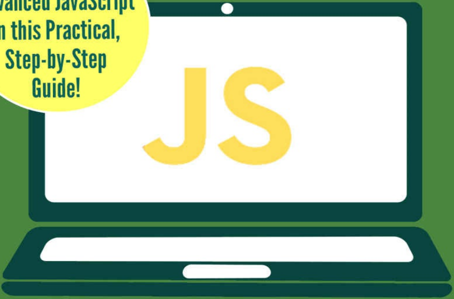
JAVASCRIPT

ADVANCED GUIDE TO PROGRAMMING CODE WITH JAVASCRIPT

Learn Advanced JavaScript in this Practical, Step-by-Step Guide!



CHARLIE MASTERSON

JavaScript:

Advanced Guide to Programming Code with JavaScript

Charlie Masterson

Table of Contents

Introduction

I want to thank you and congratulate you for owning my book, "*JavaScript: Advanced Guide to Programming Code with JavaScript*".

This book contains a whole set of proven steps and strategies to help you learn more advanced JavaScript code. I must stress the word "Advanced" here – this book is not for you if you have not learnt the basics of JavaScript; it is aimed at those with prior knowledge and experience who want to further their knowledge and become better JavaScript programmers.

You can go a long way in JavaScript just by using the basics but if you want to make the most of the power that JavaScript has to offer you need to learn a few more advanced concepts and techniques. These are designed to help you write code that is more expressive and can be reused. These are patterns that define a few features that you might not find in other computer programming languages and that is what makes this such a unique and powerful language.

If this is your first computer programming language, you will more than likely find some of the patterns a little on the strange side but, with the explanations and the examples I have included, as well as me exercises for you to do (With the answers!), you will soon become familiar with it. You will find some of the chapters in this book are quite short while others are somewhat longer. I have tried to keep everything as concise as possible to make it easier for you to take it all in. You won't find any long introductions in each chapter either; you should be at a level now where I can jump straight in and explain as I go along!

Thank you again for reading my book; I truly hope that you find it useful and I wish you luck as you work your way through these advanced JavaScript techniques.

Chapter 1: Optional Function Arguments

When a function is declared in JavaScript, the function expects would usually have an argument list in it:

Example:

CODE:

```
        function sumValues(val1, val2, val3) {
        return val1 + val2 + val3;
        }
```

Unfortunately, this won’t guarantee that your function is going to be called, every time, with three arguments. It is OK for the function to be passed with less or more than three:

Example:

```
var result1 = sumValues(3, 5, 6, 2, 7);
var result2 = sumValues(12, 20);
```

Both of these calls are going to result in some very surprising results from the perspective of the caller.

In the first example, as we are only expecting three arguments, 2 and 7, which are the extra values, are going to be ignored. This isn't good because the value that is returned won't be what was expected by the calling code.

The second example really isn't any better; in fact, it's worse. In this example, we only passed two arguments so what happened to val3? The value of val3 will be undefined and the result of this is the sum of NaN which really isn't what we wanted so we need to fix this so that this type of situation can be avoided in the future:

Example:

```
function sumValues(val1, val2, val3) {
if (val1 === undefined) {
val1 = 0;
}

if (val2 === undefined) {
val2 = 0;
}

if (val3 === undefined) {
val3 = 0;
```

```
    }

    return val1 + val2 + val3;
    }

    var result1 = sumValues(3, 5, 6, 2, 7);
    var result2 = sumValues(12, 20);

    alert(result1);
    alert(result2);
```

If you were to run that again now, you would see that the second function call now no longer results in NaN. Instead, we get the result of 32, most likely what was expected by the calling code.

Now we have a function that will add three numbers but there is still something missing; it isn’t quite right or very useful. At some time, you are going to want to add more numbers and you really don’t want to have to update your code each time so it will accept additional parameters. Fortunately, JavaScript has the answer.

Each function, when it is called, will have a variable that is called arguments. This variable is an array containing every argument that has been passed to the function. So, let’s go back to that example; when you call sumValues for the first time, the array will have [3, 5, 6, 2, 7] and the second time it will have [12, 20].

What we mean by this is that the passed

parameters can be ignored and we can concentrate on dealing with the arguments array only. An update to the function would look like this:

Example:

```
        function sumValues() {
        var sum = 0;
        for (var i = 0; i < arguments.length; i++)
{
        sum += arguments[i];
        }
        return sum;
        }

        var result1 = sumValues(3, 5, 6, 2, 7);
        var result2 = sumValues(12, 20);
        alert(result1);
        alert(result2);
```

Did you spot that we removed the parameter list? Now the values come straight from the array and, when the example is run again, the returned values are right and are exactly what the calling code expected.

What we have now is a full function that will accept any number of parameters that you throw at it and it will always return the value as the sum of all arguments.

Chapter 2: Truthy and Falsy

As you already know, JavaScript contains a Boolean data type. This data type only has two values – true or false. Boolean expressions, for example, a == b, will also evaluate to Boolean values but that isn’t all you need to know.

When you use a non-Boolean expression in a Boolean context, for example, an if condition, those expressions will implicitly convert to Booleans, in a process called type coercion. For example, if you have the expression **if (1) {alert('Hi')}**, 1 would be interpreted as true, meaning that the alert will appear. Value 1 is known as truthy while value 0 is falsy.

When we use that in a context that is expecting a Boolean value, you can use any JavaScript expression:

Example:

```
var truthies = [
        "false",
        "0",
        -1,
        "null",
        "undefined",
        "NaN",
        5/0,
        Infinity
];

var foo;
var falsies = [
        null,
        undefined,
        foo,
        0,
        NaN,
        ""
];


        for (var t=0; t<truthies.length; t++) {
        document.write("<li>" + truthies[t] + " :
" + Boolean(truthies[t]) + "</li>");
```

```
        }


        for (var f=0; f<falsies.length; f++) {
        document.write("<li>" + falsies[f] + " : "
+ Boolean(falsies[f]) + "</li>");
        }
```

Go to a browser and open that page above; what do you see? You will notice ow all the different values are evaluated when they are implicitly converted into Booleans.

Below is a list of how the different values will be converted:

Value	**Truthy or Falsy**	**Explanation**
0	falsy	0 is always falsy
“0”	truthy	all non-sera strings are

		always truthy
-1	truthy	any number that is not 0 is truthy
null	falsy	null is always falsy
“null”	truthy	another non-zero string, therefore truthy
undefined	falsy	undefined is always falsy
“undefined”	truthy	non-zero string, truthy
NaN	falsy	NaN is always falsy
“NaN”	truthy	non-zero string
Infinity	truthy	a reserved keyword and non-zero
5/0	truthy	evaluates to Infinity
““(empty string)	falsy	Zero strings are always falsy
“false”	truthy	Non-zero string

The reason we have the strictly or triple

= (===) comparison operator is type coercion. ==, which is the regular equality operator, will apply type coercion and, on occasion, your comparisons will not give you the result that you expect. Have a look at this example:

Example:

```
var num = 0;
if (num == "") {
        alert('Hey, I did not expect to see this.');
}
if (num === "") {
        alert('This will not be displayed.');
}
```

We have compared two falsy values in conditional one and they will both be resolved, through type coercion, to false. This makes the comparison result as true, most likely not what the intent of this code was.

In order to detect the differences in type, i.e. between string and number, you would use the === operator, as you can see in the second if statement.

Type Coercion

This is an exercise for you, it should take about 5 to 10 minutes to complete. You are going to debug a simple script and fix it:

- In your editor, open this page: AdvancedTechniques/Exercises/type-coercion.html
- Work out what is wrong with the

code
- Make the required fixes
- Test your solution in your browser

Solution:

This is what you should have got:

```
var answer = prompt("What is 10-10?","");
if (answer === "0") {
        alert("Right!");
} else {
        alert("Wrong!");
}
```

You probably found that a bit harder than you thought it would be, In the original code, if the user were to press on OK on their prompt without inputting anything, the alert would be “Right!”. This is because it is an empty string – remember, empty strings and 0 are both falsy.

So what do you need to do? Initial thoughts might be that you just change == to === as a way of differentiating between types but if you were to do that and test it, the alert will always read “Wrong!” even if you used 0.

The problem here is that the value in the prompt is actually a string and string “0” doesn’t strictly equal to number 0. The answer is to change the condition so that it says answer===”0”.

Chapter 3: Default Operators

&& and || are Boolean operators that use truthy and falsy to resolve the operands to Boolean values. If you have used other programming languages, you will probably be thinking that Boolean operator results are always true or false. In JavaScript, that isn't strictly true.

With the JavaScript language, Boolean operation results are always the value of the particular operand that was the determining factor in the operation result. Let me make that a little clearer:

Example:

```
var a = 0, b = NaN, c = 1, d = "hi";
var result = ( a || b || c || d );
alert("Result: " + result);

result = ( d && c && b && a );
alert("Result: " + result);
```

(a || b || c || d) is the first Boolean expression and it will be evaluated left to right until it reaches a conclusion. || is a Boolean OR operator and this only requires that one operand is a truthy for the entire result to be true. In the case of the example above:

- a = 0 and zero is always falsy. The valuation will continue with the rest of the operands because falsy isn't yet determining anything. Only one truthy value is required for the entire statement to be true

- b is Nan, which is also falsy. The same situation occurs and we must continue to evaluate
- c is 1 and this is truthy. From here on, we have no need to evaluate any more operands because the expression is going to result in true.

There is a catch though – have you worked it out? Instead of a strictly true result, we will get a truthy value that stopped the evaluation, in the case of this example, c. the displayed message will read "Result: 1"

You may already expect that the And operator, && works in pretty much the same way but using opposite conditions. This operator will return the value of the last operand that was evaluated and that will be either the first falsy or, where all the operands are truthy, the final one in the expression. Let's say that we follow the sequence for the expression (d && c && b && a) that we did in the case of the OR operator. You will see that both d and c are truthy so you must keep on going; when you get to b, a falsy, the evaluation will halt and the return will be b with the displayed message reading "Result: NaN".

You might be sitting there thinking, what use is any of this? The behavior you have seen here, the return of the first value that is conclusive, is a very handy behavior to ensure the initialization of a variable.

Look at the example of a function below:

```
function checkLength(text, min, max){
        min = min || 1;
        max = max || 10000;

        if (text.length < min || text.length > max)
{
        return false;
        }
        return true;
}
```

In line one and two of the example, we have made sure that a valid value is always assigned to min and max.

Because of this, the function can be called like checkValue("abc") . The parameters for both the min and max will start with the value undefined.

When we get to min = min || 1; all we are doing is assigning a value of 1 to min, just to make sure that it will override undefined, In the same way, max is assigned a value of 10000.

Were we to have passed the real values for the parameters, as in the example checkLength("abc", 2, 10) these values would have been retained as they are truthy.

When we use the || operator, we are, in effect giving default values to the parameters and that is why we call the operator the default operator, at least in this context. The operator is used to replace longer code like:

```
if (min === undefined) {
        min = 1;
}
```

To become:

```
min = min || 1;
```

Have a look at another example of how to use the default operator to shorten your code. This code

```
var contactInfo;
if (email) {
        contactInfo = email;
} else if (phone) {
        contactInfo = phone;
} else if (streetAddress) {
        contactInfo = streetAddress;
}
```

Can be significantly shortened to:

```
var contactInfo = email || phone || streetAddress;
```

Note: do be careful when you use this operator with a variable that accepts falsy values. Look at this code example to see where the danger lies:

Example:

```
function calculatePrice(basePrice, tax) {
        basePrice = basePrice || 0;
        tax = tax || .1; //default tax is 10%
        var price = (basePrice * (1 + tax)).toFixed(2);
        alert("The price including tax is " + price);
}

calculatePrice(100, 0); //no tax
```

The function `calculatePrice()` is using the default operator as a way setting a default tax of 1. The problem arises when 0 is passed as the tax value. Your intent with the code is an indication of no tax but, as 0 is falsy, `tax = tax || .1` will evaluate to .1.

This can be fixed by changing `tax = tax || .1` to `tax = (tax === 0) ? 0 : tax || .1;`.

Applying Defaults to Function Parameters

This exercise should take you about 5 to

15 minutes to complete. We are going to go back to an example we used earlier and use || as a way of handling optional parameters.

- Open AdvancedTechniques/Exercises/sum defaults.html in your editor
- Edit the sumValues() to make use of the || operator instead of using if blocks

Example:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Sum all numbers, default
operator</title>
```

```
<script type="text/javascript">
        function sumValues(val1, val2, val3) {
        if (val1 === undefined) {
        val1 = 0;
        }
        if (val2 === undefined) {
        val2 = 0;
        }
        if (val3 === undefined) {
        val3 = 0;
        }
        return val1 + val2 + val3;
        }
        var result1 = sumValues(3, 5, 6, 2, 7);
        var result2 = sumValues(12, 20);
        alert(result1);
        alert(result2);
```

```
</script>
</head>
<body>
<p>Nothing to show here.</p>
</body>
</html>
```

The Solution:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Sum all numbers, default operator</title>
<script type="text/javascript">
        function sumValues(val1, val2, val3) {
        val1 = val1 || 0;
        val2 = val2 || 0;
        val3 = val3 || 0;
        return val1 + val2 + val3;
        }
        var result1 = sumValues(3, 5, 6, 2, 7);
        var result2 = sumValues(12, 20);
        alert(result1);
        alert(result2);
</script>
</head>
<body>
<p>Nothing to show here.</p>
</body>
</html>
```

Chapter 4: Functions Passed as Arguments

Functions in JavaScript are first-class objects, not just a block of immutable code that is only able to be invoked. Instead, every function that is declared then becomes an object, complete with its own methods and properties, and these can be passed in the same way as any object can.

Have a look at the following example to see how a function can be used as a parameter to another function:

Example:

```
        //let's make an array that contains 5 values
        var values = [5, 2, 11, -7, 1];
        //this function will add the values of 'a' and 'b' and will return the sum
        function add(a, b) {
        return a+b;
        }
        //this function will multiply the value of 'a' times 'b' and will return the result

        function multiply(a, b) {
        return a*b;
        }
        /* the 1st param passed is the numbers
(in this case, the array of values)
        * the 2nd param passed is the initial
Value (could be 0, 1, or whatever)
        * the 3rd param passed is the operation -
which will be one of the functions that is
declared above (add or subtract)
        */
        function combineAll(nums, initialValue,
operation) {
        //let's initialize the local variable
'runningResult' with the initialValue passed to it
        var runningResult = initialValue;
        //loop the nums array
        for (var i=0; i < nums.length; i++) {
        /* for each iteration, call the function
(which can be add or multiply) passing the 2
params
        * on the first iteration, runningResult is the initialValue; after that, value returned from operation(runningResult, nums[i])
        * nums[i] refers to the current number in the iteration; the 1st is 5, then 2, and so on.
        */
        runningResult = operation(runningResult, nums[i]);
        }
        return runningResult;
        }
        //notice the 3rd param passes the "add" function - you can pass a function into another function!
        var sum = combineAll(values, 0, add);
        //notice the 3rd param passes the "multiply" function - you can pass a function into another function!
        var product = combineAll(values, 1, multiply);
        alert("Sum: " + sum); //should be 12: 5+2+11+(-7)+1
        alert("Product: " + product); //should be -770: 5x2x11x-7x1
```

You might be wondering what this function means - `var sum = combineAll(values, 0, add);`. In the above example, the function add is being passed as parameter three of `combineAll`. At this moment, we are not going to invoke add, merely pass a reference. Did you spot that the parentheses were not used after add? That should be a little tip that we are not invoking any function.

The code line that will invoke add

reads:

```
runningResult = operation(runningResult, nums[i]);
```

A reference to add the operation parameter was received by this and, when the operation is invoked, you will see that add is being called and it returns the value of both of the sums that were passed to it.

This is a vital technique and we often call the `combineAll` function to reduce. Take some time to look over the code and run it until you are comfortable with it. We are going to be using this in the following chapters so you need to understand it.

Anonymous Functions

Looking back at the example we used before, the add and multiply functions were only referred to on one occasion, in the call to combineAll . If you were to stick with that pattern, you would need to make a new function for every new combination desire that you wanted, for example, concatenation, just so you could pass it to combineAll . Doesn't that seem like rather a complicated way of doing something that is actually quite simple?

You will be pleased to learn that you don't have to declare each function. In fact, you don't even need to come up with any names for them. In JavaScript, we can create functions on the fly, whenever you need one that is only going to be used in that one location. Even better, the syntax used is quite simple:

```
function (arg1, arg2) {
        //function statements here
        }
```

Because these functions are not named, we call them anonymous functions.

Let’s go back to a previous example and use these anonymous functions to take the place of the single-use functions:

Example:

```
        var values = [5, 2, 11, -7, 1];
        function combineAll(nums, initialValue,
operation) {
        var runningResult = initialValue;
for (var i=0; i < nums.length; i++) {
runningResult = operation(runningResult, nums[i]);
}
return runningResult;
}
var sum = combineAll(values, 0, function (a, b) {
return a+b;
});
var product = combineAll(values, 1, function (a, b) {
return a*b;
});
var list = combineAll(values, 1, function (a, b) {
return a + ", " + b;
});
```

```
        alert("Sum: " + sum);
        alert("Product: " + product);
        alert("Number List: " + list);
```

There are three anonymous functions in here; the first and second ones replace where the add and multiply functions used to be defined and the third is used as a way of concatenating integers into a list, each separated by a comma.

Chapter 5: Nested Functions

As JavaScript functions are another type of JavaScript object, we are able to make a function that is inside another function. We call these nested functions, sometimes known as inner functions. Have a look at the example that shows you how to create one function and then use it inside another function:

Example:

```
        var values = [5, 2, 11, -7, 1];
        function combineAll(nums, initialValue,
operator) {
        var runningResult = initialValue;
        for (var i=0; i < nums.length; i++) {
        runningResult = calculate(nums[i]);
        }
        return runningResult;
        function calculate(num) {
        switch (operator.toLowerCase()) {
        case "+":
        return runningResult + num;
        case "x":
        return runningResult * num;
        case "*":
        return runningResult * num;
        case "a" :
        return runningResult + ", " + num;
        }
        }
        }
        var sum = combineAll(values, 0, "+");
        var product = combineAll(values, 1, "x");
        var list = combineAll(values, 1, "a");
        alert("Sum: " + sum);
        alert("Product: " + product);
        alert("Number List: " + list);
```

You are going to see more of these but, for now, take note of the way in which calculate() accesses both operator and runningResult, both of which have been scoped to the function, combineAll() .

Chapter 6: Variable Scope

JavaScript variables are declared with the use of the var keyword and they will be either locally or globally scoped. To be globally scoped, a variable must meet any of the following conditions:

- The variable has been declared outside a function
- The variable has been used without the use of the var keyword to declare it

To be locally scoped, a variable has to be declared in another function with the var keyword.

Global variables can be accessed from anywhere in the JavaScript code:

Example:

```
        var maxNum; //global
        function prepare() {
        var maxNum = 5; //local to function
        //let's forget the "var" in the "for"
declaration
        for (i=0; i <= maxNum; ++i) { //no var,
so i will be global
        //do something
        }
        }
        prepare();
        alert(maxNum); //undefined
        alert(i); //6 (because it is globally
scoped)
```

Note that maxNum is a globally declared variable but has not got a value assigned to it. It has been declared inside the prepare() function and has been given a value of 5.

Have a look at the maxNum value after prepare() , has been called. The value has been left as undefined. The reason for this is because the global variable is being rea and we can't have access to the function variable from the outside of the function.

We can, however, access variable i, simply because we gave it a value inside the function without using var to declare it.

Because there is no block scope within JavaScript, variables will never be local to a for loop or an if statement.

Chapter 7: Observing and Capturing Events

Most likely you are more used to using HTML event handlers, such as "on" attributes, as a way of capturing events – see the example below:

Example:

```
<ul>
        <li
onclick="document.bgColor='red';">Red</li>
        <li
onclick="document.bgColor='orange';">Orange
        <li
onclick="document.bgColor='green';">Green</
```

```
    <li
onclick="document.bgColor='blue';">Blue</li>
</ul>
```

However, it is far better to keep HTML and JavaScript separate from one another.

The `eval()` Function

Why are we talking about this function here? It is to simply acknowledge that it exists and to tell you NOT to use it. I will tell you why but first you need to know what it does.

eval is used to compile a string and execute it, the string will contain JavaScript code and it can be something very simple, an expression like “2 + 3” or it can be a longer more complex script, with all sorts of functions and other things in it.

Look at the following example: this is similar to what you might find on the internet in live web sites:

Example:

```
        function getProperty(objectName, propertyName) {
        //expression will be: document.title
        var expression = objectName + "." + propertyName;
        console.log(expression);
        //the eval() function will evaluate the expression, so document.title gives us the title of the document
        var propertyValue = eval(expression);
        return propertyValue;
        }
        var prop = "title"; //assume that this was given by the user
        alert(getProperty("document", prop));
```

This function will come up with an expression by taking an object name and concatenating it with a dot and the name of a property. It then evaluates the expression using `eval()` . While it is an incredibly powerful function, it also has the potential to be very dangerous and is not terribly efficient. Why is it dangerous? Because it usually used as way of evaluating input from a user and this isn’t always the safest thing to do. The inefficiency side comes from the fact that every call to `eval()` will start a

JavaScript compiler.

Using the function normally indicates a real lack of knowledge on the part of the developer. In the above example, we could have gone down the route of using the [] property accessor. We would have got the same from alert(window[prop]); without the need to start the compiler just to get the value of the property.

Remember: avoid using eval as much as you possibly can because it will only cause problems in your code.

Chapter 8: Error Handling

It doesn’t matter how carefully you write your code, errors will always appear. Mostly, they will be runtime errors that arise from things you can’t predict. Other times, they are bugs or your code not behaving as it is supposed to. Fortunately, JavaScript provides you with the tools you need to deal with both types of error and that is what we are

going to talk about in this chapter.

Runtime Errors

Web browsers can be incredibly hostile places and it is pretty much guaranteed that you will be forever dealing with runtime errors. Users put in wrong inputs in ways you truly didn’t think was possible. A new version of the browser can change the way things work and Ajax calls could fail for many different reasons.

Much of the time, there is nothing you can do to stop a runtime error but there are ways to deal with them to make things a little less traumatic for the user.

Unhandled Errors:

Have a look at this example, it seems trivial:

Example:

```
function getInput() {
        var name = prompt('Type your name');
        alert('Your name has ' + name.length + '
letters.');
}
getInput();
```

Look at the code properly; it might not be that obvious to begin with but this is a bug waiting to happen. If a user were to click or press ESC or Cancel, a null will be returned by the prompt() function and this will give the net line a null reference error and make it fail.

If the developer doesn't do anything about this, the end user will get a browser message that means absolutely nothing to them. Depending on how the user has configured their settings or browser, they may not get an on-screen message; it might show up as a small icon in the status bar, an icon which could be missed easily and leave users believing that the application is just not responding.

Globally Handed Errors:

There is an event known as on error in a window object and this might be invoked whenever an unhandled error shows up on the page. For example:

Example:

```
window.onerror = function (message, url,
lineNo) {
        alert(
        'Error: ' + message +
        '\n Url: ' + url +
        '\n Line Number: ' + lineNo);
        return true;
}

function getInput() {
        var name = prompt('Type your name');
        alert('Your name has ' + name.length + '
letters.');
}
getInput();
```

As you can see from this example, the event passes three arguments to the function. The first is the error message, the second is the URL of the file that has the error in it, very useful if this is an external .js file, and the third shows the line number that the error is on in that specific file.

The return true is telling the browser that the developer has dealt with this; if false were returned, the browser would treat this as an unhandled error and the on-screen or status bar message would show up on the user's screen.

Structured Error Handling

As with anything, the very best way to deal with an error is to detect it as near to where it occurs as you can. This will make it easier for you to know what to do with it. JavaScript will use try…catch…finally block to implement structured error handling:

```
try {
        //try statements
} catch (error) {
        //catch statements
} finally {
        //finally statements
}
```

This is a very simple premise; if the error occurs in statements that are in the try block, the statements in the catch block are executed and the error is passed straight to the error variable. The finally block is an optional one and, if it used, it will always be executed last whether there is an error or not.

What can we do to make our code catch the error?

Example:

```
window.onerror = function (message, url, lineNo) {
        alert(
        'Error: ' + message +
        '\n Url: ' + url +
        '\n Line Number: ' + lineNo);
        return true;
}

function getInput() {
        try {
```

```
        var name = window.prompt('Type your name');
        alert('Your name has ' + name.length + ' letters.');
        } catch (error) {
        alert('The error was: ' + error.name +
        '\n The error message was: ' + error.message);
        } finally {
        //do cleanup
        }
}
getInput();
```

There are two important properties in the error object – message and name. Message has the error message that we already know about while name contains information about what type of error it was. We use this information to determine what to do about the error.

One piece of good advice for any programmer is to handle an error on the spot only if you are 100% certain you know what it is and that you know exactly how to fix it, rather than suppressing it. To target the handling code better, it can be changed just to handle errors that are called TypeError – this is the name of the error that has been identified for the bug:

Example:

```
window.onerror = function (message, url, lineNo) {
        alert(
        'Error: ' + message +
        '\n Url: ' + url +
        '\n Line Number: ' + lineNo);
        return true;
}

function getInput() {
        try {
        var name = window.prompt('Type your
name');
        alert('Your name has ' + name.length + '
letters.');
        } catch (error) {
        if (error.name == 'TypeError') {
        alert('Please try again.');
        getInput();
        } else {
        throw error;
        }
        } finally {
        //do cleanup
        }
}
getInput();
```

If a different error were to occur, not likely in this simple example, it would not be handled. The throw statement will send the error on just as though we hadn't used try…catch…finally block. The error will appear later on down the line.

Throwing Custom Errors

Th throw statement can be used to throw up custom errors and there is only one recommendation here – the error object must have name and message properties that are consistent with the built-in handling method. Look at this example:

```
throw {
        name: 'InvalidColorError',
        message: 'The given color is not a valid
color value.'
};
```

Chapter 9: ‘delete’ Operator

We use the ‘delete’ operator as way of deleting object properties and array elements. Be very clear on this – deleting is not the same as setting a value to null. Deleting removes the element or property, making it undefined, while setting as value as null will retain the element or object while setting it a value of null. Have a look at this example as an illustration:

Example:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<script type="text/javascript">
        var colors = [];
        colors["red"] = "#f00";
        colors["green"] = "#060";
        colors["blue"] = "#00f";
        colors["yellow"] = "#ff0";
        colors["orange"] = "#0ff";
        function deleteItem() {
```

```
        delete colors["blue"];
        showArray();
        revealArray();
        }
        function setToNull() {
        colors["blue"] = null;
        showArray();
        revealArray();
        }
        function showArray() {
        var                  output                  =
document.getElementById("colors");
        var strOutput = "<ol>";
        for (var i in colors) {
        strOutput    +=    "<li    style='color:"    +
colors[i] + ";'>" + i + "</li>";
        }
        strOutput += "</ol>";
        output.innerHTML=strOutput;
        }
        function revealArray() {
        var                 colorsCode                    =
document.getElementById("colors-code");
        var strOutput = "&lt;ol&gt;";
        for (var i in colors) {
        strOutput+="\n\t&lt;li   style='color:"   +
colors[i] + ";'&gt;" + i + "&lt;/li&gt;";
        }
        strOutput += "\n&lt;/ol&gt;";
        colorsCode.innerHTML=strOutput;
        }
</script>
<title>Delete vs. Setting to NULL</title>
</head>
```

```
<body onload="showArray();revealArray();">
<button id="set-to-null" onclick="setToNull();">Set Blue to Null</button>
<button id="del" onclick="deleteItem();">Delete Blue</button>
<output id="colors"></output>
<pre id="colors-code"></pre>
</body>
</html>
```

Forget about the functions `showArray()` and `revealArray()` fc now. Do note however that when the button for Set Blue to Null is clicked, the array element will stay put but will be an invalid color of null. When Delete Blue button is clicked, the array element will be deleted.

Regular Expressions

This is a quick introduction to the next few chapters, where we go over regular expressions. These are used for pattern matching and this can be very helpful when it comes to form validation. For example, you can use a regular expression to check that an email address that has been input into a form filed is correct in terms of syntax. You can create regular expressions in two ways in JavaScript:

Literal Syntax:

```
var reExample = /pattern/;
```

RegExp() Constructor

```
var reExample = new RegExp("pattern");
```

If you already know the pattern you are

going to use, there isn’t any real difference between these. However, if you don’t know which patter you are going to use, the easiest one is the RegExp() constructor.

Chapter 10: Regular Expressions – Methods

The JavaScript regular expression method contains two main methods for string testing: test() and exec().

The exec() Method

This method takes an argument and a string and checks to see if there are one or more matches in the string to a pattern that is specified in the regular expressions. If the matches are found, a result array is returned by the method showing the starting point of each match. If there are no matches, null is returned.

The test() Method

This method will also take a string and an argument and check the string for pattern matches. If there is a match, true will be returned; if not, false is returned. This is incredibly useful in form validations scripts.

The following example show how to use this to test a social security number. Ignore the regular expression syntax for now, we will be looking at that in the next chapter:

Example:

RegularExpressions/Demos/SsnChecker.html

```
<!DOCTYPE HTML>
<html>
<head>
```

```
<meta charset="UTF-8">
<title>ssn Checker</title>
<script type="text/javascript">
var reSSN = /^[0-9]{3}[\- ]?[0-9]{2}[\- ]?[0-9]
{4}$/;

function checkSsn(ssn){
        if (reSSN.test(ssn)) {
        alert("VALID SSN");
        } else {
        alert("INVALID SSN");
        }
}
</script>
</head>
<body>
        <form onsubmit="return false;">
        <input type="text" name="ssn"
size="20">
        <input type="button" value="Check"
        onclick="checkSsn(this.form.ssn.value);
        </form>
</body>
</html>
```

Let’s take a closer look at the code.

First, we have declared a variable that contains a regular expression object relating to a social security number

Second, we have created a function called checkSsn(). This function takes an argument, ssn, which is also a string, and passes it to the regular expression to see if there is a match. For this, the test() method is used. If there is a match, the function will alert “VALID SSN”. If there is no match, “INVALID SSN” is alerted instead.

Lastly, a form inserted in the page body provides a field for a user to input a social security number along with a button that will pass the number to the checkSsn() function.

Flags

If a flag appears after the end slash it will modify the way the regular expression works. The i flag will turn the case of a regular expression insensitive. For example, /aeiou/i will match both uppercase and lowercase vowels.

The g flag will specify a global match

and that means that all matches of the pattern that has been specified are to be returned.

String Methods

A number of String methods will take a regular expression as an argument:

The search() Method

This method will take just one argument – a regular expression. The return is the index of the initial character of the substring that matches the regular expression. If there is no match, -1 is returned.

The split() Method

This method also take a regular expression and uses it as a delimiter to split one string into an array.

The replace() Method

This method will take two arguments – the regular expression and the string. The first regular expression match is replaced with the string and, if the g flag is used in the expression, all string matches will be replaced.

The match() Method

The match() method takes the regular expression as an argument and returns each of the substrings that are a match to the pattern of the regular expression.

Chapter 11: Regular Expressions – Syntax

Regular expressions are patterns that are used to specify a list of characters. In this chapter, we are going to look the syntax used to specify the characters:

Start and End (^ $)

- If there is a caret (^) at the start of a regular expression, it is an indicator that the string that is being researched has to begin with this pattern
- You will find pattern ^ in "food" but

you won't find it in "barfood"

- If a regular expression has a dollar ($) sign at the end of it, it is an indication that the string that is being searched has to end with the pattern
- You will find pattern foo$ in "curfoo" but you won't find it in "food"

Number of Occurrences (? + * {})

These symbols will affect how many occurrences there are of the preceding character or, if you use parenthesis, characters - ?, *, +, {}

- The question mark (?) is an indication that the preceding character must appear 0 or 1 times in the pattern

- You will find pattern foo? In “fod” and “food” but not in “faod”
- The plus sign (+) is an indication that the preceding character will appear at least once in the pattern
- You will find pattern fo+ in “food”, “fod” and “foood” but you won’t find it in “fd”
- The asterisk (*) is an indicator that the preceding character has to appear at least 0 times in the pattern
- You will find pattern fo*d in “fd”, “food” and “fod”
- The curly brackets ({}), with a single parameter {n} is an indicator that the preceding character must appear in the pattern exactly n times

- You will find pattern fo{3}d in “foood” but you won’t find it in “fooood” or in “food”
- The curly brackets with two parameters {n1, n2} is an indicator that the previous character should appear in the pattern between n1 and n2 times
- You will find pattern fo{2,4} in “fooood”, “foood” and in “food” but you won’t find it in “foooood”
- Curly brackets that have one parameter with a second empty one {n} is an indicator that the previous character must appear in the pattern at least n times.
- You will find pattern fo{2} in

"foooood" and in "food" but you won't find it in "fod"

Common Characters (. \d \D \w \W \s \S)

- A period (.) is representative of any character except for a newline
- You will find pattern fo.d in "foad", "food", "fo*f" and in "fo9d"
- The backslash-d (\d) is representative of any digit and is equivalent to [0-9]
- You will find the pattern fo\dd in "fo1d", fo0d", fo4d" but you won't find it in "fodd" or in "food"
- The backslash-D (\D) is representative of any character with the exception of a digit and is the

equivalent of [^0-9]

- You will find the pattern fo\D is "foad" and in "food" but you won't find it in "fo4d"
- The backslash-w is representative of any word character, such as digits, letters and underscore (_).
- You will find the pattern fo\w in "fo_d", "food" and in "fo4d" but you won't find it in "fo*d"
- The backslash-W (\W) is representative of any character that isn't a word character.
- "You can find the pattern fo\Wd in "fo&d", in "fo*d) and in "fo.d" but you won't find it in "fo d"
- The backslash-s (\s) character is

representative of any whitespace character, such as tab, space, newline, etc.

- You will find pattern fo\sd in “fo d” but you won’t find it in “food)
- The backslash-S (\S) is representative of any character except for a whitespace
- You can find the pattern fo\Sd in “food”, in “fo*d” and in “fo4d” but you won’t find it in “fo d”

Grouping ([])

We use square brackets ([]) as a way of grouping options together:

- You will find the pattern f[aeiou] in “fed” and “fad” but you won’t find it in “food", "faed" or "fd".

- The pattern [aeiou] will match with "a", "e", "i", "o", or "u"
- You will find pattern f[aeiou]{2}d in "faed" and in "feod", but you won’t find it in in "fod", "fed" or "fd".
- You will find the pattern [A-Za-z]+ in "Webucator, Inc.", but you won’t find it in "13078".
- The pattern [A-Za-z] will match with any letter, regardless of whether it is upper or lowercase
- The pattern [A-Z] will match any letter that is uppercase
- The pattern [a-z] will match any letter that is lowercase

Negation (^)

When the caret (^) is used within square brackets and is the first character, it is indicative of negation

- You will find pattern f[^aeiou]d in "fqd" and "f4d", but you won't find it in "fad" or "fed".

Subpatterns (())

We use parentheses () to capture subpatterns

- You will find pattern f(oo)?d in "food" and "fd", but you won't find it in "fod".

Alternatives (|)

We use the pipe (|) as a way of creating optional patterns

- You will find pattern foo$|^bar in "foo" and "bar", but you won’t find it in "foobar".

Escape Character (\ )

We use the backslash (\ ) as a way of escaping special characters.

- You will find pattern fo\.d in "fo.d", but you won’t find it in "food" or "fo4d".

Chapter 12: Regular Expressions – Form Validation

When you use regular expressions, you can create some incredibly powerful functions that are for form validation. Have a look at this example:

Example:

RegularExpressions/Demos/Login.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Login</title>
```

```
<script type="text/javascript">
var  reEmail  =  /^(\w+[\-\.])*\w+@(\w+\.)+[A-
Za-z]+$/;
var rePassword = /^[A-Za-z\d]{6,8}$/;

function validate(form){
        var email = form.Email.value;
        var password = form.Password.value;
        var errors = [];
        if (!reEmail.test(email)) {
        errors[errors.length] = "You must enter
a valid email address.";
        }
        if (!rePassword.test(password)) {
        errors[errors.length] = "You must enter
a valid password.";
        }
        if (errors.length > 0) {
```

```
        reportErrors(errors);
        return false;
        }
        return true;
}

function reportErrors(errors){
        var msg = "There were some
problems...\n";
        for (var i = 0; i<errors.length; i++) {
        var numError = i + 1;
        msg += "\n" + numError + ". " +
errors[i];
        }
        alert(msg);
}
</script>
```

```
</head>
<body>
<h1>Login Form</h1>
<form method="post" action="Process.html"
onsubmit="return validate(this);">
        Email: <input type="text" name="Email"
size="25"><br/>
        Password: <input type="password"
name="Password" size="10"><br/>
        *Password must be between 6 and 10
characters and
        can only contain letters and digits.<br/>
        <input type="submit" value="Submit">
        <input type="reset" value="Reset
Form">
        </p>
</form>
</body>
```

```
</html>
```

The code example start with the definition of a regular expression for an email address and one for a password as well so let’s look at each one in turn:

```
var reEmail = /^(\w+\.)*\w+@(\w+\.)+[A-Za-z]+$/;
```

- The caret (^) indicates that we should start at the beginning and this stops a user from inputting any invalid characters at the start of the email address.
- The pattern (\w+[\-\.])* indicates an allowance for a sequence of characters (word) that are followed by a dash or a dot. * is used as an indicator that the pattern may be

repeated 0 times or more and successful patters will include: "ndunn.", "ndunn-", "nat.s.", and "nat-s-".

- The pattern \w+ will allow for a minimum of one word character
- @ will allow for one @ symbol.
- The pattern (\w+\.)+ will allow for a sequence of characters (word) that are followed by a dot. + is indicative of the pattern being repeated at least once or more and this is the domain name minus the last bit, for example, .com
- The pattern [A-Za-z]+ will allow for at least one letter and is indicative of the last bit, for

example, .com

- The dollar sign ($) indicates that you should end here. This stops a user from inputting invalid characters at the end of their email address

```
var rePassword = /^[A-Za-z\d]{6,8}$/;
```

- Same as above. The starting caret stops a user from inputting invalid characters at the start of the password
- The pattern [A-Za-z\d]{6,8} will allow for a sequence of digits and letter that is 6 to 8 characters' long
- Same as above, the dollar sign at the end stops a user from inputting an invalid character at the end of a password

Advanced Form Validation

This exercise should take you in the region of 25 to 40 minutes.

First Exercise

- Open RegularExpressions/Exercises/ your editor.

Now write the correct regular expressions (additional ones) that will check for:

1. The proper name
    - That it begins with a capital letter
    - That it is followed by at least one letter or apostrophe
    - It may be more than one word,

for example, “New Mexico”

2. Initial
    - That it has either 0 or 1 capital letter
3. State
    - That it has two capital letters
4. US Postal Code
    - That it is made up of five digits
    - May be followed with a dash and a further four digits
5. Username
    - That it is no less than 6 and no more than 15 digits or letters

Second Exercise

- Open RegularExpressions/Exercises/ your editor.

Add the correct validation for checking

these fields:

- first name
- middle initial
- last name
- city
- state
- zip
- username

Test out your solution in your browser

A Challenge

Add in the correct regular expressions to test United Kingdom and Canadian postcodes:

- United Kingdom postcodes are two letters, one or two numbers, a whitespace, one number and two letters – for example WC12 3XY

- Canadian postcodes are one letter, one digit, one letter, a white pace, one digit, one letter and one digit – for example, M1A 2B3

Next make a modification to Register.html to check a postcode against these regular expressions as well as that of the US postcode.

Solution:

RegularExpressions/Solutions/FormValidation.j

```
//      Regular Expressions

var reEmail = /^(\w+[\-\.])*\w+@(\w+\.)+[A-Za-z]+$/;

var rePassword = /^[A-Za-z\d]{6,8}$/;

var reProperName = /^([A-Z][A-Za-z']+ )*[A-Z][A-Za-z']+$/;
var reInitial = /^[A-Z]$/;
var reState = /^[A-Z]{2}$/;
var rePostalUS = /^\d{5}(\-\d{4})?$/;
var reUsername = /^[A-Za-z\d]{6,15}$/;
```

Solution:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Registration Form</title>
<script                              type="text/javascript"
src="FormValidation.js"></script>
<script type="text/javascript">

function validate(form){
        var firstName = form.FirstName.value;
        var midInitial = form.MidInit.value;
        var lastName = form.LastName.value;
        var city = form.City.value;
        var state = form.State.value;
        var zipCode = form.Zip.value;
        var email = form.Email.value;
        var userName = form.Username.value;
        var password1 = form.Password1.value;
        var password2 = form.Password2.value;
        var errors = [];

        if (!reProperName.test(firstName)) {
        errors[errors.length] = "You must enter
a valid first name.";
        }

        if (!reInitial.test(midInitial)) {
        errors[errors.length] = "You must enter
a one-letter middle initial.";
        }

        if (!reProperName.test(lastName)) {
        errors[errors.length] = "You must enter
a valid last name.";
        }

        if (!reProperName.test(city)) {
        errors[errors.length] = "You must enter
a valid city.";
        }

        if (!reState.test(state)) {
        errors[errors.length] = "You must enter
a valid state.";
        }
        if (!rePostalUS.test(zipCode)) {
        errors[errors.length] = "You must enter
a valid zip code.";
        }

        if (!reUsername.test(userName)) {
        errors[errors.length] = "You must enter
a valid username.";
        }
```

Challenge Solution:

RegularExpressions/Solutions/FormValidation-challenge.js

```
//      Regular Expressions

var rePostalUS = /^\d{5}(\-\d{4})?$/;
var rePostalCA = /^[A-Z]\d[A-Z] \d[A-Z]\d$/;
var rePostalUK = /^[A-Z]{2}[0-9]{1,2} ?[0-9]
```

```
{1}[A-Z]{2}$/;
```

Challenge Solution:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>Registration Form</title>
<script                    type="text/javascript"
src="FormValidation-challenge.js"></script>
<script type="text/javascript">

function validate(form){


        if (!rePostalUS.test(zipCode)
        && !rePostalCA.test(zipCode)
        && !rePostalUK.test(zipCode))
        {
```

```
        errors[errors.length] = "You must enter
a valid postal code.";
        }


        return true;
}
```

Chapter 13: Cleaning Up Form Entries

Sometimes it is nice to clean user entries as soon as they are input and we can do this by using a combination of the replace() method of string objects and regular expressions.

The replace() Method Revisited

We already looked at this method and how to use it to replace a regular expression match with a string but we can also use it together with backreferences and replace a pattern that has been matched with a brand-new string that is made from substrings of that pattern. The following example shows you this:

Example:

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>ssn Cleaner</title>
<script type="text/javascript">
var reSSN = /^(\d{3})[\-   ]?(\d{2})[\-   ]?(\d{4})$/;

function cleanSsn(ssn){
        if (reSSN.test(ssn)) {
```

```
        var cleanedSsn = ssn.replace(reSSN,
"$1-$2-$3");
        return cleanedSsn;
        } else {
        alert("INVALID SSN");
        return ssn;
        }
}
</script>
</head>
<body>
        <form onsubmit="return false;">
        <input type="text" name="ssn"
size="20">
        <input type="button" value="Clean SSN"
        onclick="this.form.ssn.value =
cleanSsn(this.form.ssn.value);">
        </form>
```

```
</body>
</html>
```

We use the cleanSsn() function to clean a social security number in this example. In reSSN, ^(\d{3})[\-]?(\d{2})[\-]?(\d{4})$, the regular expression has three separate subexpressions: (\d{3}), (\d{2}), and (\d{4}). These can be referenced as $1, $2, and $3, respectively in the replace() method.

When a user clicks the button that says "Clean SSN" , they call the cleanSsn() function This function will then test to see of the number input by the user is a valid number and, if it is, it will then clean it using the code below –

this code will dash-delimit the substrings that match the subexpressions:

```
var cleanedSsn = ssn.replace(reSSN, "$1-$2-$3");
```

The properly cleaned social security number will then be returned

Cleaning Up Form Entries

This exercise should take you about 15 to 25 minutes.

- Open RegularExpressions/Exercises/ your editor
- Where it is indicated by the comment, you must declare a variable called cleanedPhone and then assign it a cleaned version of the phone number entered by the user. The clean version should be in

this format – (111) 222 3333
- Test out your solution in your browser

A Challenge

You sometimes find a phone number is a combination of numbers and letters and some of these have an extra character in them to complete the word. Your challenge is to add in a function that is named convertPhone(). This function will:

- Strip out every character that is not a letter or a number
- Convert every letter to a number as per the following list:
    - ABC -> 2
    - DEF -> 3
    - GHI -> 4

- JKL -> 5
- MNO -> 6
- PQRS -> 7
- TUV -> 8
- WXYZ -> 9

It must also:

- Pass the initial 10 characters of the string that results to the function cleanPhone()
- Return the string result
- Then modify the form, calling convertPhone() and not cleanPhone()
- Test the solution in your browser

Solution:

```
<!DOCTYPE HTML>
<html>
```

```
<head>
<meta charset="UTF-8">
<title>Phone Cleaner</title>
<script type="text/javascript">
var rePhone = /^\(?([2-9]\d\d)\)?[\-\. ]?([2-9]\d\d)[\-\. ]?(\d{4})$/;

function cleanPhone(phone){
        if (rePhone.test(phone)) {
        var cleanedPhone = phone.replace(rePhone, "($1) $2-$3");
        return cleanedPhone;
        } else {
        alert("INVALID PHONE");
        return phone;
        }
}
```

```
</script>
</head>
<body>
        <form onsubmit="return false;">
        <input type="text" name="Phone"
size="20">
        <input type="button" value="Convert
Phone"
        onclick="this.form.Phone.value =
cleanPhone(this.form.Phone.value);">
        </form>
</body>
</html>
```

Challenge Solution:

```
<!DOCTYPE HTML>
<html>
```

```
<head>
<meta charset="UTF-8">
<title>Phone Checker</title>
<script type="text/javascript">
var rePhone = /^\(?([2-9]\d\d)\)?[\-\. ]?([2-9]\d\d)[\-\. ]?(\d{4})$/;



function convertPhone(phone){
        var convertedPhone;
        convertedPhone = phone.replace(/[^A-Za-z\d]/g, "");
        convertedPhone = convertedPhone.replace(/[ABC]/gi, "2");
        convertedPhone = convertedPhone.replace(/[DEF]/gi, "3");
        convertedPhone = convertedPhone.replace(/[GHI]/gi, "4");
        convertedPhone                                  =
convertedPhone.replace(/[JKL]/gi, "5");
        convertedPhone                                  =
convertedPhone.replace(/[MNO]/gi, "6");
        convertedPhone                                  =
convertedPhone.replace(/[PQRS]/gi, "7");
        convertedPhone                                  =
convertedPhone.replace(/[TUV]/gi, "8");
        convertedPhone                                  =
convertedPhone.replace(/[WXYZ]/gi, "9");
        return
cleanPhone(convertedPhone.substr(0, 10));
}
</script>
</head>
<body>
        <form onsubmit="return false;">
        <input      type="text"      name="Phone"
```

```
size="20">
        <input type="button" value="Convert
Phone"
        onclick="this.form.Phone.value =
convertPhone(this.form.Phone.value);">
        </form>
</body>
</html>
```

Bonus Chapter: Working with AJAX

AJAX stands for Asynchronous JavaScript and XML and it is a group of technologies, such as DOM, JavaScript, HTML, XML and CSS, that are all inter-related. AJAX lets you send data and receive it asynchronously without having to reload your web page, making it incredibly fast.

AJAX also allows you to send only the most important information to the web server and not the whole page so only the valuable data will be routed over to the client side. This makes any application you build and use a good deal faster and more interactive. AJAX is used on a very large number of web applications, including Facebook, Gmail, Google Maps, Twitter, and too many more to mention.

How Does AJAX Work?

When you use the traditional JavaScript code and you want to get some information out of a server file or a database, or if you wanted to send some information over to a server, you would make an HTML form and you would use POST or GET to send or receive data to and from a data. End users would need to click a button that says SUBMIT to send or get the answer; they would then have to wait while the server responded and then a new page would load up with their results.

Each time input is submitted by a user, a new page is returned by the server and, because of this, some web applications run quite slowly and are not always very user-friendly. When you use AJAX, JavaScript will talk straight to the server via the JavaScript object XMLHttpRequest.

By using an HTTP request, web pages

can request and receive a response form web servers without the need to reload the page. The users stay on the same page and will not have a clue that there are scripts sending requests for pages or sending data to a server as this will be working away in the background.

The user will send a request that will, in turn, execute an action. The response to this action will be shown into a layer, identified via an ID and all without the page having to go through a full reload. This is an example of a page ID:

```
<div id=”ajaxResponse”></div>
```

Next, we are going to look briefly at how to create the XMLHttpRequest object and get responses from the server:

Step 1: Create the object XMLHttpRequest.

Every different browser will use a different method to create this object; Internet Explorer makes use of ActiveXObject while some of the other browsers use XMLHttpRequest, which is, of course, the built-in object in JavaScript.

To create the object and to deal with all the different browsers, we are going to make use of a statement called "catch and try"

```
function ajaxFunction()
{
var xmlHttp;
try
{
```

```
// Firefox, Opera 8.0+, Safari
xmlHttp=new XMLHttpRequest();
}
catch (e)
{
// Internet Explorer
try
{
xmlHttp=new
ActiveXObject("Msxml2.XMLHTTP");
}
catch (e)
{
try
{
xmlHttp=new
ActiveXObject("Microsoft.XMLHTTP");
}
catch (e)
{
alert("Your browser does not support AJAX!");
return false;
}
}
}
```

Step 2: Sending the Request to a Server

To send the request to a server, we will use the methods open() and send()

The first method will take three arguments – the first to define the method used when the GET or POST request is sent, the second to specify the URL of the script on the server side and the third to specify that the request must be asynchronously handled. The second method is used to send the request over to the server.

```
xmlHttp.open("GET","time.asp",true);
xmlHttp.send(null);
```

Step 3: Writing the Server Side Script

responseText is going to store any data that is returned from a server and we want to send the current time back. For this we will use time.asp and the code will look like this:

```
<%
response.expires=-1
response.write(time)
%>
```

Step 4: Consuming the Response

The next step is to consume whatever response comes back and then display it to the end user:

```
xmlHttp.onreadystatechange=function()
{
```

```
if(xmlHttp.readyState==4)
{
document.myForm.time.value=xmlHttp.respons
}
}
xmlHttp.open("GET","time.asp",true);
xmlHttp.send(null);
}
```

Step 5: Completing the Code

The final step is to determine when we should execute the AJAX function. This is going to run in the background whenever a user inputs something into a text field for the username and the entire code would look like this:

```
<html>
<body>
<script type="text/javascript">
function ajaxFunction()
```

```
{
var xmlHttp;
try
{
// Firefox, Opera 8.0+, Safari
xmlHttp=new XMLHttpRequest();
}
catch (e)
{
// Internet Explorer
try
{
xmlHttp=new
ActiveXObject("Msxml2.XMLHTTP");
}
catch (e)
{
try
{
xmlHttp=new
ActiveXObject("Microsoft.XMLHTTP");
}
catch (e)
{
alert("Your browser does not support AJAX!");
return false;
}
}
}
xmlHttp.onreadystatechange=function()
{
if(xmlHttp.readyState==4)
{
document.myForm.time.value=xmlHttp.respons
}
}
xmlHttp.open("GET","time.asp",true);
xmlHttp.send(null);
}
</script>
<form name="myForm">
Name: <input type="text"
```

```
onkeyup=”ajaxFunction();” name=”username”
/>
Time: <input type=”text” name=”time” />
</form>
</body>
</html>
```

That is a basic look at how AJAX works and what it can do. There is plenty of information available on the internet for you to research this further but do ensure you are familiar with advanced JavaScript first.

Conclusion

Thank you again for reading this book!

I hope this book was able to help you to understand and grasp some of the more advanced techniques in JavaScript and I hope that you found the exercises useful – they are designed to test your knowledge and also to help you learn much quicker. It is always better to learn from practical examples, especially with computer program languages, rather than just reading and trying to take it in.

The next step is to practice. It doesn’t matter how much you have read or how much you have learned to date; if you don’t practice you will soon lose it. The nature of computer programming is that things change on a regular basis. Keep up your learning otherwise you will have to go back to basics!

Finally, if this book has given you value and helped you in any way, then I’d like to ask you for a favor, if you would be kind enough to leave a review for this book on Amazon? It’d be greatly appreciated!

Thank you and good luck!

Check Out My Other Books:

Below you’ll find some of my other popular Java books that are popular on Amazon and Kindle as well. Simply click on the links below to check them out.

Alternatively, you can visit my author page on Amazon to see other work done by me.

Preview Of 'JavaScript: Best Practices to Programming Code with JavaScript'

Best JavaScript

Tips for Beginners

Whether you like it or not, as a programmer or web or app developer, you need to learn JavaScript. It is considered the computer language of the future simply because it is an integral part of web ad app development.

Of course, it would be next to impossible to cover every trick to help you learn the best practices for JavaScript. Instead, we want to focus on the main attributes you will come across as you learn to code. In the chapter to follow, we will provide you with some of the simpler best practices to get you on the right path. Remember that there is always more information to learn about coding.

This is just an excellent start!

DO NOT declare Boolean Objects, Strings, or Numbers

DO Initialize Any Variables

DO NOT use newObject ()

DO use Parameter Defaults

DO NOT use Automatic Type Conversions

DO use "Timer" to Optimize Code

DO NOT Pass String to "SetTimeOut" or "SetInterval"

DO use "For in" Statements

DO NOT use "With" Statements

1. DO NOT declare Boolean Objects, Strings, or Numbers

While you are coding, it is very important that you treat your Booleans, strings, and numbers as primitive values. You will not want to see these as objects. If you define any of these codes as objects, it could produce bad side effects and will just slow down the speed of your execution.

Here is a quick example:

```
var x= "Joe";
var y= new String ("Joe");
(x ===y)
```

• As you can see, this will be false due to the fact that x is a string while y is an object. It will throw an error, and this is something you can avoid!

2. DO Initialize Any Variables

Whether you are learning code, or are just looking to fix your skills, it is great practice to initialize your variables as you declare them. When you do this, it will help avoid undefined values, it will provide you with a single place to do this, and it will help give you a cleaner code.

3. DO NOT use newObject()

While this is something we touched on a bit earlier, it is important to remember. Here is a quick cheat sheet to help you remember which code to use compared to its similar code. By being organized, you can keep your code clean.

- DO USE: {} DO NOT USE: new Object()
- DO USE 0 DO NOT USE new Number()
- DO USE " " DO NOT USE new String()
- DO USE false DO NOT USE new Boolean()
- DO USE function () {} DO NOT

USE new Function()

- DO USE /()/ DO NOT USE new RegExp()
- DO USE [] DO NOT USE new Array()

4. DO use Parameter Defaults

We invite you to ask a moment to ask yourself, what do I remember about functions? What do you call it when your function is missing an argument? That's right! The value of that missing argument will be called undefined. If you weren't already aware, these undefined values can and will break your code. To break this habit, try your best to assign values to your arguments automatically.

Here is an example:

```
        function myFunction (x, y) {
        if ( x === undefined) {
        x= 1;
 }
}
```

5. DO NOT use Automatic Type Conversions

While you are coding, you will want to be aware of the numbers that could be converted to NaN or strings. If you need a refresher, NaN stands for Not a Number. This is something that can happen automatically due to the fact that JavaScript is typed loosely. The variables in your code can change the data type as well as contain a few different types of data.

Here is a quick example:

```
var x = "Hey";
```

- In this case, x is a string but if you were to continue;

```
x= 4;
```

- Then, this changes your x to a number. You can see where this could cause errors in your code.

You also should be aware that your JavaScript will convert numbers into strings if you are doing mathematical problems. If you type it the wrong way, it will give you an answer, but it may not be the one you wanted. Here is a quick example:

- In this example, typeof x will be a number

```
var x = 5+ 7;  // x.valueOf() will be 12
```

- In this next example, the typeof x will be a string

```
var x = 5+ "7"; // x.valueOf() will be 57
```

In a different case, you could try to use your function to subtract one string from another string. In this case, it will return as NaN.

Use this example:

- "Goodbye" – "Love" // This will return as NaN

6. DO use "Timer" to Optimize

Code

This practice is super simple. Are there ever times that you need a quick way to determine exactly how long your operation will take? Luckily, Firebug has a "timer" feature that will log the results for you! Sometimes it helps to have some tricks on the side to help your coding along.

Here is the code for that:

```
function TimeTracker () {
  console.time ("MyTimer");
   for (x=4000; x > 0; x--) {}
  console.timeEnd ("MyTimer");
}
```

7. DO NOT Pass String to "SetTimeOut" or "SetInterval"

Remember when we told you not to use the "eval" function? This code will function the same way. This is why we suggest never passing the string to SetTimeOut or SetInterval. Unfortunately, this code is super inefficient.

Instead of this code:

```
setInterval (
"document.getElementById('container').innerHT
_+= 'The New Number: ' + I" , 2000
);
```

Use this one:

```
setInterval(someFunction, 2000);
```

8. DO use "For in" Statements

Sometimes when you loop items in an object, you may have noticed that at the same time, you retrieve the method functions as well. If you do not want this to happen, there is a way you can wrap your code as an if statement. This way, it will filter your information.

Here is an example:

```
for (key in object) {
if(object. hasOwnProperty(key)) {
        //…do something…
   }
}
```

9. DO NOT use "With" Statements

Now, this coding tip is up for debate. On the one hand, "with" statements truly sound like a good idea to use. In retrospect, a "with" statement should be able to be used as a shorthand to access any objects that could be deeply nested in your code.

Here is an example:

```
with (person.woman.bodyparts) {
hand = true;
foot= true;
}
```

• As we noted, you would use these statements instead:

```
person.woman.bodyparts.hand= true;
person.woman.bodyparts.foot= true;
```

After some tests, it was found that these codes behaved badly, especially when setting up new members. Instead of using "with" you can use "var" and get the proper results.

About the Author

Charlie Masterson is a computer programmer and instructor who have developed several applications and computer programs.

As a computer science student, he got interested in programming early but got frustrated learning the highly complex subject matter.

Charlie wanted a teaching method that he could easily learn from and develop his programming skills. He soon discovered a teaching series that made him learn faster and better.

Applying the same approach, Charlie successfully learned different programming languages and is now teaching the subject matter through writing books.

With the books that he writes on computer programming, he hopes to provide great value and help readers interested to learn computer-related topics.