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 [???]
- 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

Chapter 3. Events

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