jQuery Fundamentals

Rebecca Murphey

jQuery Fundamentals

Rebecca Murphey

Table of Contents

Part I. jQuery Fundamentals

Chapter 1. Getting Started

Getting the Code

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

References

- jQuery documentation [http://docs.jquery.org]
- jQuery forum [http://forum.jquery.com/]
- Delicious bookmarks [http://delicious.com/rdmey/jquery-class]
- #jquery IRC channel on Freenode [http://docs.jquery.com/Discussion#Chat_.2F_IRC_Channel]

Chapter 2. 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 2.1. 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 2.2. Shorthand for $(document).ready()

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

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

Example 2.3. Passing a named function instead of an anonymous function

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


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

Selecting Elements

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

Following are a few examples of common selection techniques.

Example 2.4. Selecting elements by ID

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

Example 2.5. Selecting elements by class name

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

Example 2.6. Selecting elements by attribute

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

Example 2.7. Selecting elements by compound CSS selector

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

Example 2.8. 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

Choosing good selectors is one of the most important things you can do 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.

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 2.9. 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 2.10. Storing selections in a variable

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

Note

In Example 2.10, “Storing selections in a variable”, the variable name begins with a dollar sign. Unlike in other languages, there's nothing special about the dollar sign in JavaScript -- it's just another character. We use it here to indicate that the variable contains a jQuery object. This practice -- a sort of Hungarian notation [http://en.wikipedia.org/wiki/Hungarian_notation] -- is merely convention, and is not mandatory.

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

Note

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

Refining & Filtering Selections

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

Example 2.11. Refining selections

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

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 2.12. 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 2.13. 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 `end` method to get you back to your original selection.

Example 2.14. Restoring your original selection using `end`

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

Note

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

Getters & Setters

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

Example 2.15. The `html` method used as a setter

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

Example 2.16. The html method used as a getter

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

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

CSS, Styling, & Dimensions

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

Note

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

Example 2.17. Getting CSS properties

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

Example 2.18. Setting CSS properties

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

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

Using CSS Classes for Styling

As a getter, the `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 2.19. Working with classes

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

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

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

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

Dimensions

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

The code in Example 2.20, “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 2.20. 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 `attr` method acts as both a getter and a setter. As with the `css` method, `attr` as a setter can accept either a key and a value, or an object containing one or more key/value pairs.

Example 2.21. 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 2.22. 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 2.23. Traversal methods

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

Manipulating

Once you've made a selection, you can move, remove, and clone elements. You can also create new elements.

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

Example 2.24. Moving, removing, and cloning elements

```
$('div.panel').appendTo('#panels');
$('div.newPanel').insertAfter('div.oldPanel');
$('h1').remove();
$('div.template').clone();
```

Creating New Elements

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

Example 2.25. Creating new elements

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

Example 2.26. Creating a new element with an attribute object

```
$('<p/>', {
    text : 'This is a new paragraph',
    'class' : 'new'
});
```

Note that in the attributes object we included as the second argument, the property name class is quoted, while the property name text is 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 2.27. 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 2.28. 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(''));
```

Chapter 3. 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 `click`, `focus`, `blur`, `change`, etc. -- are shorthand for jQuery's `bind` method. The bind method is useful for binding the same hadler function to multiple events, and is also used when you want to provide data to the event hander, or when you are working with custom events.

Example 3.1. Event binding using a convenience method

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

Example 3.2. Event biding using the `bind` method

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

Example 3.3. Event binding using the `bind` method with data

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

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

Connecting Events to Run Only Once

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

Example 3.4. Switching handlers using the one method

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

The 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 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 3.5. Unbinding all click handlers on a selection

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

Example 3.6. 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 3.7. Namespacing events

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

Inside the Event Handling Function

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

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

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

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

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

Example 3.8. Preventing a link from being followed

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

Triggering Event Handlers

jQuery provides a way to trigger the event handlers bound to an element without any user interaction via the `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 `trigger`.

Example 3.9. Triggering an event handler the right way

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


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

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

Increasing Performance with Event Delegation

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

Example 3.10. Event delegation using `delegate`

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

Example 3.11. Event delegation using `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 `undelegate` for events connected with `delegate`, and `die` for events connected with `live`. As with bind, you can optionally pass in the name of the bound function.

Example 3.12. Unbinding delegated events

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

Event Helpers

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

`hover`

The `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 `hover` method required two functions.

Example 3.13. The hover helper function

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

toggle

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

Example 3.14. The toggle helper function

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

Chapter 4. Effects

Chapter 5. jQuery Core

Chapter 6. Ajax

Part II. JavaScript Primer

Chapter 7. 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 7.1. A simple variable declaration

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

Example 7.2. Whitespace has no meaning outside of quotation marks

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

Example 7.3. Parentheses indicate precedence

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

Example 7.4. 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 7.5. Concatenation

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

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

Example 7.6. Multiplication and division

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

Example 7.7. 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 7.8. Addition vs. concatenation

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

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

Example 7.8, "Addition vs. concatenation" shows how adding two numbers can have unexpected results if one of the numbers is a string.

Example 7.9. Forcing a string to act as a number

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

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

Example 7.9, "Forcing a string to act as a number" shows how to coerce a string back to a number using ***parseInt()****.*

Logical Operators

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

Example 7.10. Logical AND and OR operators

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

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

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

Though it may not be clear from Example 7.10, "Logical AND and OR operators", the `||` operator returns the value of the first truthy comparator, or false if no comparator is truthy. The `&&` operator returns the

value of the last false comparator, or the value of the last comparator if all values are truthy. Essentially, both operators return the first value that proves them true or false.

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

Comparison Operators

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

Example 7.11. Comparison Operators

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

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

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

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

Flow Control

Flow control lets you run a block of code under certain conditions. You should always use braces to mark the start and end of logical blocks.[1]

Example 7.12. Flow control

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

if (foo) {
    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
}
```

[1] Strictly speaking, braces aren't required around single-line **if** statements, but using them consistently, even when they aren't strictly required, makes for vastly more readable code.

Truthy and Falsy Things

In order to

Example 7.13. Values that evaluate to `true`

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

Example 7.14. Values that evaluate to `false`

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

Loops

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

Example 7.15. Loops

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

Note that in Example 7.15, “Loops” we use the keyword var before the variable name `i`. This "scopes" the variable `i` to the loop block. We'll discuss scope in depth later in this chapter.

Reserved Words

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

- `break`
- `case`
- `catch`
- `continue`
- `default`
- `delete`
- `do`
- `else`
- `finally`

- for
- function
- if
- in
- instanceof
- new
- return
- switch
- this
- throw
- try
- typeof
- var
- void
- while
- with
- abstract
- boolean
- byte
- char
- class
- const
- debugger
- double
- enum
- export
- extends
- final
- float
- goto
- implements

- `import`
- `int`
- `interface`
- `long`
- `native`
- `package`
- `private`
- `protected`
- `public`
- `short`
- `static`
- `super`
- `synchronized`
- `throws`
- `transient`
- `volatile`

Arrays

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

Example 7.16. A simple array

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

Example 7.17. Accessing array items by index

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

Example 7.18. Testing the size of an array

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

Example 7.19. Changing the value of an array item

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

While it's possible to change the value of an array item as shown in Example 7.19, “Changing the value of an array item”, it's generally not advised.

Example 7.20. Adding elements to an array

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

Example 7.21. Working with arrays

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

Objects

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

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

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

Example 7.22. Creating an "object literal"

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

    myName : 'Rebecca'
};

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

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 7.23. Function Declaration

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

Example 7.24. Named Function Expression

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

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

Using Functions

Example 7.25. A simple function

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


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

Example 7.26. A function that returns a value

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

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

Example 7.27. 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 7.28. A self-executing anonymous function

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


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

Functions as Arguments

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

Example 7.29. 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 7.30. Passing a named function as an argument

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

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

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

Testing Type

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

Example 7.31. Testing the type of various variables

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

var myObject = {
    foo : 'bar'
};

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

var myString = 'hello';

var myNumber = 3;

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

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

jQuery offers utility methods to help you determine whether

Scope

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

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

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

Example 7.32. 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 7.33. 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 7.34. 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 7.35. 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 7.36. Scope insanity

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

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

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

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

Closures

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

In Example 7.35, “Functions can "see" changes in variable values after the function is defined” we saw how functions have access to changing variable values. The same sort of behavior exists with functions defined within loops -- the function "sees" the change in the variable's value even after the function is defined, resulting in all clicks alerting 4.

Example 7.37. How to lock in the value of i?

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

Example 7.38. Locking in the value of i with a closure

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

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

Part III. Exercises

Chapter 8. Exercises