



# 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.

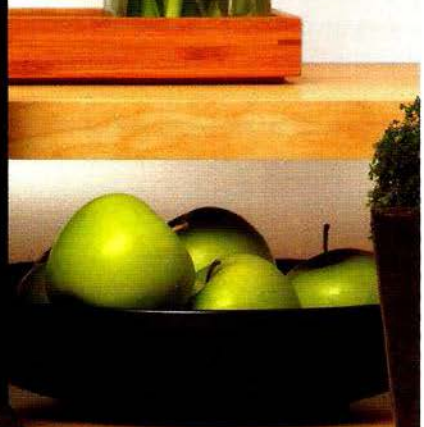
#### **OBJECTS**

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.





# 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

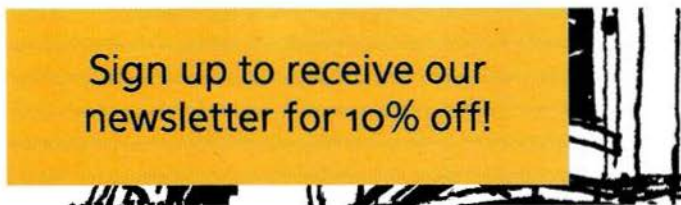
```
<!DOCTYPE html>
<html>
  <head>
    <title>Basic Function</title>
    <link rel="stylesheet" href="css/c03.css" />
  </head>
  <body>
    <h1>TravelWorthy</h1>
    <div id="message">Welcome to our site!</div>
    <script src="js/basic-function.js"></script>
  </body>
</html>
```

## 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.

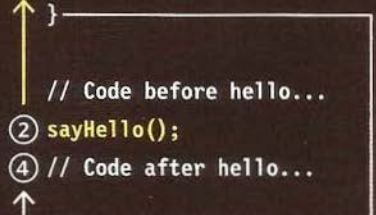
You can call the same function as many times as you want within the same JavaScript file.

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.

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

PARAMETERS

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 interpreter 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 interpreter 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; → (15)
var height = 30; → (30)
var isWall = true; → (true)
var canPaint = true; → (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.

# WHAT IS AN OBJECT?



Objects group together a set of variables and functions to create a model of a something you would recognize from the real world. In an object, variables and functions take on new names.

## IN AN OBJECT: VARIABLES BECOME KNOWN AS PROPERTIES

If a variable is part of an object, it is called a **property**. Properties tell us about the object, such as the name of a hotel or the number of rooms it has. Each individual hotel might have a different name and a different number of rooms.

## IN AN OBJECT: FUNCTIONS BECOME KNOWN AS METHODS

If a function is part of an object, it is called a **method**. Methods represent tasks that are associated with the object. For example, you can check how many rooms are available by subtracting the number of booked rooms from the total number of rooms.



This object represents a hotel. It has five properties and one method. The object is in curly braces. It is stored in a variable called `hotel`.

Like variables and named functions, properties and methods have a name and a value. In an object, that name is called a **key**.

An object cannot have two keys with the same name. This is because keys are used to access their corresponding values.

The value of a property can be a string, number, Boolean, array, or even another object. The value of a method is always a function.

```
var hotel = {
```

```
  name: 'Quay',
  rooms: 40,
  booked: 25,
  gym: true,
  roomTypes: ['twin', 'double', 'suite'],
  checkAvailability: function() {
    return this.rooms - this.booked;
  }
};
```

● KEY  
● VALUE

PROPERTIES  
These are variables

METHOD  
This is a function

Above you can see a `hotel` object. The object contains the following key/value pairs:

PROPERTIES:	KEY	VALUE
	<code>name</code>	string
	<code>rooms</code>	number
	<code>booked</code>	number
	<code>gym</code>	Boolean
	<code>roomTypes</code>	array
METHODS:	<code>checkAvailability</code>	function

As you will see over the next few pages, this is just one of the ways you can create an object.

Programmers use a lot of name/value pairs:

- HTML uses attribute names and values.
- CSS uses property names and values.

In JavaScript:

- Variables have a name and you can assign them a value of a string, number, or Boolean.
- Arrays have a name and a group of values. (Each item in an array is a name/value pair because it has an index number and a value.)
- Named functions have a name and value that is a set of statements to run if the function is called.
- Objects consist of a set of name/value pairs (but the names are referred to as keys).

# CREATING AN OBJECT: LITERAL NOTATION

Literal notation is the easiest and most popular way to create objects. (There are several ways to create objects.)

The object is the curly braces and their contents. The object is stored in a variable called `hotel`, so you would refer to it as the `hotel` object.

Separate each key from its value using a colon. Separate each property and method with a comma (but not after the last value).

```
var hotel = {
```

● OBJECT  
● KEY  
● VALUE

```
  name: 'Quay',  
  rooms: 40,  
  booked: 25,
```

PROPERTIES

```
  checkAvailability: function() {  
    return this.rooms - this.booked;  
  }
```

METHOD

```
};
```

In the `checkAvailability()` method, the `this` keyword is used to indicate that it is using the `rooms` and `booked` properties of `this` object.

When setting properties, treat the values like you would do for variables: strings live in quotes and arrays live in square brackets.



# ACCESSING AN OBJECT AND DOT NOTATION

You access the properties or methods of an object using dot notation. You can also access properties using square brackets.

To access a property or method of an object you use the name of the object, followed by a period, then the name of the property or method you want to access. This is known as **dot notation**.

The period is known as the **member operator**. The property or method on its right is a member of the object on its left. Here, two variables are created to hold the hotel name and number of vacant rooms.

```
var hotelName = hotel.name;
var roomsFree = hotel.checkAvailability();
```

OBJECT                      PROPERTY/METHOD NAME

MEMBER OPERATOR

You can also access the properties of an object (but not its methods) using square bracket syntax.

This time the object name is followed by square brackets, and the property name is inside them.

```
var hotelName = hotel['name'];
```

This notation is used most commonly used when:

- The name of the property is a number (technically allowed, but should generally be avoided)
- A variable is being used in place of the property name (you will see this technique used in Chapter 12)

# CREATING OBJECTS USING LITERAL NOTATION

This example starts by creating an object using literal notation.

This object is called `hotel` which represents a hotel called Quay with 40 rooms (25 of which have been booked).

Next, the content of the page is updated with data from this object. It shows the name of the hotel by accessing the object's name property and the number of vacant rooms using the `checkAvailability()` method.

To access a property of this object, the object name is followed by a dot (the period symbol) and the name of the property that you want.

Similarly, to use the method, you can use the object name followed by the method name. `hotel.checkAvailability()`

If the method needs parameters, you can supply them in the parentheses (just like you can pass arguments to a function).

c3/js/object-literal.js

JAVASCRIPT

```
var hotel = {
  name: 'Quay',
  rooms: 40,
  booked: 25,
  checkAvailability: function() {
    return this.rooms - this.booked;
  }
};

var elName = document.getElementById('hotelName');
elName.textContent = hotel.name;

var elRooms = document.getElementById('rooms');
elRooms.textContent = hotel.checkAvailability();
```

RESULT



hotel availability

QUAY

15  
rooms left



# CREATING MORE OBJECT LITERALS

## JAVASCRIPT

c03/js/object-literal2.js

```
var hotel = {
  name: 'Park',
  rooms: 120,
  booked: 77,
  checkAvailability: function() {
    return this.rooms - this.booked;
  }
};

var elName = document.getElementById('hotelName');
elName.textContent = hotel.name;

var elRooms = document.getElementById('rooms');
elRooms.textContent = hotel.checkAvailability();
```

## RESULT



hotel availability

PARK

43  
rooms left

Here you can see another object. Again it is called `hotel`, but this time the model represents a different hotel. For a moment, imagine that this is a different page of the same travel website.

The Park hotel is larger. It has 120 rooms and 77 of them are booked.

The only things changing in the code are the values of the `hotel` object's properties:

- The name of the hotel
- How many rooms it has
- How many rooms are booked

The rest of the page works in exactly the same way. The name is shown using the same code. The `checkAvailability()` method has not changed and is called in the same way.

If this site had 1,000 hotels, the only thing that would need to change would be the three properties of this object. Because we created a model for the hotel using data, the same code can access and display the details for any hotel that follows the same data model.

If you had two objects on the same page, you would create each one using the same notation but store them in variables with different names.



# CREATING AN OBJECT: CONSTRUCTOR NOTATION

The **new** keyword and the object constructor create a blank object. You can then add properties and methods to the object.

First, you create a new object using a combination of the **new** keyword and the **Object()** constructor function. (This function is part of the JavaScript language and is used to create objects.)

Next, having created the blank object, you can add properties and methods to it using dot notation. Each statement that adds a property or method should end with a semicolon.

```
var hotel = new Object();
```

- OBJECT
- KEY
- VALUE

```
hotel.name = 'Quay';  
hotel.rooms = 40;  
hotel.booked = 25;  
  
hotel.checkAvailability = function() {  
    return this.rooms - this.booked;  
};
```

PROPERTIES

METHOD

You can use this syntax to add properties and methods to any object you have created (no matter which notation you used to create it).

To create an empty object using literal notation use:  
**var hotel = {}**  
The curly brackets create an empty object.



# UPDATING AN OBJECT

To update the value of properties, use dot notation or square brackets. They work on objects created using literal or constructor notation. To delete a property, use the `delete` keyword.

To update a property, use the same technique that was shown on the left-hand page to add properties to the object, but give it a new value.

If the object does not have the property you are trying to update, it will be added to the object.

OBJECT      PROPERTY NAME      PROPERTY VALUE

```
hotel.name = 'Park';
```

MEMBER OPERATOR      ASSIGNMENT OPERATOR

You can also update the properties of an object (but not its methods) using square bracket syntax. The object name is followed by square brackets, and the property name is inside them.

A new value for the property is added after the assignment operator. Again, if the property you are attempting to update does not exist, it will be added to the object.

```
hotel['name'] = 'Park';
```

To delete a property, use the `delete` keyword followed by the object name and property name.

```
delete hotel.name;
```

If you just want to clear the value of a property, you could set it to a blank string.

```
hotel.name = '';
```



# CREATING MANY OBJECTS: CONSTRUCTOR NOTATION

Sometimes you will want several objects to represent similar things. Object constructors can use a function as a **template** for creating objects. First, create the template with the object's properties and methods.

A function called `Hotel` will be used as a template for creating new objects that represent hotels. Like all functions, it contains statements. In this case, they add properties or methods to the object.

The function has three parameters. Each one sets the value of a property in the object. The methods will be the same for each object created using this function.

```
function Hotel(name, rooms, booked) {
```

● KEY  
● VALUE

```
  this.name = name;
```

```
  this.rooms = rooms;
```

```
  this.booked = booked;
```

PROPERTIES

```
  this.checkAvailability = function() {
```

```
    return this.rooms - this.booked;
```

```
  };
```

METHOD

```
}
```

The `this` keyword is used instead of the object name to indicate that the property or method belongs to the object that `this` function creates.

Each statement that creates a new property or method for this object ends in a semicolon (not a comma, which is used in the literal syntax).

The name of a constructor function usually begins with a capital letter (unlike other functions, which tend to begin with a lowercase character).

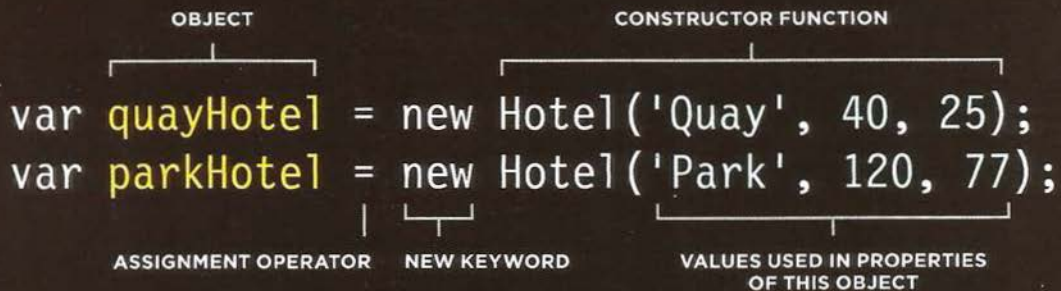
The uppercase letter is supposed to help remind developers to use the `new` keyword when they create an object using that function (see next page).



You create **instances** of the object using the constructor function. The **new** keyword followed by a call to the function creates a new object. The properties of each object are given as arguments to the function.

Here, two objects are used to represent two hotels, so each object needs a different name. When the **new** keyword calls the constructor function (defined on the left-hand page), it creates a new object.

Each time it is called, the arguments are different because they are the values for the properties of each hotel. Both objects automatically get the same method defined in the constructor function.



The first object is called **quayHotel**. Its name is 'Quay' and it has 40 rooms, 25 of which are booked.

The second object is called **parkHotel**. Its name is 'Park' and it has 120 rooms, 77 of which are booked.

Even when many objects are created using the same constructor function, the methods stay the same because they access, update, or perform a calculation on the data stored in the properties.

You might use this technique if your script contains a very complex object that needs to be available but might not be used. The object is *defined* in the function, but it is only *created* if it is needed.

# CREATING OBJECTS USING CONSTRUCTOR SYNTAX

On the right, an empty object called `hotel` is created using the constructor function.

Once it has been created, three properties and a method are then assigned to the object.

(If the object already had any of these properties, this would overwrite the values in those properties.)

To access a property of this object, you can use dot notation, just as you can with any object.

For example, to get the hotel's name you could use:

```
hotel.name
```

Similarly, to use the method, you can use the object name followed by the method name:

```
hotel.checkAvailability()
```

c3/js/object-constructor.js

JAVASCRIPT

```
var hotel = new Object();

hotel.name = 'Park';
hotel.rooms = 120;
hotel.booked = 77;
hotel.checkAvailability = function() {
    return this.rooms - this.booked;
};

var elName = document.getElementById('hotelName');
elName.textContent = hotel.name;

var elRooms = document.getElementById('rooms');
elRooms.textContent = hotel.checkAvailability();
```

RESULT



hotel availability

PARK

43  
rooms left



# CREATE & ACCESS OBJECTS CONSTRUCTOR NOTATION

## JAVASCRIPT

c03/js/multiple-objects.js

```
function Hotel(name, rooms, booked) {
  this.name = name;
  this.rooms = rooms;
  this.booked = booked;
  this.checkAvailability = function() {
    return this.rooms - this.booked;
  };
}

var quayHotel = new Hotel('Quay', 40, 25);
var parkHotel = new Hotel('Park', 120, 77);

var details1 = quayHotel.name + ' rooms: ';
  details1 += quayHotel.checkAvailability();
var elHotel1 = document.getElementById('hotel1');
elHotel1.textContent = details1;

var details2 = parkHotel.name + ' rooms: ';
  details2 += parkHotel.checkAvailability();
var elHotel2 = document.getElementById('hotel2');
elHotel2.textContent = details2;
```

## RESULT



### hotel availability

Quay rooms: 15  
Beach rooms: 30

To get a better idea of why you might want to create multiple objects on the same page, here is an example that shows room availability in two hotels.

First, a constructor function defines a template for the hotels. Next, two different instances of this type of hotel object are created. The first represents a hotel called Quay and the second a hotel called Park.

Having created instances of these objects, you can then access their properties and methods using the same dot notation that you use with all other objects.

In this example, data from both objects is accessed and written into the page. (The HTML for this example changes to accommodate the extra hotel.)

For each hotel, a variable is created to hold the hotel name, followed by space, and the word rooms.

The line after it adds to that variable with the number of available rooms in that hotel.

(The += operator is used to add content to an existing variable.)



# ADDING AND REMOVING PROPERTIES

Once you have created an object (using literal or constructor notation), you can add new properties to it.

You do this using the dot notation that you saw for adding properties to objects on p103.

In this example, you can see that an instance of the `hotel` object is created using an object literal.

Immediately after this, the `hotel` object is given two extra properties that show the facilities (whether or not it has a gym and/or a pool). These properties are given values that are Booleans (true or false).

Having added these properties to the object, you can access them just like any of the objects other properties. Here, they update the value of the `className` attribute on their respective elements to show either a check mark or a cross mark.

To delete a property, you use the keyword `delete`, and then use dot notation to identify the property or method you want to remove from the object.

In this case, the `booked` property is removed from the object.

c3/js/adding-and-removing-properties.js

JAVASCRIPT

```
var hotel = {  
  name : 'Park',  
  rooms : 120,  
  booked : 77,  
};
```

```
hotel.gym = true;  
hotel.pool = false;  
delete hotel.booked;
```

```
var elName = document.getElementById('hotelName');  
elName.textContent = hotel.name;
```

```
var elPool = document.getElementById('pool');  
elPool.className = 'Pool: ' + hotel.pool;
```

```
var elGym = document.getElementById('gym');  
elGym.className = 'Gym: ' + hotel.gym;
```

RESULT



hotel facilities

PARK

Pool ✘  
Gym ✔

If an object is created using a constructor function, this syntax only adds or removes the properties from the one instance of the object (not all objects created with that function).



# RECAP: WAYS TO CREATE OBJECTS

## CREATE THE OBJECT, THEN ADD PROPERTIES & METHODS

In both of these examples, the object is created on the first line of the code sample. The properties and methods are then added to it afterwards.

### LITERAL NOTATION

```
var hotel = {}  
  
hotel.name = 'Quay';  
hotel.rooms = 40;  
hotel.booked = 25;  
hotel.checkAvailability = function() {  
    return this.rooms - this.booked;  
};
```

Once you have created an object, the syntax for adding or removing any properties and methods from that object is the same.

### OBJECT CONSTRUCTOR NOTATION

```
var hotel = new Object();  
  
hotel.name = 'Quay';  
hotel.rooms = 40;  
hotel.booked = 25;  
hotel.checkAvailability = function() {  
    return this.rooms - this.booked;  
};
```

## CREATING AN OBJECT WITH PROPERTIES & METHODS

### LITERAL NOTATION

A colon separates the key/value pairs. There is a comma between each key/value pair.

```
var hotel = {  
    name: 'Quay',  
    rooms: 40,  
    booked: 25,  
    checkAvailability: function() {  
        return this.rooms - this.booked;  
    }  
};
```

### OBJECT CONSTRUCTOR NOTATION

The function can be used to create multiple objects. The `this` keyword is used instead of the object name.

```
function Hotel(name, rooms, booked) {  
    this.name = name;  
    this.rooms = rooms;  
    this.booked = booked;  
    this.checkAvailability = function() {  
        return this.rooms - this.booked;  
    };  
}  
  
var quayHotel = new Hotel('Quay', 40, 25);  
var parkHotel = new Hotel('Park', 120, 77);
```

# THIS (IT IS A KEYWORD)

The keyword `this` is commonly used inside functions and objects. Where the function is declared alters what `this` means. It always refers to one object, usually the object in which the function operates.

## A FUNCTION IN GLOBAL SCOPE

When a function is created at the top level of a script (that is, not inside another object or function), then it is in the **global scope** or **global context**.

The default object in this context is the `window` object, so when `this` is used inside a function in the global context it refers to the `window` object.

Below, `this` is being used to return properties of the `window` object (you meet these properties on p124).

```
function windowSize() {  
  var width = this.innerWidth;  
  var height = this.innerHeight;  
  return [height, width];  
}
```

Under the hood, the `this` keyword is a reference to the object that the function is created inside.

## GLOBAL VARIABLES

All global variables also become properties of the `window` object, so when a function is in the global context, you can access global variables using the `window` object, as well as its other properties.

Here, the `showWidth()` function is in global scope, and `this.width` refers to the `width` variable:

```
var width = 600; ←  
var shape = {width: 300};  
  
var showWidth = function() {  
  document.write(this.width);  
};  
  
showWidth();
```

Here, the function would write a value of 600 into the page (using the `document` object's `write()` method).



As you can see, the value of `this` changes in different situations. But don't worry if you do not follow these two pages on your first read through. As you write more functions and objects, these concepts will become more familiar, and if `this` is not returning the value you expected, these pages will help you work out why.

Another scenario to mention is when one function is nested inside another function. It is only done in more complicated scripts, but the value of `this` can vary (depending on which browser you are using). You could work around this by storing the value of `this` in a variable in the first function and using the variable name in child functions instead of `this`.

## A METHOD OF AN OBJECT

When a function is defined *inside* an object, it becomes a method. In a method, `this` refers to the containing object.

In the example below, the `getArea()` method appears inside the `shape` object, so `this` refers to the `shape` object it is contained in:

```
var shape = {  
  width: 600, ←  
  height: 400, ←  
  getArea: function() {  
    return this.width * this.height;  
  }  
};
```

Because the `this` keyword here refers to the `shape` object, it would be the same as writing:

```
return shape.width * shape.height;
```

If you were creating several objects using an object constructor (and each `shape` had different dimensions), `this` would refer to the individual instance of the new object you are creating. When you called `getArea()`, it would calculate the dimensions of that particular instance of the object.

## FUNCTION EXPRESSION AS METHOD

If a named function has been defined in global scope, and it is then used as a method of an object, `this` refers to the object it is contained within.

The next example uses the same `showWidth()` function expression as the one on the left-hand page, but it is assigned as a method of an object.

```
var width = 600;  
var shape = {width: 300};  
  
var showWidth = function() {  
  document.write(this.width);  
};  
  
shape.getWidth = showWidth;  
shape.getWidth();
```

The last but one line indicates that the `showWidth()` function is used as a method of the `shape` object. The method is given a different name: `getWidth()`.

When the `getWidth()` method is called, even though it uses the `showWidth()` function, `this` now refers to the `shape` object, not the global context (and `this.width` refers to the `width` property of the `shape` object). So it writes a value of 300 to the page.

# RECAP: STORING DATA

In JavaScript, data is represented using name/value pairs.

To organize your data, you can use an array or object to group a set of related values. In arrays and objects the name is also known as a key.

## VARIABLES

A variable has just one key (the variable name) and one value.

Variable names are separated from their value by an equals sign (the assignment operator):

```
var hotel = 'Quay';
```

To retrieve the value of a variable, use its name:

```
// This retrieves Quay:  
hotel;
```

When a variable has been declared but has not yet been assigned a value, it is `undefined`.

If the `var` keyword is not used, the variable is declared in global scope (you should always use it).

## ARRAYS

Arrays can store multiple pieces of information. Each piece of information is separated by a comma. The order of the values is important because items in an array are assigned a number (called an index).

Values in an array are put in square brackets, separated by commas:

```
var hotels = [  
  'Quay',  
  'Park',  
  'Beach',  
  'Bloomsbury'  
]
```

You can think of each item in the array as another key/value pair, the key is the index number, and the values are shown in the comma-separated list.

To retrieve an item, use its index number:

```
// This retrieves Park:  
hotels[1];
```

If a key is a number, to retrieve the value you must place the number in square brackets.

Generally speaking, arrays are the only times when the key would be a number.



**Note:** This recap specifically relate to storing data. You cannot store rules to perform a task in an array. They can only be stored in a function or method.

If you want to access items via a property name or key, use an object (but note that each key in the object must be unique).

If the order of the items is important, use an array.

### INDIVIDUAL OBJECTS

Objects store sets of name/value pairs. They can be properties (variables) or methods (functions).

The order of them is not important (unlike the array). You access each piece of data by its key.

In object literal notation, properties and methods of an object are given in curly braces:

```
var hotel = {  
  name: 'Quay',  
  rooms: 40  
};
```

Objects created with literal notation are good:

- When you are storing / transmitting data between applications
- For global or configuration objects that set up information for the page

To access the properties or methods of the object, use dot notation:

```
// This retrieves Quay:  
hotel.name;
```

### MULTIPLE OBJECTS

When you need to create multiple objects within the same page, you should use an object constructor to provide a template for the objects.

```
function Hotel(name, rooms) {  
  this.name = name;  
  this.rooms = rooms;  
}
```

You then create instances of the object using the new keyword and then a call to the constructor function.

```
var hotel1 = new Hotel('Quay', 40);  
var hotel2 = new Hotel('Park', 120);
```

Objects created with constructors are good when:

- You have lots of objects used with similar functionality (e.g., multiple slideshows / media players / game characters) within a page
- A complex object might not be used in code

To access the properties or methods of the object, use dot notation:

```
// This retrieves Park:  
hotel2.name;
```

# ARRAYS ARE OBJECTS

Arrays are actually a special type of object. They hold a related set of key/value pairs (like all objects), but the key for each value is its index number.

As you saw (on p72), arrays have a **length** property telling you how many items are in the array. In Chapter 12, you will see that arrays also have several other helpful methods.

## AN OBJECT

PROPERTY:	VALUE:
room1	420
room2	460
room3	230
room4	620

Here, hotel room costs are stored in an object. The example covers four rooms, and the cost for each room is a property of the object:

```
costs = {  
  room1: 420,  
  room2: 460,  
  room3: 230,  
  room4: 620  
};
```

## AN ARRAY

INDEX NUMBER:	VALUE:
0	420
1	460
2	230
3	620

Here is the the same data in an array. Instead of property names, it has index numbers:

```
costs = [420, 460, 230, 620];
```



# ARRAYS OF OBJECTS & OBJECTS IN ARRAYS

You can combine arrays and objects to create complex data structures: Arrays can store a series of objects (and remember their order). Objects can also hold arrays (as values of their properties).

In an object, the order in which the properties appear is not important. In an array, the index numbers dictate the order of the properties. You will see more examples of these data structures in Chapter 12.

## ARRAYS IN AN OBJECT

The property of any object can hold an array. On the left, each item on a hotel bill is stored separately in an array. To access the first charge for `room1` you would use:

```
costs.room1.items[0];
```

PROPERTY:

VALUE:

room1	:	items[420, 40, 10]
room2	:	items[460, 20, 20]
room3	:	items[230, 0, 0]
room4	:	items[620, 150, 60]

## OBJECTS IN AN ARRAY

The value of any element in an array can be an object (written using the object literal syntax). Here, to access the phone charge for room three, you would use:

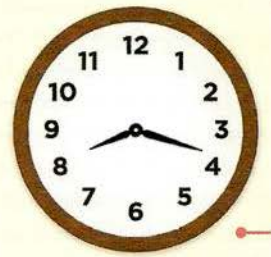
```
costs[2].phone;
```

INDEX NUMBER:

VALUE:

0	:	{accom: 420, food: 40, phone: 10}
1	:	{accom: 460, food: 20, phone: 20}
2	:	{accom: 230, food: 0, phone: 0}
3	:	{accom: 620, food: 150, phone: 60}

# WHAT ARE BUILT-IN OBJECTS?



Browsers come with a set of built-in objects that represent things like the browser window and the current web page shown in that window. These built-in objects act like a toolkit for creating interactive web pages.

The objects *you* create will usually be specifically written to suit *your* needs. They model the data used within, or contain functionality needed by, your script. Whereas, the built-in objects contain functionality commonly needed by many scripts.

As soon as a web page has loaded into the browser, these objects are available to use in your scripts.

These built-in objects help you get a wide range of information such as the width of the browser window, the content of the main heading in the page, or the length of text a user entered into a form field.

You access their properties or methods using dot notation, just like you would access the properties or methods of an object you had written yourself.



The first thing you need to do is get to know what tools are available. You can imagine that your new toolkit has three compartments:

1

### BROWSER OBJECT MODEL

The Browser Object Model contains objects that represent the current browser window or tab. It contains objects that model things like browser history and the device's screen.

3

### GLOBAL JAVASCRIPT OBJECTS

The global JavaScript objects represent things that the JavaScript language needs to create a model of. For example, there is an object that deals only with dates and times.

2

### DOCUMENT OBJECT MODEL

The Document Object Model uses objects to create a representation of the current page. It creates a new object for each element (and each individual section of text) within the page.

#### WHAT DOES THIS SECTION COVER?

You have already seen how to access the properties and methods of an object, so the purpose of this section is to let you know:

- What built-in objects are available to you
- What their main properties and methods do

There will be a few examples in the remaining part of this chapter to ensure you know how to use them. Then, throughout the rest of the entire book, you will see many practical examples of how they are used in a range of situations.

#### WHAT IS AN OBJECT MODEL?

You have seen that an object can be used to create a model of something from the real world using data.

An **object model** is a group of objects, each of which represent related things from the real world. Together they form a model of something larger.

Two pages back, it was noted that an array can hold a set of objects, or that the property of an object could be an array. It is also possible for the property of an object to be another object. When an object is nested inside another object, you may hear it referred to as a child object.

# THREE GROUPS OF BUILT-IN OBJECTS

## USING BUILT-IN OBJECTS:

The three sets of built-in objects each offer a different range of tools that help you write scripts for web pages.

Chapter 5 is dedicated to the Document Object Model because it is needed to access and update the contents of a web page.

The other two sets of objects will be introduced in this chapter, and then you will see them used throughout the rest of the book.

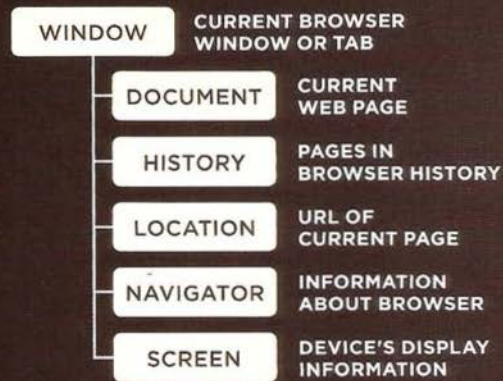
This book will teach you how to use these built-in objects and what type of information you can get from each one. You will also see examples that use many of their most popular features.

We do not have space to exhaustively document every object in each of these models in this book, so you can find a list of links to online resources at: <http://javascriptbook.com/resources>

## BROWSER OBJECT MODEL

The Browser Object Model creates a model of the browser tab or window.

The topmost object is the `window` object, which represents current browser window or tab. Its child objects represent other browser features.



## EXAMPLES

The `window` object's `print()` method will cause the browser's print dialog box to be shown:  
`window.print();`

The `screen` object's `width` property will let you find the width of the device's screen in pixels:  
`window.screen.width;`

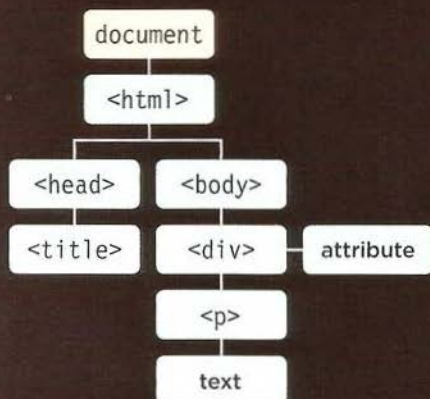
You meet the `window` object on p124 along with some properties of the `screen` and `history` objects.



## DOCUMENT OBJECT MODEL

The Document Object Model (DOM) creates a model of the current web page.

The topmost object is the **document** object, which represents the page as a whole. Its child objects represent other items on the page.



### EXAMPLES

The **document** object's `getElementById()` method gets an element by the value of its `id` attribute:  
`document.getElementById('one');`

The **document** object's `lastModified` property will tell you the date that the page was last updated:  
`document.lastModified;`

You meet the **document** object on p126. Chapter 5 goes into this object model in depth.

## GLOBAL JAVASCRIPT OBJECTS

The global objects do not form a single model. They are a group of individual objects that relate to different parts of the JavaScript language.

The names of the global objects usually start with a capital letter, e.g., the **String** and **Date** objects.

These objects represent basic data types:

**STRING**

FOR WORKING WITH STRING VALUES

**NUMBER**

FOR WORKING WITH NUMERIC VALUES

**BOOLEAN**

FOR WORKING WITH BOOLEAN VALUES

These objects help deal with real-world concepts:

**DATE**

TO REPRESENT AND HANDLE DATES

**MATH**

FOR WORKING WITH NUMBERS AND CALCULATIONS

**REGEX**

FOR MATCHING PATTERNS WITHIN STRINGS OF TEXT

### EXAMPLES

The **String** object's `toUpperCase()` method makes all letters in the following variable uppercase:  
`hotel.toUpperCase();`

The **Math** object's `PI` property will return the value of pi:  
`Math.PI();`

You meet the **String**, **Number**, **Date**, and **Math** objects later in this chapter.

# THE BROWSER OBJECT MODEL: THE WINDOW OBJECT

The window object represents the current browser window or tab. It is the topmost object in the Browser Object Model, and it contains other objects that tell you about the browser.

Here are a selection of the window object's properties and methods. You can also see some properties of the screen and history objects (which are children of the window object).

PROPERTY	DESCRIPTION
<code>window.innerHeight</code>	Height of window (excluding browser chrome/user interface) (in pixels)
<code>window.innerWidth</code>	Width of window (excluding browser chrome/user interface) (in pixels)
<code>window.pageXOffset</code>	Distance document has been scrolled horizontally (in pixels)
<code>window.pageYOffset</code>	Distance document has been scrolled vertically (in pixels)
<code>window.screenX</code>	X-coordinate of pointer, relative to top left corner of screen (in pixels)
<code>window.screenY</code>	Y-coordinate of pointer, relative to top left corner of screen (in pixels)
<code>window.location</code>	Current URL of window object (or local file path)
<code>window.document</code>	Reference to document object, which is used to represent the current page contained in window
<code>window.history</code>	Reference to history object for browser window or tab, which contains details of the pages that have been viewed in that window or tab
<code>window.history.length</code>	Number of items in history object for browser window or tab
<code>window.screen</code>	Reference to screen object
<code>window.screen.width</code>	Accesses screen object and finds value of its width property (in pixels)
<code>window.screen.height</code>	Accesses screen object and finds value of its height property (in pixels)
METHOD	DESCRIPTION
<code>window.alert()</code>	Creates dialog box with message (user must click OK button to close it)
<code>window.open()</code>	Opens new browser window with URL specified as parameter (if browser has pop-up blocking software installed, this method may not work)
<code>window.print()</code>	Tells browser that user wants to print contents of current page (acts like user has clicked a print option in the browser's user interface)



# USING THE BROWSER OBJECT MODEL

Here, data about the browser is collected from the `window` object and its children, stored in the `msg` variable, and shown in the page. The `+=` operator adds data onto the end of the `msg` variable.

1. Two of the `window` object's properties, `innerWidth` and `innerHeight`, show width and height of the browser window.

2. Child objects are stored as properties of their parent object. So dot notation is used to access them, just like you would access any other property of that object.

In turn, to access the properties of the child object, another dot is used between the child object's name and its properties, e.g., `window.history.length`

3. The element whose `id` attribute has a value of `info` is selected, and the message that has been built up to this point is written into the page.

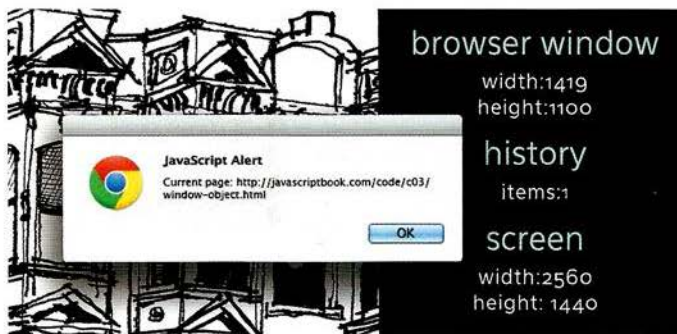
See p228 for notes on using `innerHTML` because it can be a security risk if it is not used correctly.

## JAVASCRIPT

c03/js/window-object.js

```
① var msg = '<h2>browser window</h2><p>width: ' + window.innerWidth + '</p>';  
   msg += '<p>height: ' + window.innerHeight + '</p>';  
② msg += '<h2>history</h2><p>items: ' + window.history.length + '</p>';  
   msg += '<h2>screen</h2><p>width: ' + window.screen.width + '</p>';  
   msg += '<p>height: ' + window.screen.height + '</p>';  
③ var el = document.getElementById('info');  
   el.innerHTML = msg;  
④ alert('Current page: ' + window.location);
```

## RESULT



4. The `window` object's `alert()` method is used to create a dialog box shown on top of the page. It is known as an **alert box**. Although this is a method of the `window` object, you may see it used on its own (as shown here) because the `window` object is treated as the default object if none is specified. (Historically, the `alert()` method was used to display warnings to users. These days there are better ways to provide feedback.)

# THE DOCUMENT OBJECT MODEL: THE DOCUMENT OBJECT

The topmost object in the Document Object Model (or DOM) is the document object. It represents the web page loaded into the current browser window or tab. You meet its child objects in Chapter 5.

Here are some properties of the document object, which tell you about the current page.

As you will see in Chapter 5, the DOM also creates an object for each element on the page.

PROPERTY	DESCRIPTION
<code>document.title</code>	Title of current document
<code>document.lastModified</code>	Date on which document was last modified
<code>document.URL</code>	Returns string containing URL of current document
<code>document.domain</code>	Returns domain of current document

The DOM is vital to accessing and amending the contents of the current web page.

The following are a few of the methods that select content or update the content of a page.

METHOD	DESCRIPTION
<code>document.write()</code>	Writes text to document (see restrictions on p226)
<code>document.getElementById()</code>	Returns element, if there is an element with the value of the <code>id</code> attribute that matches (full description see p195)
<code>document.querySelectorAll()</code>	Returns list of elements that match a CSS selector, which is specified as a parameter (see p202)
<code>document.createElement()</code>	Creates new element (see p222)
<code>document.createTextNode()</code>	Creates new text node (see p222)



# USING THE DOCUMENT OBJECT

This example gets information about the page, and then adds that information to the footer.

1. The details about the page are collected from properties of the document object.

These details are stored inside a variable called `msg`, along with HTML markup to display the information. Again, the `+=` operator adds the new value onto the existing content of the `msg` variable.

2. You have seen the document object's `getElementById()` method in several examples so far. It selects an element from the page using the value of its `id` attribute. You will see this method in more depth on p195.

## JAVASCRIPT

c03/js/document-object.js

```
① var msg = '<p><b>page title: </b>' + document.title + '<br />';  
  msg += '<b>page address: </b>' + document.URL + '<br />';  
  msg += '<b>last modified: </b>' + document.lastModified + '</p>';  
  
② var el = document.getElementById('footer');  
  el.innerHTML = msg;
```

## RESULT



**page title:** TravelWorthy  
**page address:** <http://javascriptbook.com/code/c03/document-object.html>  
**last modified:** 03/10/2014 14:46:23

See p228 for notes on using `innerHTML` because it can be a security risk if it is not used correctly.

The URL will look very different if you run this page locally rather than on a web server. It will likely begin with `file:///` rather than with `http://`.



# GLOBAL OBJECTS: STRING OBJECT

Whenever you have a value that is a string, you can use the properties and methods of the `String` object on that value. This example stores the phrase "Home sweet home " in a variable.

```
var saying = 'Home sweet home ';
```

These properties and methods are often used to work with text stored in variables or objects.

On the right-hand page, note how the variable name (`saying`) is followed by a dot, then the property or method that is being demonstrated (like the name of an object is followed by a dot and its properties or methods).

This is why the `String` object is known as both a **global object**, because it works anywhere within your script, and a **wrapper object** because it acts like a wrapper around any value that is a string - you can use this object's properties and methods on any value that is a string.

The `length` property counts the number of "code units" in a string. In the majority of cases, one character uses one code unit, and most programmers use it like this. But some of the rarely used characters take up two code units.

PROPERTY	DESCRIPTION
<code>length</code>	Returns number of characters in the string in most cases (see note bottom-left)
METHOD	DESCRIPTION
<code>toUpperCase()</code>	Changes string to uppercase characters
<code>toLowerCase()</code>	Changes string to lowercase characters
<code>charAt()</code>	Takes an index number as a parameter, and returns the character found at that position
<code>indexOf()</code>	Returns index number of the first time a character or set of characters is found within the string
<code>lastIndexOf()</code>	Returns index number of the last time a character or set of characters is found within the string
<code>substring()</code>	Returns characters found between two index numbers where the character for the first index number is included and the character for the last index number is not included
<code>split()</code>	When a character is specified, it splits the string each time it is found, then stores each individual part in an array
<code>trim()</code>	Removes whitespace from start and end of string
<code>replace()</code>	Like <code>find</code> and <code>replace</code> , it takes one value that should be found, and another to replace it (by default, it only replaces the first match it finds)



Each character in a string is automatically given a number, called an **index number**. Index numbers always start at zero and not one (just like for items in an array).

Home sweet home

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

EXAMPLE RESULT

`saying.length;` Home sweet home 16

EXAMPLE RESULT

`saying.toUpperCase();` Home sweet home 'HOME SWEET HOME '

`saying.toLowerCase();` Home sweet home 'home sweet home '

`saying.charAt(12);` Home sweet home 'o'

`saying.indexOf('ee');` Home sweet home 7

`saying.lastIndexOf('e');` Home sweet home 14

`saying.substring(8,14);` Home sweet home 'et hom'

`saying.split(' ');` Home sweet home ['Home', 'sweet', 'home', '']

`saying.trim();` Home sweet home 'Home sweet home'

`saying.replace('me','w');` Home sweet home 'How sweet home '

# WORKING WITH STRINGS

This example demonstrates the `length` property and many of the string object's methods shown on the previous page.

1. This example starts by storing the phrase "Home sweet home" in a variable called `saying`.

2. The next line tells you how many characters are in the string using the `length` property of the `String` object and stores the result in a variable called `msg`.

3. This is followed by examples showing several of the `String` object's methods.

The name of the variable (`saying`) is followed by a dot, then the property or method that is being demonstrated (in the same way that the other objects in this chapter used the dot notation to indicate a property or method of an object).

## JAVASCRIPT

c03/js/string-object.js

```
① var saying = 'Home sweet home ';\n② var msg = '<h2>length</h2><p>' + saying.length + '</p>';\nmsg += '<h2>uppercase</h2><p>' + saying.toUpperCase() + '</p>';\nmsg += '<h2>lowercase</h2><p>' + saying.toLowerCase() + '</p>';\nmsg += '<h2>character index: 12</h2><p>' + saying.charAt(12) + '</p>';\n③ msg += '<h2>first ee</h2><p>' + saying.indexOf('ee') + '</p>';\nmsg += '<h2>last e</h2><p>' + saying.lastIndexOf('e') + '</p>';\nmsg += '<h2>character index: 8-14</h2><p>' + saying.substring(8, 14) + '</p>';\nmsg += '<h2>replace</h2><p>' + saying.replace('me', 'w') + '</p>';\n④ [var el = document.getElementById('info');\nel.innerHTML = msg;
```

## RESULT



4. The final two lines select the element with an `id` attribute whose value is `info` and then add the value of the `msg` variable inside that element.

(Remember, security issues with using the `innerHTML` property are discussed on p228.)



# DATA TYPES REVISITED

In JavaScript there are six data types:  
Five of them are described as simple (or primitive) data types.  
The sixth is the object (and is referred to as a complex data type).

## SIMPLE OR PRIMITIVE DATA TYPES

JavaScript has five **simple** (or **primitive**) data types:

1. **String**
2. **Number**
3. **Boolean**
4. **Undefined** (a variable that has been declared, but no value has been assigned to it yet)
5. **Null** (a variable with no value - it may have had one at some point, but no longer has a value)

As you have seen, both the web browser and the current document can be modeled using objects (and objects can have methods and properties).

But it can be confusing to discover that a simple value (like a string, a number, or a Boolean) can have methods and properties. Under the hood, JavaScript treats every variable as an object in its own right.

**String:** If a variable, or the property of an object, contains a string, you can use the properties and methods of the `String` object on it.

**Number:** If a variable, or property of an object, stores a number, you can use the properties and methods of the `Number` object on it (see next page).

**Boolean:** There is a `Boolean` object. It is rarely used.

(Undefined and null values do not have objects.)

## COMPLEX DATA TYPE

JavaScript also defines a complex data type:

### 6. Object

Under the hood, arrays and functions are considered types of objects.

#### ARRAYS ARE OBJECTS

As you saw on p118, an array is a set of key/value pairs (just like any other object). But you do not specify the name in the key/value pair of an array - it is an index number.

Like other objects, arrays have properties and methods. On p72 you saw that arrays have a property called `length`, which tells you how many items are in that array. There is also a set of methods you can use with any array to add items to it, remove items from it, or reorder its contents. You will meet those methods in Chapter 12.

#### FUNCTIONS ARE OBJECTS

Technically, functions are also objects. But they have an additional feature: they are callable, which means you can tell the interpreter when you want to execute the statements that it contains.

# GLOBAL OBJECTS: NUMBER OBJECT

Whenever you have a value that is a number, you can use the methods and properties of the Number object on it.

These methods are helpful when dealing with a range of applications from financial calculations to animations.

Many calculations involving currency (such as tax rates) will need to be rounded to a specific number of decimal places.

Or, in an animation, you might want to specify that certain elements should be evenly spaced out across the page.

METHOD	DESCRIPTION
<code>isNaN()</code>	Checks if the value is not a number
<code>toFixed()</code>	Rounds to specified number of decimal places (returns a string)
<code>toPrecision()</code>	Rounds to total number of places (returns a string)
<code>toExponential()</code>	Returns a string representing the number in exponential notation

## COMMONLY USED TERMS:

- An **integer** is a whole number (not a fraction).
- A **real number** is a number that can contain a fractional part.
- A **floating point number** is a real number that uses decimals to represent a fraction. The term *floating point* refers to the decimal point.
- **Scientific notation** is a way of writing numbers that are too big or too small to be conveniently written in decimal form. For example: 3,750,000,000 can be represented as  $3.75 \times 10^9$  or  $3.75e+12$ .



# WORKING WITH DECIMAL NUMBERS

As with the String object, the properties and methods of the Number object can be used with with any value that is a number.

1. In this example, a number is stored in a variable called `originalNumber`, and it is then rounded up or down using two different techniques.

In both cases, you need to indicate how many digits you want to round to. This is provided as a parameter in the parentheses for that method.

## JAVASCRIPT

c03/js/number-object.js

```
① var originalNumber = 10.23456;  
   3 decimal places  
   var msg = '<h2>original number</h2><p>' + originalNumber + '</p>';  
② msg += '<h2>toFixed()</h2><p>' + originalNumber.toFixed(3); + '</p>';  
③ msg += '<h2>toFixed()</h2><p>' + originalNumber.toFixed(3) + '</p>';  
   var el = document.getElementById('info');  
   el.innerHTML = msg; 3 digits
```

## RESULT



2. `originalNumber.toFixed(3)` will round the number stored in the variable `originalNumber` to three decimal places. (The number of decimal places is specified in the parentheses.) It will return the number as a string. It returns a string because fractions cannot always be accurately represented using floating point numbers.

2. `toFixed(3)` uses the number in parentheses to indicate the total number of digits the number should have. It will also return the number as a string. (It may return a scientific notation if there are more digits than the specified number of positions.)

# GLOBAL OBJECTS: MATH OBJECT

The Math object has properties and methods for mathematical constants and functions.

PROPERTY	DESCRIPTION
<code>Math.PI</code>	Returns pi (approximately 3.14159265359)

METHOD	DESCRIPTION
<code>Math.round()</code>	Rounds number to the nearest integer
<code>Math.sqrt(n)</code>	Returns square root of positive number, e.g., <code>Math.sqrt(9)</code> returns 3
<code>Math.ceil()</code>	Rounds number up to the nearest integer
<code>Math.floor()</code>	Rounds number down to the nearest integer
<code>Math.random()</code>	Generates a random number between 0 (inclusive) and 1 (not inclusive)

Because it is known as a **global object**, you can just use the name of the Math object followed by the property or method you want to access.

Typically you will then store the resulting number in a variable. This object also has many trigonometric functions such as `sin()`, `cos()`, and `tan()`.

The trigonometric functions return angles in radians which can then be converted into degrees if you divide the number by  $(\pi/180)$ .



# MATH OBJECT TO CREATE RANDOM NUMBERS

This example is designed to generate a random whole number between 1 and 10.

The `Math` object's `random()` method generates a random number between 0 and 1 (with many decimal places).

To get a random whole number between 1 and 10, you need to multiply the randomly generated number by 10.

This number will still have many decimal places, so you can round it down to the nearest integer.

The `floor()` method is used to specifically round a number down (rather than up or down).

This will give you a value between 0 and 9. You then add 1 to make it a number between 1 and 10.

## JAVASCRIPT

c03/js/math-object.js

```
var randomNum = Math.floor((Math.random() * 10) + 1);  
  
var el = document.getElementById('info');  
el.innerHTML = '<h2>random number</h2><p>' + randomNum + '</p>';
```

## RESULT



random number  
7

If you used the `round()` method instead of the `floor()` method, the numbers 1 and 10 would be chosen around half of the number of times that 2-9 would be chosen.

Anything between 1.5 and 1.999 would get rounded up to 2, and anything between 9 and 9.5 would be rounded down to 9.

Using the `floor()` method ensures that the number is always rounded down to the nearest integer, and you can then add 1 to ensure the number is between 1 and 10.



# CREATING AN INSTANCE OF THE DATE OBJECT

In order to work with dates, you create an instance of the **Date** object. You can then specify the time and date that you want it to represent.

To create a **Date** object, use the **Date()** object constructor. The syntax is the same for creating any object with a constructor function (see p108). You can use it to create more than one **Date** object.

By default, when you create a **Date** object it will hold today's date and the current time. If you want it to store another date, you must explicitly specify the date and time you want it to hold.

VARIABLE NAME      NEW KEYWORD

```
var today = new Date();
```

VARIABLE DECLARATION      ASSIGNMENT OPERATOR      DATE OBJECT CONSTRUCTOR

You can think of the above as creating a variable called **today** that holds a number. This is because in JavaScript, dates are stored as a number: specifically the number of milliseconds since midnight on January 1, 1970.

Note that the current date / time is determined by the computer's clock. If the user is in a different time zone than you, their day may start earlier or later than yours. Also, if the internal clock on their computer has the wrong date or time, the **Date** object could reflect this by holding the wrong date.

The **Date()** object constructor tells the JavaScript interpreter that this variable is a date, and this in turn allows you to use the **Date** object's methods to set and retrieve dates and times from this **Date** object (see right-hand page for a list of methods).

You can set the date and/or time using any of the following formats (or methods shown on the right):

```
var dob = new Date(1996, 11, 26, 15, 45, 55);  
var dob = new Date('Dec 26, 1996 15:45:55');  
var dob = new Date(1996, 11, 26);
```



# GLOBAL OBJECTS: DATE OBJECT (AND TIME)

Once you have created a Date object, the following methods let you set and retrieve the time and date that it represents.

METHOD		DESCRIPTION
<code>getDate()</code>	<code>setDate()</code>	Returns / sets the day of the month (1-31)
<code>getDay()</code>		Returns the day of the week (0-6)
<code>getFullYear()</code>	<code>setFullYear()</code>	Returns / sets the year (4 digits)
<code>getHours()</code>	<code>setHours()</code>	Returns / sets the hour (0-23)
<code>getMilliseconds()</code>	<code>setMilliseconds()</code>	Returns / sets the milliseconds (0-999)
<code>getMinutes()</code>	<code>setMinutes()</code>	Returns / sets the minutes (0-59)
<code>getMonth()</code>	<code>setMonth()</code>	Returns / sets the month (0-11)
<code>getSeconds()</code>	<code>setSeconds()</code>	Returns / sets the seconds (0-59)
<code>getTime()</code>	<code>setTime()</code>	Number of milliseconds since January 1, 1970, 00:00:00 UTC (Coordinated Universal Time) and a negative number for any time before
<code>getTimezoneOffset()</code>		Returns time zone offset in mins for locale
<code>toDatestring()</code>		Returns "date" as a human-readable string
<code>toTimeString()</code>		Returns "time" as a human-readable string
<code>toString()</code>		Returns a string representing the specified date

The `toDatestring()` method will display the date in the following format:  
Wed Apr 16 1975.

If you want to display the date in another way, you can construct a different date format using the individual methods listed above to represent the individual parts: day, date, month, year.

`toTimeString()` shows the time. Several programming languages specify dates in milliseconds since midnight on Jan 1, 1970. This is known as Unix time.

A visitor's location may affect time zones and language spoken. Programmers use the term **locale** to refer to this kind of location-based information.

The Date object does not store the names of days or months as they vary between languages.

Instead, it uses a number from 0 to 6 for the days of the week and 0 to 11 for the months.

To show their names, you need to create an array to hold them (see p143).

# CREATING A DATE OBJECT

1. In this example, a new Date object is created using the Date() object constructor. It is called today.

If you do not specify a date when creating a Date object, it will contain the date and time when the JavaScript interpreter encounters that line of code.

Once you have an instance of the Date object (holding the current date and time), you can use any of its properties or methods.

## JAVASCRIPT

c03/js/date-object.js

```
① var today = new Date();  
② var year = today.getFullYear();  
  
③ [var e1 = document.getElementById('footer');  
   e1.innerHTML = '<p>Copyright &copy; ' + year + '</p>';
```

## RESULT



Copyright ©2014

2. In this example, you can see that getFullYear() is used to return the year of the date being stored in the Date object.

3. In this case, it is being used to write the current year in a copyright statement.



# WORKING WITH DATES & TIMES

To specify a date and time, you can use this format:

YYYY, MM, DD, HH, MM, SS  
1996, 04, 16, 15, 45, 55

This represents 3:45pm and 55 seconds on April 16, 1996.

The order and syntax for this is:

**Year** four digits  
**Month** 0-11 (Jan is 0)  
**Day** 1-31  
**Hour** 0-23  
**Minutes** 0-59  
**Seconds** 0-59  
**Milliseconds** 0-999

Another way to format the date and time is like this:

MMM DD, YYYY HH:MM:SS  
Apr 16, 1996 15:45:55

You can omit the time portion if you do not need it.

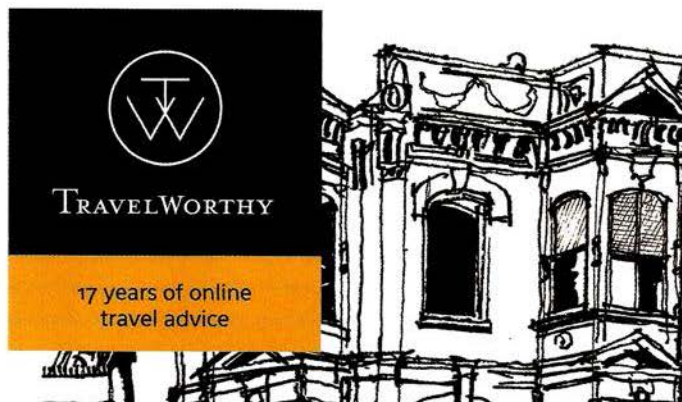
## JAVASCRIPT

c03/js/date-object-difference.js

```
1 var today = new Date();  
2 var year = today.getFullYear();  
3 var est = new Date('Apr 16, 1996 15:45:55');  
4 var difference = today.getTime() - est.getTime();  
5 difference = (difference / 31556900000);
```

```
var elMsg = document.getElementById('message');  
elMsg.textContent = Math.floor(difference) + ' years of online travel advice';
```

## RESULT



1. In this example, you can see a date being set in the past.

2. If you try to find the difference between two dates, you will end up with a result in milliseconds.

3. To get the difference in days/weeks/years, you divide this number by the number of milliseconds in a day/week/year.

Here the number is divided by 31,556,900,000 - the number of milliseconds in a year (that is not a leap year).









# EXAMPLE

## FUNCTIONS, METHODS & OBJECTS

This example is split into two parts. The first shows you the details about the hotel, room rate, and offer rate. The second part indicates when the offer expires.

All of the code is placed inside an immediately invoked function expression (IIFE) to ensure any variable names used in the script do not clash with variable names used in other scripts.

The first part of the script creates a `hotel` object; it has three properties (the hotel name, room rate, and percentage discount being offered), plus a method to calculate the offer price which is shown to the user.

The details of the discount are written into the page using information from this `hotel` object. To ensure that the discounted rate is shown with two decimal places (like most prices are shown) the `.toFixed()` method of the `Number` object is used.

The second part of the script shows that the offer will expire in seven days. It does this using a function called `offerExpires()`. The date currently set on the user's computer is passed as an argument to the `offerExpires()` function so that it can calculate when the offer ends.

Inside the function, a new `Date` object is created; and seven days is added to today's date. The `Date` object represents the days and months as numbers (starting at 0) so – to show the name of the day and month – two arrays are created storing all possible day and month names. When the message is written, it retrieves the appropriate day/month from those arrays.

The message to show the expiry date is built up inside a variable called `expiryMsg`. The code that calls the `offerExpires()` function and displays the message is at the end of the script. It selects the element where the message should appear and updates its content using the `innerHTML` property, which you will meet in Chapter 5.

# EXAMPLE

## FUNCTIONS, METHODS & OBJECTS

c03/js/example.js

JAVASCRIPT

```
/* The script is placed inside an immediately invoked function expression
   which helps protect the scope of variables */

-(function() {

    // PART ONE: CREATE HOTEL OBJECT AND WRITE OUT THE OFFER DETAILS

    // Create a hotel object using object literal syntax
    var hotel = {
        name: 'Park',
        roomRate: 240, // Amount in dollars
        discount: 15, // Percentage discount
        offerPrice: function() {
            var offerRate = this.roomRate * ((100 - this.discount) / 100);
            return offerRate;
        }
    }

    // Write out the hotel name, standard rate, and the special rate
    var hotelName, roomRate, specialRate; // Declare variables

    hotelName = document.getElementById('hotelName'); // Get elements
    roomRate = document.getElementById('roomRate');
    specialRate = document.getElementById('specialRate');

    hotelName.textContent = hotel.name; // Write hotel name
    roomRate.textContent = '$' + hotel.roomRate.toFixed(2); // Write room rate
    specialRate.textContent = '$' + hotel.offerPrice(); // Write offer price
}
```

If you read the comments in the code, you can see how this example works.



# EXAMPLE

## FUNCTIONS, METHODS & OBJECTS

JAVASCRIPT

c03/js/example.js

```
// PART TWO: CALCULATE AND WRITE OUT THE EXPIRY DETAILS FOR THE OFFER
var expiryMsg; // Message displayed to users
var today;     // Today's date
var elEnds;    // The element that shows the message about the offer ending

function offerExpires(today) {
    // Declare variables within the function for local scope
    var weekFromToday, day, date, month, year, dayNames, monthNames;
    // Add 7 days time (added in milliseconds)
    weekFromToday = new Date(today.getTime() + 7 * 24 * 60 * 60 * 1000);
    // Create arrays to hold the names of days / months
    dayNames = ['Sunday', 'Monday', 'Tuesday', 'Wednesday', 'Thursday',
    ➔ 'Friday', 'Saturday'];
    monthNames = ['January', 'February', 'March', 'April', 'May', 'June',
    ➔ 'July', 'August', 'September', 'October', 'November', 'December'];
    // Collect the parts of the date to show on the page
    day = dayNames[weekFromToday.getDay()];
    date = weekFromToday.getDate();
    month = monthNames[weekFromToday.getMonth()];
    year = weekFromToday.getFullYear();
    // Create the message
    expiryMsg = 'Offer expires next ';
    expiryMsg += day + ' <br />(' + date + ' ' + month + ' ' + year + ')';
    return expiryMsg;
}

today = new Date(); // Put today's date in variable
elEnds = document.getElementById('offerEnds'); // Get the offerEnds element
elEnds.innerHTML = offerExpires(today); // Add the expiry message

// Finish the immediately invoked function expression
}());
```

➔ This symbol indicates that the code is wrapping from the previous line and should not contain line breaks.

This is a good demonstration of several concepts relating to date, but if the user has the wrong date on their computer (perhaps their clock is set incorrectly), it will not show a date seven days from now – it will show a date seven days from the time the computer thinks it is.



# SUMMARY

## FUNCTIONS, METHODS & OBJECTS

- ▶ Functions allow you to group a set of related statements together that represent a single task.
- ▶ Functions can take parameters (information required to do their job) and may return a value.
- ▶ An object is a series of variables and functions that represent something from the world around you.
- ▶ In an object, variables are known as properties of the object; functions are known as methods of the object.
- ▶ Web browsers implement objects that represent both the browser window and the document loaded into the browser window.
- ▶ JavaScript also has several built-in objects such as `String`, `Number`, `Math`, and `Date`. Their properties and methods offer functionality that help you write scripts.
- ▶ Arrays and objects can be used to create complex data sets (and both can contain the other).