Beginning JavaScript®, Fifth Edition
By Jeremy McPeak and Paul Wilton
Copyright © 2015 by John Wiley & Sons, Inc., Indianapolis, Indiana

4 Functions and Scope

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Creating your own functions
- Identifying, creating, and using global and local variables
- Using functions as a value

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

You can find the wrox.com code downloads for this chapter at `http://www.wiley.com/go/BeginningJavaScript5E` on the Download Code tab. You can also view all of the examples and related files at `http://beginningjs.com`.

A function is something that performs a particular task. Take a pocket calculator as an example. It performs lots of basic calculations, such as addition and subtraction. However, many also have function keys that perform more complex operations. For example, some calculators have a button for calculating the square root of a number, and others even provide statistical functions, such as the calculation of an average. Most of these functions could be done with the basic mathematical operations of add, subtract, multiply, and divide, but that might take a lot of steps—it's much simpler for the user if she only needs to press one button. All she needs to do is provide the data—numbers in this case—and the function key does the rest.

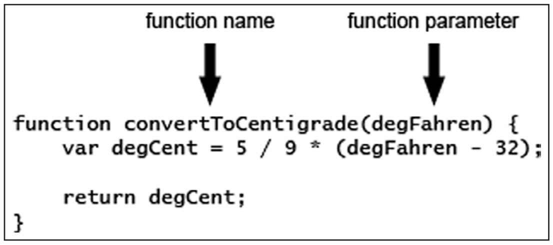
Functions in JavaScript work a little like the function buttons on a pocket calculator: They encapsulate a block of code that performs a certain task. Over the course of the book so far, you have come across a number of handy built-in functions that perform a certain task, such as the `parseInt()` and `parseFloat()` functions, which convert strings to numbers, and the `isNaN()` function, which tells you whether a particular value can be converted to a number. Some of these functions return data, such as `parseInt()`, which returns an integer number;

others simply perform an action but return no data. You'll also notice that some functions can be passed data, whereas others cannot. For example, the `isNaN()` function needs to be passed some data, which it checks to see if it is `NaN`. The data that a function requires to be passed are known as its *parameter(s)*.

As you work your way through the book, you'll be coming across many more useful built-in functions, but wouldn't it be great to be able to write your own functions? After you've worked out, written, and debugged a block of code to perform a certain task, it would be nice to be able to call it again and again when you need it. JavaScript enables us to do just that, and this is what you'll be concentrating on in this section.

CREATING YOUR OWN FUNCTIONS

Creating and using your own functions is very simple. Figure 4-1 shows an example of a function declaration.



```
function convertToCentigrade(degFahren) {
    var degCent = 5 / 9 * (degFahren - 32);

    return degCent;
}
```

FIGURE 4-1

You've probably already realized what this function does and how the code works. Yes, it's the infamous Fahrenheit-to-centigrade conversion code again.

Each function you define in JavaScript must be given a unique name for that particular page. The name comes immediately after the `function` keyword. To make life easier for yourself, try using meaningful names so that when you see them being used later in your code, you'll know exactly what they do. For example, a function that takes as its parameters someone's birthday and today's date and returns the person's age could be called `getAge()`. However, the names you can use are limited, much as variable names are. For example, you can't use words reserved by JavaScript, so you can't call your function `if()` or `while()`.

The parameters for the function are given in parentheses after the function's name. A parameter is just an item of data that the function needs to be given in order to do its job. Usually, not passing the required parameters will result in an error. A function can have zero or more parameters, though even if it has no parameters, you must still put the open and close parentheses after its name. For example, the top of your function definition must look like the following:

```
function myNoParamFunction()
```

You then write the code, which the function will execute when called on to do so. All the function code must be put in a block with a pair of curly braces.

Functions also enable you to return a value from a function to the code that called it. You use the `return` statement to return a value. In the example function given earlier, you return the value of the variable `degCent`, which you have just calculated. You don't have to return a value if you don't want to, but it's important to note that every function returns a value even if you don't use the `return` statement. Functions that do not explicitly return a value—that is, return a value with the `return` statement—return `undefined`.

When JavaScript comes across a `return` statement in a function, it treats it a bit like a `break` statement in a `for` loop—it exits the function, returning any value specified after the `return` keyword.

You'll probably find it useful to build up a "library" of functions that you use frequently in JavaScript code, which you can reference in your pages.

Having created your functions, how do you use them? Unlike the code you've seen so far, which executes when JavaScript reaches that line, functions only execute if you ask them to, which is termed *calling* or *invoking* the function. You call a function by writing its name at the point where you want it to be called and making sure that you pass any parameters it needs, separated by commas. For example:

```
myTemp = convertToCentigrade(212);
```

This line calls the `convertToCentigrade()` function you saw earlier, passing `212` as the parameter and storing the `return` value from the function (that is, `100`) in the `myTemp` variable.

Have a go at creating your own functions now, taking a closer look at how parameters are passed. Parameter passing can be a bit confusing, so you'll first create a simple function that takes just one parameter (the user's name) and writes it to the page in a friendly welcome string. First, you need to think of a name for your function. A short but descriptive name is `writeUserWelcome()`. Now you need to define what parameters the function expects to be passed. There's only one parameter—the username. Defining parameters is a little like defining variables—you need to stick to the same rules for naming, so that means no spaces, special characters, or reserved words. Let's call your parameter `userName`. You need to add it inside parentheses to the end of the function name (note that you don't put a semicolon at the end of the line):

```
function writeUserWelcome(userName)
```

Okay, now you have defined your function name and its parameters; all that's left is to create the function body—that is, the code that will be executed when the function is called. You mark out this part of the function by wrapping it in curly braces:

```
function writeUserWelcome(userName){
   document.write("Welcome to my website " + userName + "<br />");
   document.write("Hope you enjoy it!");
}
```

The code is simple enough; you write out a message to the web page using `document.write()`. You can see that `userName` is used just as you'd use any normal variable; in fact, it's best to think of parameters as normal variables. The value that the parameter has will be that specified by the JavaScript code where the function was called.

Let's see how you would call this function:

```
writeUserWelcome("Paul");
```

Simple, really—just write the name of the function you want to call, and then in parentheses add the data to be passed to each of the parameters, here just one piece. When the code in the function is executed, the variable `userName`, used in the body of the function code, will contain the text `"Paul"`.

Suppose you wanted to pass two parameters to your function—what would you need to change? Well, first you'd have to alter the function definition. Imagine that the second parameter will hold the user's age—you could call it `userAge` because that makes it pretty clear what the parameter's data represents. Here is the new code:

```
function writeUserWelcome(userName, userAge) {
   document.write("Welcome to my website" + userName + "<br />");
   document.write("Hope you enjoy it<br />");
   document.write("Your age is " + userAge);
}
```

You've added a line to the body of the function that uses the parameter you have added. To call the function, you'd write the following:

```
writeUserWelcome("Paul",31);
```

The second parameter is a number, so there is no need for quotes around it. Here the `userName` parameter will be `Paul`, and the second parameter, `userAge`, will be `31`.

TRY IT OUT Fahrenheit to Centigrade Function

Let's rewrite the temperature converter page using functions. You can cut and paste most of this code from `ch3_example4.html`—the parts that have changed have been highlighted. When you've finished, save it as `ch4_example1.html`.

```
<!DOCTYPE html>

<html lang="en">
<head>
    <title>Chapter 4, Example 1</title>
</head>
<body>
    <script>
        function convertToCentigrade(degFahren) {
            var degCent = 5 / 9 * (degFahren - 32);

            return degCent;
        }


        var degFahren = [212, 32, -459.15];
        var degCent = [];
        var loopCounter;

        for (loopCounter = 0; loopCounter <= 2; loopCounter++) {
            degCent[loopCounter] = convertToCentigrade(degFahren[loopCounter]);
        }

        for (loopCounter = 2; loopCounter >= 0; loopCounter--) {
                document.write("Value " + loopCounter +
                               " was " + degFahren[loopCounter] +
                               " degrees Fahrenheit");

                document.write(" which is " + degCent[loopCounter] +
                               " degrees centigrade<br />");
            }
        </script>
    </body>
    </html>
```

When you load this page into your browser, you should see exactly the same results that you had with `ch3_example4.html`.

At the top of the script block you declare your `convertToCentigrade()` function. You saw this function earlier:

```
function convertToCentigrade(degFahren) {
   var degCent = 5/9 * (degFahren - 32);

   return degCent;
}
```

If you're using a number of separate `script` blocks in a page, it's very important that the function be defined before any script calls it. If you have a number of functions, you may want to put them all in their own script file and load it before all other scripts. That way you know where to find all your functions, and you can be sure that they have been declared before they have been used.

You should be pretty familiar with how the code in the function works. You declare a variable `degCent`, do your calculation, and then return `degCent` back to the calling code. The function's parameter is `degFahren`, which provides the information the calculation needs.

Following the function declaration is the code that executes when the page loads. First you define the variables you need, and then you have the two loops that calculate and then output the results. This is mostly the same as before, apart from the first `for` loop:

```
for (loopCounter = 0; loopCounter <= 2; loopCounter++) {
    degCent[loopCounter] = convertToCentigrade(degFahren[loopCounter]);
}
```

The code inside the first `for` loop puts the value returned by the function `convertToCentigrade()` into the `degCent` array.

There is a subtle point to the code in this example. Notice that you declare the variable `degCent` within your function `convertToCentigrade()`, and you also declare it as an array after the function definition.

Surely this isn't allowed?

Well, this leads neatly to the next topic of this chapter—scope.

SCOPE AND LIFETIME

What is meant by *scope*? Well, put simply, it's the scope or extent of a variable's or function's availability—which parts of your code can access a variable and the data it contains. Scope is important to any programming language—even more so in JavaScript—so it's imperative that you understand how scope works in JavaScript.

Global Scope

Any variables or functions declared outside of a function will be available to all JavaScript code on the page, whether that code is inside a function or otherwise—we call this *global scope*. Look at the following example:

```
var degFahren = 12;

function convertToCentigrade() {
    var degCent = 5/9 * (degFahren - 32);

    return degCent;
}
```

In this code, the `degFahren` variable is a global variable because it is created outside of a function, and because it is global, it can be used anywhere in the page. The `convertToCentigrade()` function accesses the `degFahren` variable, using it as part of the calculation to convert Fahrenheit to centigrade.

This also means you can change the value of a global variable, and the following code does just that:

```
var degFahren = 12;

function convertToCentigrade() {
    degFahren = 20;
    var degCent = 5/9 * (degFahren - 32);

    return degCent;
}
```

This new line of code changes the value of `degFahren` to `20`; so the original value of `12` is no longer used in the calculation. This change in value isn't seen only inside of the `convertToCentigrade()` function. The `degFahren` variable is a global variable, and thus its value is `20` everywhere it is used.

Additionally, the `covertToCentigrade()` function is a global function because it is defined outside of another function (yes, you can create a function within a function … funception!), and they too can be accessed anywhere in the page.

In practice, you want to avoid creating global variables and functions because they can be easily and unintentionally modified. You can use some tricks to avoid globals, and you will see them throughout this book, but they all boil down to creating variables and functions in functional scope.

Functional Scope

Variables and functions declared inside a function are visible *only* inside that function—no code outside the function can access them. For example, consider our standard `convertToCentigrade()` function:

```
function convertToCentigrade(degFahren) {
    var degCent = 5/9 * (degFahren - 32);

    return degCent;
}
```

The `degCent` variable is defined inside the `convertToCentigrade()` function. Therefore, it can only be accessed from within `convertToCentigrade()`. This is commonly referred to as *functional* or *local scope*, and `degCent` is commonly called a *local variable*.

Function parameters are similar to variables; they have local scope, and thus can only be accessed from within the function. So in the case of the previous `convertToCentigrade()` function, `degFahren` and `degCent` are local variables.

So what happens when the code inside a function ends and execution returns to the point at which the code was called? Do the variables defined within the function retain their value when you call the function the next time?

The answer is no: Variables not only have the scope property—where they are visible—but they also have a *lifetime*. When the function finishes executing, the variables in that function die and their values are lost, unless you return one of them to the calling code. Every so often JavaScript performs garbage collection (which we talked about in Chapter 2), whereby it scans through the code and sees if any variables are no longer in use; if so, the data they hold are freed from memory to make way for the data of other variables.

Identifier Lookup

What happens if you use the same variable name for both a global and local variable? JavaScript handles this seemingly catastrophic event with a process called *identifier lookup*. An *identifier* is simply the name you give a variable or function. So, identifier lookup is the process that the JavaScript engine uses to find a variable or function with a given name. Consider the following code:

```
var degCent = 10;

function convertToCentigrade(degFahren) {
    var degCent = 5/9 * (degFahren - 32);

    return degCent;
}
```

This code contains two `degCent` variables: One is global, and the other is local to `convertToCentigrade()`. When you execute the function, the JavaScript engine creates the local `degCent` variable and assigns it the result of the Fahrenheit-to-centigrade conversion—the global degCent variable is left alone and still contains 10. But what value does the return statement return: the global or local degCent?

The JavaScript engine begins the identifier lookup process in the current level of scope. Therefore, it starts looking within the functional scope of convertToCentigrade() for a variable or function with the name degCent, it finds the local variable, and uses its value in the return statement.

But if degCent was not created within convertToCentigrade(), the JavaScript engine would then look in the next level of scope—the global scope in this case—for the degCent identifier. It would find the global variable and use its value.

So now that you understand how scope works, revisit Example 1 in the "Fahrenheit to Centigrade Function" Try It Out. Even though it has two degCent variables—one global and one local to convertToCentigrade()—the code executes without a problem. Inside the function, the local degCent variable takes precedence over the global. And outside of the function, the local variable is no longer in scope; therefore, the global degCent is used.

Although it's perfectly valid to use the same identifier for global and local variables, it is highly recommended that you avoid doing so. It adds extra, and often unnecessary, complexity and confusion to your code. It can also make it easier to introduce bugs that are difficult to find and fix. Imagine that, within a function, you modified a local variable when you meant to modify a global variable. That is a bug, and if you replicated it in many other functions, you will spend precious time finding and fixing those errors.

FUNCTIONS AS VALUES

JavaScript is a powerful language, and a lot of that power comes from functions. Unlike many other languages, functions are *first-class citizens* in JavaScript; in other words, we can treat functions just like any other type of value. For example, let's take the convertToCentigrade() function and assign it to a variable:

```
function convertToCentigrade(degFahren) {
    var degCent = 5/9 * (degFahren - 32);

    return degCent;
}

var myFunction = convertToCentigrade;
```

This code assigns the convertToCentigrade() function to the myFunction variable, but look closely at the right-hand side of the assignment—the opening and closing parentheses are missing at the end of the convertToCentigrade identifier. It looks a lot like a variable!

In this statement, we are not executing convertToCentigrade(); we are referring to the actual function itself. This means that we now have two ways of executing the same function. We can call it normally by executing convertToCentigrade(), or we can execute myFunction(), like this:

```
var degCent = myFunction(75); // 23.88888889
var degCent2 = convertToCentigrade(75); // 23.88888889
```

This also means we can pass a function to another function's parameter. Take a look at the following code:

```
function doSomething(fn) {
    fn("Hello, World");
}

doSomething(alert);
```

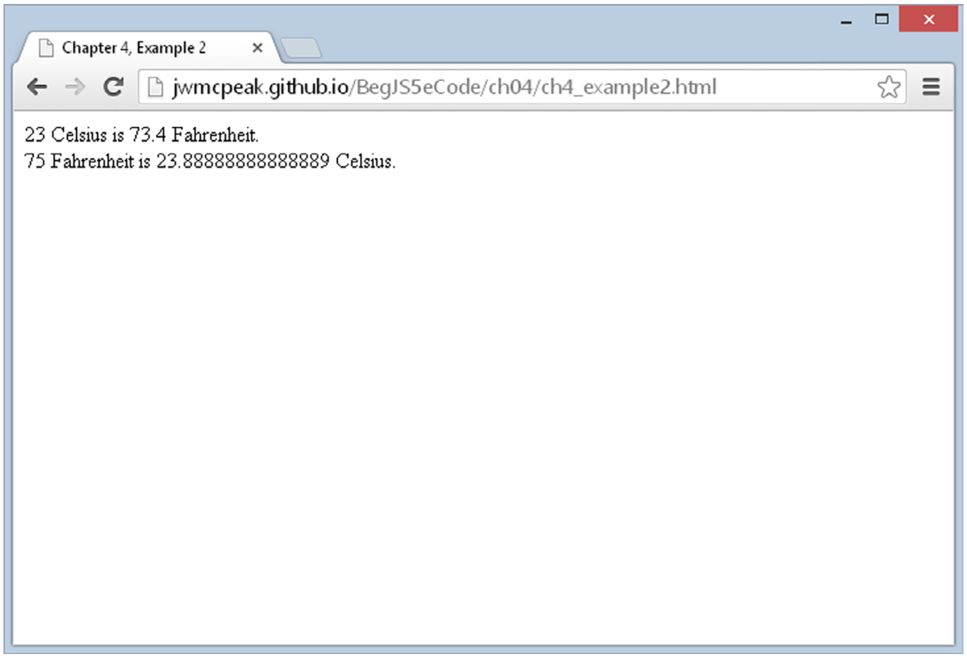
This code defines a function called `doSomething()`, and it has a single parameter called `fn`. Inside the function, the `fn` variable is used as a function; it's executed by using the `fn` identifier followed by a pair of parentheses. The final line of code executes the `doSomething()` function and passes the `alert` function as the parameter. When this code executes, an `alert` box displays the message `"Hello, World"`.

TRY IT OUT Passing Functions

Let's rewrite the temperature converter page to use more functions. You can cut and paste some of the code from `ch4_example1.html`, but the majority of this example will be new code. When you've finished, save it as `ch4_example2.html`.

```
<!DOCTYPE html>

<html lang="en">
<head>
    <title>Chapter 4, Example 2</title>
</head>
<body>
        <script>
        function toCentigrade(degFahren) {
            var degCent = 5 / 9 * (degFahren - 32);

            document.write(degFahren + " Fahrenheit is " +
                           degCent + " Celsius.<br/>");
        }

        function toFahrenheit(degCent) {
            var degFahren = 9 / 5 * degCent + 32;

            document.write(degCent + " Celsius is " +
                           degFahren + " Fahrenheit.<br/>");
        }

        function convert(converter, temperature) {
            converter(temperature);
        }

        convert(toFahrenheit, 23);
        convert(toCentigrade, 75);
    </script>
</body>
</html>
```

When you load this page into your browser, you should see the results shown in Figure 4-2.



```
23 Celsius is 73.4 Fahrenheit.
75 Fahrenheit is 23.88888888888889 Celsius.
```

FIGURE 4-2

At the top of the script block is the `toCentigrade()` function. It is somewhat similar to the `convertToCentigrade()` function from `ch4_example1.html`; instead of returning the converted value, it simply writes the conversion information to the document:

```
function toCentigrade(degFahren) {
    var degCent = 5 / 9 * (degFahren - 32);

    document.write(degFahren + " Fahrenheit is " +
                   degCent + " Celsius.<br/>");
}
```

The next function, `toFahrenheit()`, is similar to `toCentigrade()` except that it converts the supplied value to Fahrenheit. It then writes the conversion information to the document:

```
function toFahrenheit(degCent) {
    var degFahren = 9 / 5 * degCent + 32;

    document.write(degCent + " Celsius is " +
                   degFahren + " Fahrenheit.<br/>");
}
```

Admittedly, you could use these functions as is without any problem, but that wouldn't result in a very interesting example. Instead, the third function, `convert()`, will be used to execute `toCentigrade()` and `toFahrenheit()`:

```
function convert(converter, temperature) {
    return converter(temperature);
}
```

This function takes the first parameter, `converter`, and uses it as a function. The second parameter, `temperature`, is then passed to `converter()` to perform the conversion and write the results to the document.

The final two lines of code use `convert()` and pass it the appropriate converter function and temperature value:

```
convert(toFahrenheit, 23);
convert(toCentigrade, 75);
```

Although this is certainly a more complex solution to a relatively simple problem, it demonstrates the fact that functions are values in JavaScript. We can assign them to variables and pass them to other functions. This is an extremely important concept to understand, and you'll see why in Chapter 10 when you learn about events.

SUMMARY

In this chapter you concluded your look at the core of the JavaScript language and its syntax. Everything from now on builds on these foundations, and with the less interesting syntax under your belt, you can move on to more interesting things in the remainder of the book.

The chapter looked at the following:

- **Functions are reusable bits of code.** JavaScript has a lot of built-in functions that provide programmers services, such as converting a string to a number. However, JavaScript also enables you to define and use your own functions using the `function` keyword. Functions can have zero or more parameters passed to them and can return a value if you so wish.
- **Variable scope and lifetime:** Variables declared outside a function are available globally—that is, anywhere in the page. Any variables defined inside a function are private to that function and can't be accessed outside of it. Variables have a lifetime, the length of which depends on where the variable was declared. If it's a global variable, its lifetime is that of the page—while the page is loaded in the browser, the variable remains alive. For variables defined in a function, the lifetime is limited to the execution of that function. When the function is finished executing, the variables die, and their values are lost. If the function is called again later in the code, the variables will be empty.
- **Identifier lookup:** When you use a variable or function, the JavaScript engine goes through the identifier lookup process to find the value associated with the identifier.
- **Functions are first-class citizens in JavaScript.** You can assign functions to variables and pass them to other functions.

EXERCISES

You can find suggested solutions to these questions in Appendix A.

1. Change the code of Question 2 from Chapter 3 so that it's a function that takes as parameters the times table required and the values at which it should start and end. For example, you might try the 4 times table displayed starting with `4 * 4` and ending at `4 * 9`.

2. Modify the code of Question 1 to request the times table to be displayed from the user; the code should continue to request and display times tables until the user enters `-1`. Additionally, do a check to make sure that the user is entering a valid number; if the number is not valid, ask the user to re-enter it.