jQuery Fundamentals

Rebecca Murphey [http://www.rebeccamurphey.com]

jQuery Fundamentals

Rebecca Murphey [http://www.rebeccamurphey.com]

Table of Contents

Chapter 1. Welcome

jQuery is fast becoming a must-have skill for front-end developers. The purpose of this book is to provide an overview of the jQuery JavaScript library; when you're done with the book, you should be able to complete basic tasks using jQuery, and have a solid basis from which to continue your learning. This book was designed as material to be used in a classroom setting, but you may find it useful for individual study.

This is a hands-on class. We will spend a bit of time covering a concept, and then you'll have the chance to work on an exercise related to the concept. Some of the exercises may seem trivial; others may be downright daunting. In either case, there is no grade; the goal is simply to get you comfortable working your way through problems you'll commonly be called upon to solve using jQuery. Example solutions to all of the exercises are included in the sample code.

Getting the Code

The code we'll be using in this book is hosted in a repository on Github [http://github.com/rmurphey/jqfundamentals]. You can download a .zip or .tar file of the code, then uncompress it to use it on your server. If you're git-inclined, you're welcome to clone or fork the repository.

Software

You'll want to have the following tools to make the most of the class:

- The Firefox browser
- The Firebug extension for Firefox
- A plain text editor
- For the Ajax portions: A local server (such as MAMP or WAMP), or an FTP or SSH client to access a remote server.

Adding JavaScript to Your Page

JavaScript can be included inline or by including an external file via a script tag. The order in which you include JavaScript is important: dependencies must be included before the script that depends on them.

For the sake of page performance, JavaScript should be included as close to the end of your HTML as is practical. Multiple JavaScript files should be combined for production use.

Example 1.1. An example of inline Javascript

```
<script>
console.log('hello');
</script>
```

Example 1.2. An example of including external JavaScript

```
<script src='/js/jquery.js'></script>
```

JavaScript Debugging

A debugging tool is essential for JavaScript development. Firefox provides a debugger via the Firebug extension; Safari and Chrome provide built-in consoles.

Each console offers:

- single- and multi-line editors for experimenting with JavaScript
- an inspector for looking at the generated source of your page
- a Network or Resources view, to examine network requests

When you are writing JavaScript code, you can use the following methods to send messages to the console:

- `console.log()` for sending general log messages
- `console.dir()` for logging a browseable object
- `console.warn()` for logging warnings
- `console.error()` for logging error messages

Other console methods are also available, though they may differ from one browser to another. The consoles also provide the ability to set break points and watch expressions in your code for debugging purposes.

Exercises

Most chapters in the book conclude with one or more exercises. For some exercises, you'll be able to work directly in Firebug; for others, you will need to include other scripts after the jQuery script tag as directed in the individual exercises.

In some cases, you will need to consult the jQuery documentation in order to complete an exercise, as we won't have covered all of the relevant information in the book. This is by design; the jQuery library is large, and learning to find answers in the documentation is an important part of the process.

Here are a few suggestions for tackling these problems:

- First, make sure you thoroughly understand the problem you're being asked to solve.
- Next, figure out which elements you'll need to access in order to solve the problem, and determine how you'll get those elements. Use Firebug to verify that you're getting the elements you're after.
- Finally, figure out what you need to do with the elements to solve the problem. It can be helpful to write comments explaining what you're going to do before you try to write the code to do it.

Do not be afraid to make mistakes! Do not try to make your code perfect on the first try! Making mistakes and experimenting with solutions is part of learning the library, and you'll be a better developer for it. Examples of solutions for these exercises are located in the `/solutions` directory in the sample code.

Conventions used in this book

Methods that can be called on jQuery objects will be referred to as `$.fn.methodName`. Methods that exist in the jQuery namespace but that cannot be called on jQuery objects will be referred to as

`$.methodName`. If this doesn't mean much to you, don't worry — it should become clearer as you progress through the book.

Example 1.3. Example of an example

```
// code examples will appear like this
```

Remarks will appear like this.

> **Note**
>
> Notes about a topic will appear like this.

Reference Material

There are any number of articles and blog posts out there that address some aspect of jQuery. Some are phenomenal; some are downright wrong. When you read an article about jQuery, be sure it's talking about the same version as you're using, and resist the urge to just copy and paste — take the time to understand the code in the article.

Here are some excellent resources to use during your jQuery learning. The most important of all is the jQuery source itself: it contains, in code form, complete documentation of the library. It is not a black box — your understanding of the library will grow exponentially if you spend some time visiting it now and again — and I highly recommend bookmarking it in your browser and referring to it often.

- The jQuery source [http://ajax.googleapis.com/ajax/libs/jquery/1/jquery.js]
- jQuery documentation [http://api.jquery.com]
- jQuery forum [http://forum.jquery.com/]
- Delicious bookmarks [http://delicious.com/rdmey/jquery-class]
- #jquery IRC channel on Freenode [http://docs.jquery.com/Discussion#Chat_.2F_IRC_Channel]

Chapter 2. JavaScript Basics

Overview

jQuery is built on top of JavaScript, a rich and expressive language in its own right. This section covers the basic concepts of JavaScript, as well as some frequent pitfalls for people who have not used JavaScript before. While it will be of particular value to people with no programming experience, even people who have used other programming languages may benefit from learning about some of the peculiarities of JavaScript.

If you're interested in learning more about the JavaScript language, I highly recommend *JavaScript: The Good Parts* by Douglas Crockford.

Syntax Basics

Understanding statements, variable naming, whitespace, and other basic JavaScript syntax.

Example 2.1. A simple variable declaration

```
var foo = 'hello world';
```

Example 2.2. Whitespace has no meaning outside of quotation marks

```
var foo =         'hello world';
```

Example 2.3. Parentheses indicate precedence

```
2 * 3 + 5;    // returns 11; multiplication happens first
2 * (3 + 5);  // returns 16; addition happens first
```

Example 2.4. Tabs enhance readability, but have no special meaning

```
var foo = function() {
    console.log('hello');
};
```

Operators

Basic Operators

Basic operators allow you to manipulate values.

Example 2.5. Concatenation

```
var foo = 'hello';
var bar = 'world';

console.log(foo + ' ' + bar); // 'hello world'
```

Example 2.6. Multiplication and division

```
2 * 3;
2 / 3;
```

Example 2.7. Incrementing and decrementing

```
var i = 1;

var j = ++i;  // pre-increment:  j equals 2; i equals 2
var k = i++;  // post-increment: k equals 2; i equals 3
```

Operations on Numbers & Strings

In JavaScript, numbers and strings will occasionally behave in ways you might not expect.

Example 2.8. Addition vs. concatenation

```
var foo = 1;
var bar = '2';

console.log(foo + bar);  // 12. uh oh
```

Example 2.9. Forcing a string to act as a number

```
var foo = 1;
var bar = '2';

// coerce the string to a number
console.log(foo + parseInt(bar));
```

Logical Operators

Logical operators allow you to evaluate a series of comparators using AND and OR operations.

Example 2.10. Logical AND and OR operators

```
var foo = 1;
var bar = 0;
var baz = 2;

foo || bar;   // returns 1, which is true
bar || foo;   // returns 1, which is true

foo && bar;   // returns 0, which is false
foo && baz;   // returns 2, which is true
baz && foo;   // returns 1, which is true
```

Though it may not be clear from Example 2.10, “Logical AND and OR operators”, the `||` operator returns the value of the first truthy comparator, or false if no comparator is truthy. The `&&` operator returns the value of the last false comparator, or the value of the last comparator if all values are truthy. Essentially, both operators return the first value that proves them true or false.

Be sure to consult the section called “Truthy and Falsy Things” for more details on which values evaluate to `true` and which evaluate to `false`.

Note

You'll sometimes see developers use these logical operators for flow control instead of using `if` statements. For example:

```
// do something with foo if foo is truthy
foo && doSomething(foo);

// set bar to baz if baz is truthy;
// otherwise, set it to the return
// value of createBar()
var bar = baz || createBar();
```

This style is quite elegant and pleasantly terse; that said, it can be really hard to read, especially for beginners. I bring it up here so you'll recognize it in code you read, but I don't recommend using it until you're extremely comfortable with what it means and how you can expect it to behave.

Comparison Operators

Comparison operators allow you to test whether values are equivalent or whether values are identical.

Example 2.11. Comparison operators

```
var foo = 1;
var bar = 0;
var baz = '1';
var bim = 2;

foo == bar;   // returns false
foo != bar;   // returns true
foo == baz;   // returns true; careful!

foo === baz;             // returns false
foo !== baz;             // returns true
foo === parseInt(baz);   // returns true

foo > bim;    // returns false
bim > baz;    // returns true
foo <= baz;   // returns true
```

Conditional Code

Sometimes you only want to run a block of code under certain conditions. Flow control — via `if`, `else if`, and `else` blocks — lets you run code only under certain conditions.

Example 2.12. Flow control

```
var foo = true;
var bar = false;

if (bar) {
    // this code will never run
    console.log('hello!');
}

if (bar) {
    // this code won't run
} else if (foo) {
    // this code will run
} else {
    // this code would run if foo and bar were both false
}
```

> **Note**
>
> While curly braces aren't strictly required around single-line `if` statements, using them consistently, even when they aren't strictly required, makes for vastly more readable code.

Be mindful not to define functions with the same name multiple times within separate `if`/`else` blocks, as doing so may not have the expected result.

Truthy and Falsy Things

In order to use flow control successfully, it's important to understand which kinds of values are "truthy" and which kinds of values are "falsy." Sometimes, values that seem like they should evaluate one way actually evaluate another.

Example 2.13. Values that evaluate to `true`

```
'0';
'any string';
[];  // an empty array
{};  // an empty object
1;   // any non-zero number
```

Example 2.14. Values that evaluate to `false`

```
0;
'';  // an empty string
NaN; // JavaScript's "not-a-number" variable
null;
undefined;  // be careful -- undefined can be redefined!
```

Conditional Variable Assignment with The Ternary Operator

Sometimes you want to set a variable to a value depending on some condition. You could use an `if`/`else` statement, but in many cases the ternary operator is more convenient. [Definition: The *ternary operator* tests a condition; if the condition is true, it returns a certain value, otherwise it returns a different value.]

Example 2.15. The ternary operator

```
// set foo to 1 if bar is true;
// otherwise, set foo to 0
var foo = bar ? 1 : 0;
```

While the ternary operator can be used without assigning the return value to a variable, this is generally discouraged.

Switch Statements

Rather than using a series of if/else if/else blocks, sometimes it can be useful to use a switch statement instead. [Definition: *Switch statements* look at the value of a variable or expression, and run different blocks of code depending on the value.]

Example 2.16. A switch statement

```
switch (foo) {

    case 'bar':
        alert('the value was bar -- yay!');
    break;

    case 'baz':
        alert('boo baz :(');
    break;

    default:
        alert('everything else is just ok');
    break;

}
```

Switch statements have somewhat fallen out of favor in JavaScript, because often the same behavior can be accomplished by creating an obect that has more potential for reuse, testing, etc. For example:

```
var stuffToDo = {
    'bar' : function() {
        alert('the value was bar -- yay!');
    },

    'baz' : function() {
        alert('boo baz :(');
    },

    'default' : function() {
        alert('everything else is just ok');
    }
};

if (stuffToDo[foo]) {
    stuffToDo[foo]();
} else {
    stuffToDo['default']();
```

```
}
```

We'll look at objects in greater depth later in this chapter.

Loops

Loops let you run a block of code a certain number of times.

Example 2.17. Loops

```
// logs 'try 1', 'try 2', ..., 'try 5'
for (var i=0; i<5; i++) {
    console.log('try ' + i);
}
```

Note that in Example 2.17, "Loops" we use the keyword var before the variable name `i`. This "scopes" the variable `i` to the loop block. We'll discuss scope in depth later in this chapter.

Reserved Words

JavaScript has a number of "reserved words," or words that have special meaning in the language. You should avoid using these words in your code except when using them with their intended meaning.

- `break`
- `case`
- `catch`
- `continue`
- `default`
- `delete`
- `do`
- `else`
- `finally`
- `for`
- `function`
- `if`
- `in`
- `instanceof`
- `new`
- `return`

- switch
- this
- throw
- try
- typeof
- var
- void
- while
- with
- abstract
- boolean
- byte
- char
- class
- const
- debugger
- double
- enum
- export
- extends
- final
- float
- goto
- implements
- import
- int
- interface
- long
- native

- `package`
- `private`
- `protected`
- `public`
- `short`
- `static`
- `super`
- `synchronized`
- `throws`
- `transient`
- `volatile`

Arrays

Arrays are zero-indexed lists of values. They are a handy way to store a set of related items of the same type (such as strings), though in reality, an array can include multiple types of items, including other arrays.

Example 2.18. A simple array

```
var myArray = [ 'hello', 'world' ];
```

Example 2.19. Accessing array items by index

```
var myArray = [ 'hello', 'world', 'foo', 'bar' ];
console.log(myArray[3]);   // logs 'bar'
```

Example 2.20. Testing the size of an array

```
var myArray = [ 'hello', 'world' ];
console.log(myArray.length);   // logs 2
```

Example 2.21. Changing the value of an array item

```
var myArray = [ 'hello', 'world' ];
myArray[1] = 'changed';
```

While it's possible to change the value of an array item as shown in Example 2.21, “Changing the value of an array item”, it's generally not advised.

Example 2.22. Adding elements to an array

```
var myArray = [ 'hello', 'world' ];
myArray.push('new');
```

Example 2.23. Working with arrays

```
var myArray = [ 'h', 'e', 'l', 'l', 'o' ];
var myString = myArray.join('');   // 'hello'
var mySplit = myString.split('');  // [ 'h', 'e', 'l', 'l', 'o' ]
```

Objects

Objects contain one or more key-value pairs. The key portion can be any string; if the string contains spaces or is a reserved word, then it must be quoted. The value portion can be any type of value: a number, a string, an array, a function, or even another object.

[Definition: When one of these values is a function, it's called a *method* of the object.] Otherwise, they are called properties.

As it turns out, nearly everything in JavaScript is an object — arrays, functions, numbers, even strings — and they all have properties and methods.

Example 2.24. Creating an "object literal"

```
var myObject = {
    sayHello : function() {
        console.log('hello');
    },

    myName : 'Rebecca'
};

myObject.sayHello();            // logs 'hello'
console.log(myObject.myName);   // logs 'Rebecca'
```

Object literals can be extremely useful for code organization; for more information, read Using Objects to Organize Your Code [http://blog.rebeccamurphey.com/2009/10/15/using-objects-to-organize-your-code/] by Rebecca Murphey.

Functions

Functions contain blocks of code that need to be executed repeatedly. Functions can take zero or more arguments, and can optionally return a value.

Functions can be created in a variety of ways:

Example 2.25. Function Declaration

```
function foo() { /* do something */ }
```

Example 2.26. Named Function Expression

```
var foo = function() { /* do something */ }
```

I prefer the named function expression method of setting a function's name, for some rather in-depth and technical reasons [http://yura.thinkweb2.com/named-function-expressions/]. You are likely to see both methods used in others' JavaScript code.

Using Functions

Example 2.27. A simple function

```
var greet = function(person, greeting) {
    var text = greeting + ', ' + person;
    console.log(text);
};


greet('Rebecca', 'Hello');
```

Example 2.28. A function that returns a value

```
var greet = function(person, greeting) {
    var text = greeting + ', ' + person;
    return text;
};

console.log(greet('Rebecca','hello'));
```

Example 2.29. A function that returns another function

```
var greet = function(person, greeting) {
    var text = greeting + ', ' + person;
    return function() { console.log(text); };
};


var greeting = greet('Rebecca', 'Hello');
greeting();
```

Self-Executing Anonymous Functions

A common pattern in JavaScript is the self-executing anonymous function. This pattern creates a function expression and then immediately executes the function. This pattern is extremely useful for cases where you want to avoid polluting the global namespace with your code -- no variables declared inside of the function are visible outside of it.

Example 2.30. A self-executing anonymous function

```
(function(){
    var foo = 'Hello world';
})();


console.log(foo);   // undefined!
```

Functions as Arguments

In JavaScript, functions are "first-class citizens" -- they can be assigned to variables or passed to other functions as arguments. Passing functions as arguments is an extremely common idiom in jQuery.

Example 2.31. Passing an anonymous function as an argument

```
var myFn = function(fn) {
    var result = fn();
    console.log(result);
};

myFn(function() { return 'hello world'; });   // logs 'hello world'
```

Example 2.32. Passing a named function as an argument

```
var myFn = function(fn) {
    var result = fn();
    console.log(result);
};

var myOtherFn = function() {
    return 'hello world';
};

myFn(myOtherFn);   // logs 'hello world'
```

Testing Type

JavaScript offers a way to test the "type" of a variable. However, the result can be confusing -- for example, the type of an Array is "object".

Example 2.33. Testing the type of various variables

```
var myFunction = function() {
    console.log('hello');
};

var myObject = {
    foo : 'bar'
};

var myArray = [ 'a', 'b', 'c' ];

var myString = 'hello';

var myNumber = 3;

typeof(myFunction);   // returns 'function'
typeof(myObject);     // returns 'object'
typeof(myArray);      // returns 'object' -- careful!
typeof(myString);     // returns 'string';
typeof(myNumber);     // returns 'number'

if (myArray.push && myArray.slice && myArray.join) {
    // probably an array
    // (this is called "duck typing")
}
```

jQuery offers utility methods to help you determine whether

Scope

"Scope" refers to the variables that are available to a piece of code at a given time. A lack of understanding of scope can lead to frustrating debugging experiences.

When a variable is declared inside of a function using the `var` keyword, it is only available to code inside of that function -- code outside of that function cannot access the variable. On the other hand, functions defined *inside* that function *will* have access to to the declared variable.

Furthermore, variables that are declared inside a function without the `var` keyword are not local to the function -- JavaScript will traverse the scope chain all the way up to the window scope to find where the variable was previously defined. If the variable wasn't previously defined, it will be defined in the global scope, which can have extremely unexpected consequences;

Example 2.34. Functions have access to variables defined in the same scope

```
var foo = 'hello';

var sayHello = function() {
    console.log(foo);
};

sayHello();         // logs 'hello'
console.log(foo);   // also logs 'hello'
```

Example 2.35. Code outside the scope in which a variable was defined does not have access to the variable

```
var sayHello = function() {
    var foo = 'hello';
    console.log(foo);
};

sayHello();         // logs 'hello'
console.log(foo);   // doesn't log anything
```

Example 2.36. Variables with the same name can exist in different scopes with different values

```
var foo = 'world';

var sayHello = function() {
    var foo = 'hello';
    console.log(foo);
};

sayHello();         // logs 'hello'
console.log(foo);   // logs 'world'
```

Example 2.37. Functions can "see" changes in variable values after the function is defined

```
var myFunction = function() {
    var foo = 'hello';

    var myFn = function() {
        console.log(foo);
    };

    foo = 'world';

    return myFn;
};

var f = myFunction();
f();  // logs 'world' -- uh oh
```

Example 2.38. Scope insanity

```
// a self-executing anonymous function
(function() {
    var baz = 1;
    var bim = function() { alert(baz); };
    bar = function() { alert(baz); };
})();

console.log(baz);  // baz is not defined outside of the function

bar();  // bar is defined outside of the anonymous function
        // because it wasn't declared with var; furthermore,
        // because it was defined in the same scope as baz,
        // it has access to baz even though other code
        // outside of the function does not

bim();  // bim is not defined outside of the anonymous function,
        // so this will result in an error
```

Closures

Closures are an extension of the concept of scope — functions have access to variables that were available in the scope where the function was created. If that’s confusing, don’t worry: closures are generally best understood by example.

In Example 2.37, “Functions can "see" changes in variable values after the function is defined” we saw how functions have access to changing variable values. The same sort of behavior exists with functions defined within loops -- the function "sees" the change in the variable's value even after the function is defined, resulting in all clicks alerting 4.

Example 2.39. How to lock in the value of `i`?

```
/* this won't behave as we want it to; */
/* every click will alert 4 */
for (var i=0; i<5; i++) {
    $('<p>click me</p>').appendTo('body').click(function() {
        alert(i);
    });
}
```

Example 2.40. Locking in the value of `i` with a closure

```
/* fix: “close” the value of i inside createFunction, so it won't change */
var createFunction = function(i) {
    return function() { alert(i); };
};

for (var i=0; i<5; i++) {
    $('p').appendTo('body').click(createFunction(i));
}
```

Chapter 3. jQuery Basics

$(document).ready()

You cannot safely manipulate a page until the document is "ready." jQuery detects this state of readiness for you; code included inside `$(document).ready()` will only run once the page is ready for JavaScript code to execute.

Example 3.1. A $(document).ready() block

```
$(document).ready(function() {
    console.log('ready!');
});
```

There is a shorthand for `$(document).ready()` that you will sometimes see; however, I recommend against using it if you are writing code that people who aren't experienced with jQuery may see.

Example 3.2. Shorthand for $(document).ready()

```
$(function() {
    console.log('ready!');
});
```

You can also passed a named function to `$(document).ready()` instead of passing an anonymous function.

Example 3.3. Passing a named function instead of an anonymous function

```
var readyFn = function() {
    // code to run when the document is ready
};


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

Selecting Elements

The most basic concept of jQuery is to "select some elements and do something with them." jQuery supports most CSS3 selectors, as well as some non-standard selectors. For a complete selector reference, visit http://api.jquery.com/category/selectors/.

Following are a few examples of common selection techniques.

Example 3.4. Selecting elements by ID

```
$('#myId'); // note IDs must be unique per page
```

Example 3.5. Selecting elements by class name

```
$('div.myClass'); // performance improves if you specify element type
```

Example 3.6. Selecting elements by attribute

```
$('input[name=first_name]'); // beware, this can be very slow
```

Example 3.7. Selecting elements by compound CSS selector

```
$('#contents ul.people li');
```

Example 3.8. Pseudo-selectors

```
$('a.external:first');
$('tr:odd');
$('#myForm :input');   // select all input-like elements in a form
$('div:visible');
$('div:gt(2)');        // all except the first three divs
$('div:animated');     // all currently animated divs
```

Choosing Selectors

Choosing good selectors is one way to improve the performance of your JavaScript. A little specificity — for example, including an element type such as `div` when selecting elements by class name — can go a long way. Generally, any time you can give jQuery a hint about where it might expect to find what you're looking for, you should. On the other hand, too much specificity can be a bad thing. A selector such as `#myTable thead tr th.special` is overkill if a selector such as `#myTable th.special` will get you what you want.

jQuery offers many attribute-based selectors, allowing you to make selections based on the content of arbitrary attributes using simplified regular expressions.

```
// find all <a>s whose rel attribute
// ends with "thinger"
$("a[rel$='thinger']");
```

While these can be useful in a pinch, they can also be extremely slow — I once wrote an attribute-based selector that locked up my page for multiple seconds. Wherever possible, make your selections using IDs, class names, and tag names.

Want to know more? Paul Irish has a great presentation about improving performance in JavaScript [http://paulirish.com/perf], with several slides focused specifically on selector performance.

Does My Selection Contain Any Elements?

Once you've made a selection, you'll often want to know whether you have anything to work with. You may be inclined to try something like:

```
if ($('div.foo')) { ... }
```

This won't work. When you make a selection using `$()`, an object is always returned, and objects always evaluate to `true`. Even if your selection doesn't contain any elements, the code inside the `if` statement will still run.

Instead, you need to test the selection's length property, which tells you how many elements were selected. If the answer is 0, the length property will evaluate to false when used as a boolean value.

Example 3.9. Testing whether a selection contains elements

```
if ($('div.foo').length) { ... }
```

Saving Selections

Every time you make a selection, a lot of code runs, and jQuery doesn't do caching of selections for you. If you've made a selection that you might need to make again, you should save the selection in a variable rather than making the selection repeatedly.

Example 3.10. Storing selections in a variable

```
var $divs = $('div');
```

Note

In Example 3.10, “Storing selections in a variable”, the variable name begins with a dollar sign. Unlike in other languages, there's nothing special about the dollar sign in JavaScript -- it's just another character. We use it here to indicate that the variable contains a jQuery object. This practice -- a sort of Hungarian notation [http://en.wikipedia.org/wiki/Hungarian_notation] -- is merely convention, and is not mandatory.

Once you've stored your selection, you can call jQuery methods on the variable you stored it in just like you would have called them on the original selection.

Note

A selection only fetches the elements that are on the page when you make the selection. If you add elements to the page later, you'll have to repeat the selection or otherwise add them to the selection stored in the variable. Stored selections don't magically update when the DOM changes.

Refining & Filtering Selections

Sometimes you have a selection that contains more than what you're after; in this case, you may want to refine your selection. jQuery offers several methods for zeroing in on exactly what you're after.

Example 3.11. Refining selections

```
$('div.foo').has('p');          // div.foo elements that contain <p>'s
$('h1').not('.bar');            // h1 elements that don't have a class of bar
$('ul li').filter('.current');  // unordered list items with class of current
$('ul li').first();             // just the first unordered list item
$('ul li').eq(5);               // the fifth
```

Form-Related Selectors

jQuery offers several pseudo-selectors that help you find elements in your forms; these are especially helpful because it can be difficult to distinguish between form elements based on their state or type using standard CSS selectors.

:button Selects `<button>` elements and elements with `type="button"`

:checkbox Selects inputs with `type="checkbox"`

:checked	Selects checked inputs
:disabled	Selects disabled form elements
:enabled	Selects enabled form elements
:file	Selects inputs with `type="file"`
:image	Selects inputs with `type="image"`
:input	Selects `<input>`, `<textarea>`, and `<select>` elements
:password	Selects inputs with `type="password"`
:radio	Selects inputs with `type="radio"`
:reset	Selects inputs with `type="reset"`
:selected	Selects options that are selected
:submit	Selects inputs with `type="submit"`
:text	Selects inputs with `type="text"`

Example 3.12. Using form-related pseduo-selectors

```
$("#myForm :input'); // get all elements that accept input
```

Working with Selections

Once you have a selection, you can call methods on the selection. Methods generally come in two different flavors: getters and setters. Getters return a property of the first selected element; setters set a property on all selected elements.

Chaining

If you call a method on a selection and that method returns a jQuery object, you can continue to call jQuery methods on the object without pausing for a semicolon.

Example 3.13. Chaining

```
$('#content').find('h3').eq(2).html('new text for the third h3!');
```

If you are writing a chain that includes several steps, you (and the person who comes after you) may find your code more readable if you break the chain over several lines.

Example 3.14. Formatting chained code

```
$('#content')
    .find('h3')
    .eq(2)
    .html('new text for the third h3!');
```

If you change your selection in the midst of a chain, jQuery provides the `$.fn.end` method to get you back to your original selection.

Example 3.15. Restoring your original selection using `$.fn.end`

```
$('#content')
    .find('h3')
    .eq(2)
        .html('new text for the third h3!')
    .end() // restores the selection to all h3's in #content
    .eq(0)
        .html('new text for the first h3!');
```

Note

Chaining is extraordinarily powerful, and it's a feature that many libraries have adapted since it was made popular by jQuery. However, it must be used with care. Extensive chaining can make code extremely difficult to modify or debug. There is no hard-and-fast rule to how long a chain should be -- just know that it is easy to get carried away.

Getters & Setters

jQuery “overloads” its methods, so the method used to set a value generally has the same name as the method used to get a value. [Definition: When a method is used to set a value, it is called a *setter*]. [Definition: When a method is used to get (or read) a value, it is called a *getter*]. Setters affect all elements in a selection; getters get the requested value only for the first element in the selection.

Example 3.16. The `$.fn.html` method used as a setter

```
$('h1').html('hello world');
```

Example 3.17. The html method used as a getter

```
$('h1').html();
```

Setters return a jQuery object, allowing you to continue to call jQuery methods on your selection; getters return whatever they were asked to get, meaning you cannot continue to call jQuery methods on the value returned by the getter.

CSS, Styling, & Dimensions

jQuery includes a handy way to get and set CSS properties of elements.

Note

CSS properties that normally include a hyphen need to be *camel cased* in JavaScript. For example, the CSS property `font-size` is expressed as `fontSize` in JavaScript.

Example 3.18. Getting CSS properties

```
$('h1').css('fontSize'); // returns a string such as "19px"
```

Example 3.19. Setting CSS properties

```
$('h1').css('fontSize', '100px'); // setting an individual property
$('h1').css({ 'fontSize' : '100px', 'color' : 'red' }); // setting multiple proper
```

Note the style of the argument we use on the second line -- it is an object that contains multiple properties. This is a common way to pass multiple arguments to a function, and many jQuery setter methods accept objects to set mulitple values at once.

Using CSS Classes for Styling

As a getter, the `$.fn.css` method is valuable; however, it should generally be avoided as a setter in production-ready code, because you don't want presentational information in your JavaScript. Instead, write CSS rules for classes that describe the various visual states, and then simply change the class on the element you want to affect.

Example 3.20. Working with classes

```
var $h1 = $('h1');

$h1.addClass('big');
$h1.removeClass('big');
$h1.toggleClass('big');

if ($h1.hasClass('big')) { ... }
```

Classes can also be useful for storing state information about an element, such as indicating that an element is selected.

Dimensions

jQuery offers a variety of methods for obtaining and modifying dimenion and position information about an element.

The code in Example 3.21, “Basic dimensions methods” is just a very brief overview of the dimensions functionality in jQuery; for complete details about jQuery dimension methods, visit http://api.jquery.com/category/dimensions/.

Example 3.21. Basic dimensions methods

```
$('h1').width('50px');   // sets the width of all H1 elements
$('h1').width();         // gets the width of the first H1

$('h1').height('50px');  // sets the height of all H1 elements
$('h1').height();        // gets the height of the first H1

$('h1').position();      // returns an object containing position
                         // information for the first H1 relative to
                         // its "offset (positioned) parent"
```

Attributes

An element's attributes can contain useful information for your application, so it's important to be able to get and set them.

The `$.fn.attr` method acts as both a getter and a setter. As with the `$.fn.css` method, `$.fn.attr` as a setter can accept either a key and a value, or an object containing one or more key/value pairs.

Example 3.22. Setting attributes

```
$('a').attr('href', 'allMyHrefsAreTheSameNow.html');
$('a').attr({
    'title' : 'all titles are the same too!',
    'href' : 'somethingNew.html'
});
```

This time, we broke the object up into multiple lines. Remember, whitespace doesn't matter in JavaScript, so you should feel free to use it liberally to make your code more legible! You can use a minification tool later to strip out unnecessary whitespace for production.

Example 3.23. Getting attributes

```
$('a').attr('href');  // returns the href for the first a element in the document
```

Traversing

Once you have a jQuery selection, you can find other elements using your selection as a starting point.

For complete documentation of jQuery traversal methods, visit http://api.jquery.com/category/traversing/.

> ## Note
>
> Be cautious with traversing long distances in your documents -- complex traversal makes it imperative that your document's structure remain the same, something that's difficult to guarantee even if you're the one creating the whole application from server to client. One- or two-step traversal is fine, but you generally want to avoid traversals that take you from one container to another.

Example 3.24. Moving around the DOM using traversal methods

```
$('h1').next('p');
$('div:visible').parent();
$('input[name=first_name]').closest('form');
$('#myList').children();
$('li.selected').siblings();
```

You can also iterate over a selection using `$.fn.each`. This method iterates over all of the elements in a selection, and runs a function for each one. The function receives the index of the current element and the DOM element itself as arguments. Inside the function, the DOM element is also available as `this` by default.

Example 3.25. Iterating over a selection

```
$('#myList li').each(function(idx, el) {
    console.log(
        'Element ' + idx +
        'has the following html: ' +
        $(el).html()
    );
});
```

Manipulating Elements

Once you've made a selection, the fun begins. You can change, move, remove, and clone elements. You can also create new elements via a simple syntax.

For complete documentation of jQuery manipulation methods, visit http://api.jquery.com/category/manipulation/.

Getting and Setting Information about Elements

There are any number of ways you can change an existing element. Among the most common tasks you'll perform is changing the inner HTML or attribute of an element. jQuery offers simple, cross-browser methods for these sorts of manipulations. You can also get information about elements using many of the same methods in their getter incarnations. We'll see examples of these throughout this section, but specifically, here are a few methods you can use to get and set information about elements.

Note

Changing things about elements is trivial, but remember that the change will affect *all* elements in the selection, so if you just want to change one element, be sure to specify that in your selection before calling a setter method.

Note

When methods act as getters, they generally only work on the first element in the selection, and they do not return a jQuery object, so you can't chain additional methods to them. One notable exception is `$.fn.text`; as mentioned below, it gets the text for all elements in the selection.

$.fn.html	Get or set the html contents.
$.fn.text	Get or set the text contents; HTML will be stripped.
$.fn.attr	Get or set the value of the provided attribute.
$.fn.width	Get or set the width in pixels of the first element in the selection as an integer.
$.fn.height	Get or set the height in pixels of the first element in the selection as an integer.
$.fn.position	Get an object with position information for the first element in the selection, relative to its first positioned ancestor. *This is a getter only.*
$.fn.val	Get or set the value of form elements.

Example 3.26. Changing the HTML of an element

```
$('#myDiv p:first')
    .html('New <strong>first</strong> paragraph!');
```

Moving, Copying, and Removing Elements

There are a variety of ways to move elements around the DOM; generally, there are two approaches:

- Place the selected element(s) relative to another element
- Place an element relative to the selected element(s)

For example, jQuery provides `$.fn.insertAfter` and `$.fn.after`. The `$.fn.insertAfter` method places the selected element(s) after the element that you provide as an argument; the `$.fn.after` method places the element provided as an argument after the selected element. Several other methods follow this pattern: `$.fn.insertBefore` and `$.fn.before`; `$.fn.appendTo` and `$.fn.append`; and `$.fn.prependTo` and `$.fn.prepend`.

The method that makes the most sense for you will depend on what elements you already have selected, and whether you will need to store a reference to the elements you're adding to the page. If you need to store a reference, you will always want to take the first approach -- placing the selected elements relative to another element -- as it returns the element(s) you're placing. In this case, `$.fn.insertAfter`, `$.fn.insertBefore`, `$.fn.appendTo`, and `$.fn.prependTo` will be your tools of choice.

Example 3.27. Moving elements using different approaches

```
// make the first list item the last list item
var $li = $('#myList li:first').appendTo('#myList');

// another approach to the same problem
$('#myList').append($('#myList li:first'));

// note that there's no way to access the
// list item that we moved, as this returns
// the list itself
```

Cloning Elements

When you use methods such as $.fn.appendTo, you are moving the element; sometimes you want to make a copy of the element instead. In this case, you'll need to use $.fn.clone first.

Example 3.28. Making a copy of an element

```
// copy the first list item to the end of the list
$('#myList li:first').clone().appendTo('#myList');
```

Note

If you need to copy related data and events, be sure to pass `true` as an argument to `$.fn.clone`.

Removing Elements

There are two ways to remove elements from the page: `$.fn.remove` and `$.fn.detach`. You'll use `$.fn.remove` when you want to permanently remove the selection from the page; while the method does return the removed element(s), those elements will not have their associated data and events attached to them if you return them to the page.

If you need the data and events to persist, you'll want to use `$.fn.detach` instead. Like `$.fn.remove`, it returns the selection, but it also maintains the data and events associated with the selection, so you can restore the selection to the page at a later time.

Note

The `$.fn.detach` method is extremely valuable if you are doing heavy manipulation to an element. In that case, it's beneficial to `$.fn.detach` the element from the page, work on it

in your code, and then restore it to the page when you're done. This saves you from expensive "DOM touches" while maintaining the element's data and events.

If you want to leave the element on the page but simply want to remove its contents, you can use `$.fn.empty` to dispose of the element's inner HTML.

Creating New Elements

jQuery offers a trivial and elegant way to create new elements using the same `$()` method you use to make selections.

Example 3.29. Creating new elements

```
$('<p>This is a new paragraph</p>');
$('<li class="new">new list item</p>');
```

Example 3.30. Creating a new element with an attribute object

```
$('<a/>', {
    html : 'This is a <strong>new</strong> link',
    'class' : 'new',
    href : 'foo.html'
});
```

Note that in the attributes object we included as the second argument, the property name class is quoted, while the property names text and href are not. Property names generally do not need to be quoted unless they are reserved words (as class is in this case).

When you create a new element, it is not immediately added to the page. There are several ways to add an element to the page once it's been created.

Example 3.31. Getting a new element on to the page

```
var $myNewElement = $('<p>New element</p>');
$myNewElement.appendTo('#content');

$myNewElement.insertAfter('ul:last'); // this will remove the p from #content!
$('ul').last().after($myNewElement.clone());  // clone the p so now we have 2
```

Strictly speaking, you don't have to store the created element in a variable -- you could just call the method to add the element to the page directly after the $(). However, most of the time you will want a reference to the element you added, so you don't need to select it later.

You can even create an element as you're adding it to the page, but note that in this case you don't get a reference to the newly created element.

Example 3.32. Creating and adding an element to the page at the same time

```
$('ul').append('<li>list item</li>');
```

Note

The syntax for adding new elements to the page is so easy, it's tempting to forget that there's a huge performance cost for adding to the DOM repeatedly. If you are adding many elements to the same container, you'll want to concatenate all the html into a single string, and then append

that string to the container instead of appending the elements one at a time. You can use an array to gather all the pieces together, then `join` them into a single string for appending.

```
var myItems = [], $myList = $('#myList');

for (var i=0; i<100; i++) {
    myItems.push('<li>item ' + i + '</li>');
}

$myList.append(myItems.join(''));
```

Manipulating Attributes

jQuery's attribute manipulation capabilities are extensive. Basic changes are simple, but the $.fn.attr method also allows for more complex manipulations.

Example 3.33. Manipulating a single attribute

```
$('#myDiv a:first').attr('href', 'newDestination.html');
```

Example 3.34. Manipulating multiple attributes

```
$('#myDiv a:first').attr({
    href : 'newDestination.html',
    rel : 'super-special'
});
```

Example 3.35. Using a function to determine an attribute's new value

```
$('#myDiv a:first').attr({
    rel : 'super-special',
    href : function() {
        return '/new/' + $(this).attr('href');
    }
});

$('#myDiv a:first').attr('href', function() {
    return '/new/' + $(this).attr('href');
});
```

Exercises

Selecting

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/sandbox.js` or work in Firebug to accomplish the following:

1. Select all of the div elements that have a class of "module".
2. Come up with three selectors that you could use to get the third item in the #myList unordered list. Which is the best to use? Why?
3. Select the label for the search input using an attribute selector.

4. Figure out how many elements on the page are hidden (hint: .length).
5. Figure out how many image elements on the page have an alt attribute.
6. Select all of the odd table rows in the table body.

Traversing

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/sandbox.js` or work in Firebug to accomplish the following:

1. Select all of the image elements on the page; log each image's alt attribute.
2. Select the search input text box, then traverse up to the form and add a class to the form.
3. Select the list item inside #myList that has a class of "current" and remove that class from it; add a class of "current" to the next list item.
4. Select the select element inside #specials; traverse your way to the submit button.
5. Select the first list item in the #slideshow element; add the class "current" to it, and then add a class of "disabled" to its sibling elements.

Manipulating

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/sandbox.js` or work in Firebug to accomplish the following:

1. Add five new list items to the end of the unordered list #myList. Hint:

   ```
   for (var i = 0; i<5; i++) { ... }
   ```

2. Remove the odd list items
3. Add another h2 and another paragraph to the last div.module
4. Add another option to the select element; give the option the value "Wednesday"
5. Add a new div.module to the page after the last one; put a copy of one of the existing images inside of it.

Chapter 4. jQuery Core

$ vs $()

Until now, we've been dealing entirely with methods that are called on a jQuery object. For example:

```
$('h1').remove();
```

Most jQuery methods are called on jQuery objects as shown above; these methods are said to be part of the `$.fn` namespace, or the "jQuery prototype," and are best thought of as jQuery object methods.

However, there are several methods that do not act on a selection; these methods are said to be part of the jQuery namespace, and are best thought of as core jQuery methods.

This distinction can be incredibly confusing to new jQuery users. Here's what you need to remember:

- Methods called on jQuery selections are in the `$.fn` namespace, and automatically receive and return the selection as this.
- Methods in the $ namespace are generally utility-type methods, and do not work with selections; they are not automatically passed any arguments, and their return value will vary.

There are a few cases where object methods and core methods have the same names, such as `$.each` and `$.fn.each`. In these cases, be extremely careful when reading the documentation that you are exploring the correct method.

Utility Methods

jQuery offers several utility methods in the $ namespace. These methods are helpful for accomplishing routine programming tasks. Below are examples of a few of the utility methods; for a complete reference on jQuery utility methods, visit http://api.jquery.com/category/utilities/.

$.trim — Removes leading and trailing whitespace.

```
$.trim('    lots of extra whitespace    ');
// returns 'lots of extra whitespace'
```

$.each — Iterates over arrays and objects.

```
$.each([ 'foo', 'bar', 'baz' ], function(idx, val) {
    console.log('element ' + idx + 'is ' + val);
});

$.each({ foo : 'bar', baz : 'bim' }, function(k, v) {
    console.log(k + ' : ' + v);
});
```

> **Note**
>
> There is also a method `$.fn.each`, which is used for iterating over a selection of elements.

$.inArray — Returns a value's index in an array, or -1 if the value is not in the array.

```
var myArray = [ 1, 2, 3, 5 ];
```

```
if ($.inArray(4, myArray) !== -1) {
    console.log('found it!');
}
```

$.extend Changes the properties of the first object using the properties of subsequent objects.

```
var firstObject = { foo : 'bar', a : 'b' };
var secondObject = { foo : 'baz' };

var newObject = $.extend(firstObject, secondObject);
console.log(firstObject.foo); // 'baz'
console.log(newObject.foo);   // 'baz'
```

If you don't want to change any of the objects you pass to `$.extend`, pass an empty object as the first argument.

```
var firstObject = { foo : 'bar', a : 'b' };
var secondObject = { foo : 'baz' };

var newObject = $.extend({}, firstObject, secondObject);
console.log(firstObject.foo); // 'bar'
console.log(newObject.foo);   // 'baz'
```

Data Methods

As your work with jQuery progresses, you'll find that there's often data about an element that you want to store with the element. In plain JavaScript, you might do this by adding a property to the DOM element, but you'd have to deal with memory leaks in some browsers. jQuery offers a straightforward way to store data related to an element, and it manages the memory issues for you.

Example 4.1. Storing and retrieving data related to an element

```
$('#myDiv').data('keyName', { foo : 'bar' });
$('#myDiv').data('keyName'); // { foo : 'bar' }
```

You can store any kind of data on an element, and it's hard to overstate the importance of this when you get into complex application development. For the purposes of this class, we'll mostly use `$.fn.data` to store references to other elements.

For example, we may want to establish a relationship between a list item and a div that's inside of it. We could establish this relationship every single time we interact with the list item, but a better solution would be to establish the relationship once, and then store a pointer to the div on the list item using `$.fn.data`:

Example 4.2. Storing a relationship between elements using `$.fn.data`

```
$('#myList li').each(function() {
    var $li = $(this), $div = $li.find('div.content');
    $li.data('contentDiv', $div);
});

// later, we don't have to find the div again;
// we can just read it from the list item's data
var $firstLi = $('#myList li:first');
$firstLi.data('contentDiv').html('new content');
```

In addition to passing `$.fn.data` a single key-value pair to store data, you can also pass an object containing one or more pairs.

Feature & Browser Detection

Although jQuery eliminates most JavaScript browser quirks, there are still occasions when your code needs to know about the browser environment.

jQuery offers the `$.support` object, as well as the deprecated `$.browser` object, for this purpose. For complete documentation on these objects, visit http://api.jquery.com/jQuery.support/ and http://api.jquery.com/jQuery.browser/.

The `$.support` object is dedicated to determining what features a browser supports; it is recommended as a more "future-proof" method of customizing your JavaScript for different browser environments.

The `$.browser` object was deprecated in favor of the `$.support` object, but it will not be removed from jQuery anytime soon. It provides direct detection of the browser brand and version.

Avoiding Conflicts with Other Libraries

If you are using another JavaScript library that uses the $ variable, you can run into conflicts with jQuery. In order to avoid these conflicts, you need to put jQuery in no-conflict mode immediately after it is loaded onto the page and before you attempt to use jQuery in your page.

When you put jQuery into no-conflict mode, you have the option of assigning a variable name to replace $.

Example 4.3. Putting jQuery into no-conflict mode

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>var $j = jQuery.noConflict();</script>
```

You can continue to use the standard $ by wrapping your code in a self-executing anonymous function; this is a standard pattern for plugin authoring, where the author cannot know whether another library will have taken over the $.

Example 4.4. Using the $ inside a self-executing anonymous function

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
jQuery.noConflict();

(function($) {
   // your code here, using the $
})(jQuery);
</script>
```

Chapter 5. Events

Overview

jQuery provides simple methods for attaching event handlers to selections. When an event occurs, the provided function is executed. Inside the function, this refers to the element that was clicked.

For details on jQuery events, visit http://api.jquery.com/category/events/.

The event handling function can receive an event object. This object can be used to determine the nature of the event, and to prevent the event's default behavior.

For details on the event object, visit http://api.jquery.com/category/events/event-object/.

Connecting Events to Elements

jQuery offers convenience methods for most common events, and these are the methods you will see used most often. These methods -- including `$.fn.click`, `$.fn.focus`, `$.fn.blur`, `$.fn.change`, etc. -- are shorthand for jQuery's `$.fn.bind` method. The bind method is useful for binding the same hadler function to multiple events, and is also used when you want to provide data to the event hander, or when you are working with custom events.

Example 5.1. Event binding using a convenience method

```
$('p').click(function() {
    console.log('click');
});
```

Example 5.2. Event biding using the `$.fn.bind` method

```
$('p').bind('click', function() {
    console.log('click');
});
```

Example 5.3. Event binding using the `$.fn.bind` method with data

```
$('input').bind(
    'click change',  // bind to multiple events
    { foo : 'bar' }, // pass in data

    function(eventObject) {
        console.log(eventObject.type, eventObject.data);
    }
);
```

Connecting Events to Run Only Once

Sometimes you need a particular handler to run only once -- after that, you may want no handler to run, or you may want a different handler to run. jQuery provides the `$.fn.one` method for this purpose.

Example 5.4. Switching handlers using the `$.fn.one` method

```
$('p').one('click', function() {
    $(this).click(function() { console.log('You clicked this before!'); });
});
```

The `$.fn.one` method is especially useful if you need to do some complicated setup the first time an element is clicked, but not subsequent times.

Disconnecting Events

To disconnect an event handler, you use the `$.fn.unbind` method and pass in the event type to unbind. If you attached a named function to the event, then you can isolate the unbinding to that named function by passing it as the second argument.

Example 5.5. Unbinding all click handlers on a selection

```
$('p').unbind('click');
```

Example 5.6. Unbinding a particular click handler

```
var foo = function() { console.log('foo'); };
var bar = function() { console.log('bar'); };

$('p').bind('click', foo).bind('click', bar);
$('p').unbind('click', bar); // foo is still bound to the click event
```

Namespacing Events

For complex applications and for plugins you share with others, it can be useful to namespace your events so you don't unintentionally disconnect events that you didn't or couldn't know about.

Example 5.7. Namespacing events

```
$('p').bind('click.myNamespace', function() { /* ... */ });
$('p').unbind('click.myNamespace');
$('p').unbind('.myNamespace'); // unbind all events in the namespace
```

Inside the Event Handling Function

As mentioned in the overview, the event handling function receives an event object, which contains many properties and methods. The event object is most commonly used to prevent the default action of the event via the preventDefault method. However, the event object contains a number of other useful properties and methods, including:

pageX, pageY	The mouse position at the time the event occurred, relative to the top left of the page.
type	The type of the event (e.g. "click").
which	The button or key that was pressed.
data	Any data that was passed in when the event was bound.
target	The DOM element that initiated the event.

preventDefault()	Prevent the default action of the event (e.g. following a link).
stopPropagation()	Stop the event from bubbling up to other elements.

In addition to the event object, the event handling function also has access to the DOM element that the handler was bound to via the keyword `this`. To turn the DOM element into a jQuery object that we can use jQuery methods on, we simply do `$(this)`, often following this idiom:

```
var $this = $(this);
```

Example 5.8. Preventing a link from being followed

```
$('a').click(function(e) {
    var $this = $(this);
    if ($this.attr('href').match('evil')) {
        e.preventDefault();
        $this.addClass('evil');
    }
});
```

Triggering Event Handlers

jQuery provides a way to trigger the event handlers bound to an element without any user interaction via the `$.fn.trigger` method. While this method has its uses, it should not be used simply to call a function that was bound as a click handler. Instead, you should store the function you want to call in a variable, and pass the variable name when you do your binding. Then, you can call the function itself whenever you want, without the need for `$.fn.trigger`.

Example 5.9. Triggering an event handler the right way

```
var foo = function(e) {
    if (e) {
        console.log(e);
    } else {
        console.log('this didn\'t come from an event!');
    }
};


$('p').click(foo);

foo(); // instead of $('p').trigger('click')
```

Increasing Performance with Event Delegation

You'll frequently use jQuery to add new elements to the page, and when you do, you may need to bind events to those new elements -- events you already bound to similar elements that were on the page originally. Instead of repeating your event binding every time you add elements to the page, you can use event delegation. With event delegation, you bind your event to a container element, and then when the event occurs, you look to see which contained element it occurred on. If this sounds complicated, luckily jQuery makes it easy with its `$.fn.live` and `$.fn.delegate` methods.

While most people discover event delegation while dealing with elements added to the page later, it has some performance benefits even if you never add more elements to the page. The time required to bind

event handlers to hundreds of individual elements is non-trivial; if you have a large set of elements, you should consider delegating related events to a container element.

Note

The `$.fn.live` method was introduced in jQuery 1.3, and at that time only certain event types were supported. As of jQuery 1.4.2, the `$.fn.delegate` method is available, and is the preferred method.

Example 5.10. Event delegation using `$.fn.delegate`

```
$('#myUnorderedList').delegate('li', 'click', function(e) {
    var $myListItem = $(this);
    // ...
});
```

Example 5.11. Event delegation using `$.fn.live`

```
$('#myUnorderedList li').live('click', function(e) {
    var $myListItem = $(this);
    // ...
});
```

Unbinding Delegated Events

If you need to remove delegated events, you can't simply unbind them. Instead, use `$.fn.undelegate` for events connected with `$.fn.delegate`, and `$.fn.die` for events connected with `$.fn.live`. As with bind, you can optionally pass in the name of the bound function.

Example 5.12. Unbinding delegated events

```
$('#myUnorderedList').undelegate('li', 'click');
$('#myUnorderedList li').die('click');
```

Event Helpers

jQuery offers two event-related helper functions that save you a few keystrokes.

`$.fn.hover`

The `$.fn.hover` method lets you pass one or two functions to be run when the `mouseenter` and `mouseleave` events occur on an element. If you pass one function, it will be run for both events; if you pass two functions, the first will run for `mouseenter`, and the second will run for `mouseleave`.

Note

Prior to jQuery 1.4, the `$.fn.hover` method required two functions.

Example 5.13. The hover helper function

```
$('#menu li').hover(function() {
    $(this).toggleClass('hover');
});
```

`$.fn.toggle`

Much like `$.fn.hover`, the `$.fn.toggle` method receives two or more functions; each time the event occurs, the next function in the list is called. Generally, `$.fn.toggle` is used with just two functions, but technically you can use as many as you'd like.

Example 5.14. The toggle helper function

```
$('p.expander').toggle(
    function() {
        $(this).prev().addClass('open');
    },
    function() {
        $(this).prev().removeClass('open');
    }
);
```

Exercises

Create an Input Hint

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/inputHint.js` or work in Firebug. Your task is to use the text of the label for the search input to create "hint" text for the search input. The steps are as follows:

1. Set the value of the search input to the text of the label element
2. Add a class of "hint" to the search input
3. Remove the label element
4. Bind a focus event to the search input that removes the hint text and the "hint" class
5. Bind a blur event to the search input that restores the hint text and "hint" class if no search text was entered

What other considerations might there be if you were creating this functionality for a real site?

Add Tabbed Navigation

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/tabs.js`. Your task is to create tabbed navigation for the two div.module elements. To accomplish this:

1. Hide all of the modules.
2. Create an unordered list element before the first module.
3. Iterate over the modules using `$.fn.each`. For each module, use the text of the h2 element as the text for a list item that you add to the unordered list element.
4. Bind a click even to the list item that:
    - Shows the related module, and hides any other modules

- Adds a class of "current" to the clicked list item
- Removes the class "current" from the other list item

5. Finally, show the first tab.

Chapter 6. Effects

Overview

jQuery makes it trivial to add simple effects to your page. Effects can use the built-in settings, or provide a customized duration. You can also create custom animations of arbitrary CSS properties.

For complete details on jQuery effects, visit http://api.jquery.com/category/effects/.

Built-in Effects

Frequently used effects are built into jQuery as methods:

$.fn.show	Show the selected element.
$.fn.hide	Hide the selected elements.
$.fn.fadeIn	Animate the opacity of the selected elements to 100%.
$.fn.fadeOut	Animate the opacity of the selected elements to 0%.
$.fn.slideDown	Display the selected elements with a vertical sliding motion.
$.fn.slideUp	Hide the selected elements with a vertical sliding motion.
$.fn.slideToggle	Show or hide the selected elements with a vertical sliding motion, depending on whether the elements are currently visible.

Example 6.1. A basic use of a built-in effect

```
$('h1').show();
```

Changing the Duration of Built-in Effects

With the exception of `$.fn.show` and `$.fn.hide`, all of the built-in methods are animated over the course of 400ms by default. Changing the duration of an effect is simple.

Example 6.2. Setting the duration of an effect

```
$('h1').fadeIn(300);      // fade in over 300ms
$('h1').fadeOut('slow');  // using a built-in speed definition
```

jQuery.fx.speeds

jQuery has an object at `jQuery.fx.speeds` that contains the default speed, as well as settings for `"slow"` and `"fast"`.

```
speeds: {
    slow: 600,
    fast: 200,
    // Default speed
    _default: 400
}
```

It is possible to override or add to this object. For example, you may want to change the default duration of effects, or you may want to create your own effects speed.

Example 6.3. Augmenting `jQuery.fx.speeds` with custom speed definitions

```
jQuery.fx.speeds.blazing = 100;
jQuery.fx.speeds.turtle = 2000;
```

Doing Something when an Effect is Done

Often, you'll want to run some code once an animation is done -- if you run it before the animation is done, it may affect the quality of the animation, or it may remove elements that are part of the animation. [Definition: *Callback functions* provide a way to register your interest in an event that will happen in the future.] In this case, the event we'll be responding to is the conclusion of the animation. Inside of the callback function, the keyword `this` refers to the element that the effect was called on; as we did inside of event handler functions, we can turn it into a jQuery object via `$(this)`.

Example 6.4. Running code when an animation is complete

```
$('div.old').fadeOut(300, function() { $(this).remove(); });
```

Note that if your selection doesn't return any elements, your callback will never run! You can solve this problem by testing whether your selection returned any elements; if not, you can just run the callback immediately.

Example 6.5. Run a callback even if there were no elements to animate

```
var $thing = $('#nonexistent');

var cb = function() {
    console.log('done!');
};

if ($thing.length) {
    $thing.fadeIn(300, cb);
} else {
    cb();
}
```

Custom Effects with `$.fn.animate`

jQuery makes it possible to animate arbitrary CSS properties via the `$.fn.animate` method. The `$.fn.animate` method lets you animate to a set value, or to a value relative to the current value.

Example 6.6. Custom effects with `$.fn.animate`

```
$('div.funtimes').animate(
    {
        left : "+=50",
        opacity : 0.25
    },
    300, // duration
    function() { console.log('done!'); // calback
});
```

Note

Color-related properties cannot be animated with `$.fn.animate` using jQuery out of the box. Color animations can easily be accomplished by including the color plugin [http://plugins.jquery.com/files/jquery.color.js.txt]. We'll discuss using plugins later in the book.

Easing

[Definition: *Easing* describes the manner in which an effect occurs -- whether the rate of change is steady, or varies over the duration of the animation.] jQuery includes only two methods of easing: swing and linear. If you want more natural transitions in your animations, various easing plugins are available.

As of jQuery 1.4, it is possible to do per-property easing when using the $.fn.animate method.

Example 6.7. Per-property easing

```
$('div.funtimes').animate(
    {
        left : [ "+=50", "swing ],
        opacity : [ 0.25, "linear" ]
    },
    300
);
```

For more details on easing options, see http://api.jquery.com/animate/.

Managing Effects

jQuery provides several tools for managing animations.

$.fn.stop	Stop currently running animations on the selected elements.
$.fn.delay	Wait the specified number of milliseconds before running the next animation. `$('h1').show(300).delay(1000).hide(300);`
jQuery.fx.off	If this value is true, there will be no transition for animations; elements will immediately be set to the target final state instead. This can be especially useful when dealing with older browsers; you also may want to provide the option to your users.

Exercises

Reveal Hidden Text

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/blog.js`. Your task is to add some interactivity to the blog section of the page. The spec for the feature is as follows:

- Clicking on a headline in the #blog div should slide down the excerpt paragraph
- Clicking on another headline should slide down its excerpt paragraph, and slide up any other currently showing excerpt paragraphs.

Hint: don't forget about the `:visible` selector!

Create Dropdown Menus

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/navigation.js`. Your task is to add dropdowns to the main navigation at the top of the page.

- Hovering over an item in the main menu should show that item's submenu items, if any.
- Exiting an item should hide any submenu items.

To accomplish this, use the `$.fn.hover` method to add and remove a class from the submenu items to control whether they're visible or hidden. (The file at `/exercises/css/styles.css` includes the "hover" class for this purpose.)

Create a Slideshow

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/slideshow.js`. Your task is to take a plain semantic HTML page and enhance it with JavaScript by adding a slideshow.

1. Move the #slideshow element to the top of the body.
2. Write code to cycle through the list items inside the element; fade one in, display it for a few seconds, then fade it out and fade in the next one.
3. When you get to the end of the list, start again at the beginning.

For an extra challenge, create a navigation area under the slideshow that shows how many images there are and which image you're currently viewing. (Hint: $.fn.prevAll will come in handy for this.)

Chapter 7. Ajax

Overview

The XMLHttpRequest method (XHR) allows browsers communicate with the server without requiring a page reload. This method, also known as Ajax (Asynchronous JavaScript and XML), allows for web pages that provide rich, interactive experiences.

Ajax requests are triggered by JavaScript code; your code sends a request to a URL, and when it receives a response, a callback function can be triggered to handle the response. Because the request is asynchronous, the rest of your code continues to execute while the request is being processed, so it's imperative that a callback be used to handle the response.

jQuery provides Ajax support that abstracts away painful browser differences. It offers both a full-featured `$.ajax()` method, and simple convenience methods such as `$.get()`, `$.getScript()`, `$.getJSON()`, `$.post()`, and `$().load()`.

Most jQuery applications don't in fact use XML, despite the name "Ajax"; instead, they transport data as plain HTML or JSON (JavaScript Object Notation).

In general, Ajax does not work across domains. Exceptions are services that provide JSONP (JSON with Padding) support, which allow limited cross-domain functionality.

Key Concepts

Proper use of Ajax-related jQuery methods requires understanding some key concepts first.

GET vs. Post

The two most common "methods" for sending a request to a server are GET and POST. It's important to understand the proper application of each.

The GET method should be used for non-destructive operations — that is, operations where you are only "getting" data from the server, not changing data on the server. For example, a query to a search service might be a GET request. GET requests may be cached by the browser, which can lead to unpredictable behavior if you are not expecting it. GET requests generally send all of their data in a query string.

The POST method should be used for destructive operations — that is, operations where you are changing data on the server. For example, a user saving a blog post should be a POST request. POST requests are generally not cached by the browser; a query string can be part of the URL, but the data tends to be sent separately as post data.

Data Types

jQuery generally requires some instruction as to the type of data you expect to get back from an Ajax request; in some cases the data type is specified by the method name, and in other cases it is provided as part of a configuration object. There are several options:

text — For transporting simple strings

html — For transporting blocks of HTML to be placed on the page

script — For ading a new script to the page

json For transporting JSON-formatted data, which can include strings, arrays, and objects

> **Note**
>
> As of jQuery 1.4, if the JSON data sent by your server isn't properly formatted, the request may fail silently. See http://json.org for details on properly formatting JSON, but as a general rule, use built-in language methods for generating JSON on the server to avoid syntax issues.

jsonp For transporting JSON data from another domain

xml For transporting data in a custom XML schema

I am a strong proponent of using the JSON format in most cases, as it provides the most flexibility. It is especially useful for sending both HTML and data at the same time.

A is for Asynchronous

The asynchronicity of Ajax catches many new jQuery users off guard. Because Ajax calls are asynchronous by default, the response is not immediately available. Responses can only be handled using a callback. So, for example, the following code will not work:

```
$.get('foo.php');
console.log(response);
```

Instead, we need to pass a callback function to our request; this callback will run when the request succeeds, at which point we can access the data that it returned, if any.

```
$.get('foo.php', function(response) { console.log(response); });
```

Same-Origin Policy and JSONP

In general, Ajax requests are limited to the same protocol (http or https), the same port, and the same domain as the page making the request. This limitation does not apply to scripts that are loaded via jQuery's Ajax methods.

The other exception is requests targeted at a JSONP service on another domain. In the case of JSONP, the provider of the service has agreed to respond to your request with a script that can be loaded into the page using a `<script>` tag, thus avoiding the same-origin limitation; that script will include the data you requested, wrapped in a callback function you provide.

Ajax and Firebug

Firebug (or the Webkit Inspector in Chrome or Safari) is an invaluable tool for working with Ajax requests. You can see Ajax requests as they happen in the Console tab of Firebug (and in the Resources > XHR panel of Webkit Inspector), and you can click on a request to expand it and see details such as the request headers, response headers, response content, and more. If something isn't going as expected with an Ajax request, this is the first place to look to track down what's wrong.

jQuery's Ajax-Related Methods

While jQuery does offer many Ajax-related convenience methods, the core `$.ajax` method is at the heart of all of them, and understanding it is imperative. We'll review it first, and then touch briefly on the convenience methods.

I generally use the $.ajax method and do not use convenience methods. As you'll see, it offers features that the convenience methods do not, and its syntax is more easily understandable, in my opinion.

$.ajax

jQuery's core `$.ajax` method is a powerful and straightforward way of creating Ajax requests. It takes a configuration object that contains all the instructions jQuery requires to complete the request. The `$.ajax` method is particularly valuable because it offers the ability to specify both success and failure callbacks. Also, its ability to take a configuration object that can be defined separately makes it easier to write reusable code. For complete documentation of the configuration options, visit http://api.jquery.com/jQuery.ajax/.

Example 7.1. Using the core $.ajax method

```
$.ajax({
    // the URL for the request
    url : 'post.php',

    // the data to send
    // (will be converted to a query string)
    data : { id : 123 },

    // whether this is a POST or GET request
    method : 'GET',

    // the type of data we expect back
    dataType : 'json',

    // code to run if the request succeeds;
    // the response is passed to the function
    success : function(json) {
        $('<h1/>').text(json.title).appendTo('body');
        $('<div class="content"/>')
            .html(json.html).appendTo('body');
    },

    // code to run if the request fails;
    // the raw request and status codes are
    // passed to the function
    error : function(xhr, status) {
        alert('Sorry, there was a problem!');
    },

    // code to run regardless of success or failure
    complete : function(xhr, status) {
        alert('The request is complete!');
    }
});
```

Note

A note about the `dataType` setting: if the server sends back data that is in a different format than you specify, your code may fail, and the reason will not always be clear, because the HTTP response code will not show an error. When working with Ajax requests, make sure your server is sending back the data type you're asking for, and verify that the Content-type header is accurate for

the data type. For example, for JSON data, the Content-type header should be `application/json`.

`$.ajax` Options

There are many, many options for the $.ajax method, which is part of its power. For a complete list of options, visit http://api.jquery.com/jQuery.ajax/; here are several that you will use frequently:

async — Set to `false` if the request should be sent synchronously. Defaults to `true`. Note that if you set this option to false, your request will block execution of other code until the response is received.

cache — Whether to use a cached response if available. Defaults to `true` for all dataTypes except "script" and "jsonp". When set to false, the URL will simply have a cachebusting parameter appended to it.

complete — A callback function to run when the request is complete, regardless of success or failure. The function receives the raw request object and the text status of the request.

context — The scope in which the callback function(s) should run (i.e. what `this` will mean inside the callback function(s)). By default, `this` inside the callback function(s) refers to the object originally passed to `$.ajax`.

data — The data to be sent to the server. This can either be an object or a query string, such as `foo=bar&baz=bim`.

dataType — The type of data you expect back from the server. By default, jQuery will look at the MIME type of the response if no dataType is specified.

error — A callback function to run if the request results in an error. The function receives the raw request object and the text status of the request.

jsonp — The callback name to send in a query string when making a JSONP request. Defaults to "callback".

success — A callback function to run if the request succeeds. The function receives the response data (converted to a JavaScript object if the dataType was JSON), as well as the text status of the request and the raw request object.

timeout — The time in milliseconds to wait before considering the request a failure.

traditional — Set to true to use the param serialization style in use prior to jQuery 1.4. For details, see http://api.jquery.com/jQuery.param/.

type — The type of the request, "POST" or "GET". Defaults to "GET". Other request types, such as "PUT" and "DELETE" can be used, but they may not be supported by all browsers.

url — The URL for the request.

The `url` option is the only required property of the `$.ajax` configuration object; all other properties are optional.

Convenience Methods

If you don't need the extensive configurability of `$.ajax`, and you don't care about handling errors, the Ajax convenience functions provided by jQuery can be useful, terse ways to accomplish Ajax requests.

These methods are just "wrappers" around the core `$.ajax` method, and simply pre-set some of the options on the `$.ajax` method.

The convenience methods provided by jQuery are:

$.get	Perform a GET request to the provided URL.
$.post	Perform a POST request to the provided URL.
$.getScript	Add a script to the page.
$.getJSON	Perform a GET request, and expect JSON to be returned.

In each case, the methods take the following arguments, in order:

url	The URL for the request. Required.
data	The data to be sent to the server. Optional. This can either be an object or a query string, such as `foo=bar&baz=bim`.

> ## Note
>
> This option is not valid for `$.getScript`.

success callback	A callback function to run if the request succeeds. Optional. The function receives the response data (converted to a JavaScript object if the data type was JSON), as well as the text status of the request and the raw request object.
data type	The type of data you expect back from the server. Optional.

> ## Note
>
> This option is only applicable for methods that don't already specify the data type in their name.

Example 7.2. Using jQuery's Ajax convenience methods

```
// get plain text or html
$.get('/users.php', { userId : 1234 }, function(resp) {
    console.log(resp);
});

// add a script to the page, then run a function defined in it
$.getScript('/static/js/myScript.js', function() {
    functionFromMyScript();
});

// get JSON-formatted data from the server
$.getJSON('/details.php', function(resp) {
    $.each(resp, function(k, v) {
        console.log(k + ' : ' + v);
    });
});
```

$.fn.load

The $.fn.load method is unique among jQuery's Ajax methods in that it is called on a selection. The $.fn.load method fetches HTML from a URL, and uses the returned HTML to populate the selected element(s). In addition to providing a URL to the method, you can optionally provide a selector; jQuery will fetch only the matching content from the returned HTML.

Example 7.3. Using $.fn.load to populate an element

```
$('#newContent').load('/foo.html');
```

Example 7.4. Using $.fn.load to populate an element based on a selector

```
$('#newContent').load('/foo.html #myDiv h1:first', function(html) {
  alert('Content updated!');
});
```

Ajax and Forms

jQuery's ajax capabilities can be especially useful when dealing with forms. The jQuery Form Plugin [http://jquery.malsup.com/form/] is a well-tested tool for adding Ajax capabilities to forms, and you should generally use it for handling forms with Ajax rather than trying to roll your own solution for anything remotely complex. That said, there are a two jQuery methods you should know that relate to form processing in jQuery: $.fn.serialize and $.fn.serializeArray.

Example 7.5. Turning form data into a query string

```
$('#myForm').serialize();
```

Example 7.6. Creating an array of objects containing form data

```
$('#myForm').serializeArray();

// creates a structure like this:
[
    { name : 'field1', value : 123 },
    { name : 'field2', value : 'hello world' }
]
```

Working with JSONP

The advent of JSONP -- essentially a consensual cross-site scripting hack -- has opened the door to powerful mashups of content. Many prominent sites provide JSONP services, allowing you access to their content via a predefined API. A particularly great source of JSONP-formatted data is the Yahoo! Query Language [http://developer.yahoo.com/yql/console/], which we'll use in the following example to fetch news about cats.

Example 7.7. Using YQL and JSONP

```
$.ajax({
    url : 'http://query.yahooapis.com/v1/public/yql',

    // the name of the callback parameter,
    // as specified by the YQL service
    jsonp : 'callback',

    // tell jQuery we're expecting JSONP
    dataType : 'jsonp',

    // tell YQL what we want and that we want JSON
    data : {
        q : 'select title,abstract,url from search.news where query="cat"',
        format : 'json'
    },

    // work with the response
    success : function(response) {
        console.log(response);
    }
});
```

jQuery handles all the complex aspects of JSONP behind-the-scenes -- all we have to do is tell jQuery the name of the JSONP callback parameter specified by YQL ("callback" in this case), and otherwise the whole process looks and feels like a normal Ajax request.

Ajax Events

Often, you'll want to perform an operation whenever an Ajax requests starts or stops, such as showing or hiding a loading indicator. Rather than defining this behavior inside every Ajax request, you can bind Ajax events to elements just like you'd bind other events. For a complete list of Ajax events, visit http://docs.jquery.com/Ajax_Events.

Example 7.8. Setting up a loading indicator using Ajax Events

```
$('#loading_indicator')
    .ajaxStart(function() { $(this).show(); })
    .ajaxStop(function() { $(this).hide(); });
```

Exercises

Load External Content

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/load.js`. Your task is to load the content of a blog item when a user clicks on the title of the item.

1. Create a target div after the headline for each blog post and store a reference to it on the headline element using `$.fn.data`.
2. Bind a click event to the headline that will use the `$.fn.load` method to load the appropriate content from `/exercises/data/blog.html` into the target div. Don't forget to prevent the default action of the click event.

Note that each blog headline in index.html includes a link to the post. You'll need to leverage the href of that link to get the proper content from blog.html. Once you have the href, here's one way to process it into an ID that you can use as a selector in `$.fn.load`:

```
var href = 'blog.html#post1';
var tempArray = href.split('#');
var id = '#' + tempArray[1];
```

Remember to make liberal use of `console.log` to make sure you're on the right path!

Load Content Using JSON

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/specials.js`. Your task is to show the user details about the special for a given day when the user selects a day from the select dropdown.

1. Append a target div after the form that's inside the #specials element; this will be where you put information about the special once you receive it.

2. Bind to the change event of the select element; when the user changes the selection, send an Ajax request to `/exercises/data/specials.json`.

3. When the request returns a response, use the value the user selected in the select (hint: `$.fn.val`) to look up information about the special in the JSON response.

4. Add some HTML about the special to the target div you created.

5. Finally, because the form is now Ajax-enabled, remove the submit button from the form.

Note that we're loading the JSON every time the user changes their selection. How could we change the code so we only make the request once, and then use a cached response when the user changes their choice in the select?

Chapter 8. Plugins

Finding & Evaluating Plugins

Plugins extend the basic jQuery functionality, and one of the most celebrated aspects of the library is its extensive plugin ecosystem. From table sorting to form validation to autocompletion ... if there's a need for it, chances are good that someone has written a plugin for it.

The quality of jQuery plugins varies widely. Many plugins are extensively tested and well-maintained, but others are hastily created and then ignored. More than a few fail to follow best practices.

Google is your best initial resource for locating plugins, though the jQuery team is working on an improved plugin repository. Once you've identified some options via a Google search, you may want to consult the jQuery mailing list or the #jquery IRC channel to get input from others.

When looking for a plugin to fill a need, do your homework. Ensure that the plugin is well-documented, and look for the author to provide lots of examples of its use. Be wary of plugins that do far more than you need; they can end up adding substantial overhead to your page. For more tips on spotting a subpar plugin, read Signs of a poorly written jQuery plugin [http://remysharp.com/2010/06/03/signs-of-a-poorly-written-jquery-plugin/] by Remy Sharp.

Once you choose a plugin, you'll need to add it to your page. Download the plugin, unzip it if necessary, place it your application's directory structure, then include the plugin in your page using a script tag (after you include jQuery).

Writing Plugins

Sometimes you want to make a piece of functionality available throughout your code; for example, perhaps you want a single method you can call on a jQuery selection that performs a series of operations on the selection. In this case, you may want to write a plugin.

Most plugins are simply methods created in the `$.fn` namespace. jQuery guarantees that a method called on a jQuery object will be able to access that jQuery object as `this` inside the method. In return, your plugin needs to guarantee that it returns the same object it received, unless explicitly documented otherwise.

Here is an example of a simple plugin:

Example 8.1. Creating a plugin to add and remove a class on hover

```
// defining the plugin
$.fn.hoverClass = function(c) {
    return this.hover(
        function() { $(this).toggleClass(c); }
    );
};

// using the plugin
$('li').hoverClass('hover');
```

Exercises

Make a Table Sortable

For this exercise, your task is to identify, download, and implement a table sorting plugin on the index.html page. When you're done, all columns in the table on the page should be sortable.

Write a Table-Striping Plugin

Open the file `/exercises/index.html` in your browser. Use the file `/exercises/js/stripe.js`. Your task is to write a plugin called "stripe" that you can call on any table element. When the plugin is called on a table element, it should change the color of odd rows in the table body to a user-specified color.

```
$('#myTable').stripe('#cccccc');
```

Don't forget to return the table so other methods can be chained after the plugin!