

jQuery Fundamentals Training

JavaScript Basics

Lesson 1, Activity 2: 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.

Syntax Basics

We will cover understanding statements, variable naming, whitespace, and other basic JavaScript syntax.

A Simple Variable Declaration

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

The first line creates a variable with no defined value. In JavaScript, this results in a special value, `undefined`. The second variable is initialized to a *string* value.

Whitespace

Whitespace has no meaning outside of quotation marks:

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

Tabs enhance readability, but have no special meaning:

```
var foo = function(a) {  
  if (a < 10) {
```

```
    alert('Hello');  
}  
}
```

Comments

There are two types of comments: block comments and comments-to-end-of-line:

```
var x = 3;    // double slash is comment to end of line  
  
/*  
    This is a block comment  
*/
```

Lesson 1, Activity 3: **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. Note that some of them are truly *reserved*, they are not currently implemented in the language.

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	

Lesson 1, Activity 5: Operators

Basic Operators

Basic operators allow you to manipulate values.

Addition and Subtraction

```
2 + 3;  
2 - 3;
```

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
```

Combination Operators

```
var x = 0;  
x += 3; // same as x = x + 3;  
x -= 4; // same as x = x - 4;  
x *= 5; // same as x = x * 5;  
x /= 6; // same as x = x / 6;  
  
var y = "Hello";  
y += " World"; // y is now "Hello World"
```

Parentheses Affect Precedence:

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

Lesson 1, Activity 6: Operations on Numbers and 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

// divide and multiply don't work for strings
console.log(foo / bar); // so JavaScript coerces the string to a
                        // number resulting in 0.5
```

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);
```

Lesson 1, Activity 8: Logical Operators

Logical operators allow you to evaluate a series of operands using *AND* and *OR* operations. These operations are called short-circuiting, because they stop evaluating an expression when the overall result has been determined (for example, in an AND operation, if the first value is false, then there is no way that the overall expression can be true, so processing of the expression stops; similarly for an OR operation if the first value is true).

This behavior treats all values as having a logical (true/false) aspect. The concept that a value is treated as true is called *truthy*, and behaving as false is called *falsy*.

Truthy and Falsy Values

In order to use logical operations and 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 Act as True

```
'any string'    // any non-empty string, even '0' and 'false'
[]              // any array, even an empty array
{}              // any object, even an empty object
function() { } // any function
1;              // any non-zero number
true            // the literal true value
```

Values That Act as False

```
0
''              // an empty string
NaN             // JavaScript's Not-a-Number value
```



```

null
undefined
false           // the literal boolean value

```

You should note that "0" and "false" are *truthy*, since they are non-empty strings.

Logical AND and OR operators

In Javascript, `&&` = *AND* while `||` = *OR*

```

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

foo || bar;    // foo OR bar, returns 1, which is truthy
bar || foo;    // bar OR foo, returns 1, which is truthy

foo && bar;     // foo AND bar, returns 0, which is falsy
bar && foo;     // bar AND foo, returns 0, which is falsy
foo && baz;     // foo AND baz, returns 2, which is truthy
baz && foo;     // baz AND foo, returns 1, which is truthy

```

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 will return the last of both operands. The `&&` operator returns the value of the first *falsy* operand, or the value of the last operand if both operands are *truthy*. In essence a logical operation evaluates to the last value it processed.

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.

Lesson 1, Activity 9: Comparison Operators

Comparison operators allow you to test whether values are equivalent or whether values are identical. `||` is the OR operator, and `&&` is the AND operator.

Comparison Operators in Detail

Operator	Description	Example
<code>==</code>	is equal to	<ul style="list-style-type: none"> • <code>5==7</code> returns false • <code>5==5</code> returns true
<code>===</code>	is exactly equal to (value and type)	<ul style="list-style-type: none"> • <code>5==="5"</code> returns false • <code>5===5</code> returns true
<code>!=</code>	is not equal	<ul style="list-style-type: none"> • <code>4!=3</code> returns true • <code>4!=4</code> returns false
<code>!==</code>	is not equal (neither value or type)	<ul style="list-style-type: none"> • <code>4!== "4"</code> returns true • <code>4!==4</code> returns false
<code>></code>	is greater than	<ul style="list-style-type: none"> • <code>5>3</code> returns true • <code>3>5</code> returns false
<code><</code>	is less than	<ul style="list-style-type: none"> • <code>1<2</code> returns true • <code>2<1</code> returns false
<code>>=</code>	is greater than or equal to	<ul style="list-style-type: none"> • <code>5>=5</code> returns true • <code>5>=4</code> returns true • <code>5>=6</code> returns false
<code><=</code>	is less than or equal to	<ul style="list-style-type: none"> • <code>5<=5</code> returns true • <code>5<=7</code> returns true • <code>5<=4</code> returns false

Comparison Operators - More Examples

```
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
```

Lesson 1, Activity 11: 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 statement
if (bar) {
  // this code will never run
  console.log('hello!');
}

// if ... else statement
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.

Lesson 1, Activity 12: 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.

Lesson 1, Activity 13: Switch Statements

Rather than using a series of if/else if/else blocks, sometimes it can be useful to use a `switch` statement instead. `switch` statements are essentially a variable "goto" statement - the `switch` looks at the value of a variable or expression, and jumps to a matching labelled line within block controlled by the switch. At that point, all controlling behavior of the `switch` is done. All of the code after the matching label will be executed, even that belonging to later cases, unless you put in a `break` statement to exit the `switch`.

A switch Statement

```
switch (userResponse) {  
  case 'yes':    // these cases are stacked - whichever one matches  
  case 'yeah':  // will cause execution to fall through to the first  
  case 'yowsa': // executable line below  
  case 'yup':  
    alert('Proceeding ...');  
    break;      // now we exit the switch  
  
  case 'no':  
    alert('OK, bye!');  
    break;  
  
  default:  
    alert('Sorry - I did not understand your response');  
}
```

Lesson 1, Activity 14: Loops

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

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}
```

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 ({ . . . }). If there is only a single statement, then the curly braces are not necessary, but recommended as a good coding practice. The same rule holds for the other looping tags below.

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 = 0;
while (++i <= 100) {
  // This block will be executed 100 times
  console.log('Currently at ' + i);
}
```

Notice that we're starting at 0 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);
```

A **do ... while** Loop Example

This type of loop always executes the body at least once, since the test is performed after each iteration.

```
var i = 10;
do {
  alert(i--);
} while (i >= 0);
alert('Blastoff!');

do {
  // Even though the condition evaluates to false
  // this loop's body will still execute once.
  alert('Hi there!');
} while (false);
```

The **for** Loop

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

```
for (initialization; conditional; iteration) {loopBody}
```

The *initialization* 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 (`{ . . . }`). If there is only a single statement, then the curly braces are not necessary, but recommended as a good coding practice. The same rule holds for the other looping tags below.

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

Note: In this example, 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 ... in` Loop

There is a special version of the for loop that iterates through the elements of an associative array (or, equivalently, the properties of an object).

```
for (loopVariable in object) {loopBody using loopVariable}
```

This loops through the set of element names within `object`, so that `loopVariable` is each name in turn. The associated values would be retrieved as `object[loopVariable]`.

```
var person = { firstName: 'Joe', lastName: 'Smith' };

for (var item in person) {
  console.log(item + ' is' + person[item]);
  // Should log 'firstName is Joe' and 'lastName is Smith'
}
```

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 for this iteration. 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');  
}
```

Lesson 1, Activity 15: 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 Size of an Array

```
var myArray = [ 'hello', 'world' ];  
myArray.length = 0; // truncates the array to zero elements  
console.log(myArray.length); // logs 0
```

Changing the Value of an Array Item

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

Adding Elements to an Array

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

A lot of online code uses the first approach. Since the indexing is zero-based, the length value coincidentally matches the index of the next available spot, so the `'new'` item gets added at that location, and the length goes up by one. The second line uses the object nature of arrays, invoking the array's `push` method to add an item. This, of course, also causes the `length` to increase by one.

Working with Arrays

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

Array Methods

As full-fledged JavaScript objects, arrays contain a number of built-in methods.

Some Useful Array Methods

Method	Description
<code>push(item)</code>	Adds the item at the end of the array, increasing the length by 1.
<code>pop()</code>	Removes and returns the last item from the array, decreasing the length by 1.
<code>join(delimiter)</code>	Returns a single string containing all the elements, concatenated using the specified delimiter. The default delimiter is the comma character. The values used are the string values of the elements - which, for objects, is the result of the objects' <code>toString()</code> method.
<code>split(delimiter)</code>	Splits a string into an array of substrings, and returns the new array. Note that the <code>split()</code> method does not change the original string.

Lesson 1, Activity 16: **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 usually 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*

An object literal is an object written in a shorthand syntax, using a pair of curly braces ({ }) to surround a set of properties: value pairs, separated by commas. The values can be literal strings, numbers, booleans, functions, or nested literal objects, or values from variables or other expressions.

Code Sample:

<jqy-basics/Demos/object-literal.html>

```
<html>
<head>
<script>
var myObject = {
  sayHello : function() {
    alert('hello');
  },

  myName : 'Rebecca'
};

myObject.sayHello();    // alerts 'hello'
alert(myObject.myName); // alerts 'Rebecca'
```

```

</script>
</head>
<body>
  <h1>Object Literals</h1>
</body>

```

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, which does not need to be a valid JavaScript identifier), or a number:

```

var myObject = {
  validIdentifier: 123,
  'some string': 456, // need quotes because of the space
  'class': 'abc',     // because class is a reserved word
  99999: 789
};

```

Object literals can be extremely useful for code organization.

Accessing Elements of an Object

You can work with properties and methods of an object by either:

- Using a dot after the object variable name, followed by the property or method name.

```

person.firstName
person.getFullName()

```

- Using *associative array* notation, using the property or method name as a string index.

```

person["firstName"]
person["getFullName"]()

```

This form is particularly useful when the element name comes

from a variable.

Lesson 1, Activity 18: **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 several ways.

Function Declaration

```
function functionName(parameterList) {  
    /* do something */  
}
```

Named Function Expression

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

This approach creates an *inline function*, then assigns it to a variable. Since functions in JavaScript are also objects, they are treated as having a value, which can be stored in a variable, passed to a function, or returned from a function. When passed to a function as a parameter, or returned from a function, an inline function is also known as an *anonymous function*, since it does not have a name in the current section of 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 task = greet('Rebecca', 'hello');
task();

// or
greet('Rebecca', 'hello') ();
```

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'; } );    // logs 'hello'
```

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'
```

Lesson 1, Activity 20: 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;  
var nothing;  
  
typeof myFunction;    // returns 'function'  
typeof myObject;     // returns 'object'  
typeof myArray;       // returns 'object' -- careful!  
typeof myString;      // returns 'string';  
typeof myNumber;      // returns 'number'  
typeof nothing;       // returns 'undefined';  
  
typeof null;          // returns 'object' -- careful!
```

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

Lesson 1, Activity 21: **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 the declared variable.

Furthermore, variables that are created 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

```

Lesson 1, Activity 23: 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. All of the examples in the "Scope" section used closures to hold on to the value of a local variable declared in the enclosing function.

How to Lock in the Value of i? Use a Closure Combined With Pass-By-Value

In the previous section we saw how functions have access to changing variable values. In the demo below, a function is defined within a loop, using the looping variable inside the function. But, perhaps unexpectedly, the function "sees" the change in the variable's value even after the function is defined, resulting in all clicks alerting 5.

But, if we define a function that takes a parameter, and pass in the value of the looping variable. By definition, a method parameter is a local variable to the function. The passed-in value is captured and held for use when the click event occurs.

Code Sample:

jqy-basics/Demos/lock-i-closure.html

```
<html>
<head>
<title>Using Closures</title>
<script src="../../jqy-lib/jquery.js"></script>
</head>
<body>
<h1>Using Closures</h1>
<script>
// 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);
  });
}

$('<hr />').appendTo('body');

// fix: 'close' the value of the parameter j inside createFunction
// since it was passed by value, each invocation will get its own copy
var createFunction = function(j) {
  return function() { alert(j); };
};

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