3 FUNCTIONS, METHODS & OBJECTS

Browsers require very detailed instructions about what we want them to do. Therefore, complex scripts can run to hundreds (even thousands) of lines. Programmers use functions, methods, and objects to organize their code. This chapter is divided into three sections that introduce:

FUNCTIONS & METHODS

Functions consist of a series of statements that have been grouped together because they perform a specific task. A method is the same as a function, except methods are created inside (and are part of) an object.

OBJECTS

In Chapter 1 you saw that programmers use objects to create models of the world using data, and that objects are made up of properties and methods. In this section, you learn how to create your own objects using JavaScript.

BUILT-IN OBJECTS

The browser comes with a set of objects that act like a toolkit for creating interactive web pages. This section introduces you to a number of built-in objects, which you will then see used throughout the rest of the book.

latest hotel offer
PARK
$204
room rate when you book 2 or more nights

WHAT IS A FUNCTION?

Functions let you group a series of statements together to perform a specific task. If different parts of a script repeat the same task, you can reuse the function (rather than repeating the same set of statements).

Grouping together the statements that are required to answer a question or perform a task helps organize your code.

Furthermore, the statements in a function are not always executed when a page loads, so functions also offer a way to *store* the steps needed to achieve a task. The script can then ask the function to perform all of those steps as and when they are required. For example, you might have a task that you only want to perform if the user clicks on a specific element in the page.

If you are going to ask the function to perform its task later, you need to give your function a name. That name should describe the task it is performing. When you ask it to perform its task, it is known as **calling** the function.

The steps that the function needs to perform in order to perform its task are packaged up in a code block. You may remember from the last chapter that a code block consists of one or more statements contained within curly braces. (And you do not write a semicolon after the closing curly brace - like you do after a statement.)

Some functions need to be provided with information in order to achieve a given task. For example, a function to calculate the area of a box would need to know its width and height. Pieces of information passed to a function are known as **parameters**.

When you write a function and you expect it to provide you with an answer, the response is known as a **return value**.

On the right, there is an example of a function in the JavaScript file. It is called `updateMessage()`.

Don't worry if you do not understand the syntax of the example on the right; you will take a closer look at how to write and use functions in the pages that follow.

Remember that programming languages often rely upon on name/value pairs. The function has a name, `updateMessage`, and the value is the code block (which consists of statements). When you call the function by its name, those statements will run.

You can also have anonymous functions. They do not have a name, so they cannot be called. Instead, they are executed as soon as the interpreter comes across them.

A BASIC FUNCTION

In this example, the user is shown a message at the top of the page. The message is held in an HTML element whose `id` attribute has a value of `message`. The message is going to be changed using JavaScript.

Before the closing `</body>` tag, you can see the link to the JavaScript file. The JavaScript file starts with a variable used to hold a new message, and is followed by a function called `updateMessage()`.

You do not need to worry about *how* this function works yet - you will learn about that over the next few pages. For the moment, it is just worth noting that inside the curly braces of the function are two statements.

HTML

c03/basic-function.html

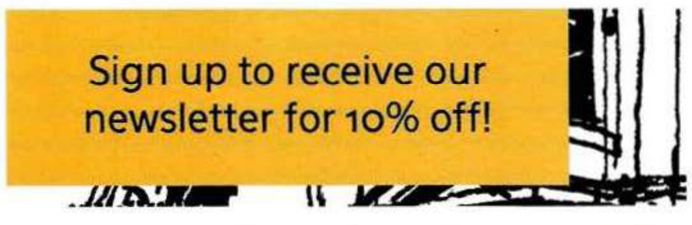
```
<!DOCTYPE html>
<html>
  <head>
    <title>Basic Function</title>
    <link rel="stylesheet" href="css/c03.css" />
  </head>
  <body>
    <h1>TravelWorthy</h1>
    <div id="message">Welcome to our site!</div>
    <script src="js/basic-function.js"></script>
  </body>
</html>
```

JAVASCRIPT

c03/js/basic-function.js

```
var msg = 'Sign up to receive our newsletter for 10% off!';
function updateMessage() {
  var el = document.getElementById('message');
  el.textContent = msg;
}
updateMessage();
```

RESULT



These statements update the message at the top of the page. The function acts like a store; it holds the statements that are contained in the curly braces until you are ready to use them. Those statements are not run until the function is **called**. The function is only called on the last line of this script.

DECLARING A FUNCTION

To create a function, you give it a name and then write the statements needed to achieve its task inside the curly braces.
This is known as a **function declaration**.

You declare a **function** using the **`function`** keyword.

You give the function a **name** (sometimes called an **identifier**) followed by parentheses.

The **statements** that perform the task sit in a code block. (They are inside curly braces.)

FUNCTION KEYWORD

FUNCTION NAME

```
function sayHello() {
  document.write('Hello!');
}
```

CODE BLOCK (IN CURLY BRACES)

This function is very basic (it only contains one statement), but it illustrates how to write a function. Most functions that you will see or write are likely to consist of more statements.

The point to remember is that functions store the code required to perform a specific task, and that the script can ask the function to perform that task whenever needed.

If different parts of a script need to perform the same task, you do not need to repeat the same statements multiple times - you use a function to do it (and reuse the same code).

CALLING A FUNCTION

Having declared the function, you can then execute all of the statements between its curly braces with just one line of code. This is known as **calling the function**.

To run the code in the function, you use the function name followed by parentheses.

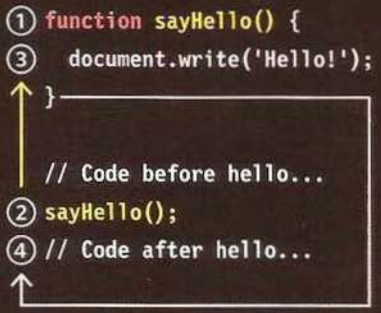
In programmer-speak, you would say that this code **calls** a function.

You can call the same function as many times as you want within the same JavaScript file.

FUNCTION NAME

```
sayHello();
```

1. The function can store the instructions for a specific task.
2. When you need the script to perform that task, you call the function.
3. The function executes the code in that code block.
4. When it has finished, the code continues to run from the point where it was initially called.



```
① function sayHello() {
③   document.write('Hello!');
  }

  // Code before hello...
② sayHello();
④ // Code after hello...
```

Sometimes you will see a function called *before* it has been declared. This still works because the interpreter runs through a script before executing each statement, so it will know that a function declaration appears later in the script. But for the moment, we will declare the function before calling it.

DECLARING FUNCTIONS THAT NEED INFORMATION

Sometimes a function needs specific information to perform its task. In such cases, when you declare the function you give it **parameters.** Inside the function, the parameters act like variables.

If a function needs information to work, you indicate what it needs to know in parentheses after the function name.

The items that appear inside these parentheses are known as the **parameters** of the function. Inside the function those words act like variable names.

PARAMETERS

```
function getArea(width, height) {
  return width * height;
}
```

THE PARAMETERS ARE USED LIKE VARIABLES WITHIN THE FUNCTION

This function will calculate and return the area of a rectangle. To do this, it needs the rectangle's width and height. Each time you call the function these values could be different.

This demonstrates how the code can perform a task without knowing the *exact* details in advance, as long as it has rules it can follow to achieve the task.

So, when you design a script, you need to note the information the function will require in order to perform its task.

If you look inside the function, the parameter names are used just as you would use variables. Here, the parameter names `width` and `height` represent the width and height of the wall.

CALLING FUNCTIONS THAT NEED INFORMATION

When you call a function that has parameters, you specify the values it should use in the parentheses that follow its name. The values are called **arguments**, and they can be provided as values or as variables.

ARGUMENTS AS VALUES

When the function below is called, the number 3 will be used for the width of the wall, and 5 will be used for its height.

```
getArea(3, 5);
```

ARGUMENTS AS VARIABLES

You do not have to specify actual values when calling a function - you can use variables in their place. So the following does the same thing.

```
wallWidth = 3;
wallHeight = 5;
getArea(wallWidth, wallHeight);
```

PARAMETERS VS ARGUMENTS

People often use the terms **parameter** and **argument** interchangeably, but there *is* a subtle difference.

On the left-hand page, when the function is declared, you can see the words **`width`** and **`height`** used (in parentheses on the first line). Inside the curly braces of the function, those words act like variables. These names are the parameters.

On this page, you can see that the **`getArea()`** function is being called and the code specifies real numbers that will be used to perform the calculation (or variables that hold real numbers).

These values that you pass into the code (the information it needs to calculate the size of this particular wall) are called arguments.

GETTING A SINGLE VALUE OUT OF A FUNCTION

Some functions return information to the code that called them. For example, when they perform a calculation, they return the result.

This **calculateArea()** function returns the area of a rectangle to the code that called it.

Inside the function, a variable called **area** is created. It holds the calculated area of the box.

The **return** keyword is used to return a value to the code that called the function.

```
function calculateArea(width, height) {
  var area = width * height;
  return area;
}
var wallOne = calculateArea(3, 5);
var wallTwo = calculateArea(8, 5);
```

Note that the intrepreter leaves the function when **return** is used. It goes back to the statement that called it. If there had been any subsequent statements in this function, they would not be processed.

The **wallOne** variable holds the value **15**, which was calculated by the **calculateArea()** function.

The **wallTwo** variable holds the value **40**, which was calculated by the same **calculateArea()** function.

This also demonstrates how the same function can be used to perform the same steps with different values.

GETTING MULTIPLE VALUES OUT OF A FUNCTION

Functions can return more than one value using an array.
For example, this function calculates the area and volume of a box.

First, a new function is created called **getSize()**. The area of the box is calculated and stored in a variable called **area**.

The volume is calculated and stored in a variable called **volume**. Both are then placed into an array called **sizes**.

This array is then returned to the code that called the **getSize()** function, allowing the values to be used.

```
function getSize(width, height, depth) {
  var area = width * height;
  var volume = width * height * depth;
  var sizes = [area, volume];
  return sizes;
}
var areaOne = getSize(3, 2, 3)[0];
var volumeOne = getSize(3, 2, 3)[1];
```

The **areaOne** variable holds the area of a box that is 3 x 2. The area is the *first* value in the **sizes** array.

The **volumeOne** variable holds the volume of a box that is 3 x 2 x 3. The volume is the *second* value in the **sizes** array.

ANONYMOUS FUNCTIONS & FUNCTION EXPRESSIONS

Expressions produce a value. They can be used where values are expected. If a function is placed where a browser expects to see an expression, (e.g., as an argument to a function), then it gets treated as an expression.

FUNCTION DECLARATION

A **function declaration** creates a function that you can call later in your code. It is the type of function you have seen so far in this book.

In order to call the function later in your code, you must give it a name, so these are known as **named functions**. Below, a function called `area()` is declared, which can then be called using its name.

```
function area(width, height) {
  return width * height;
};

var size = area(3, 4);
```

As you will see on p456, the interpreter always looks for variables and function declarations *before* going through each section of a script, line-by-line. This means that a function created with a function declaration can be called *before* it has even been declared.

For more information about how variables and functions are processed first, see the discussion about execution context and hoisting on p452 - p457.

FUNCTION EXPRESSION

If you put a function where the interpreter would expect to see an expression, then it is treated as an expression, and it is known as a **function expression**. In function expressions, the name is usually omitted. A function with no name is called an **anonymous function**. Below, the function is stored in a variable called `area`. It can be called like any function created with a function declaration.

```
var area = function(width, height) {
  return width * height;
};

var size = area(3, 4);
```

In a function expression, the function is not processed until the interpreter gets to that statement. This means you cannot call this function *before* the interpreter has discovered it. It also means that any code that appears up to that point could potentially alter what goes on inside this function.

IMMEDIATELY INVOKED FUNCTION EXPRESSIONS

This way of writing a function is used in several different situations. Often functions are used to ensure that the variable names do not conflict with each other (especially if the page uses more than one script).

IMMEDIATELY INVOKED FUNCTION EXPRESSIONS (IIFE)

Pronounced "iffy," these functions are not given a name. Instead, they are executed once as the interpreter comes across them.

Below, the variable called area will hold the value returned from the function (rather than storing the function itself so that it can be called later).

```
var area = (function() {
  var width = 3;
  var height = 2;
  return width * height;
}());
```

The **final parentheses** (shown on green) after the closing curly brace of the code block tell the interpreter to call the function immediately.
The **grouping operators** (shown on pink) are parentheses there to ensure the intrepreter treats this as an expression.

You may see the final parentheses in an IIFE placed *after* the closing grouping operator but it is commonly considered better practice to place the final parentheses *before* the closing grouping operator, as shown in the code above.

WHEN TO USE ANONYMOUS FUNCTIONS AND IIFES

You will see many ways in which anonymous function expressions and IIFEs are used throughout the book.

They are used for code that only needs to run once within a task, rather than repeatedly being called by other parts of the script. For example:

- As an argument when a function is called (to calculate a value for that function).
- To assign the value of a property to an object.
- In event handlers and listeners (see Chapter 6) to perform a task when an event occurs.
- To prevent conflicts between two scripts that might use the same variable names (see p99).

IIFEs are commonly used as a wrapper around a set of code. Any variables declared within that anonymous function are effectively protected from variables in other scripts that might have the same name. This is due to a concept called scope, which you meet on the next page. It is also a very popular technique with jQuery.

VARIABLE SCOPE

The location where you declare a variable will affect where it can be used within your code. If you declare it within a function, it can only be used within that function. This is known as the variable's **scope**.

LOCAL VARIABLES

When a variable is created *inside* a function using the `var` keyword, it can only be used in that function. It is called a **local** variable or **function-level** variable. It is said to have **local scope** or **function-level scope**. It cannot be accessed outside of the function in which it was declared. Below, `area` is a local variable.

The interpreter creates local variables when the function is run, and removes them as soon as the function has finished its task. This means that:

- If the function runs twice, the variable can have different values each time.
- Two different functions can use variables with the same name without any kind of naming conflict.

GLOBAL VARIABLES

If you create a variable *outside* of a function, then it can be used anywhere within the script. It is called a **global** variable and has **global scope**. In the example shown, `wallSize` is a global variable.

Global variables are stored in memory for as long as the web page is loaded into the web browser. This means they take up more memory than local variables, and it also increases the risk of naming conflicts (see next page). For these reasons, you should use local variables wherever possible.

If you forget to declare a variable using the `var` keyword, the variable will work, but it will be treated as a *global* variable (this is considered bad practice).

```
function getArea(width, height) {
  var area = width * height;
  return area;
}

var wallSize = getArea(3, 2);
document.write(wallSize);
```

🟥 LOCAL (OR FUNCTION-LEVEL) SCOPE

🟩 GLOBAL SCOPE

HOW MEMORY & VARIABLES WORK

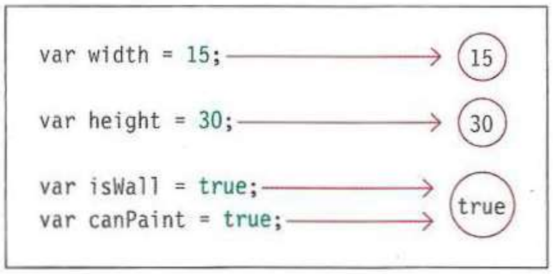
Global variables use more memory. The browser has to remember them for as long as the web page using them is loaded. Local variables are only remembered during the period of time that a function is being executed.

CREATING THE VARIABLES IN CODE

Each variable that you declare takes up memory. The more variables a browser has to remember, the more memory your script requires to run. Scripts that require a lot of memory can perform slower, which in turn makes your web page take longer to respond to the user.

```
var width = 15;
var height = 30;
var isWall = true;
var canPaint = true;
```

A variable actually references a value that is stored in memory. The same value can be used with more than one variable:



```
var width = 15;
var height = 30;
var isWall = true;
var canPaint = true;
```

Here the values for the `width` and `height` of the wall are stored separately, but the same value of `true` can be used for both `isWall` and `canPaint`.

NAMING COLLISIONS

You might think you would avoid naming collisions; after all *you* know which variables you are using. But many sites use scripts written by several people. If an HTML page uses two JavaScript files, and both have a global variable with the same name, it can cause errors. Imagine a page using these two scripts:

```
// Show size of the building plot
function showPlotSize(){
  var width = 3;
  var height = 2;
  return 'Area: " + (width * height);
}
var msg = showArea()
```

```
// Show size of the garden
function showGardenSize() {
  var width = 12;
  var height = 25;
  return width * height;
}
var msg = showGardenSize();
```

● Variables in global scope: have naming conflicts.
● Variables in function scope: there is no conflict between them.