JavaScript Basics

Based on “jQuery Fundamentals” by Rebecca Murphey
http://github.com/rmurphey/jqfundamentals

Overview

JavaScript is 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.

A simple variable declaration

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

Whitespace has no meaning outside of quotation marks

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

Parentheses indicate precedence

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

Tabs enhance readability, but have no special meaning

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

Operators

Basic Operators

Basic operators allow you to manipulate values.

Concatenation

```
var foo = 'hello';

var bar = 'world';

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

Multiplication and division

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

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.

Addition vs. concatenation

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

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

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:

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.

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.

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.

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.

Values that evaluate to `true`

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

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.]

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.]

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.

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:

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:

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:

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:

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.

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.

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.

A simple array

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

Accessing array items by index

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

Testing the size of an array

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

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.

Adding elements to an array

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

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.

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:

Function Declaration

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

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

A simple function

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

A function that returns a value

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

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

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.

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.

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'
```

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.

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.

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'
```

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'
```

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;

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'
```

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
```

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'
```

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
```

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.

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);
    });
}
```

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:

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 () {
                // logs 'outer inner'
                console.log(self.myName, this.myName);
            }
        };

        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.