jQuery Fundamentals

By Rebecca Murphey

http://github.com/rmurphey/jqfundamentals

With contributions by James Padolsey, Paul Irish, and others. See the GitHub repository for a complete history of contributions.

Contents

List of Examples

Back to top

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. 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
- jQuery documentation
- jQuery forum
- Delicious bookmarks
- #jquery IRC channel on Freenode

Part I. JavaScript 101

Back to top

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 + Number(bar));
```

The Number constructor, when called as a function (like above) will have the effect of casting its argument into a number. You could also use the unary plus operator, which does the same thing:

Example 2.10: Forcing a string to act as a number (using the unary-plus operator)

```
console.log(foo + +bar);
```

Logical Operators

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

Example 2.11: 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 the example, the `||` operator returns the value of the first truthy operand, or, in cases where neither operand is truthy, it'll return the last of both operands. The `&&` operator returns the value of the first false operand, or the value of the last operand if both operands are truthy.

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 2.12: 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` and `else` blocks — lets you run code only under certain conditions.

Example 2.13: 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 2.14: Values that evaluate to true

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

Example 2.15: 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 2.16: 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 2.17: 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 object 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 2.18: Loops

```
// logs 'try 0', 'try 1', ..., 'try 4'
for (var i=0; i<5; i++) {
    console.log('try ' + i);
}
```

Note that in Loops even though we use the keyword var before the variable name `i`, this does not "scope" the variable `i` to the loop block. We'll discuss scope in depth later in this chapter.

The for loop

A `for` loop is made up of four statements and has the following structure:

```
for ([initialisation]; [conditional];
[iteration])
 [loopBody]
```

The *initialisation* statement is executed only once, before the loop starts. It gives you an opportunity to prepare or declare any variables.

The *conditional* statement is executed before each iteration, and its return value decides whether or not the loop is to continue. If the conditional statement evaluates to a falsey value then the loop stops.

The *iteration* statement is executed at the end of each iteration and gives you an opportunity to change the state of important variables. Typically, this will involve incrementing or decrementing a counter and thus bringing the loop ever closer to its end.

The *loopBody* statement is what runs on every iteration. It can contain anything you want. You'll typically have multiple statements that need to be executed and so will wrap them in a block (`{...}`).

Here's a typical `for` loop:

Example 2.19: A typical for loop

```
for (var i = 0, limit = 100; i < limit; i++) {
    // This block will be executed 100 times
    console.log('Currently at ' + i);
    // Note: the last log will be "Currently at
99"
}
```

The while loop

A `while` loop is similar to an `if` statement, except that its body will keep executing until the condition evaluates to false.

```
while ([conditional]) [loopBody]
```

Here's a typical `while` loop:

Example 2.20: A typical while loop

```
var i = 0;
while (i < 100) {

    // This block will be executed 100 times
    console.log('Currently at ' + i);

    i++; // increment i

}
```

You'll notice that we're having to increment the counter within the loop's body. It is possible to combine the conditional and incrementer, like so:

Example 2.21: A while loop with a combined conditional and incrementer

```
var i = -1;
while (++i < 100) {
    // This block will be executed 100 times
    console.log('Currently at ' + i);
}
```

Notice that we're starting at `-1` and using the prefix incrementer (`++i`).

The do-while loop

This is almost exactly the same as the `while` loop, except for the fact that the loop's body is executed at least once before the condition is tested.

```
do [loopBody] while ([conditional])
```

Here's a `do-while` loop:

Example 2.22: A do-while loop

```
do {

    // Even though the condition evaluates to
false
    // this loop's body will still execute
once.

    alert('Hi there!');

} while (false);
```

These types of loops are quite rare since only few situations require a loop that blindly executes at least once. Regardless, it's good to be aware of it.

Breaking and continuing

Usually, a loop's termination will result from the conditional statement not evaluating to true, but it is possible to stop a loop in its tracks from within the loop's body with the `break` statement.

Example 2.23: Stopping a loop

```
for (var i = 0; i < 10; i++) {
    if (something) {
        break;
    }
}
```

You may also want to continue the loop without executing more of the loop's body. This is done using the `continue` statement.

Example 2.24: Skipping to the next iteration of a loop

```
for (var i = 0; i < 10; i++) {

    if (something) {
        continue;
    }

    // The following statement will only be
executed
    // if the conditional 'something' has not
been met
    console.log('I have been reached');

}
```

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.

- `abstract`
- `boolean`
- `break`
- `byte`
- `case`
- `catch`
- `char`
- `class`
- `const`
- `continue`
- `debugger`
- `default`
- `delete`
- `do`
- `double`
- `else`
- `enum`
- `export`
- `extends`
- `final`
- `finally`
- `float`
- `for`

- `function`
- `goto`
- `if`
- `implements`
- `import`
- `in`
- `instanceof`
- `int`
- `interface`
- `long`
- `native`
- `new`
- `package`
- `private`
- `protected`
- `public`
- `return`
- `short`
- `static`
- `super`
- `switch`
- `synchronized`
- `this`
- `throw`
- `throws`
- `transient`
- `try`
- `typeof`
- `var`
- `void`
- `volatile`
- `while`
- `with`

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 2.25: A simple array

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

Example 2.26: Accessing array items by index

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

Example 2.27: Testing the size of an array

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

Example 2.28: 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 "Changing the value of an array item", it's generally not advised.

Example 2.29: Adding elements to an array

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

Example 2.30: 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. 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 2.31: Creating an "object literal"

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

    myName : 'Rebecca'
};

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

Note

When creating object literals, you should note that the key portion of each key-value pair can be written as any valid JavaScript identifier, a string (wrapped in quotes) or a number:

```
var myObject = {
    validIdentifier: 123,
    'some string': 456,
    99999: 789
};
```

Object literals can be extremely useful for code organization; for more information, read 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 2.32: Function Declaration

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

Example 2.33: 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. You are likely to see both methods used in others' JavaScript code.

Using Functions

Example 2.34: A simple function

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


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

Example 2.35: A function that returns a value

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

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

Example 2.36: 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 2.37: A self-executing anonymous function

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


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

Functions as Arguments

In JavaScript, functions are "first-class citizens" &mdash 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 2.38: 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 2.39: 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".

It's common practice to use the `typeof` operator when trying to determining the type of a specific value.

Example 2.40: 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'

typeof null;         // returns 'object' -- careful!


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

if (Object.prototype.toString.call(myArray) === '[object Array]') {
    // Definitely an array!
    // This is widely considered as the most robust way
    // to determine if a specific value is an Array.
}
```

jQuery offers utility methods to help you determine the type of an arbitrary value. These will be covered later.

The `this` keyword

In JavaScript, as in most object-oriented programming languages, `this` is a special keyword that is used within methods to refer to the object on which a method is being invoked. The value of `this` is determined using a simple series of steps:

1. If the function is invoked using Function.call or Function.apply, `this` will be set to the first argument passed to call/apply. If the first argument passed to call/apply is `null` or `undefined`, `this` will refer to the global object (which is the `window` object in Web browsers).
2. If the function being invoked was created using Function.bind, `this` will be the first argument that was passed to bind at the time the function was created.
3. If the function is being invoked as a method of an object, `this` will refer to that object.
4. Otherwise, the function is being invoked as a standalone function not attached to any object, and `this` will refer to the global object.

Example 2.41: A function invoked using Function.call

```
var myObject = {
    sayHello : function() {
        console.log('Hi! My name is ' + this.myName);
    },

    myName : 'Rebecca'
};

```

```
var secondObject = {
    myName : 'Colin'
};

myObject.sayHello();                    // logs
'Hi! My name is Rebecca'
myObject.sayHello.call(secondObject); // logs
'Hi! My name is Colin'
```

Example 2.42: A function created using Function.bind

```
var myName = 'the global object',

    sayHello = function () {
        console.log('Hi! My name is ' +
this.myName);
    },

    myObject = {
        myName : 'Rebecca'
    };

var myObjectHello = sayHello.bind(myObject);

sayHello();       // logs 'Hi! My name is the
global object'
myObjectHello();  // logs 'Hi! My name is
Rebecca'
```

Example 2.43: A function being attached to an object at runtime

```
var myName = 'the global object',

    sayHello = function() {
        console.log('Hi! My name is ' +
this.myName);
    },

    myObject = {
        myName : 'Rebecca'
    },

    secondObject = {
        myName : 'Colin'
    };

myObject.sayHello = sayHello;
secondObject.sayHello = sayHello;

sayHello();               // logs 'Hi! My name
is the global object'
myObject.sayHello();      // logs 'Hi! My name
is Rebecca'
secondObject.sayHello();  // logs 'Hi! My name
is Colin'
```

Note

When invoking a function deep within a long namespace, it is often tempting to reduce the amount of code you need to type by storing a reference to the actual function as a single, shorter variable. It is important not to do this with instance methods as this will cause the value of `this` within the function to change, leading to incorrect code operation. For instance:

```
var myNamespace = {
    myObject : {
        sayHello : function() {
            console.log('Hi! My name is ' +
this.myName);
        },

        myName : 'Rebecca'
    }
};

var hello = myNamespace.myObject.sayHello;

hello();  // logs 'Hi! My name is
undefined'
```

You can, however, safely reduce everything up to the object on which the method is invoked:

```
var myNamespace = {
    myObject : {
        sayHello : function() {
            console.log('Hi! My name is ' +
this.myName);
        },

        myName : 'Rebecca'
    }
};

var obj = myNamespace.myObject;

obj.sayHello();  // logs 'Hi! My name is
Rebecca'
```

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 2.44: 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 2.45: 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 2.46: 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 2.47: 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 2.48: 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 “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 5.

Example 2.49: How to lock in the value of i?

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

Example 2.50: 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>click me</p>').appendTo('body').click(createFunction(i));
}
```

Closures can also be used to resolve issues with the `this` keyword, which is unique to each scope:

Example 2.51: Using a closure to access inner and outer object instances simultaneously

```
var outerObj = {
    myName : 'outer',
    outerFunction : function () {

        // provide a reference to outerObj through innerFunction's closure
        var self = this;

        var innerObj = {
            myName : 'inner',
            innerFunction : function () {
                console.log(self.myName, this.myName); // logs 'outer inner'
            }
        };

        innerObj.innerFunction();

        console.log(this.myName); // logs 'outer'
    }
};

outerObj.outerFunction();
```

This mechanism can be particularly useful when dealing with callbacks, though in those cases, it is often better to use Function.bind, which will avoid any overhead associated with scope traversal.

Part II. jQuery: Basic Concepts

Back to top

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 pass a named function to `$(document).ready()` instead of passing an anonymous function.

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

```
function readyFn() {
    // 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
```

Note

When you use the `:visible` and `:hidden` pseudo-selectors, jQuery tests the actual visibility of the element, not its CSS visibility or display — that is, it looks to see if the element's *physical height and width on the page* are both greater than zero. However, this test doesn't work with `<tr>` elements; in this case, jQuery does check the CSS `display` property, and considers an element hidden if its `display` property is set to `none`. Elements that have not been added to the DOM will always be considered hidden, even if the CSS that would affect them would render them visible. (See the Manipulation section later in this chapter to learn how to create and add elements to the DOM.)

For reference, here is the code jQuery uses to determine whether an element is visible or hidden, with comments added for clarity:

```
jQuery.expr.filters.hidden = function( elem ) {
    var width = elem.offsetWidth, height =
elem.offsetHeight,
        skip = elem.nodeName.toLowerCase() ===
"tr";

    // does the element have 0 height, 0 width,
    // and it's not a <tr>?
    return width === 0 && height === 0 && !skip
?

        // then it must be hidden
        true :

        // but if it has width and height
        // and it's not a <tr>
        width > 0 && height > 0 && !skip ?

            // then it must be visible
            false :

            // if we get here, the element has
width
            // and height, but it's also a
<tr>,
            // so check its display property to
            // decide whether it's hidden
            jQuery.curCSS(elem, "display") ===
"none";
};

jQuery.expr.filters.visible = function( elem )
{
    return !jQuery.expr.filters.hidden( elem );
};
```

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, 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 “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 — 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 sixth
```

Selecting Form Elements

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. When a method is used to set a value, it is called a setter. 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` when used as a property name in JavaScript. This does not apply, however, when passing the name of a CSS property to the `$.fn.css` method as a string — in that case, either the camel cased or hyphenated form will work.

Example 3.18: Getting CSS properties

```
$('h1').css('fontSize'); // returns a string such as "19px"
$('h1').css('font-size'); // also works
```

Example 3.19: Setting CSS properties

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

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 dimension and position information about an element.

The code in “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</li>');
```

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. It can either set an explicit value, or set a value using the return value of a function. When the function syntax is used, the function receives two arguments: the zero-based index of the element whose attribute is being changed, and the current value of the attribute being changed.

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(idx, href) {
        return '/new/' + href;
    }
});

$('#myDiv a:first').attr('href', function(idx,
href) {
    return '/new/' + 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.

Back to top

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'
```

$.proxy

Returns a function that will always run in the provided scope — that is, sets the meaning of `this` inside the passed function to the second argument.

```
var myFunction = function() { console.log(this); };
var myObject = { foo : 'bar' };

myFunction(); // logs window object

var myProxyFunction = $.proxy(myFunction, myObject);
myProxyFunction(); // logs myObject object
```

If you have an object with methods, you can pass the object and the name of a method to return a function that will always run in the scope of the object.

```
var myObject = {
    myFn : function() {
        console.log(this);
    }
};

$('#foo').click(myObject.myFn); // logs DOM element #foo
$('#foo').click($.proxy(myObject, 'myFn')); // logs myObject
```

Checking types

As mentioned in the "JavaScript basics" section, jQuery offers a few basic utility methods for determining the type of a specific value.

Example 4.1: Checking the type of an arbitrary value

```
var myValue = [1, 2, 3];

// Using JavaScript's typeof operator to test for primative types
typeof myValue == 'string'; // false
typeof myValue == 'number'; // false
typeof myValue == 'undefined'; // false
typeof myValue == 'boolean'; // false

// Using strict equality operator to check for null
myValue === null; // false

```

```
// Using jQuery's methods to check for non-
primative types
jQuery.isFunction(myValue); // false
jQuery.isPlainObject(myValue); // false
jQuery.isArray(myValue); // true
```

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 4.2: 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 4.3: 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 4.4: 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 4.5: 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>
```

Back to top

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 handler function to multiple events, when you want to provide data to the event hander, when you are working with custom events, or when you want to pass an object of multiple events and handlers.

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);
        // logs event type, then { foo : 'bar'
}
    }
);
```

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() {
    console.log('You just clicked this for the first time!');
    $(this).click(function() { console.log('You have 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
```

Binding Multiple Events

Quite often elements in your application will be bound to multiple events, each having a different function for handing the event. In these cases you can pass an object into `$.fn.bind` with one or more key/value pairs, with the key being the event name and the value being the function to handle the event.

Example 5.8: Binding Multiple Events

```
$('p').bind({
  'click': function() { console.log
('clicked!'); },
  'mouseover': function() { console.log
('hovered!'); }
});
```

Note

The option to pass an object of multiple events and handlers to `$.fn.bind` was introduced in jQuery 1.4.4.

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);
```

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`.

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.

Event delegation using `$.fn.delegate`

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

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.

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.

The hover helper function

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

`$.fn.toggle`

The `$.fn.toggle` method is triggered by the "click" event and accepts two or more functions. Each time the click event occurs, the next function in the list is called. Generally, `$.fn.toggle` is used with just two functions; however, it will accept an unlimited number of functions. Be careful, though: providing a long list of functions can be difficult to debug).

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 event 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.

Back to top

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. 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.)

Back to top

Ajax

Overview

The XMLHttpRequest method (XHR) allows browsers to 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 adding 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:

```
var response;

```

```
$.get('foo.php', function(r) { response = r;
});
console.log(response); // undefined!
```

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
    type : '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 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, 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?

Back to top

Plugins

What exactly is a plugin?

A jQuery plugin is simply a new method that we use to extend jQuery's prototype object. By extending the prototype object you enable all jQuery objects to inherit any methods that you add. As established, whenever you call `jQuery()` you're creating a new jQuery object, with all of jQuery's methods inherited.

The idea of a plugin is to do something with a collection of elements. You could consider each method that comes with the jQuery core a plugin, like `fadeOut` or `addClass`.

You can make your own plugins and use them privately in your code or you can release them into the wild. There are thousands of jQuery plugins available online. The barrier to creating a plugin of your own is so low that you'll want to do it straight away!

How to create a basic plugin

The notation for creating a typical plugin is as follows:

```
(function($){
    $.fn.myNewPlugin = function() {
        return this.each(function(){
            // do something
        });
    };
}(jQuery));
```

Don't let that confuse you though. The point of a jQuery plugin is to extend jQuery's prototype object, and that's what's happening on this line:

```
$.fn.myNewPlugin = function() { //...
```

We wrap this assignment in an immediately-invoked function:

```
(function($){
    //...
}(jQuery));
```

This has the effect of creating a "private" scope that allows us to extend jQuery using the dollar symbol without having to risk the possibility that the dollar has been overwritten by another library.

So our actual plugin, thus far, is this:

```
$.fn.myNewPlugin = function() {
    return this.each(function(){
        // do something
    });
};
```

The `this` keyword within the new plugin refers to the jQuery object on which the plugin is being called.

```
var somejQueryObject = $('#something');

$.fn.myNewPlugin = function() {
    alert(this === somejQueryObject);
};

somejQueryObject.myNewPlugin(); // alerts
'true'
```

Your typical jQuery object will contain references to any number of DOM elements, and that's why jQuery objects are often referred to as collections.

So, to do something with a collection we need to loop through it, which is most easily achieved using jQuery's `each()` method:

```
$.fn.myNewPlugin = function() {
    return this.each(function(){

    });
};
```

jQuery's `each()` method, like most other jQuery methods, returns a jQuery object, thus enabling what we've all come to know and love as 'chaining' (`$(...).css().attr()...`). We wouldn't want to break this convention so we return the `this` object. Within this loop you can do whatever you want with each element. Here's an example of a small plugin using some of the techniques we've discussed:

```
(function($){
    $.fn.showLinkLocation = function() {
        return this.filter('a').each(function()
{
            $(this).append(
                ' (' + $(this).attr('href') +
')'
            );
        });
    };
}(jQuery));

// Usage example:
$('a').showLinkLocation();
```

This handy plugin goes through all anchors in the collection and appends the `href` attribute in brackets.

```
<!-- Before plugin is called: -->
<a href="page.html">Foo</a>

<!-- After plugin is called: -->
<a href="page.html">Foo (page.html)</a>
```

Our plugin can be optimised though:

```
(function($){
    $.fn.showLinkLocation = function() {
        return this.filter('a').append(function
(){
            return ' (' + this.href + ')';
        });
    };
}(jQuery));
```

We're using the `append` method's capability to accept a callback, and the return value of that callback will determine what is appended to each element in the collection. Notice also that we're not using the `attr` method to retrieve the `href` attribute, because the native DOM API gives us easy access with the aptly named `href` property.

Here's another example of a plugin. This one doesn't require us to loop through every elememt with the `each()` method. Instead, we're simply going to delegate to other jQuery methods directly:

```
(function($){
    $.fn.fadeInAndAddClass = function(duration,
className) {

        return this.fadeIn(duration, function()
{
            $(this).addClass(className);
        });
    };
}(jQuery));

// Usage example:
$('a').fadeInAndAddClass(400,
'finishedFading');
```

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 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
(function($){
    $.fn.hoverClass = function(c) {
        return this.hover(
            function() { $(this).toggleClass
(c); }
        );
    };
})(jQuery);

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

For more on plugin development, read Mike Alsup's essential post, A Plugin Development Pattern. In it, he creates a plugin called `$.fn.hilight`, which provides support for the metadata plugin if it's present, and provides a centralized method for setting global and instance options for the plugin.

Example 8.2: The Mike Alsup jQuery Plugin Development Pattern

```
//
// create closure
//
(function($) {
  //
  // plugin definition
  //
  $.fn.hilight = function(options) {
    debug(this);
    // build main options before element
iteration
    var opts = $.extend({},
$.fn.hilight.defaults, options);
    // iterate and reformat each matched
element
    return this.each(function() {
      $this = $(this);
      // build element specific options
      var o = $.meta ? $.extend({}, opts,
$this.data()) : opts;
      // update element styles
      $this.css({
        backgroundColor: o.background,
        color: o.foreground
      });
      var markup = $this.html();
      // call our format function
      markup = $.fn.hilight.format(markup);
      $this.html(markup);
    });
  };
  //
  // private function for debugging
  //
  function debug($obj) {
    if (window.console && window.console.log)
      window.console.log('hilight selection
count: ' + $obj.size());
  };
  //
  // define and expose our format function
  //
  $.fn.hilight.format = function(txt) {
    return '<strong>' + txt + '</strong>';
  };
  //
  // plugin defaults
  //
  $.fn.hilight.defaults = {
    foreground: 'red',
    background: 'yellow'
  };
//
// end of closure
//
})(jQuery);
```

Writing Stateful Plugins with the jQuery UI Widget Factory

Note

This section is based, with permission, on the blog post Building Stateful jQuery Plugins by Scott Gonzalez.

While most existing jQuery plugins are stateless — that is, we call them on an element and that is the extent of our interaction with the plugin — there's a large set of functionality that doesn't fit into the basic plugin pattern.

In order to fill this gap, jQuery UI has implemented a more advanced plugin system. The new system manages state, allows multiple functions to be exposed via a single plugin, and provides various extension points. This system is called the widget factory and is exposed as `jQuery.widget` as part of jQuery UI 1.8; however, it can be used independently of jQuery UI.

To demonstrate the capabilities of the widget factory, we'll build a simple progress bar plugin.

To start, we'll create a progress bar that just lets us set the progress once. As we can see below, this is done by calling `jQuery.widget` with two parameters: the name of the plugin to create and an object literal containing functions to support our plugin. When our plugin gets called, it will create a new plugin instance and all functions will be executed within the context of that instance. This is different from a standard jQuery plugin in two important ways. First, the context is an object, not a DOM element. Second, the context is always a single object, never a collection.

Example 8.3: A simple, stateful plugin using the jQuery UI widget factory

```
$.widget("nmk.progressbar", {
    _create: function() {
        var progress = this.options.value + "%";
        this.element
            .addClass("progressbar")
            .text(progress);
    }
});
```

The name of the plugin must contain a namespace; in this case we've used the `nmk` namespace. There is a limitation that namespaces be exactly one level deep — that is, we can't use a namespace like `nmk.foo`. We can also see that the widget factory has provided two properties for us. `this.element` is a jQuery object containing exactly one element. If our plugin is called on a jQuery object containing multiple elements, a separate plugin instance will be created for each element, and each instance will have its own `this.element`. The second property, `this.options`, is a hash containing key/value pairs for all of our plugin's options. These options can be passed to our plugin as shown here.

Note

In our example we use the `nmk` namespace. The `ui` namespace is reserved for official jQuery UI plugins. When building your own plugins, you should create your own namespace. This makes it clear where the plugin came from and whether it is part of a larger collection.

Example 8.4: Passing options to a widget

```
$("<div></div>")
    .appendTo( "body" )
    .progressbar({ value: 20 });
```

When we call `jQuery.widget` it extends jQuery by adding a method to `jQuery.fn` (the same way we'd create a standard plugin). The name of the function it adds is based on the name you pass to `jQuery.widget`, without the namespace; in our case it will create `jQuery.fn.progressbar`. The options passed to our plugin get set in `this.options` inside of our plugin instance. As shown below, we can specify default values for any of our options. When designing your API, you should figure out the most common use case for your plugin so that you can set appropriate default values and make all options truly optional.

Example 8.5: Setting default options for a widget

```
$.widget("nmk.progressbar", {
    // default options
    options: {
        value: 0
    },

    _create: function() {
        var progress = this.options.value +
"%";
        this.element
            .addClass( "progressbar" )
            .text( progress );
    }
});
```

Adding Methods to a Widget

Now that we can initialize our progress bar, we'll add the ability to perform actions by calling methods on our plugin instance. To define a plugin method, we just include the function in the object literal that we pass to `jQuery.widget`. We can also define "private" methods by prepending an underscore to the function name.

Example 8.6: Creating widget methods

```
$.widget("nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        var progress = this.options.value +
"%";
        this.element
            .addClass("progressbar")
            .text(progress);
    },

    // create a public method
    value: function(value) {
        // no value passed, act as a getter
        if (value === undefined) {
            return this.options.value;
        // value passed, act as a setter
        } else {
            this.options.value =
this._constrain(value);
            var progress = this.options.value +
"%";
            this.element.text(progress);
        }
    },

    // create a private method
    _constrain: function(value) {
        if (value > 100) {
            value = 100;
        }
        if (value < 0) {
            value = 0;
        }
        return value;
    }
});
```

To call a method on a plugin instance, you pass the name of the method to the jQuery plugin. If you are calling a method that accepts parameters, you simply pass those parameters after the method name.

Example 8.7: Calling methods on a plugin instance

```
var bar = $("<div></div>")
    .appendTo("body")
    .progressbar({ value: 20 });

// get the current value
alert(bar.progressbar("value"));

// update the value
bar.progressbar("value", 50);

// get the current value again
alert(bar.progressbar("value"));
```

Note

Executing methods by passing the method name to the same jQuery function that was used to initialize the plugin may seem odd. This is done to prevent pollution of the jQuery namespace while maintaining the ability to chain method calls.

Working with Widget Options

One of the methods that is automatically available to our plugin is the `option` method. The option method allows you to get and set options after initialization. This method works exactly like jQuery's css and attr methods: you can pass just a name to use it as a setter, a name and value to use it as a single setter, or a hash of name/value pairs to set multiple values. When used as a getter, the plugin will return the current value of the option that corresponds to the name that was passed in. When used as a setter, the plugin's `_setOption` method will be called for each option that is being set. We can specify a `_setOption` method in our plugin to react to option changes.

Example 8.8: Responding when an option is set

```
$.widget("nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
    },

    _update: function() {
        var progress = this.options.value +
"%";
        this.element.text(progress);
    }
});
```

Adding Callbacks

One of the easiest ways to make your plugin extensible is to add callbacks so users can react when the state of your plugin changes. We can see below how to add a callback to our progress bar to signify when the progress has reached 100%. The `_trigger` method takes three parameters: the name of the callback, a native event object that initiated the callback, and a hash of data relevant to the event. The callback name is the only required parameter, but the others can be very useful for users who want to implement custom functionality on top of your plugin. For example, if we were building a draggable plugin, we could pass the native mousemove event when triggering a drag callback; this would allow users to react to the drag based on the x/y coordinates provided by the event object.

Example 8.9: Providing callbacks for user extension

```
$.widget("nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
    },

    _update: function() {
        var progress = this.options.value +
"%";
        this.element.text(progress);
        if (this.options.value == 100) {
            this._trigger("complete", null, {
value: 100 });
        }
    }
});
```

Callback functions are essentially just additional options, so you can get and set them just like any other option. Whenever a callback is executed, a corresponding event is triggered as well. The event type is determined by concatenating the plugin name and the callback name. The callback and event both receive the same two parameters: an event object and a hash of data relevant to the event, as we'll see below.

If your plugin has functionality that you want to allow the user to prevent, the best way to support this is by creating cancelable callbacks. Users can cancel a callback, or its associated event, the same way they cancel any native event: by calling `event.preventDefault()` or using `return false`. If the user cancels the callback, the `_trigger` method will return false so you can implement the appropriate functionality within your plugin.

Example 8.10: Binding to widget events

```
var bar = $("<div></div>")
    .appendTo("body")
    .progressbar({
        complete: function(event, data) {
            alert( "Callbacks are great!" );
        }
    })
    .bind("progressbarcomplete", function
(event, data) {
        alert("Events bubble and support many
handlers for extreme flexibility.");
        alert("The progress bar value is " +
data.value);
    });

bar.progressbar("option", "value", 100);
```

The Widget Factory: Under the Hood

When you call `jQuery.widget`, it creates a constructor function for your plugin and sets the object literal that you pass in as the prototype for your plugin instances. All of the functionality that automatically gets added to your plugin comes from a base widget prototype, which is defined as `jQuery.Widget.prototype`. When a plugin instance is created, it is stored on the original DOM element using `jQuery.data`, with the plugin name as the key.

Because the plugin instance is directly linked to the DOM element, you can access the plugin instance directly instead of going through the exposed plugin method if you want. This will allow you to call methods directly on the plugin instance instead of passing method names as strings and will also give you direct access to the plugin's properties.

```
var bar = $("<div></div>")
    .appendTo("body")
    .progressbar()
    .data("progressbar" );

// call a method directly on the plugin
instance
bar.option("value", 50);

// access properties on the plugin instance
alert(bar.options.value);
```

One of the biggest benefits of having a constructor and prototype for a plugin is the ease of extending the plugin. By adding or modifying methods on the plugin's prototype, we can modify the behavior of all instances of our plugin. For example, if we wanted to add a method to our progress bar to reset the progress to 0% we could add this method to the prototype and it would instantly be available to be called on any plugin instance.

```

```

```
$.nmk.progressbar.prototype.reset = function()
{
    this._setOption("value", 0);
};
```

Cleaning Up

In some cases, it will make sense to allow users to apply and then later unapply your plugin. You can accomplish this via the destroy method. Within the `destroy` method, you should undo anything your plugin may have done during initialization or later use. The `destroy` method is automatically called if the element that your plugin instance is tied to is removed from the DOM, so this can be used for garbage collection as well. The default `destroy` method removes the link between the DOM element and the plugin instance, so it's important to call the base function from your plugin's `destroy` method.

Example 8.11: Adding a destroy method to a widget

```
$.widget( "nmk.progressbar", {
    options: {
        value: 0
    },

    _create: function() {
        this.element.addClass("progressbar");
        this._update();
    },

    _setOption: function(key, value) {
        this.options[key] = value;
        this._update();
    },

    _update: function() {
        var progress = this.options.value +
"%";
        this.element.text(progress);
        if (this.options.value == 100 ) {
            this._trigger("complete", null, {
value: 100 });
        }
    },

    destroy: function() {
        this.element
            .removeClass("progressbar")
            .text("");

        // call the base destroy function
        $.Widget.prototype.destroy.call(this);
    }
});
```

Conclusion

The widget factory is only one way of creating stateful plugins. There are a few different models that can be used and each have their own advantages and disadvantages. The widget factory solves lots of common problems for you and can greatly improve productivity, it also greatly improves code reuse, making it a great fit for jQuery UI as well as many other stateful plugins.

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!

Part III. Advanced Topics

This Section is a Work in Progress

Please visit https://github.com/jquery/web-learn-jquery-com to contribute!

Back to top

Performance Best Practices

This chapter covers a number of jQuery and JavaScript best practices, in no particular order. Many of the best practices in this chapter are based on the jQuery Anti-Patterns for Performance presentation by Paul Irish.

Cache length during loops

In a for loop, don't access the length property of an array every time; cache it beforehand.

```
var myLength = myArray.length;

for (var i = 0; i < myLength; i++) {
    // do stuff
}
```

Append new content outside of a loop

Touching the DOM comes at a cost; if you're adding a lot of elements to the DOM, do it all at once, not one at a time.

```
// this is bad
$.each(myArray, function(i, item) {
   var newListItem = '<li>' + item + '</li>';
   $('#ballers').append(newListItem);
});

// better: do this
var frag = document.createDocumentFragment();

$.each(myArray, function(i, item) {
    var newListItem = '<li>' + item + '</li>';
    frag.appendChild(newListItem);
});
$('#ballers')[0].appendChild(frag);

// or do this
var myHtml = '';

$.each(myArray, function(i, item) {
    html += '<li>' + item + '</li>';
});
$('#ballers').html(myHtml);
```

Keep things DRY

Don't repeat yourself; if you're repeating yourself, you're doing it wrong.

```
// BAD
if ($eventfade.data('currently') != 'showing')
{
    $eventfade.stop();
}

if ($eventhover.data('currently') != 'showing')
{
    $eventhover.stop();
}

if ($spans.data('currently') != 'showing') {
    $spans.stop();
}

// GOOD!!
var $elems = [$eventfade, $eventhover, $spans];
```

```
$.each($elems, function(i,elem) {
    if (elem.data('currently') != 'showing') {
        elem.stop();
    }
});
```

Beware anonymous functions

Anonymous functions bound everywhere are a pain. They're difficult to debug, maintain, test, or reuse. Instead, use an object literal to organize and name your handlers and callbacks.

```
// BAD
$(document).ready(function() {
    $('#magic').click(function(e) {
        $('#yayeffects').slideUp(function() {
            // ...
        });
    });

    $('#happiness').load(url + ' #unicorns',
function() {
        // ...
    });
});

// BETTER
var PI = {
    onReady : function() {
        $('#magic').click(PI.candyMtn);
        $('#happiness').load(PI.url + '
#unicorns', PI.unicornCb);
    },

    candyMtn : function(e) {
        $('#yayeffects').slideUp(PI.slideCb);
    },

    slideCb : function() { ... },

    unicornCb : function() { ... }
};

$(document).ready(PI.onReady);
```

Optimize Selectors

Selector optimization is less important than it used to be, as more browsers implement `document.querySelectorAll()` and the burden of selection shifts from jQuery to the browser. However, there are still some tips to keep in mind.

ID-Based Selectors

Beginning your selector with an ID is always best.

```
// fast
$('#container div.robotarm');

// super-fast
```

```
$('#container').find('div.robotarm');
```

The `$.fn.find` approach is faster because the first selection is handled without going through the Sizzle selector engine — ID-only selections are handled using `document.getElementById()`, which is extremely fast because it is native to the browser.

Specificity

Be specific on the right-hand side of your selector, and less specific on the left.

```
// unoptimized
$('div.data .gonzalez');

// optimized
$('.data td.gonzalez');
```

Use `tag.class` if possible on your right-most selector, and just `tag` or just `.class` on the left.

Avoid excessive specificity.

```
$('.data table.attendees td.gonzalez');

// better: drop the middle if possible
$('.data td.gonzalez');
```

A "flatter" DOM also helps improve selector performance, as the selector engine has fewer layers to traverse when looking for an element.

Avoid the Universal Selector

Selections that specify or imply that a match could be found anywhere can be very slow.

```
$('.buttons > *');  // extremely expensive
$('.buttons').children();  // much better

$('.gender :radio');  // implied universal selection
$('.gender *:radio'); // same thing, explicit now
$('.gender input:radio'); // much better
```

Use Event Delegation

Event delegation allows you to bind an event handler to one container element (for example, an unordered list) instead of multiple contained elements (for example, list items). jQuery makes this easy with $.fn.live and $.fn.delegate. Where possible, you should use `$.fn.delegate` instead of `$.fn.live`, as it eliminates the need for an unnecessary selection, and its explicit context (vs. `$.fn.live`'s context of `document`) reduces overhead by approximately 80%.

In addition to performance benefits, event delegation also allows you to add new contained elements to your page without having to re-bind the event handlers for them as they're added.

```
// bad (if there are lots of list items)
$('li.trigger').click(handlerFn);

// better: event delegation with $.fn.live
$('li.trigger').live('click', handlerFn);

// best: event delegation with $.fn.delegate
// allows you to specify a context easily
$('#myList').delegate('li.trigger', 'click',
handlerFn);
```

Detach Elements to Work With Them

The DOM is slow; you want to avoid manipulating it as much as possible. jQuery introduced `$.fn.detach` in version 1.4 to help address this issue, allowing you to remove an element from the DOM while you work with it.

```
var $table = $('#myTable');
var $parent = $table.parent();

$table.detach();
// ... add lots and lots of rows to table
$parent.append(table);
```

Use Stylesheets for Changing CSS on Many Elements

If you're changing the CSS of more than 20 elements using $.fn.css, consider adding a style tag to the page instead for a nearly 60% increase in speed.

```
// fine for up to 20 elements, slow after that
$('a.swedberg').css('color', '#asd123');
$('<style type="text/css">a.swedberg { color :
#asd123 }</style>')
    .appendTo('head');
```

Use `$.data` Instead of `$.fn.data`

Using $.data on a DOM element instead of calling $.fn.data on a jQuery selection can be up to 10 times faster. Be sure you understand the difference between a DOM element and a jQuery selection before doing this, though.

```
// regular
$(elem).data(key,value);
```

```

// 10x faster
$.data(elem,key,value);
```

Don't Act on Absent Elements

jQuery won't tell you if you're trying to run a whole lot of code on an empty selection — it will proceed as though nothing's wrong. It's up to you to verify that your selection contains some elements.

```
// BAD: this runs three functions
// before it realizes there's nothing
// in the selection
$('#nosuchthing').slideUp();

// Better
var $mySelection = $('#nosuchthing');
if ($mySelection.length) { $mySelection.slideUp
(); }

// BEST: add a doOnce plugin
jQuery.fn.doOnce = function(func){
    this.length && func.apply(this);
    return this;
}

$('li.cartitems').doOnce(function(){
    // make it ajax! \o/
});
```

This guidance is especially applicable for jQuery UI widgets, which have a lot of overhead even when the selection doesn't contain elements.

Variable Definition

Variables can be defined in one statement instead of several.

```
// old & busted
var test = 1;
var test2 = function() { ... };
var test3 = test2(test);

// new hotness
var test = 1,
    test2 = function() { ... },
    test3 = test2(test);
```

In self-executing functions, variable definition can be skipped all together.

```
(function(foo, bar) { ... })(1, 2);
```

Conditionals

```
// old way
if (type == 'foo' || type == 'bar') { ... }

```

```
// better
if (/^(foo|bar)$/.test(type)) { ... }

// object literal lookup
if (({ foo : 1, bar : 1 })[type]) { ... }
```

Don't Treat jQuery as a Black Box

Use the source as your documentation — bookmark http://bit.ly/jqsource and refer to it often.

Back to top

Code Organization

Overview

When you move beyond adding simple enhancements to your website with jQuery and start developing full-blown client-side applications, you need to consider how to organize your code. In this chapter, we'll take a look at various code organization patterns you can use in your jQuery application and explore the RequireJS dependency management and build system.

Key Concepts

Before we jump into code organization patterns, it's important to understand some concepts that are common to all good code organization patterns.

- Your code should be divided into units of functionality — modules, services, etc. Avoid the temptation to have all of your code in one huge `$(document).ready()` block. This concept, loosely, is known as encapsulation.
- Don't repeat yourself. Identify similarities among pieces of functionality, and use inheritance techniques to avoid repetitive code.
- Despite jQuery's DOM-centric nature, JavaScript applications are not all about the DOM. Remember that not all pieces of functionality need to — or should — have a DOM representation.
- Units of functionality should be loosely coupled — a unit of functionality should be able to exist on its own, and communication between units should be handled via a

> messaging system such as custom events or pub/sub. Stay away from direct communication between units of functionality whenever possible.

The concept of loose coupling can be especially troublesome to developers making their first foray into complex applications, so be mindful of this as you're getting started.

Encapsulation

The first step to code organization is separating pieces of your application into distinct pieces; sometimes, even just this effort is sufficient to lend

The Object Literal

An object literal is perhaps the simplest way to encapsulate related code. It doesn't offer any privacy for properties or methods, but it's useful for eliminating anonymous functions from your code, centralizing configuration options, and easing the path to reuse and refactoring.

Example 10.1: An object literal

```
var myFeature = {
myProperty : 'hello',

myMethod : function() {
    console.log(myFeature.myProperty);
},

init : function(settings) {
    myFeature.settings = settings;
},

readSettings : function() {
    console.log(myFeature.settings);
}
};

myFeature.myProperty; // 'hello'
myFeature.myMethod(); // logs 'hello'
myFeature.init({ foo : 'bar' });
myFeature.readSettings(); // logs { foo : 'bar'
}
```

The object literal above is simply an object assigned to a variable. The object has one property and several methods. All of the properties and methods are public, so any part of your application can see the properties and call methods on the object. While there is an init method, there's nothing requiring that it be called before the object is functional.

How would we apply this pattern to jQuery code? Let's say that we had this code written in the traditional jQuery style:

```
// clicking on a list item loads some content
// using the list item's ID and hides content
// in sibling list items
```

```
$(document).ready(function() {
$('#myFeature li')
.append('<div/>')
.click(function() {
  var $this = $(this);
  var $div = $this.find('div');
  $div.load('foo.php?item=' +
    $this.attr('id'),
    function() {
      $div.show();
      $this.siblings()
        .find('div').hide();
    }
  );
});
});
```

If this were the extent of our application, leaving it as-is would be fine. On the other hand, if this was a piece of a larger application, we'd do well to keep this functionality separate from unrelated functionality. We might also want to move the URL out of the code and into a configuration area. Finally, we might want to break up the chain to make it easier to modify pieces of the functionality later.

Example 10.2: Using an object literal for a jQuery feature

```
var myFeature = {
init : function(settings) {
    myFeature.config = {
        $items : $('#myFeature li'),
        $container : $('<div
class="container"></div>'),
        urlBase : '/foo.php?item='
    };

    // allow overriding the default config
    $.extend(myFeature.config, settings);

    myFeature.setup();
},

setup : function() {
    myFeature.config.$items
        .each(myFeature.createContainer)
        .click(myFeature.showItem);
},

createContainer : function() {
    var $i = $(this),
        $c = myFeature.config.$container.clone
()
                 .appendTo($i);

    $i.data('container', $c);
},

buildUrl : function() {
    return myFeature.config.urlBase +
           myFeature.$currentItem.attr('id');
},

showItem : function() {
    var myFeature.$currentItem = $(this);
    myFeature.getContent
(myFeature.showContent);
},

getContent : function(callback) {
    var url = myFeature.buildUrl();
    myFeature.$currentItem
        .data('container').load(url, callback);
},

showContent : function() {
    myFeature.$currentItem
        .data('container').show();
    myFeature.hideContent();
},

hideContent : function() {
    myFeature.$currentItem.siblings()
        .each(function() {
            $(this).data('container').hide();
        });
}
};

$(document).ready(myFeature.init);
```

The first thing you'll notice is that this approach is obviously far longer than the original — again, if this were the extent of our application, using an object literal would likely be overkill. Assuming it's not the extent of our application, though, we've gained several things:

- We've broken our feature up into tiny methods. In the future, if we want to change how content is shown, it's clear where to change it. In the original code, this step is much harder to locate.
- We've eliminated the use of anonymous functions.
- We've moved configuration options out of the body of the code and put them in a central location.
- We've eliminated the constraints of the chain, making the code easier to refactor, remix, and rearrange.

For non-trivial features, object literals are a clear improvement over a long stretch of code stuffed in a $(document).ready() block, as they get us thinking about the pieces of our functionality. However, they aren't a whole lot more advanced than simply having a bunch of function declarations inside of that $(document).ready() block.

The Module Pattern

The module pattern overcomes some of the limitations of the object literal, offering privacy for variables and functions while exposing a public API if desired.

Example 10.3: The module pattern

```
var feature =(function() {
```

```

// private variables and functions
var privateThing = 'secret',
    publicThing = 'not secret',

    changePrivateThing = function() {
        privateThing = 'super secret';
    },

    sayPrivateThing = function() {
        console.log(privateThing);
        changePrivateThing();
    };

// public API
return {
    publicThing : publicThing,
    sayPrivateThing : sayPrivateThing
}

})();

feature.publicThing; // 'not secret'

feature.sayPrivateThing();
// logs 'secret' and changes the value
// of privateThing
```

In the example above, we self-execute an anonymous function that returns an object. Inside of the function, we define some variables. Because the variables are defined inside of the function, we don't have access to them outside of the function unless we put them in the return object. This means that no code outside of the function has access to the `privateThing` variable or to the `changePrivateThing` function. However, `sayPrivateThing` does have access to `privateThing` and `changePrivateThing`, because both were defined in the same scope as `sayPrivateThing`.

This pattern is powerful because, as you can gather from the variable names, it can give you private variables and functions while exposing a limited API consisting of the returned object's properties and methods.

Below is a revised version of the previous example, showing how we could create the same feature using the module pattern while only exposing one public method of the module, `showItemByIndex()`.

Example 10.4: Using the module pattern for a jQuery feature

```
$(document).ready(function() {
var feature = (function() {

    var $items = $('#myFeature li'),
        $container = $('<div
class="container"></div>'),
        $currentItem,

        urlBase = '/foo.php?item=',

        createContainer = function() {
            var $i = $(this),
                $c = $container.clone().appendTo($i);

            $i.data('container', $c);
        },

        buildUrl = function() {
            return urlBase + $currentItem.attr('id');
        },

        showItem = function() {
            var $currentItem = $(this);
            getContent(showContent);
        },

        showItemByIndex = function(idx) {
            $.proxy(showItem, $items.get(idx));
        },

        getContent = function(callback) {
            $currentItem.data('container').load(buildUrl(), callback);
        },

        showContent = function() {
            $currentItem.data('container').show();
            hideContent();
        },

        hideContent = function() {
            $currentItem.siblings()
                .each(function() {
                    $(this).data('container').hide();
            });
        };

    $items
        .each(createContainer)
        .click(showItem);

    return { showItemByIndex : showItemByIndex };
})();

feature.showItemByIndex(0);
});
```

Managing Dependencies

Note

This section is based heavily on the excellent RequireJS documentation at http://requirejs.org/docs/jquery.html, and is used with the permission of RequireJS author James Burke.

When a project reaches a certain size, managing the script modules for a project starts to get tricky. You need to be sure to sequence the scripts in the right order, and you need to start seriously thinking about combining scripts together into a bundle for deployment, so that only one or a very small number of requests are made to load the scripts. You may also want to load code on the fly, after page load.

RequireJS, a dependency management tool by James Burke, can help you manage the script modules, load them in the right order, and make it easy to combine the scripts later via the RequireJS optimization tool without needing to change your markup. It also gives you an easy way to load scripts after the page has loaded, allowing you to spread out the download size over time.

RequireJS has a module system that lets you define well-scoped modules, but you do not have to follow that system to get the benefits of dependency management and build-time optimizations. Over time, if you start to create more modular code that needs to be reused in a few places, the module format for RequireJS makes it easy to write encapsulated code that can be loaded on the fly. It can grow with you, particularly if you want to incorporate internationalization (i18n) string bundles, to localize your project for different languages, or load some HTML strings and make sure those strings are available before executing code, or even use JSONP services as dependencies.

Getting RequireJS

The easiest way to use RequireJS with jQuery is to download a build of jQuery that has RequireJS built in. This build excludes portions of RequireJS that duplicate jQuery functionality. You may also find it useful to download a sample jQuery project that uses RequireJS.

Using RequireJS with jQuery

Using RequireJS in your page is simple: just include the jQuery that has RequireJS built in, then require your application files. The following example assumes that the jQuery build, and your other scripts, are all in a `scripts/` directory.

Example 10.5: Using RequireJS: A simple example

```
<!DOCTYPE html>
<html>
<head>
    <title>jQuery+RequireJS Sample Page</title>
    <script src="scripts/require-jquery.js"></script>
    <script>require(["app"]);</script>
</head>
<body>
    <h1>jQuery+RequireJS Sample Page</h1>
</body>
</html>
```

The call to `require(["app"])` tells RequireJS to load the `scripts/app.js` file. RequireJS will load any dependency that is passed to `require()` without a `.js` extension from the same directory as `require-jquery.js`, though this can be configured to behave differently. If you feel more comfortable specifying the whole path, you can also do the following:

```
<script>require(["scripts/app.js"]);</script>
```

What is in `app.js`? Another call to `require.js` to load all the scripts you need and any init work you want to do for the page. This example `app.js` script loads two plugins, `jquery.alpha.js` and `jquery.beta.js` (not the names of real plugins, just an example). The plugins should be in the same directory as `require-jquery.js`:

Example 10.6: A simple JavaScript file with dependencies

```
require(["jquery.alpha", "jquery.beta"],
function() {
//the jquery.alpha.js and jquery.beta.js
plugins have been loaded.
$(function() {
    $('body').alpha().beta();
});
});
```

Creating Reusable Modules with RequireJS

RequireJS makes it easy to define reusable modules via `require.def()`. A RequireJS module can have dependencies that can be used to define a module, and a RequireJS module can return a value — an object, a function, whatever — that can then be consumed by yet other modules.

If your module does not have any dependencies, then just specify the name of the module as the first argument to `require.def()`. The second argument is just an object literal that defines the module's properties. For example:

Example 10.7: Defining a RequireJS module that has no dependencies

```
require.def("my/simpleshirt",
{
    color: "black",
    size: "unisize"
}
);
```

This example would be stored in a my/simpleshirt.js file.

If your module has dependencies, you can specify the dependencies as the second argument to `require.def()` (as an array) and then pass a function as the third argument. The function will be called to define the module once all dependencies have loaded. The function receives the values returned by the dependencies as its arguments (in the same order they were required in the array), and the function should return an object that defines the module.

Example 10.8: Defining a RequireJS module with dependencies

```
require.def("my/shirt",
["my/cart", "my/inventory"],
function(cart, inventory) {
    //return an object to define the "my/shirt"
module.
    return {
        color: "blue",
        size: "large"
        addToCart: function() {
            inventory.decrement(this);
            cart.add(this);
        }
    }
}
);
```

In this example, a my/shirt module is created. It depends on my/cart and my/inventory. On disk, the files are structured like this:

```
my/cart.js
my/inventory.js
my/shirt.js
```

The function that defines `my/shirt` is not called until the `my/cart` and `my/inventory` modules have been loaded, and the function receives the modules as the `cart` and `inventory` arguments. The order of the function arguments must match the order in which the dependencies were required in the dependencies array. The object returned by the function call defines the `my/shirt` module. Be defining modules in this way, `my/shirt` does not exist as a global object. Modules that define globals are explicitly discouraged, so multiple versions of a module can exist in a page at a time.

Modules do not have to return objects; any valid return value from a function is allowed.

Example 10.9: Defining a RequireJS module that returns a function

```
require.def("my/title",
["my/dependency1", "my/dependency2"],
function(dep1, dep2) {
    //return a function to define "my/title".
It gets or sets
    //the window title.
    return function(title) {
        return title ? (window.title = title) :
window.title;
    }
}
);
```

Only one module should be required per JavaScript file.

Optimizing Your Code: The RequireJS Build Tool

Once you incorporate RequireJS for dependency management, your page is set up to be optimized very easily. Download the RequireJS source and place it anywhere you like, preferrably somewhere outside your web development area. For the purposes of this example, the RequireJS source is placed as a sibling to the `webapp` directory, which contains the HTML page and the scripts directory with all the scripts. Complete directory structure:

```
requirejs/ (used for the build tools)
webapp/app.html
webapp/scripts/app.js
webapp/scripts/require-jquery.js
webapp/scripts/jquery.alpha.js
webapp/scripts/jquery.beta.js
```

Then, in the scripts directory that has `require-jquery.js` and app.js, create a file called app.build.js with the following contents:

A RequireJS build configuration file

```
{
appDir: "../",
baseUrl: "scripts/",
dir: "../../webapp-build",
//Comment out the optimize line if you want
//the code minified by Closure Compiler using
//the "simple" optimizations mode
optimize: "none",

modules: [
    {
        name: "app"
    }
]
}
```

To use the build tool, you need Java 6 installed. Closure Compiler is used for the JavaScript minification step (if `optimize: "none"` is commented out), and it requires Java 6.

To start the build, go to the webapp/scripts directory, execute the following command:

```
# non-windows systems
../../requirejs/build/build.sh app.build.js

# windows systems
..\..\requirejs\build\build.bat app.build.js
```

Now, in the webapp-build directory, `app.js` will have the `app.js` contents, `jquery.alpha.js` and `jquery.beta.js` inlined. If you then load the `app.html` file in the `webapp-build` directory, you should not see any network requests for `jquery.alpha.js` and `jquery.beta.js`.

Exercises

Create a Portlet Module

Open the file `/exercises/portlets.html` in your browser. Use the file `/exercises/js/portlets.js`. Your task is to create a portlet creation function that uses the module pattern, such that the following code will work:

```
var myPortlet = Portlet({
title : 'Curry',
source : 'data/html/curry.html',
initialState : 'open' // or 'closed'
});

myPortlet.$element.appendTo('body');
```

Each portlet should be a div with a title, a content area, a button to open/close the portlet, a button to remove the portlet, and a button to refresh the portlet. The portlet returned by the Portlet function should have the following public API:

```
myPortlet.open(); // force open state
myPortlet.close(); // force close state
myPortlet.toggle(); // toggle open/close state
myPortlet.refresh(); // refresh the content
myPortlet.destroy(); // remove the portlet from
the page
myPortlet.setSource('data/html/onions.html');
// change the source
```

Back to top

Custom Events

Introducing Custom Events

We're all familiar with the basic events — click, mouseover, focus, blur, submit, etc. — that we can latch on to as a user interacts with the browser. Custom events open up a whole new world of event-driven programming. In this chapter, we'll use jQuery's custom events system to make a simple Twitter search application.

It can be difficult at first to understand why you'd want to use custom events, when the built-in events seem to suit your needs just fine. It turns out that custom events offer a whole new way of thinking about event-driven JavaScript. Instead of focusing on the element that triggers an action, custom events put the spotlight on the element being acted upon. This brings a bevy of benefits, including:

- Behaviors of the target element can easily be triggered by different elements using the same code.
- Behaviors can be triggered across multiple, similar, target elements at once.

- Behaviors are more clearly associated with the target element in code, making code easier to read and maintain.

Why should you care? An example is probably the best way to explain. Suppose you have a lightbulb in a room in a house. The lightbulb is currently turned on, and it's controlled by two three-way switches and a clapper:

```
<div class="room" id="kitchen">
    <div class="lightbulb on"></div>
    <div class="switch"></div>
    <div class="switch"></div>
    <div class="clapper"></div>
</div>
```

Triggering the clapper or either of the switches will change the state of the lightbulb. The switches and the clapper don't care what state the lightbulb is in; they just want to change the state.

Without custom events, you might write some code like this:

```
$('.switch, .clapper').click(function() {
    var $light = $(this).parent().find
('.lightbulb');
    if ($light.hasClass('on')) {
        $light.removeClass('on').addClass
('off');
    } else {
        $light.removeClass('off').addClass
('on');
    }
});
```

With custom events, your code might look more like this:

```
$('.lightbulb').bind('changeState', function(e)
{
    var $light = $(this);
    if ($light.hasClass('on')) {
        $light.removeClass('on').addClass
('off');
    } else {
        $light.removeClass('off').addClass
('on');
    }
});

$('.switch, .clapper').click(function() {
    $(this).parent().find('.lightbulb').trigger
('changeState');
});
```

This last bit of code is not that exciting, but something important has happened: we've moved the behavior of the lightbulb to the lightbulb, and away from the switches and the clapper.

Let's make our example a little more interesting. We'll add another room to our house, along with a master switch, as shown here:

```
<div class="room" id="kitchen">
    <div class="lightbulb on"></div>
    <div class="switch"></div>
    <div class="switch"></div>
    <div class="clapper"></div>
</div>
<div class="room" id="bedroom">
    <div class="lightbulb on"></div>
    <div class="switch"></div>
    <div class="switch"></div>
    <div class="clapper"></div>
</div>
<div id="master_switch"></div>
```

If there are any lights on in the house, we want the master switch to turn all the lights off; otherwise, we want it to turn all lights on. To accomplish this, we'll add two more custom events to the lightbulbs: `turnOn` and `turnOff`. We'll make use of them in the `changeState` custom event, and use some logic to decide which one the master switch should trigger:

```
$('.lightbulb')
    .bind('changeState', function(e) {
        var $light = $(this);
        if ($light.hasClass('on')) {
            $light.trigger('turnOff');
        } else {
            $light.trigger('turnOn');
        }
    })
    .bind('turnOn', function(e) {
        $(this).removeClass('off').addClass
('on');
    })
    .bind('turnOff', function(e) {
        $(this).removeClass('off').addClass
('on');
    });

$('.switch, .clapper').click(function() {
    $(this).parent().find('.lightbulb').trigger
('changeState');
});

$('#master_switch').click(function() {
    if ($('.lightbulb.on').length) {
        $('.lightbulb').trigger('turnOff');
    } else {
        $('.lightbulb').trigger('turnOn');
    }
});
```

Note how the behavior of the master switch is attached to the master switch; the behavior of a lightbulb belongs to the lightbulbs.

Note

If you're accustomed to object-oriented programming, you may find it useful to think of custom events as methods of objects. Loosely speaking, the object to which the method belongs is created via the jQuery selector. Binding the changeState custom event to all `$('.light')` elements is akin to having a class called `Light` with a method of `changeState`, and then instantiating new `Light` objects for each element with a classname of light.

Recap: $.fn.bind and $.fn.trigger

In the world of custom events, there are two important jQuery methods: `$.fn.bind` and `$.fn.trigger`. In the Events chapter, we saw how to use these methods for working with user events; for this chapter, it's important to remember two things:

- The `$.fn.bind` method takes an event type and an event handling function as arguments. Optionally, it can also receive event-related data as its second argument, pushing the event handling function to the third argument. Any data that is passed will be available to the event handling function in the `data` property of the event object. The event handling function always receives the event object as its first argument.
- The `$.fn.trigger` method takes an event type as its argument. Optionally, it can also take an array of values. These values will be passed to the event handling function as arguments after the event object.

Here is an example of the usage of `$.fn.bind` and `$.fn.trigger` that uses custom data in both cases:

```
$(document).bind('myCustomEvent', { foo : 'bar' }, function(e, arg1, arg2) {
    console.log(e.data.foo); // 'bar'
    console.log(arg1); // 'bim'
    console.log(arg2); // 'baz'
});

$(document).trigger('myCustomEvent', [ 'bim', 'baz' ]);
```

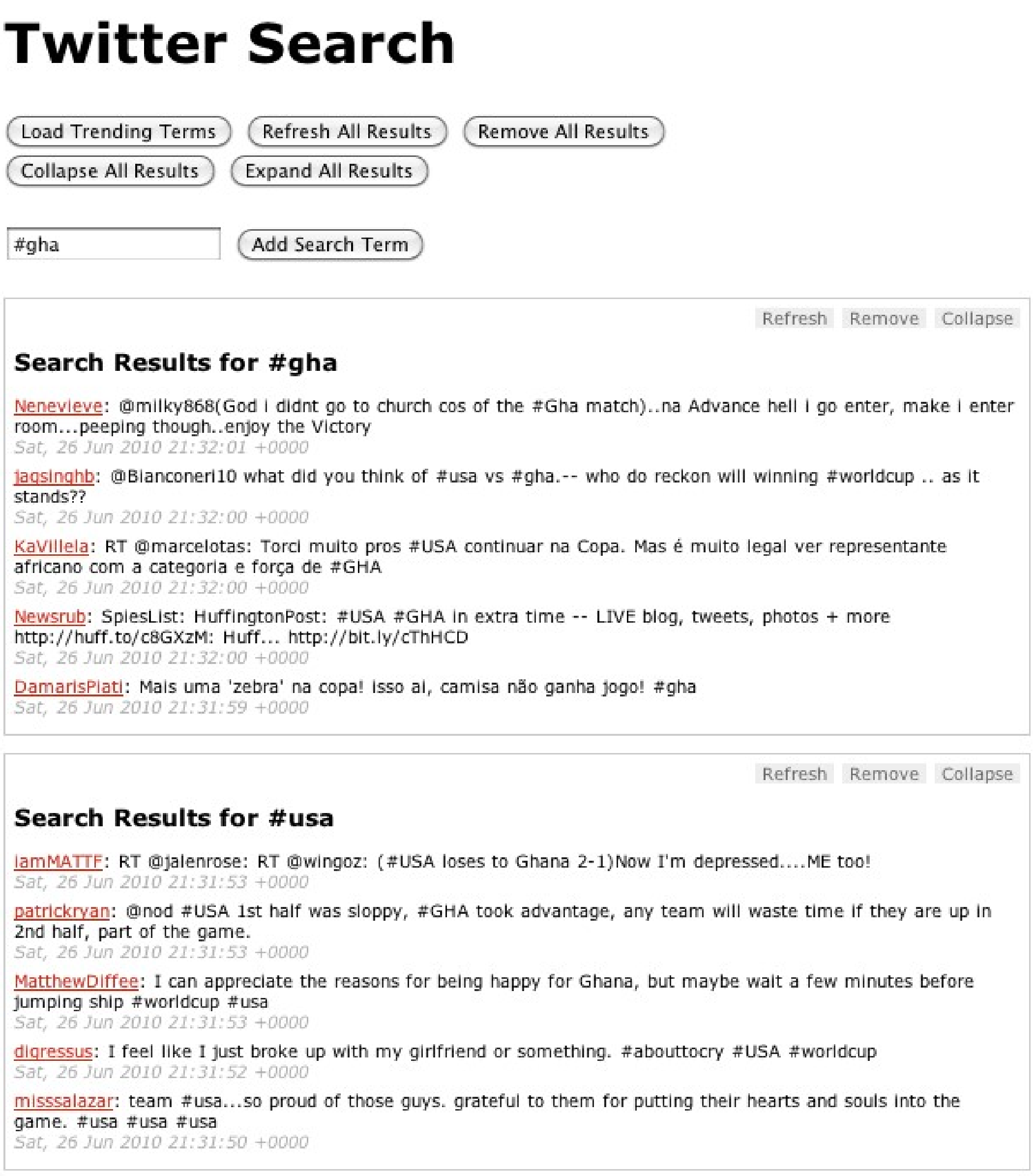
A Sample Application

To demonstrate the power of custom events, we're going to create a simple tool for searching Twitter. The tool will offer several ways for a user to add search terms to the display: by entering a search term in a text box, by entering multiple search terms in the URL, and by querying Twitter for trending terms.

The results for each term will be shown in a results container; these containers will be able to be expanded, collapsed, refreshed, and removed, either individually or all at once.

When we're done, it will look like this:

Figure 11.1. Our finished application



Twitter Search

Search Results for #gha

Nenevieve: @milky868(God i didnt go to church cos of the #Gha match)..na Advance hell i go enter, make i enter room...peeping though..enjoy the Victory
Sat, 26 Jun 2010 21:32:01 +0000

jagsinghb: @Bianconeri10 what did you think of #usa vs #gha.-- who do reckon will winning #worldcup .. as it stands??
Sat, 26 Jun 2010 21:32:00 +0000

KaVillela: RT @marcelotas: Torci muito pros #USA continuar na Copa. Mas é muito legal ver representante africano com a categoria e força de #GHA
Sat, 26 Jun 2010 21:32:00 +0000

Newsrub: SpiesList: HuffingtonPost: #USA #GHA in extra time -- LIVE blog, tweets, photos + more http://huff.to/c8GXzM: Huff... http://bit.ly/cThHCD
Sat, 26 Jun 2010 21:32:00 +0000

DamarisPiati: Mais uma 'zebra' na copa! isso aí, camisa não ganha jogo! #gha
Sat, 26 Jun 2010 21:31:59 +0000

Search Results for #usa

iamMATTF: RT @jalenrose: RT @wingoz: (#USA loses to Ghana 2-1)Now I'm depressed....ME too!
Sat, 26 Jun 2010 21:31:53 +0000

patrickryan: @nod #USA 1st half was sloppy, #GHA took advantage, any team will waste time if they are up in 2nd half, part of the game.
Sat, 26 Jun 2010 21:31:53 +0000

MatthewDiffee: I can appreciate the reasons for being happy for Ghana, but maybe wait a few minutes before jumping ship #worldcup #usa
Sat, 26 Jun 2010 21:31:53 +0000

digressus: I feel like I just broke up with my girlfriend or something. #abouttocry #USA #worldcup
Sat, 26 Jun 2010 21:31:52 +0000

misssalazar: team #usa...so proud of those guys. grateful to them for putting their hearts and souls into the game. #usa #usa #usa
Sat, 26 Jun 2010 21:31:50 +0000

The Setup

We'll start with some basic HTML:

```
<h1>Twitter Search</h1>
<input type="button" id="get_trends"
    value="Load Trending Terms" />

<form>
    <input type="text" class="input_text"
        id="search_term" />
    <input type="submit" class="input_submit"
        value="Add Search Term" />
</form>

<div id="twitter">
    <div class="template results">
        <h2>Search Results for
        <span class="search_term"></span></h2>
    </div>
</div>
```

This gives us a container (#twitter) for our widget, a template for our results containers (hidden via CSS), and a simple form where users can input a search term. (For the sake of simplicity, we're going to assume that our application is JavaScript-only and that our users will always have CSS.)

There are two types of objects we'll want to act on: the results containers, and the Twitter container.

The results containers are the heart of the application. We'll create a plugin that will prepare each results container once it's added to the Twitter container. Among other things, it will bind the custom events for each container and add the action buttons at the top right of each container. Each results container will have the following custom events:

refresh

> Mark the container as being in the "refreshing" state, and fire the request to fetch the data for the search term.

populate

> Receive the returned JSON data and use it to populate the container.

remove

> Remove the container from the page after the user verifies the request to do so. Verification can be bypassed by passing true as the second argument to the event handler. The remove event also removes the term associated with the results container from the global object containing the search terms.

collapse

> Add a class of collapsed to the container, which will hide the results via CSS. It will also turn the container's "Collapse" button into an "Expand" button.

expand

> Remove the collapsed class from the container. It will also turn the container's "Expand" button into a "Collapse" button.

The plugin is also responsible for adding the action buttons to the container. It binds a click event to each action's list item, and uses the list item's class to determine which custom event will be triggered on the corresponding results container.

```
$.fn.twitterResult = function(settings) {
    return this.each(function() {
        var $results = $(this),
            $actions = $.fn.twitterResult.actions =
                $.fn.twitterResult.actions ||
                $.fn.twitterResult.createActions(),
            $a = $actions.clone().prependTo($results),
            term = settings.term;

        $results.find('span.search_term').text(term);

        $.each(
            ['refresh', 'populate', 'remove', 'collapse', 'expand'],
            function(i, ev) {
                $results.bind(
                    ev,
                    { term : term },
                    $.fn.twitterResult.events[ev]
                );
            }
        );

        // use the class of each action to figure out
        // which event it will trigger on the results panel
        $a.find('li').click(function() {
            // pass the li that was clicked to the function
            // so it can be manipulated if needed
            $results.trigger($(this).attr('class'), [ $(this) ]);
        });
    });
};

$.fn.twitterResult.createActions = function() {
    return $('<ul class="actions" />').append(
        '<li class="refresh">Refresh</li>' +
        '<li class="remove">Remove</li>' +
        '<li class="collapse">Collapse</li>'
    );
};

$.fn.twitterResult.events = {
    refresh : function(e) {
           // indicate that the results are refreshing
        var $this = $(this).addClass('refreshing');

        $this.find('p.tweet').remove();
        $results.append('<p class="loading">Loading ...</p>');

        // get the twitter data using jsonp
        $.getJSON(
            'http://search.twitter.com/search.json?q=' +
                escape(e.data.term) + '&rpp=5&callback=?',
            function(json) {
                $this.trigger('populate', [ json ]);
            }
        );
    },

    populate : function(e, json) {
        var results = json.results;
        var $this = $(this);

        $this.find('p.loading').remove();

        $.each(results, function(i,result) {
            var tweet = '<p class="tweet">' +
                '<a href="http://twitter.com/' +
                result.from_user +
                '">' +
                result.from_user +
                '</a>: ' +
                result.text +
                ' <span class="date">' +
                result.created_at +
                '</span>' +
            '</p>';
            $this.append(tweet);
        });

        // indicate that the results
        // are done refreshing
        $this.removeClass('refreshing');
    },

    remove : function(e, force) {
        if (
            !force &&
            !confirm('Remove panel for term ' + e.data.term + '?')
        ) {
            return;
        }
        $(this).remove();

        // indicate that we no longer
        // have a panel for the term
        search_terms[e.data.term] = 0;
    },

    collapse : function(e) {
        $(this).find('li.collapse').removeClass('collapse')
            .addClass('expand').text('Expand');

        $(this).addClass('collapsed');
    },

    expand : function(e) {
        $(this).find('li.expand').removeClass('expand')
            .addClass('collapse').text('Collapse');

        $(this).removeClass('collapsed');
    }
};
```

The Twitter container itself will have just two custom events:

getResults

Receives a search term and checks to determine whether there's already a results container for the term; if not, adds a results container using the results template, set up the results container using the `$.fn.twitterResult` plugin discussed above, and then triggers the `refresh` event on the results container in order to actually load the results. Finally, it will store the search term so the application knows not to re-fetch the term.

getTrends

Queries Twitter for the top 10 trending terms, then iterates over them and triggers the `getResults` event for each of them, thereby adding a results container for each term.

Here's how the Twitter container bindings look:

```
$('#twitter')
    .bind('getResults', function(e, term) {
        // make sure we don't have a box for this term already
        if (!search_terms[term]) {
            var $this = $(this);
            var $template = $this.find('div.template');

            // make a copy of the template div
            // and insert it as the first results box
            $results = $template.clone().
                removeClass('template').
                insertBefore($this.find('div:first')).
                twitterResult({
                    'term' : term
                });

            // load the content using the "refresh"
            // custom event that we bound to the results container
            $results.trigger('refresh');
            search_terms[term] = 1;
        }
    })
    .bind('getTrends', function(e) {
        var $this = $(this);
        $.getJSON('http://search.twitter.com/trends.json?callback=?', function(json) {
                var trends = json.trends;
                $.each(trends, function(i, trend) {
                    $this.trigger('getResults', [ trend.name ]);
                });
            });
    });
```

So far, we've written a lot of code that does approximately nothing, but that's OK. By specifying all the behaviors that we want our core objects to have, we've created a solid framework for rapidly building out the interface.

Let's start by hooking up our text input and the "Load Trending Terms" button. For the text input, we'll capture the term that was entered in the input and pass it as we trigger the Twitter container's `getResults` event. Clicking the “Load Trending Terms” will trigger the Twitter container’s `getTrends` event:

```
$('form').submit(function(e) {
    e.preventDefault();
    var term = $('#search_term').val();
    $('#twitter').trigger('getResults', [ term
]);
});

$('#get_trends').click(function() {
    $('#twitter').trigger('getTrends');
});
```

By adding a few buttons with the appropriate IDs, we can make it possible to remove, collapse, expand, and refresh all results containers at once, as shown below. For the remove button, note how we’re passing a value of true to the event handler as its second argument, telling the event handler that we don’t want to verify the removal of individual containers.

```
$.each(['refresh', 'expand', 'collapse'],
function(i, ev) {
    $('#' + ev).click(function(e) { $('#twitter
div.results').trigger(ev); });
});

$('#remove').click(function(e) {
    if (confirm('Remove all results?')) {
        $('#twitter div.results').trigger
('remove', [ true ]);
    }
});
```

Conclusion

Custom events offer a new way of thinking about your code: they put the emphasis on the target of a behavior, not on the element that triggers it. If you take the time at the outset to spell out the pieces of your application, as well as the behaviors those pieces need to exhibit, custom events can provide a powerful way for you to “talk” to those pieces, either one at a time or en masse. Once the behaviors of a piece have been described, it becomes trivial to trigger those behaviors from anywhere, allowing for rapid creation of and experimentation with interface options. Finally, custom events can enhance code readability and maintainability, by making clear the relationship between an element and its behaviors.

You can see the full application at `demos/custom-events.html` and `demos/js/custom-events.js` in the sample code.