the five tabs. Make sure you save these files in the same **/jQuery** folder where you created **tabsWithAjax.html** and **tabsWithAjax.js**:

CODE TO TYPE:

```
<div id="home-content">
  <h2>Best Ice Cream Shop Ever</h2>
  <p>
    We offer many flavors and toppings for your
    ice cream eating experience. Try our
    flavor of the month: Mocha Madness.
  </p>
</div>
```

Save this file in your **/jQuery** folder as **home.html**. Create another new HTML file as shown:

CODE TO TYPE:

```
<div id="about-content">
  <h2>About Best Ice Cream Shop Ever</h2>
  <p>
    We began in 1929 with just two flavors: chocolate
    and vanilla. But our ice cream was so good that
    people flocked from around the world to taste it!
    And now we've grown to 24 flavors.  People call us
    BICSE ("bik-see") for short.
  </p>
</div>
```

 Save this file in your **/jQuery** folder as **about.html**. Create another new HTML file as shown:

CODE TO TYPE:

```
<div id="solutions-content">
  <h2>Solutions from Best Ice Cream Shop Ever</h2>
  <p>
    We can cater your next birthday or office party
    with the best ice cream ever! So just call us
    and let the fun begin.
  </p>
</div>
```

 Save this file in your **/jQuery** folder as **solutions.html**. Create another new HTML file as shown:

CODE TO TYPE:

```
<div id="blog-content">
  <h2>Best Ice Cream Shop Ever Blog</h2>
  <h3>Special of the day</h3>
  <p>
    Special of the day is 20% discount on pints of
    Cherry Chocolate Chip!
  </p>
  <h3>BICSE donates ice cream</h3>
  <p>
    Best Ice Cream Shop Ever donated 10 pints of
    ice cream to families in need this Holiday Season
    to make their holidays a little jollier.
  </p>
</div>
```

 Save this file in your **/jQuery** folder as **blog.html**. Create another new HTML file as shown:

CODE TO TYPE:

```
<div id="contact-content">
  <h2>Contact the Best Ice Cream Shop Ever</h2>
  <p>
    Call us at 555-1212 or stop by our store at 121 Main Street, in FunTown, USA
.
  </p>
</div>
```

 Save this file in your **/jQuery** folder as **contact.html**.

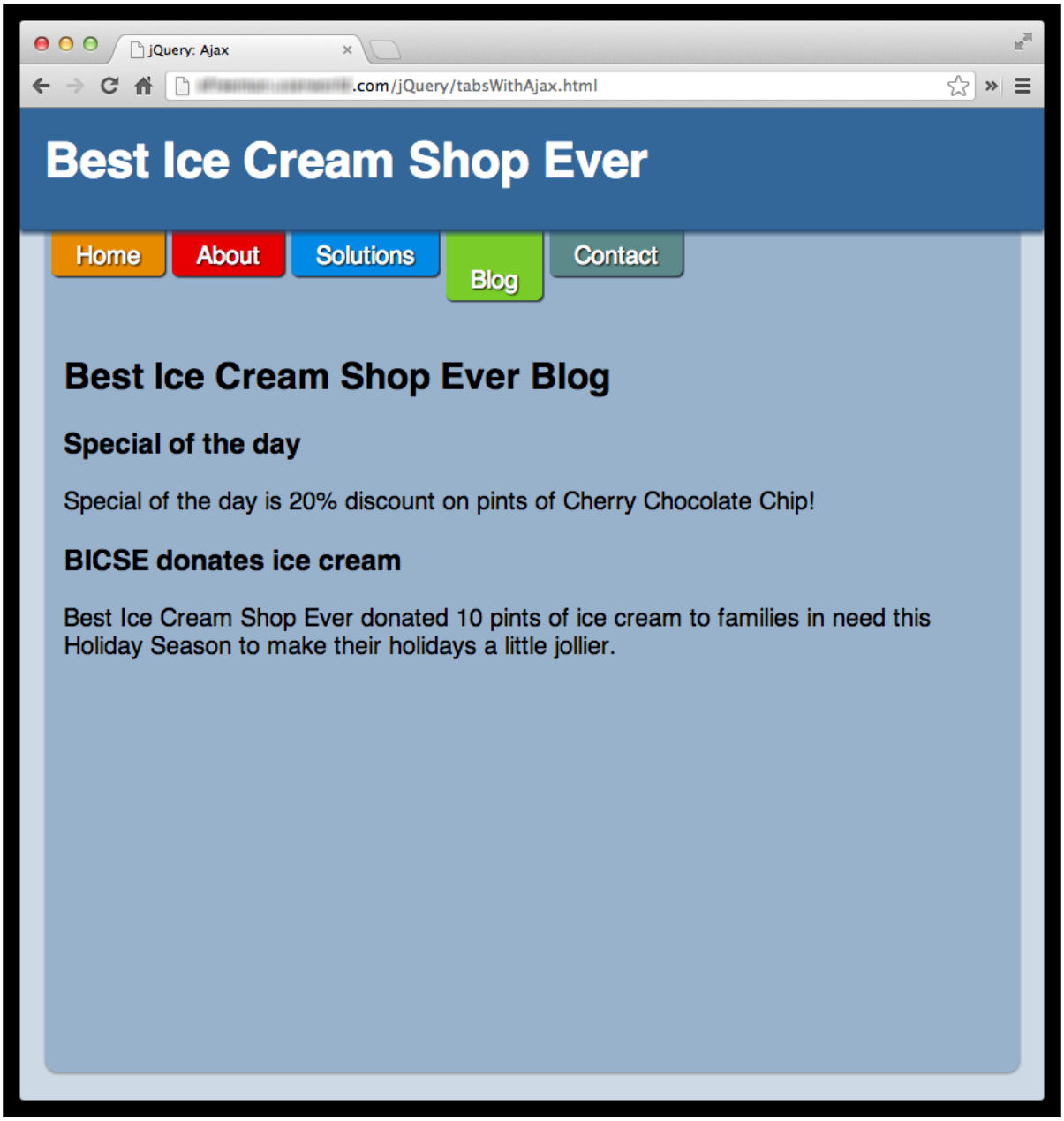
Each of these files is an incomplete HTML file; but that's okay, because they aren't meant to be viewed on their own. We're going to load the contents of these files into the tabs using Ajax. To do that, edit **tabsWithAjax.js**:

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the tabs except the home tab
    $("div#content div").hide();
    //Load all tabs with content from HTML files
    $("div.tab-content").each(function() {
        var id = $(this).attr("id");
        $(this).find("p").load(id + ".html");
    });
    $("div#content div#home").show();
    $("ul#tabs li.selected").css("top", 20);

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        var $link = $(this);
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);

        $div.css("z-index", 1);
        $div.slideDown("slow", function() {
            $selectedDiv.hide();
            $div.css("z-index", 0);
        });

        $("ul#tabs li.selected").animate({
            top: 0
        });
        $("ul#tabs li.selected").removeClass("selected");

        $(this).parent().animate({
            top: 20
        });
        $(this).parent().addClass("selected");
    });
});
```

 Save **tabsWithAjax.js** and 

preview your **tabsWithAjax.html** file. Try clicking on some of the tabs; the content from the HTML files you just created appears in each of the appropriate content panels:



Let's go over the code:

OBSERVE:

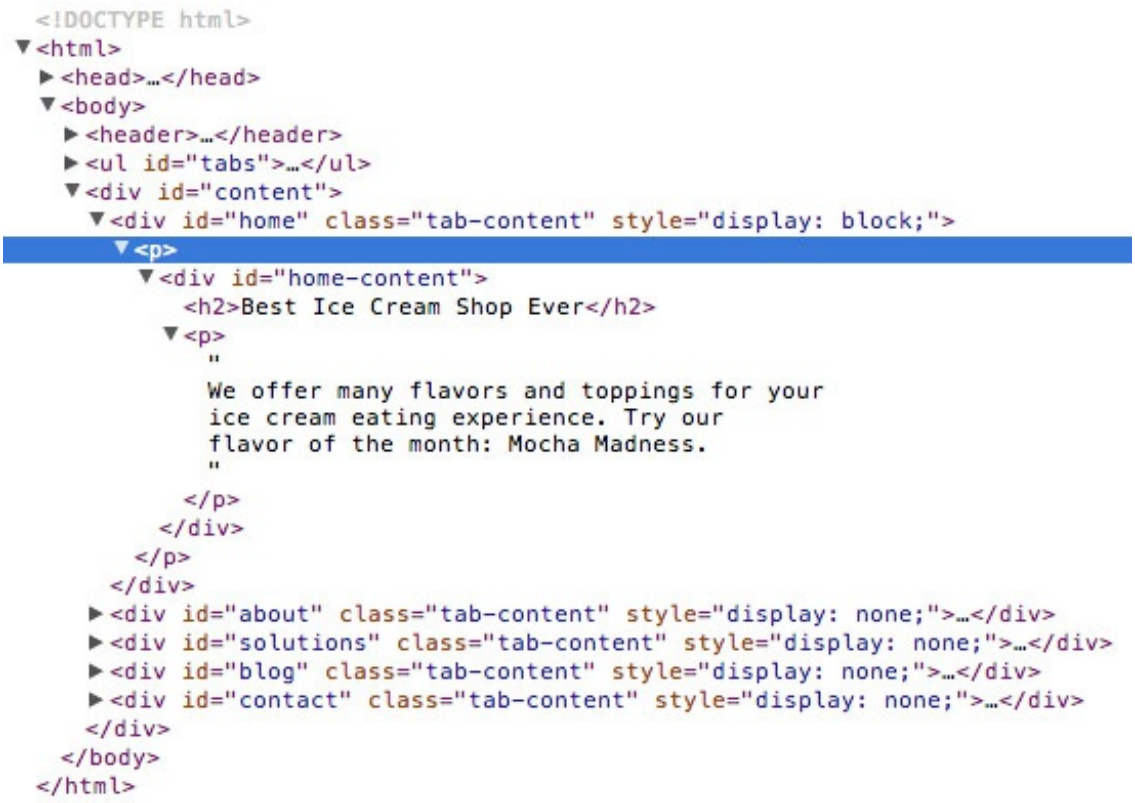
```
$("div.tab-content").each(function() {
    var id = $(this).attr("id");
    $(this).find("p").load(id + ".html");
});
```

First, we **select all the elements with the class "tab-content" from the HTML**. These are all the s with content that we're going to replace with data from the files. Then we **iterate though each of these s**. For each , we first **get that 's id attribute**. We'll use the id of each of the s to create the file name from which to pull the data. For instance, if the id is "home," the file name will be "home.html." We **get the <p> element nested in the **, and use the jQuery **load() method to load the data that's in the file** corresponding to the id of the we're working on currently. So, if that is the "home" , then we **load("home.html")**.

The **load()** method is a jQuery Ajax shortcut method that is often used for loading HTML into an element. It's a method you call on a selected element (in our case, the <p> element nested within the selected) to load content into that element. "Loading content" into the element replaces any content that's already there (just like using the **html()** method), so the default text we left inside each of the <p> elements gets replaced with the content from each of the files. If there's something wrong, and the content can't be loaded for some

reason, the default text will still be there and you'll see it in the page.

If you use the JavaScript console to inspect the s, you'll see the content from the files inside the <p> elements:



If you're familiar with using Ajax in JavaScript without jQuery, you'll recognize just how handy it is to use the **load()** method instead!

Options with load()

One of the advantages to using **load()** to load content from HTML files, is that you can select elements from the file you're loading right in your call to **load()**. Try that now. Update your jQuery as shown (inserting a space and "p" in the line **$(this).find("p").load(id + ".html"**):

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the tabs except the home tab
    $("div#content div").hide();
    //Load all tabs with content from HTML files
    $("div.tab-content").each(function() {
        var id = $(this).attr("id");
        $(this).find("p").load(id + ".html p");
    });
    $("div#content div#home").show();
    $("ul#tabs li.selected").css("top", 20);

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        var $link = $(this);
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);

        $div.css("z-index", 1);
        $div.slideDown("slow", function() {
            $selectedDiv.hide();
            $div.css("z-index", 0);
        });

        $("ul#tabs li.selected").animate({
            top: 0
        });
        $("ul#tabs li.selected").removeClass("selected");

        $(this).parent().animate({
            top: 20
        });
        $(this).parent().addClass("selected");
    });
});
```

 Save **tabsWithAjax.js** and 

preview your **tabsWithAjax.html** file. Now, the content of each of the tabs consists only of file content that is contained within <p> elements. Each of the content files you created has <p> elements containing part of the content. That's the content that shows in the content panels. To do this, we add a selector to the string we pass to the **load()** method with the file name. For instance, if we are loading content for the "home" tab, we pass the string "home.html p". Notice the space between the filename and the selector. This is really important: it's how jQuery knows which part of the string is the filename and which part is the selector. You can use any selector here. Of course, if you use a selector that doesn't match any content from the file, you won't get a result, and so the content will not show up in the page.

Another option with **load()** is the **complete** function. You've seen complete functions before (with animations), so you can probably guess that if you pass a complete function to **load()**, it will execute once **load()** has loaded the data from the file. Update your jQuery:

CODE TO TYPE:

```
$(document).ready(function() {
    // add colors to the tabs
    $("ul#tabs li a[href='about']").parent().css("background-color", "#E60000");
    $("ul#tabs li a[href='solutions']").parent().css("background-color", "#008AE
6");
    $("ul#tabs li a[href='blog']").parent().css("background-color", "#7ACC29");
    $("ul#tabs li a[href='contact']").parent().css("background-color", "#5C8A8A"
);

    // hide all the tabs except the home tab
    $("div#content div").hide();
    //Load all tabs with content from HTML files
    $("div.tab-content").each(function() {
        var id = $(this).attr("id");
        $(this).find("p").load(id + ".html", function() {
            if (id == "home") {
                $("div#content div#home").show();
            }
        });
    });
    $("div#content div#home").show();
    $("ul#tabs li.selected").css("top", 20);

    // click handler for all tabs
    $("ul#tabs li a").click(function(e) {
        var $link = $(this);
        // prevent the a link from causing a page load
        e.preventDefault();
        // get the currently selected tab, and associated div
        var selectedTabName = $("ul#tabs li.selected a").attr("href");
        var $selectedDiv = $("div#" + selectedTabName);
        // get the newly selected tab, and associated div
        var tabName = $(this).attr("href");
        var $div = $("div#" + tabName);

        $div.css("z-index", 1);
        $div.slideDown("slow", function() {
            $selectedDiv.hide();
            $div.css("z-index", 0);
        });

        $("ul#tabs li.selected").animate({
            top: 0
        });
        $("ul#tabs li.selected").removeClass("selected");

        $(this).parent().animate({
            top: 20
        });
        $(this).parent().addClass("selected");
    });
});
```

 Save **tabsWithAjax.js** and  preview your **tabsWithAjax.html** file. Now, we don't show the "home" content panel until we're sure that the **load()** method has completed loading the data (the complete function will also run even if the load fails, but in that case, we'll see the default content for the "home" panel). We check to make sure the **id** is "home," so we only **show()** the "home" once (rather than every time we call **load()** for each of the tabs, which would be overkill).

Using get()

Next, we'll use another jQuery Ajax shortcut method: **get()**. We'll use it to load JSON data for the graph we created back in the Viewport lesson. Again, you can start by making a copy of your **graph.html** file, or you can start from scratch with the code below:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery: Data Visualization - Bar Chart with Ajax</title>
  <meta charset="utf-8">
  <style>
html, body {
    width: 100%;
    height: 100%;
}
h1 {
    margin: auto;
    margin-bottom: 20px;
    width: 800px;
    font-size: 150%;
    font-family: Helvetica, Arial, sans-serif;
    color: gray;
}
div.container {
    margin: auto;
    position: relative;
    width: 800px;
    height: 600px;
    border: 1px solid gray;
}
div.year {
    font-size: 300%;
    font-family: Helvetica, Arial, sans-serif;
    color: gray;
}
div.dataContainer {
    border-bottom: 1px solid black;
    border-right: 1px solid black;
}
span.X {
    font-size: 50%;
    font-family: Helvetica, Arial, sans-serif;
}
span.Y {
    font-size: 50%;
    font-family: Helvetica, Arial, sans-serif;
}
div.F {
    background-color: rgba(238, 173, 218, .5);
}
div.M {
    background-color: rgba(121, 173, 210, .5);
}
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="graph.js"></script>
  <script src="graphWithAjax.js"></script>
</head>
<body>
</body>
</html>
```

 Save the file in your **/jQuery** folder as **graphWithAjax.html**. All we changed is the name of the JavaScript file we're linking to (from **graph.js** to **graphWithAjax.js**).

Next, we'll modify the jQuery we used to create the graph, in **graph.js**. We'll delete the contents of the **data** variable from that file, and put that array into a JSON file instead. So open **graph.js**, and copy the data into a new file (or you can copy it from the listing below):

CODE TO TYPE:

```
[
    [0,1,9735380],
    [0,2,9310714],
    [5,1,10552146],
    [5,2,10069564],
    [10,1,10563233],
    [10,2,10022524],
    [15,1,10237419],
    [15,2,9692669],
    [20,1,9731315],
    [20,2,9324244],
    [25,1,9659493],
    [25,2,9518507],
    [30,1,10205879],
    [30,2,10119296],
    [35,1,11475182],
    [35,2,11635647],
    [40,1,11320252],
    [40,2,11488578],
    [45,1,9925006],
    [45,2,10261253],
    [50,1,8507934],
    [50,2,8911133],
    [55,1,6459082],
    [55,2,6921268],
    [60,1,5123399],
    [60,2,5668961],
    [65,1,4453623],
    [65,2,4804784],
    [70,1,3792145],
    [70,2,5184855],
    [75,1,2912655],
    [75,2,4355644],
    [80,1,1902638],
    [80,2,3221898],
    [85,1,970357],
    [85,2,1981156],
    [90,1,336303],
    [90,2,1064581]
]
```

 Save this file in your **/jQuery** folder as **graph.json**. Make sure your file looks just like the above; that is, make sure you don't have **var data =** in front of the array (like we did with the code), and remove the semicolon (**;**) at the end of the array.

Okay, now that we have the data from the file, make another new file and copy your code from **graph.js** (or copy the code from the listing below):

CODE TO TYPE:

```
$(document).ready(function() {

var data = [
    [0,1,9735380],
    [0,2,9310714],
    [5,1,10552146],
    [5,2,10069564],
    [10,1,10563233],
    [10,2,10022524],
    [15,1,10237419],
    [15,2,9692669],
    [20,1,9731315],
    [20,2,9324244],
    [25,1,9659493],
    [25,2,9518507],
    [30,1,10205879],
    [30,2,10119296],
    [35,1,11475182],
    [35,2,11635647],
    [40,1,11320252],
    [40,2,11488578],
    [45,1,9925006],
    [45,2,10261253],
    [50,1,8507934],
    [50,2,8911133],
    [55,1,6459082],
    [55,2,6921268],
    [60,1,5123399],
    [60,2,5668961],
    [65,1,4453623],
    [65,2,4804784],
    [70,1,3792145],
    [70,2,5184855],
    [75,1,2912655],
    [75,2,4355644],
    [80,1,1902638],
    [80,2,3221898],
    [85,1,970357],
    [85,2,1981156],
    [90,1,336303],
    [90,2,1064501]
];

$("body").append("<h1>Population Distribution by Age</h1>");
$("body").append("<div class=\"container\"></div>");
$("<div>2000</div>").appendTo("div.container").addClass("year").css({
    position: "relative",
    top: 10,
    left: 10
});
$("<div></div>").appendTo("div.container").addClass("dataContainer").css({
    position: "absolute",
    top: 40,
    left: 40,
    width: $("div.container").width() - 80,
    height: $("div.container").height() - 80
});

var data;
$.get("graph.json", function(graphData) {
    console.log("Data loaded successfully");
    data = graphData;
    addAxes();
    addData();
}, "json");
```

```
addAxes();
addData();

function addAxes() {
    var numPoints = data.length / 2;
    var ptWidth = $("div.dataContainer").width() / numPoints / 2;
    var dataContainerLeft = $("div.dataContainer").position().left;
    for (var i = 0; i < data.length; i += 2) {
        var dataPt = data[i];
        var left = dataContainerLeft + (ptWidth * i);
        $("<span>" + dataPt[0] + "</span>").appendTo("div.container").addClass("
X").css({
            position: "absolute",
            bottom: 20,
            left: left + $("span.X").width()
        });
    }
    var max = findMax();
    var magMax = findMag(max);
    var ptHeight = $("div.dataContainer").height() / (magMax+1);
    var dataContainerTop = $("div.dataContainer").position().top;
    var left = $("div.dataContainer").width() + 50;
    for (var i = 0; i <= magMax; i++) {
        var top = dataContainerTop + (ptHeight * i);
        $("<span>" + (magMax-i) + "M</span>").appendTo("div.container").addClass
("Y").css({
            position: "absolute",
            top: top + $("span.Y").height(),
            left: left
        });
    }
}
function addData() {
    var numPoints = data.length / 2;
    var ptWidth = $("div.dataContainer").width() / numPoints / 2;
    var max = findMax();
    for (var i = 0; i < data.length; i++) {
        var dataPt = data[i];
        var sexClass;
        var pos;
        if (dataPt[1] == 1) {
            sexClass = "M";
            pos = i;
        } else {
            sexClass = "F";
            pos = i-1;
        }
        //var left = dataContainerLeft + ((ptWidth+2) * pos/2);
        var left = ptWidth * pos;
        var barHeight = (dataPt[2] * $("div.dataContainer").height()) / max;
        $("<div></div>").appendTo("div.dataContainer").addClass("dataPoint").add
Class(sexClass).css({
            position: "absolute",
            width: ptWidth,
            height: barHeight,
            bottom: 0,
            left: left
        });
    }
}

function findMax() {
    var max = 0;
    for (var i = 0; i < data.length; i++) {
        var dataPt = data[i];
        if (dataPt[2] > max) {
            max = dataPt[2];
        }
```

```
    }
    return max;
}

function findMag(num) {
    while (num > 1) {
        num = num / 10;
    }
    num = num * 100;
    num = Math.floor(num);
    return num;
}

});
```

 Save the file in your **/jQuery** folder as **graphWithAjax.js**, and  preview your **graphWithAjax.html** file. Your graph will look just like it did before, but now all the data for the graph is being loaded from the **graph.json** file, rather than being defined in the **data** variable in the jQuery file. We still need that variable though, because we're putting the data that we load from the file into it. Let's go over how we used the **$.get()** function to load the data, and talk about **$.get()** in detail:

OBSERVE:

```
    $.get("graph.json", function(graphData) {
    console.log("Data loaded successfully");
    data = graphData;
    addAxes();
    addData();
}, "json");
```

First, you might have noticed that **$.get()** looks a bit odd. So far we've used **$** as a function, to select elements from the page (like when we say **$("div")**, to select all <div>s), to create new elements (like when we say **$(" ")**), and to wrap elements (like when we say $(this)), so we can use jQuery methods on those elements. Okay, so what on earth is **$.get()**?

$.get() is a property of **$** that also happens to be a function. So, **$** is a function, and it has properties. In JavaScript, functions are objects, and just like other kinds of objects, functions can have properties too. Also, just like other objects, those properties can have functions as values (we call them methods, because they are functions in objects).

When you write: **$.get("graph.json", ...);** you're calling the method **get()** which is a property of **$**. We're not *calling* **$()** here; we're just accessing one of its properties, a property that happens to contain a method.

Compare this to when we call a method on the result of using **$** as a function:

OBSERVE:

```
$("div").addClass("highlight");
```

In this small example, we call the function **$()** to select all elements. This results in an array of elements wrapped in a special jQuery object that knows about methods you can call to do more things with those elements. In this case, we call the **addClass()** method to add the "highlight" class to each of those s. We call a method, **addClass()**, that is a property of that special jQuery object that wraps the array of elements.

Don't worry too much about the details of how functions work as objects; just recognize the difference between these two ways of calling methods using **$**. We call **$.get()** a **jQuery utility method**, because it's a method of the **$()** function itself, rather than a method of the *result* of calling that function. We'll experiment with other utility methods later, so you'll see more of these soon.

Now that you have a basic idea of what **$.get()** is, we can get back to the code:

OBSERVE:

```
    $.get("graph.json", function(graphData) {
    console.log("Data loaded successfully");
    data = graphData;
    addAxes();
    addData();
}, "json");
```

$.get() takes three arguments: the **name of the file** from which to load JSON data, a **callback function** that is called once that data has been loaded from the file successfully, and a **parameter that indicates what type of data** the **$.get()** method should expect. Since our data is represented as JSON, we're using "json" as the parameter (you can use "xml," "json," "script," or "html").

If **get()** is able to load the data from the file you specify successfully, the **callback function** is called, and the data is **passed into the function** as an argument; in our case, that's **graphData**. We **save this data in the data variable**, and then call **the addAxes() and addData() functions**. Notice that we moved these function inside the callback function, because we don't want them to execute until after we know the data has loaded successfully. If you moved these function calls back out of the callback function, below the call to **$.get()**, what do you think would happen? (Hint: remember that Ajax is asynchronous; that is, JavaScript doesn't wait for the data to be loaded before continuing to execute code!)

Using getJSON() to Get JSON Data

JSON is such a popular format for data that jQuery has another utility method, **$.getJSON()** that works exactly like **$.get()**, except you can skip that third parameter (the one that specifies the data type). Update your jQuery as shown:

CODE TO TYPE:

```
...
$.getJSON("graph.json", function(graphData) {
    console.log("Data loaded successfully");
    data = graphData;
    addAxes();
    addData();
}, "json");
...
```

 Save the file and 

preview your **graphWithAjax.html** file. Your page looks and behaves just like before.

So, what if **$.get()** (or **$.getJSON()**) encounters a problem and can't load the data from the file? All we've provided is a **success callback function**; we didn't specify what to do in the case that the Ajax fails.

Both **$.get()** and **$.getJSON()** have a slightly different alternative syntax that allows you to specify the **success()**, **error()**, and **complete()** functions. Let's see how that works. Update you jQuery as shown:

CODE TO TYPE:

```
$(document).ready(function() {

$("body").append("<h1>Population Distribution by Age</h1>");
$("body").append("<div class=\"container\"></div>");
$("<div>2000</div>").appendTo("div.container").addClass("year").css({
    position: "relative",
    top: 10,
    left: 10
});
$("<div></div>").appendTo("div.container").addClass("dataContainer").css({
    position: "absolute",
    top: 40,
    left: 40,
    width: $("div.container").width() - 80,
    height: $("div.container").height() - 80
});

var data;
$.getJSON("graph.json", function(graphData) {
    console.log("Data loaded successfully");
    data = graphData;
    addAxes();
    addData();
});
$.getJSON("graph.json")
    .success(function(graphData) {
        console.log("Data loaded successfully");
        data = graphData;
        addAxes();
        addData();
    })
    .error(function() {
        alert("Couldn't load graph data");
    })
    .complete(function(xhr, status) {
        // run after success or error
        console.log("Status: " + status);
    });

function addAxes() {
    var numPoints = data.length / 2;
    var ptWidth = $("div.dataContainer").width() / numPoints / 2;
    var dataContainerLeft = $("div.dataContainer").position().left;
    for (var i = 0; i < data.length; i += 2) {
        var dataPt = data[i];
        var left = dataContainerLeft + (ptWidth * i);
        $("<span>" + dataPt[0] + "</span>").appendTo("div.container").addClass("
X").css({
            position: "absolute",
            bottom: 20,
            left: left + $("span.X").width()
        });
    }
    var max = findMax();
    var magMax = findMag(max);
    var ptHeight = $("div.dataContainer").height() / (magMax+1);
    var dataContainerTop = $("div.dataContainer").position().top;
    var left = $("div.dataContainer").width() + 50;
    for (var i = 0; i <= magMax; i++) {
        var top = dataContainerTop + (ptHeight * i);
        $("<span>" + (magMax-i) + "M</span>").appendTo("div.container").addClass
("Y").css({
            position: "absolute",
            top: top + $("span.Y").height(),
            left: left
        });
```

```
    }
}
...
```

 Save the file and Preview preview your **graphWithAjax.html** file. Again, your graph looks the same. Check in the console, and you see the status message "success."

Now, temporarily rename the **graph.json** file to something else, like **graph2.json**, and Preview preview your **graphWithAjax.html** file. Your code will not be able to find the JSON file, which will cause an error—you may not see it because the data is cached, so try reloading the page. You'll see an alert message, "Couldn't load graph data," and in the console you see a status of "error." Rename the file back to **graph.json**, and make sure your program works properly again:

OBSERVE:

```
$.getJSON("graph.json")
    .success(function(graphData) {
        console.log("Data loaded successfully");
        data = graphData;
        addAxes();
        addData();
    })
    .error(function() {
        alert("Couldn't load graph data");
    })
    .complete(function(xhr, status) {
        // run after success or error
        console.log("Status: " + status);
    });
```

Now, we only pass one argument to **$.getJSON()**: the file name from which we want to load the JSON data. We also specify what happens on **success**, **failure**, and **when the Ajax is complete**, by using the **success()**, **error()**, and **complete()** methods. Each of these methods takes a function as an argument. The function argument to **success()** is exactly the same as the previous success function we used. The **error function** alerts you that the data couldn't be loaded. Only one of these two functions will run, because the Ajax call will either succeed or fail. Once the success or error function has run, then the complete function runs. This function has two parameters: the **XMLHttpRequest object, and a status**. We don't need the XMLHttpRequest object, but we use the status to provide more information about the status of the Ajax call in the console.

We've chained these three methods together; we call all of the methods to set up the appropriate callback functions, but only two of those callback functions will actually get called by jQuery (the success and complete callback functions, or the error and complete callback functions), depending on the success or failure of the XHR request.

Prevent Caching with ajaxSetup()

Some browsers will cache the data loaded by the jQuery Ajax calls, so if you make the same request repeatedly, you might get old data. In our case that doesn't matter because our data's not changing, but if you are generating your JSON data dynamically and frequently changing it, you will want to make sure that caching is turned off. To do that, update your jQuery as shown:

CODE TO TYPE:

```
...
$.ajaxSetup({ cache: false });
$.getJSON("graph.json")
    .success(function(graphData) {
        console.log("Data loaded successfully");
        data = graphData;
        addAxes();
        addData();
    })
    .error(function() {
        alert("Couldn't load graph data");
    })
    .complete(function(xhr, status) {
        // run after success or error
        console.log("Status: " + status);
    });
...
```

The Most Flexible Ajax Method: ajax()

The jQuery Ajax calls we've covered—**load()**, **$.get()**, and **$.getJSON()**—will cover your Ajax needs most of the time. However, if you need more control over your Ajax calls than these methods provide, you can use the **$.ajax()** method. We're not going to cover it in the lesson, but take some time to review the documentation online at http://api.jquery.com/jQuery.ajax/. You'll see that you have many options for customizing your Ajax calls with this method! The methods **load()**, **$.get()**, and **$.getJSON()** all use **$.ajax()** behind the scenes; they're just shortcut methods that make it a little easier to use Ajax.

There are two other Ajax shortcut methods: **$.post()** and **$.getScript()**. The **$.post()** method is analogous to **$.get()**, except that it makes a POST request to the web server, rather than a GET. You can explore this method online at http://api.jquery.com/jQuery.post/. The **$.getScript()** method is used to load other JavaScript dynamically into your existing JavaScript. The script is executed once it's loaded, and it executes as part of the page you're loading it into, so it has access to all of your functions and variables. Be careful with this one; make sure you're loading a script that you trust! You can get all the details about **$.getScript()** here: http://api.jquery.com/jQuery.getScript/.

In this lesson, you learned how to use jQuery Ajax methods to load data dynamically into your page from external files. We used **load()** to load HTML and add it to the page; and we used the utility methods **$.get()** and **$.getJSON()** to load JSON data for a graph. Excellent work! See you at the finish line!

jQuery Utilities

Lesson Objectives

- You will use the window object in jQuery.
- You will use jQuery to complete your real project.

More About jQuery, and a Few jQuery Utilities

In the previous lessons, almost everything we've done related to jQuery has involved the **$()** function. We've used it with selectors as a powerful way to select elements in our page for manipulation and styling; we've used it to create new elements; we've used it to wrap DOM elements so we can make use of the myriad jQuery methods, like **addClass()** and **show()** and **animate()**; and we've even used it to access utility methods, like **getJSON()**. This one function does a whole heck of a lot, doesn't it?

In this final lesson of the course, we'll revisit **$()**: it is the heart of jQuery, so it's important that you understand it thoroughly. We'll also look at the importance of structuring your code well, and finally we'll look at another utility built in to jQuery.

Once you're done with this lesson, you'll be ready to go out and put all this jQuery knowledge to good use in the real world. So hang in there with us for one more lesson (and, of course, the final project). It'll be fun, we promise!

jQuery and $

We mentioned at the very beginning of the course that **$** is actually a shortcut for a function named **jQuery**. So, we could write our code like this (create a new file and add the following HTML and jQuery):

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery</title>
  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>
jQuery(document).ready(function() {
    jQuery("#myDiv").text("Hello jQuery!");
});
  </script>
</head>
<body>
  <div id="myDiv"></div>
</body>
</html>
```

 Save this file in your **/jQuery** folder as **jquery.html**, and  preview. The "myDiv" has the text "Hello jQuery!" in the page.

Now, rewrite that code like this:

Update your jQuery:

```
jQuery$(document).ready(function() {
    jQuery$("#myDiv").text("Hello jQuery!");
});
```

 Save the file and  preview. Both work because **$** is just a shortcut name for **jQuery**. The **jQuery()** (or **$()**) function makes jQuery work; everything that is part of the jQuery library is defined in this function, including all the methods you use to manipulate element content and style. Look again at the (jQuery source code), near the bottom of the file, for the this line:

OBSERVE:
`window.jQuery = window.$ = jQuery;`

This line sets up the **jQuery** function so that it's accessible for your JavaScript (by adding the function to the global window object), and also sets up the **$** shortcut for jQuery.

You probably already know from your JavaScript experience that the **window** object is the context for everything in JavaScript: it's where all the globally available, built-in properties and methods are stored. Anything you add to **window** is immediately available to all the functions and methods in your JavaScript. So, when your browser loads the jQuery library, it's basically loading a large function with the name **jQuery** and adding it to the **window** object.

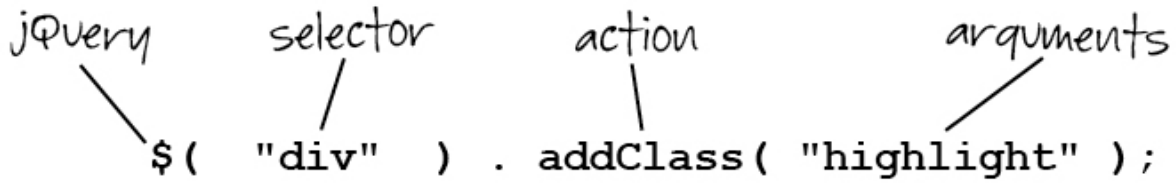
Just like the **window** object gives you access to all things JavaScript, the **jQuery** function gives you access to all things jQuery. All the properties, objects, and methods associated with jQuery are available through this **jQuery** function, which we usually refer to by its shortcut, or alias, **$**.

Dissecting a jQuery Statement

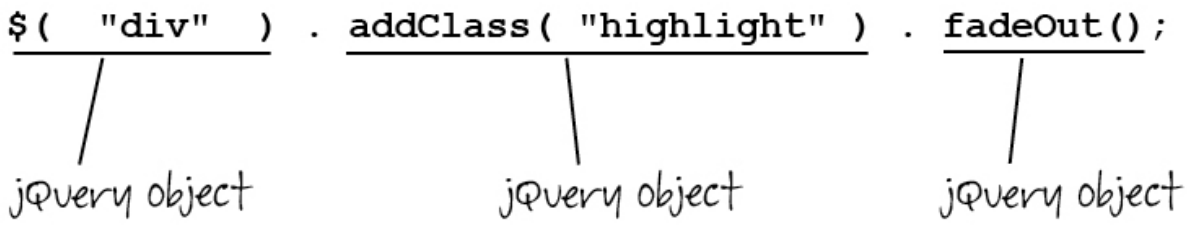
In our jQuery code, we've used the **$** function primarily to select elements in our page:

OBSERVE:
`$("#myDiv").text("Hello jQuery!");`

As you know, this selects the with the id "myDiv," and then changes the text content of the to the string "Hello jQuery!". Much of what you do in jQuery will fall into this same pattern:



You use the **$** function to select one or more elements in your page, and then call a jQuery method, like **addClass()** or **text()**, on those elements. Selecting with **$** always returns a jQuery object, and most of the methods you can call on that object also return a jQuery object, which allows you to chain methods together, like this:



The jQuery object that's returned from each part of the statement above contains a collection of DOM elements that have been selected from the document. Sometimes that collection will contain one element, sometimes many, and sometimes perhaps even none. One of the features of jQuery that makes it so great to work with is that most of the jQuery methods you can call on the selection result work with zero, one, or many elements, so you don't have to worry about how many elements matched the selector. So, in the example above, you can call the **addClass()** method, regardless of how many elements are included in the result of the selector.

While using **$()** with a selector (like in the above examples) is probably the most common way you'll use the **$()** function, as you've seen, there are other ways to use **$()**. Look at the (documentation), and you'll see a list of all the different ways you can use this function:

jQuery(selector [, context])

jQuery(element)
jQuery(elementArray)
jQuery(object)
jQuery(jQuery object)
jQuery()
jQuery(html [, ownerDocument])
jQuery(html, attributes)
jQuery(callback)

We've touched on most of these uses throughout this course, as we've used jQuery to select elements, create elements, wrap objects, and so on.

$() and the ready Function

One of the ways we've used **$()** is to wrap DOM objects, turning them into jQuery objects so we can then call jQuery methods on those objects. You've seen this happen often in callback functions, where **this** represents the currently selected element (a DOM object), and we wrap **this** to create a jQuery object, **$(this)**. Modify your **jquery.html** as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery</title>
  <meta charset="utf-8">
  <style>
     .highlight {
         background-color: lightyellow;
     }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>
$(document).ready(function() {
    $("#myDiv").text("Hello jQuery!").click(function() {
        $(this).toggleClass("highlight");
    });
});
  </script>
</head>
<body>
  <div id="myDiv"></div>
</body>
</html>
```

 Save the file and 

preview. When you click on the "myDiv" in your page, the background toggles between yellow and plain. **$(this)** refers to the element you click.

So this is an example where we've used the **$()** to wrap a DOM object so we can then use jQuery methods on it—in this case, the **toggleClass()** method.

Look at the beginning of the jQuery code though; notice that we wrap another object there:

OBSERVE:

```
$(document).ready(function() {
    $("#myDiv").text("Hello jQuery!").click(function() {
        $(this).toggleClass("highlight");
    });
});
```

Here, we wrap the **document** object, which represents the document, or web page; that is, it is a data

structure containing all the elements in your page. Then we call the **ready()** method on the **wrapped document object**, passing this method a **function** that gets called once the web page has been fully loaded into the browser. This is how your jQuery code runs once the page has loaded.

There is a shortcut for setting up the ready function that you should be familiar with it. You can actually set up the ready function like this:

CODE TO TYPE:

```
$(document).ready(function() {
    $("#myDiv").text("Hello jQuery!").click(function() {
        $(this).toggleClass("highlight");
    });
});
```

 Save **jquery.html** and 

preview; everything looks and works the same—but now we're passing the ready function directly to the **$()** function, instead of calling the **ready()** method.

We don't recommend that you write the ready function like this because it's not quite as clear, but you'll likely run across this shortcut sometime, and now you'll know what it means.

Before we leave the topic of the ready function, you should also know that it's perfectly legal in jQuery to write:

CODE TO TYPE:

```
$(document).ready(function() {
    $("#myDiv").text("Hello jQuery!").click(function() {
        $(this).toggleClass("highlight");
    });
});
$(document).ready(function() {
    $("body").append("<p>I'm here now!</p>");
});
```

 Save **jquery.html** and 

preview; a paragraph has been added to the page, below the . This jQuery program has *two* ready functions! In this example, the first ready function will be called when the page has loaded, and then the second one. In JavaScript you can't really do the equivalent thing—that is, set the **window.onload** property to two different functions—because you'd just be overwriting the previous onload handler with the second function, and only the second function would actually run.

Again, we don't actually recommend that you do this. You might see two (or more!) ready functions defined like this, and the code will work, but it's not considered a best practice and there's no real reason to structure your code this way. Remember, you can always define your own functions, either outside your ready function, or within your ready function, and call those in sequence, like this:

CODE TO TYPE:

```
$(document).ready(function() {
    $("#myDiv").text("Hello jQuery!").click(function() {
        $(this).toggleClass("highlight");
    });
    modifyDiv();
        addPara();
    });
$(document).ready(function() {
    $("body").append("<p>I'm here now!</p>");
});

function modifyDiv() {
    $("#myDiv").text("Hello jQuery!").click(function() {
        $(this).toggleClass("highlight");
    });
}
function addPara() {
    $("body").append("<p>I'm here now!</p>");
}
```

Here, we define two functions (external to the ready function), and call these functions from the ready function in sequence.  Save the file and 

preview; the page looks and works the same way as before.

If you want to hide the functions you use to structure your jQuery code, you can place them inside the ready function itself, like this:

Update your jQuery:

```
$(document).ready(function() {
    modifyDiv();
    addPara();

    function modifyDiv() {
        $("#myDiv").text("Hello jQuery!").click(function() {
            $(this).toggleClass("highlight");
        });
    }
    function addPara() {
        $("body").append("<p>I'm here now!</p>");
    }
});
function modifyDiv() {
    $("#myDiv").text("Hello jQuery!").click(function() {
        $(this).toggleClass("highlight");
    });
}
function addPara() {
    $("body").append("<p>I'm here now!</p>");
}
```

This works in exactly the same way, only now, the functions **modifyDiv()** and **addPara()** are not visible outside your jQuery ready function (which might help to avoid name clashes). This pattern of hiding functions within another function is a fairly common pattern in JavaScript, because JavaScript has no concept of packages or modules, like other languages do. When you define a function in JavaScript, unless you hide it like this, that function is in the *global namespace*, which means it's visible everywhere. Normally this isn't a problem, but if you're writing code that you expect other programmers to include and use with their own, it's useful to hide functions that you're using internally to your own scripts that will never be used outside of your own code. This helps to keep your function names from clashing with anyone else's, and also helps keep your code neat and tidy.

Detecting Features with jQuery.support

One of the benefits of using jQuery is that it hides just about all the browser differences you're likely to run into, unless you're trying some of the most cutting-edge CSS and HTML features. This even includes browser

differences going back to IE 7 and 8 (which many people still use), and even most of the funny quirks (of which there were many!) in IE 6.

However there are a few browser differences that might matter to you if you are using some of the more esoteric features in CSS, HTML, or JavaScript. For these, jQuery provides a **support** object. This object contains many properties that you can test in your code to see if the browser supports a certain feature, like **opacity**, for example. Update your jQuery as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery</title>
  <meta charset="utf-8">
  <style>
     .highlight {
         background-color: lightyellowred;
         opacity: .5;
     }
     .red {
         background-color: red;
     }
  </style>
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>
$(document).ready(function() {
    modifyDiv();
    addPara();

    function modifyDiv() {
        $("#myDiv").text("Hello jQuery!").click(function() {
            $(this).toggleClass("highlight");
        });
    }
    function addPara() {
        $("body").append("<p>I'm here now!</p>");
    }
});
$(document).ready(function() {
    $("#myDiv").text("Hello jQuery!");
    if ($.support.opacity) {
        $("#myDiv").addClass("highlight");
    } else {
        $("#myDiv").addClass("red");
    }
});
  </script>
</head>
<body>
  <div id="myDiv"></div>
</body>
</html>
```

 Save the file and 

preview; if it's supported in the browser, you should see the with a red background that's been faded by 50% using the **opacity** CSS property. If **opacity** is *not* supported by the browser, you see the with a fully opaque red background. (Chances are your browser does support the **opacity** property).

Because jQuery already hides the more common differences between browsers, you won't find too many properties in this object that you'll use often, but it's worth looking through the documentation to see what's available.

Notice that the **support** object is a property of the **$** function, so you access it using the dot notation, but with *no* parentheses after **$**. Also, notice that **support** is an *object*, not a function, so don't use parentheses after **support** either!

Utility Functions

The **$.support** object is an example of the many utilities that jQuery offers. Most of these utilities are functions, rather than objects, and in fact, you've already seen and used a couple of these functions: **$.get()** and **$.getJSON()** (in the Ajax lesson).

In this example, we'll look closer at another of these utility functions: **$.each()**. You've already used **each()** in the context of selection results; for instance, if you have a web page with a list, you can iterate over the selected list like this:

OBSERVE:

```
$("li").each(function(i, el) {
    // do something with each list item
});
```

Recall that the function you pass to **each()** has two parameters: the index of the element in the selection results array, and the element at that index. As we saw in a previous lesson, using **each()** is similar to doing a **for** loop.

$.each() works in basically the same way, except that you can apply it to any array, not just to selection results. So, if you have an array of data in your code, you can use **$.each()** whenever you need to iterate over that array and apply a function to each item in the array. Create a new file and add this HTML and jQuery:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery Utilities</title>
  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script>
$(document).ready(function() {
    var temps = [50, 54, 48, 46, 46];
    $.each(temps, convertToC);

    function convertToC(i, f) {
        var c = Math.floor(((f - 32) * 5)/9);
        $("<li>Day " + (i+1) + ": " + c + "</li>").appendTo("ul");
    };
});
  </script>
</head>
<body>
  <ul>
  </ul>
</body>
</html>
```

 Save the file in your **/jQuery** folder as **mapUtil.html**, and  preview. You'll see a list of temperatures in Celsius (that have been converted from the Fahrenheit temperatures in the **temps** array).

Here, we use **$.each()** to iterate over the **temps** array, and apply the function **convertToC()** to each item in the array. Just like the **each()** method, the function you pass to **$.each()** has two parameters: the index of the item in the array (**i**), and the item itself (**f**). So, when **convertToC()** is applied to the first item in the array, **i** is set to 0, and **f** is 50, and so on, for the rest of the items. In this example, we convert the number in **f** to Celsius, and then create a new <li> element with content showing the day of the week (from **i**) and the Celsius temperature, and append this new <li> element to the list in the page.

There are many more utility functions like **$.each()** that you can make use of in your jQuery code; check out the full list of utilities on the jQuery API site.

Using the Latest and Greatest Version of jQuery

Throughout this course we've been using version 1.8.3 of jQuery (the most recent version as we wrote the

course). However, as we mentioned at the beginning of the course, jQuery is updated frequently, so by the time you read this, it's likely that there will be a new version of jQuery.

Some developers prefer to pick a version of the library to use and stick with that as they develop a website, and there may not always be a compelling reason to update that version (unless there is a major bug fix or security issue, in which case, it's always a good idea to update).

Others prefer to use the latest and greatest version of jQuery always. There is a small risk in doing this; something in the jQuery library may have been changed that will cause your code to break. In practice, this hardly ever happens (for the most part, the jQuery library is consistent between releases), but it's definitely something to consider.

If you're one of those who always wants to use the latest version, change your code as shown:

CODE TO TYPE:

```
<!doctype html>
<html>
<head>
  <title>jQuery Utilities</title>
  <meta charset="utf-8">
  <script src="http://code.jquery.com/jquery-1.8.3.min.js"></script>
  <script src="http://code.jquery.com/jquery-latest.js"></script>
  <script>
$(document).ready(function() {
    var temps = [50, 54, 48, 46, 46];
    $.each(temps, convertToC);

    function convertToC(i, f) {
        var c = Math.floor(((f - 32) * 5)/9);
        $("<li>Day " + (i+1) + ": " + c + "</li>").appendTo("ul");
    };
});
  </script>
</head>
<body>
  <ul>
  </ul>
</body>
</html>
```

 Save the file and 

preview; everything looks and behaves exactly as before, except now you're linking to the latest version of jQuery. To check, you can load the code at the URL http://code.jquery.com/jquery-latest.js into your browser and look at the version number at the top. As jQuery is updated to a new version, this URL will always load that latest version.

jQuery is a hugely popular JavaScript library, compatible with all the major browsers, that simplifies the task of writing web applications. The library offers a large variety of powerful selectors and methods for manipulating HTML elements, creating and modifying CSS style, handling events, making Ajax calls, and more. It's used on millions of websites, and by companies big and small, precisely because it is so versatile. Even though it's such a useful library, it's not that big, so the overhead of using it (the extra download required by the browser) is well worth it.

We hope you've had fun learning jQuery. You now have a great new skill in your toolbelt for creating web applications. Practice one last time with the final project, and then you'll be ready to start building your own applications! Thanks so much for taking this journey with me.