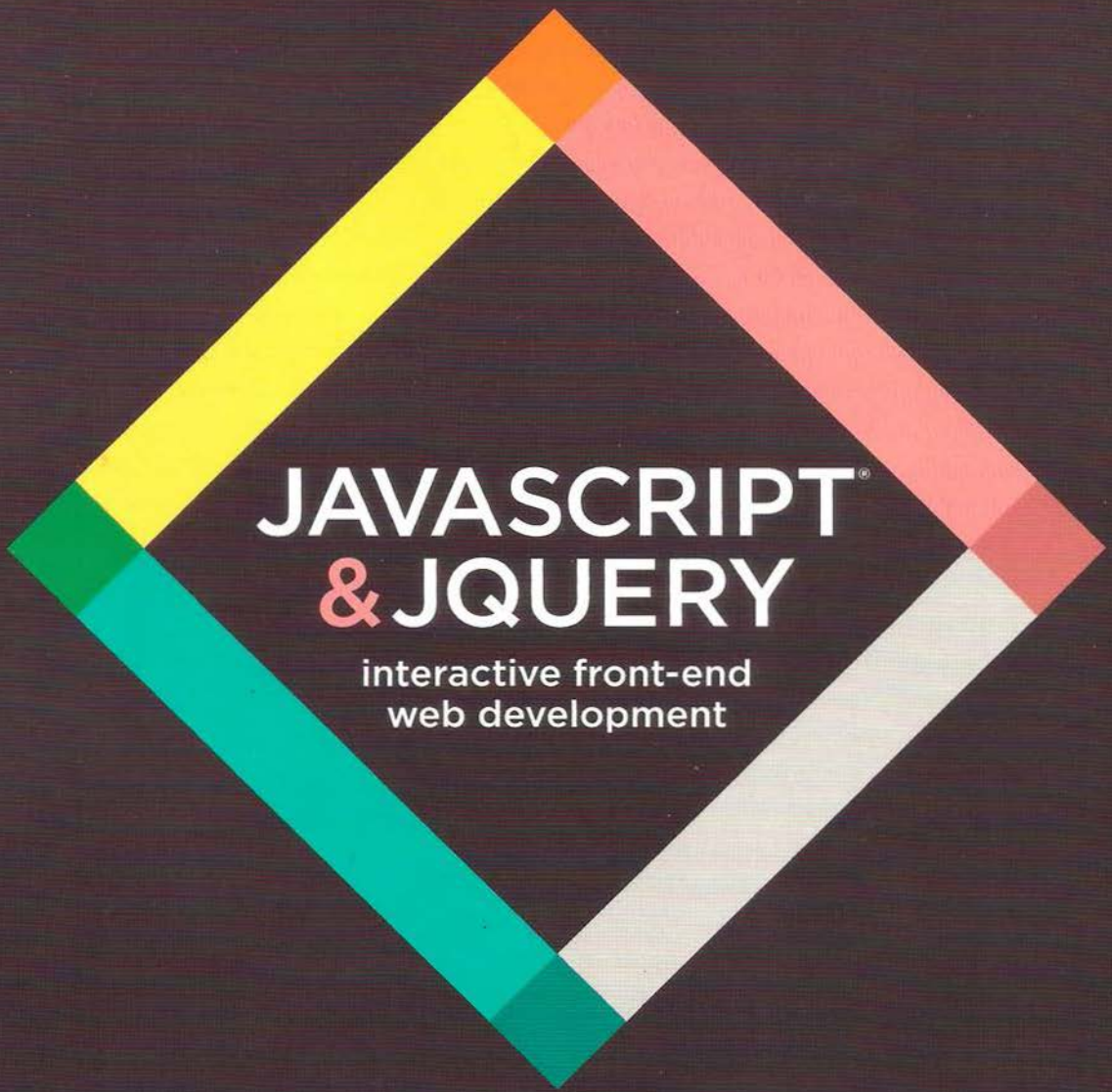


JAVASCRIPT® & JQUERY

interactive front-end web development

JON DUCKETT

JAVASCRIPT & JQUERY

Interactive Front-End Web Development

JON DUCKETT

Additional material by:

GILLES RUPPERT & JACK MOORE

WILEY

The text stock is SFI certified

Published by
John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

ISBN: 978-1-118-53164-8
Manufactured in the United States of America
10 9 8 7 6 5 4 3 2 1

Library of Congress Control Number: 2013933932

TABLE OF CONTENTS

Try out & download the code in this book

www.javascriptbook.com

CREDITS

For John Wiley & Sons, Inc.

Executive Editor
Carol Long

Project Editor
Kevin Kent

Production Editor
Daniel Scribner

Editorial Manager
Mary Beth Wakefield

Associate Director of Marketing
David Mayhew

Marketing Manager
Lorna Mein

Business Manager
Amy Knies

Vice President and Executive Group Publisher
Richard Swadley

Associate Publisher
Jim Minatel

Project Coordinator, Cover
Todd Klemme

For Wagon Ltd.

Author
Jon Duckett

Co-Authors
Jack Moore
(Chapters 11 & 12)

Gilles Ruppert
(Chapter 13)

Technical Review
Mathias Bynens

Review Team
Chris Ullman
David Lean
Harrison Thrift
Jay Bursky
Richard Eskins
Scott Robin
Stachu Korick

Thank you
Annette Loudon
Michael Tomko
Michael Vella Zarb
Pam Coca
Rishabh Pugalia

Cover Design
Emme Stone

Design
Emme Stone
Jon Duckett

Photography
John Stewardson
`johnstewardson.com`

Illustration
Matthew Cencich
(Hotel in Chapter 3)

Emme Stone
(Teacher in Chapter 4)

Additional Photography

Electronics in Chapters 8 & 9:
Aaron Nielsen
Arkadiusz Jan Sikorski
Matt Mets
Mirsad Dedović
Steve Lodefink

`javascriptbook.com/credits`

INTRODUCTION

This book explains how JavaScript can be used in browsers to make websites more interactive, interesting, and user-friendly. You will also learn about jQuery because it makes writing JavaScript a lot easier.

To get the most out of this book, you will need to know how to build web pages using HTML and CSS. Beyond that, no prior experience with programming is necessary. Learning to program with JavaScript involves:

1

Understanding some **basic programming concepts** and the terms that JavaScript programmers use to describe them.

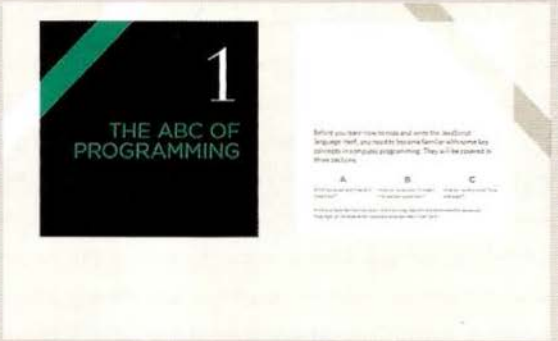
2

Learning **the language** itself, and, like all languages, you need to know its vocabulary and how to structure your sentences.

3

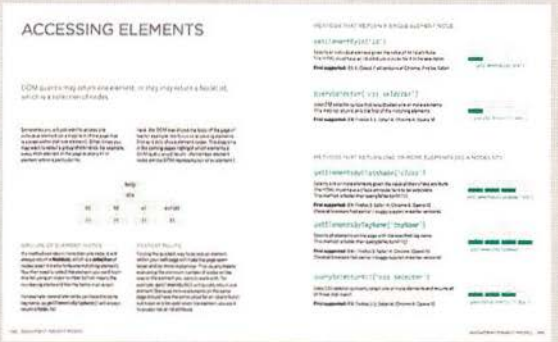
Becoming familiar with **how it is applied** by looking at examples of how JavaScript is commonly used in websites today.

The only equipment you need to use this book are a computer with a modern web browser installed, and your favorite code editor, (e.g., Notepad, TextEdit, Sublime Text, or Coda).



Introduction pages come at the beginning of each chapter. They introduce the key topics you will learn about.

Reference pages introduce key pieces of JavaScript. HTML code is shown in blue, CSS code in pink, and JavaScript in green.



Background pages appear on white. They explain the context of the topics covered that are discussed in each chapter.

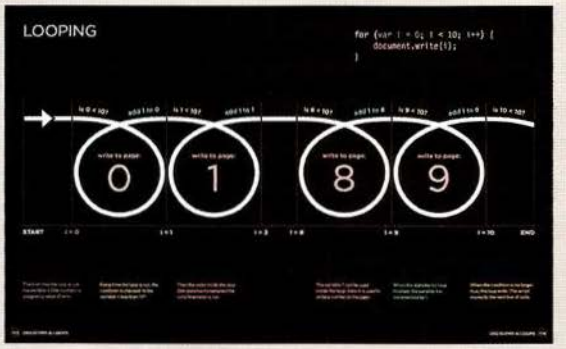


Diagram and infographics pages are shown on a dark background. They provide a simple, visual reference to topics discussed.

Example pages bring together the topics you have learned in that chapter and demonstrate how they can be applied.

Summary pages come at the end of each chapter. They remind you of the key topics that were covered in each chapter.

HOW JAVASCRIPT MAKES WEB PAGES MORE INTERACTIVE

JavaScript allows you to make web pages more interactive by accessing and modifying the content and markup used in a web page while it is being viewed in the browser.

1 ACCESS CONTENT

You can use JavaScript to select any element, attribute, or text from an HTML page. For example:

- Select the text inside all of the `<h1>` elements on a page
- Select any elements that have a `class` attribute with a value of `note`
- Find out what was entered into a text input whose `id` attribute has a value of `email`

2 MODIFY CONTENT

You can use JavaScript to add elements, attributes, and text to the page, or remove them. For example:

- Add a paragraph of text after the first `<h1>` element
- Change the value of `class` attributes to trigger new CSS rules for those elements
- Change the size or position of an `<img>` element

3
PROGRAM RULES

You can specify a set of steps for the browser to follow (like a recipe), which allows it to access or change the content of a page. For example:

- A gallery script could check which image a user clicked on and display a larger version of that image.
- A mortgage calculator could collect values from a form, perform a calculation, and display repayments.
- An animation could check the dimensions of the browser window and move an image to the bottom of the viewable area (also known as the viewport).

JavaScript encompasses many of the traditional rules of programming.

It can make the web page feel interactive by responding to what the user does.

4
REACT TO EVENTS

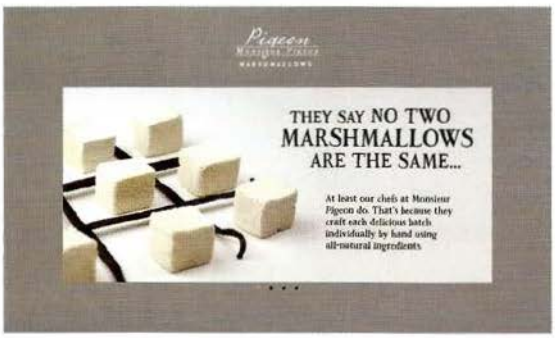
You can specify that a script should run when a specific event has occurred. For example, it could be run when:

- A button is pressed
- A link is clicked (or tapped) on
- A cursor hovers over an element
- Information is added to a form
- An interval of time has passed
- A web page has finished loading

EXAMPLES OF JAVASCRIPT IN THE BROWSER

Being able to change the content of an HTML page while it is loaded in the browser is very powerful. The examples below rely on the ability to:

Access the content of the page
Modify the content of the page
Program rules or instructions the browser can follow
React to events triggered by the user or browser



SLIDESHOWS

Shown in Chapter 11

Slideshows can display a number of different images (or other HTML content) within the same space on a given page. They can play automatically as a sequence, or users can click through the slides manually. They allow more content to be displayed within a limited amount of space.

React: Script triggered when the page loads
Access: Get each slide from the slideshow
Modify: Only show the first slide (hide others)
Program: Set a timer: when to show next slide
Modify: Change which slide is shown
React: When user clicks button for different slide
Program: Determine which slide to show
Modify: Show the requested slide

FORMS

Shown in Chapter 13

Validating forms (checking whether they have been filled in correctly) is important when information is supplied by users. JavaScript lets you alert the user if mistakes have been made. It can also perform sophisticated calculations based on any data entered and reveal the results to the user.

React: User presses the submit button when they have entered their name
Access: Get value from form field
Program: Check that the name is long enough
Modify: Show a warning message if the name is not long enough

The examples on these two pages give you a taste of what JavaScript can do within a web page, and of the techniques you will be learning throughout this book.

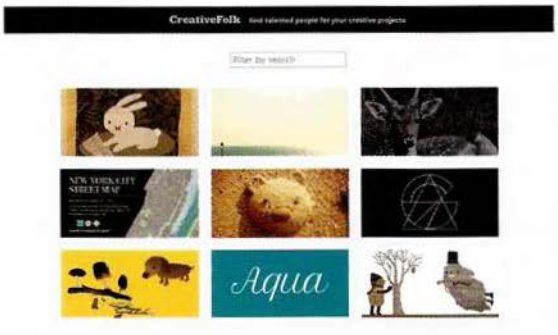
In the coming chapters, you will learn how and when to access or modify content, add programming rules, and react to events.

RELOAD PART OF PAGE

Shown in Chapter 8

You might not want to force visitors to reload the content of an entire web page, particularly if you only need to refresh a small portion of a page. Just reloading a section of the page can make a site feel like it is faster to load and more like an application.

React: Script triggered when user clicks on link
Access: The link that they clicked on
Program: Load the new content that was requested from that link
Access: Find the element to replace in the page
Modify: Replace that content with the new content



FILTERING DATA

Shown in Chapter 12

If you have a lot of information to display on a page, you can help users find information they need by providing filters. Here, buttons are generated using data in the attributes of the HTML `<img>` elements. When the user clicks on one of the buttons, they are only shown the images with that keyword.

React: Script triggered when page loads
Program: Collect keywords from images
Program: Turn the keywords into buttons the user can click on
React: User clicks on one of the buttons
Program: Find the relevant subset of images that should be shown
Modify: Show the subset of images that use that tag

THE STRUCTURE OF THIS BOOK

In order to teach you JavaScript, this book is divided into two sections:

CORE CONCEPTS

The first nine chapters introduce you to the basics of programming and the JavaScript language. Along the way you will learn how it is used to create more engaging, interactive, and usable websites.

Chapter 1 looks at some key concepts in computer programming, showing you how computers create models of the world using data, and how JavaScript is used to change the contents of an HTML page.

Chapters 2-4 cover the basics of the JavaScript language.

Chapter 5 explains how the Document Object Model (DOM) lets you access and change a document's contents while it is loaded into the browser.

Chapter 6 discusses how events can be used to trigger code.

Chapter 7 shows you how jQuery can make the process of writing scripts faster and easier.

Chapter 8 introduces you to Ajax, a set of techniques that allow you to just change part of a web page without reloading the entire page.

Chapter 9 covers Application Programming Interfaces (APIs), including new APIs that are part of HTML5 and those of sites like Google Maps.

PRACTICAL APPLICATIONS

By this point you will already have seen many examples of how JavaScript is used on popular websites. This section brings together all of the techniques you have learned so far, to give you practical demonstrations of how JavaScript is used by professional developers. Not only will you see a selection of in-depth examples, you will also learn more about the process of designing and writing scripts from scratch.

Chapter 10 deals with error-handling and debugging, and explains more about how JavaScript is processed.

Chapter 11 shows you techniques for creating content panels such as sliders, modal windows, tabbed panels, and accordions.

Chapter 12 demonstrates several techniques for filtering and sorting data. This includes filtering a gallery of images, and re-ordering the rows of a table by clicking on the column headings.

Chapter 13 deals with form enhancements and how to validate form entries.

Unless you are already a confident programmer, you will probably find it helpful to read the book from start to finish the first time. However, once you have grasped the basics, we hope it will continue to be a helpful reference as you create your own scripts.

HTML & CSS: A QUICK REFRESHER

Before looking at JavaScript, let's clarify some HTML & CSS terms. Note how HTML attributes and CSS properties use name/value pairs.

HTML ELEMENTS

HTML elements are added to the content of a page to describe its structure. An element consists of the opening and closing tags, plus its content.

Tags usually come in pairs with an opening tag and a closing tag. There are a few empty elements with no content, (e.g., `<img>`). They have one self-closing tag.

Opening tags can carry attributes, which tell us more about that element. Attributes have a name and a value. The value is usually given in quotes.

CSS RULES

CSS uses rules to indicate how the contents of one or more elements should be displayed in the browser. Each rule has a selector and a declaration block.

The CSS selector indicates which element(s) the rule applies to. The declaration block contains rules that indicate how those elements should appear.

Each declaration in the declaration block has a property (the aspect you want to control), and a value, which is the setting for that property.

BROWSER SUPPORT

Some early examples in this book do not work with Internet Explorer 8 and earlier (but alternative code samples that work in IE8 are available to download from `http://javascriptbook.com`). We explain techniques for dealing with older browsers in later chapters.

Each version of a web browser adds new features. Often these new features make tasks easier, or are considered better, than using older techniques.

But, website visitors do not always keep up with the latest browser releases, so website developers cannot always rely upon the latest technologies.

As you will see, there are many inconsistencies between browsers that affect JavaScript developers. jQuery will help you deal with cross-browser inconsistencies (it is one of the major reasons why jQuery rapidly gained popularity amongst web developers). But, before you learn jQuery, it helps to know what it is helping you to achieve.

To make JavaScript easier to learn, the first few chapters use some features of JavaScript that are not supported in IE8. But:

- You **will** learn how to deal with IE8 and older browsers in later chapters (because we know that many clients expect sites to work in IE8). It just requires knowledge of some extra code or requires you to be aware of some additional issues.
- Online, you will find alternatives available for each example that does not work in IE8. But please check the comments in those code samples to make sure you know about the about issues involved in using them.

1

THE ABC OF PROGRAMMING

Before you learn how to read and write the JavaScript language itself, you need to become familiar with some key concepts in computer programming. They will be covered in three sections:

A	B	C
What is a script and how do I create one?	How do computers fit in with the world around them?	How do I write a script for a web page?

Once you have learned the basics, the following chapters will show how the JavaScript language can be used to tell browsers what you want them to do.

1/a

WHAT IS A SCRIPT AND HOW DO I CREATE ONE?

A SCRIPT IS A SERIES OF INSTRUCTIONS

A script is a series of instructions that a computer can follow to achieve a goal. You could compare scripts to any of the following:

RECIPES

By following the instructions in a recipe, one-by-one in the order set out, cooks can create a dish they have never made before.

Some scripts are simple and only deal with one individual scenario, like a simple recipe for a basic dish. Other scripts can perform many tasks, like a recipe for a complicated three-course meal.

Another similarity is that, if you are new to cooking or programming, there is a lot of new terminology to learn.

HANDBOOKS

Large companies often provide handbooks for new employees that contain procedures to follow in certain situations.

For example, hotel handbooks may contain steps to follow in different scenarios such as when a guest checks in, when a room needs to be tidied, when a fire alarm goes off, and so forth.

In any of these scenarios, the employees need to follow only the steps for that one type of event. (You would not want someone going through every single step in the entire handbook while you were waiting to check in.) Similarly, in a complex script, the browser might use only a subset of the code available at any given time.

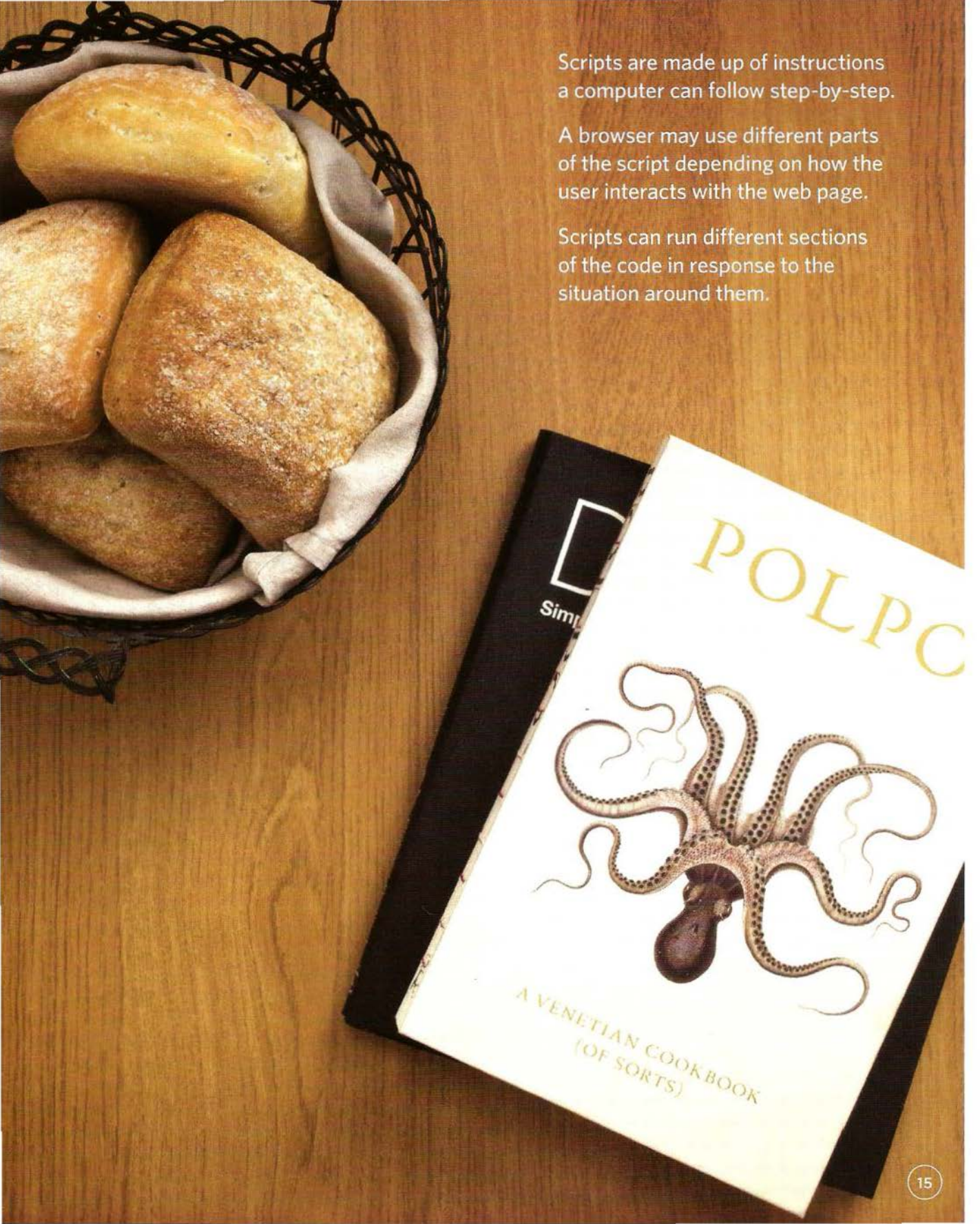
MANUALS

Mechanics often refer to car repair manuals when servicing models they are not familiar with. These manuals contain a series of tests to check the key functions of the car are working, along with details of how to fix any issues that arise.

For example, there might be details about how to test the brakes. If they pass this test, the mechanic can then go on to the next test without needing to fix the brakes. But, if they fail, the mechanic will need to follow the instructions to repair them.

The mechanic can then go back and test the brakes again to see if the problem is fixed. If the brakes now pass the test, the mechanic knows they are fixed and can move onto the next test.

Similarly, scripts can allow the browser to check the current situation and only perform a set of steps if that action is appropriate.



Scripts are made up of instructions a computer can follow step-by-step.

A browser may use different parts of the script depending on how the user interacts with the web page.

Scripts can run different sections of the code in response to the situation around them.

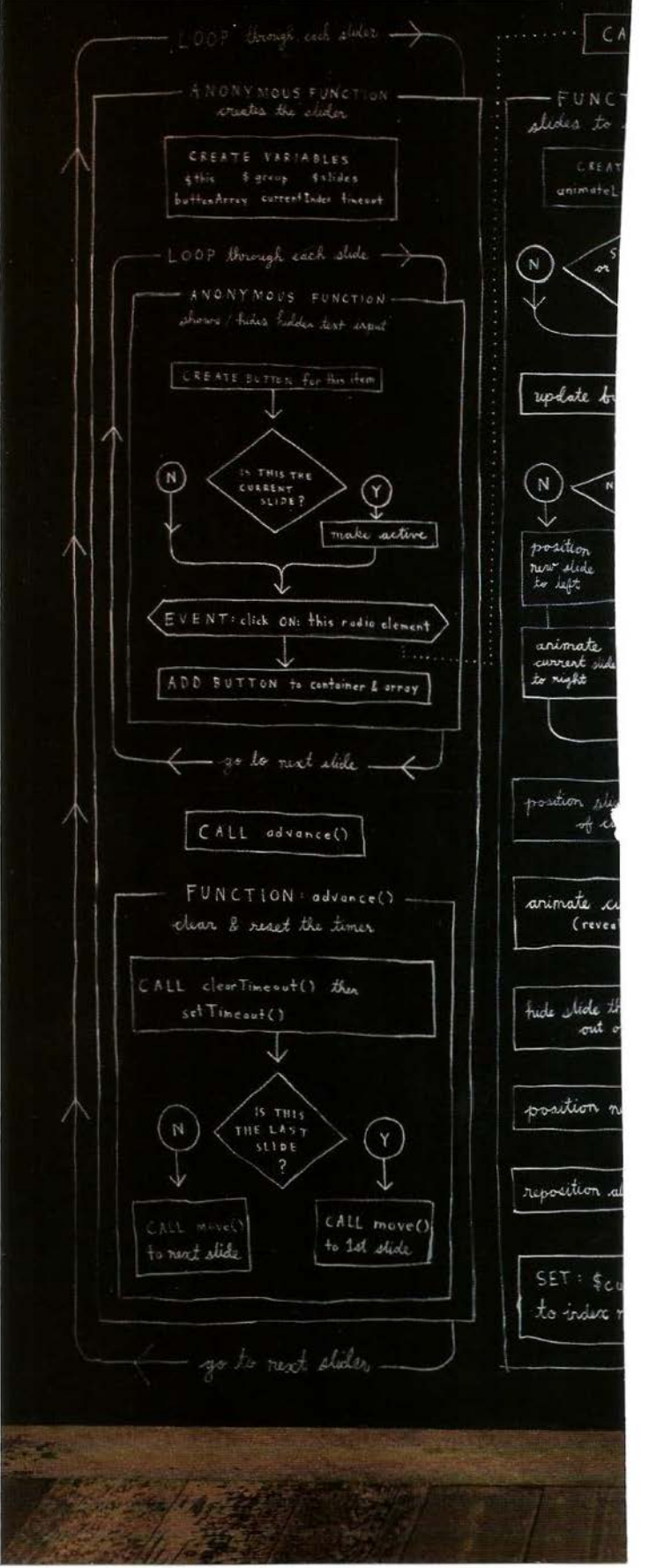
WRITING A SCRIPT

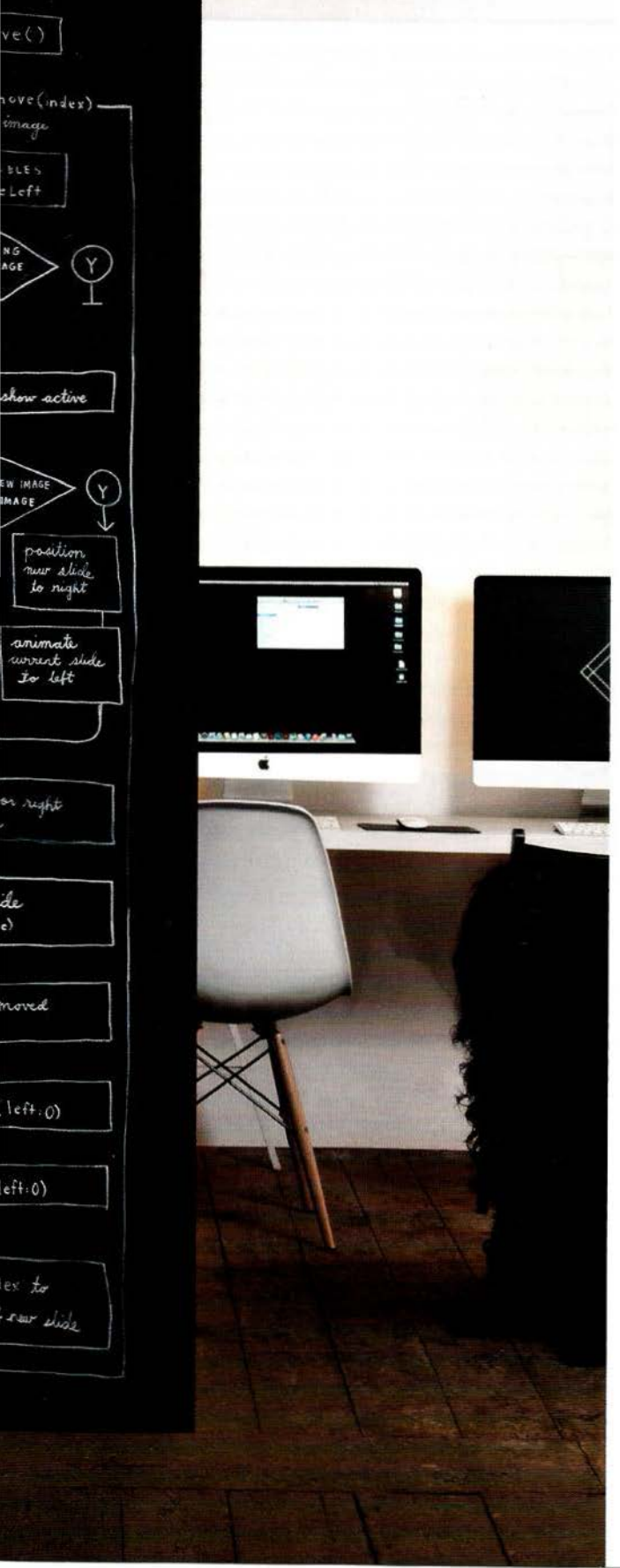
To write a script, you need to first state your goal and then list the tasks that need to be completed in order to achieve it.

Humans can achieve complex goals without thinking about them too much, for example you might be able to drive a car, cook breakfast, or send an email without a set of detailed instructions. But the first time we do these things they can seem daunting. Therefore, when learning a new skill, we often break it down into smaller tasks, and learn one of these at a time. With experience these individual tasks grow familiar and seem simpler.

Some of the scripts you will be reading or writing when you have finished this book will be quite complicated and might look intimidating at first. However, a script is just a series of short instructions, each of which is performed in order to solve the problem in hand. This is why creating a script is like writing a recipe or manual that allows a computer to solve a puzzle one step at a time.

It is worth noting, however, that a computer doesn't learn how to perform tasks like you or I might; it needs to follow instructions every time it performs the task. So a program must give the computer enough detail to perform the task as if every time were its first time.





Start with the big picture of what you want to achieve, and break that down into smaller steps.

1: DEFINE THE GOAL

First, you need to define the task you want to achieve. You can think of this as a puzzle for the computer to solve.

2: DESIGN THE SCRIPT

To design a script you split the goal out into a series of tasks that are going to be involved in solving this puzzle. This can be represented using a flowchart.

You can then write down individual steps that the computer needs to perform in order to complete each individual task (and any information it needs to perform the task), rather like writing a recipe that it can follow.

3: CODE EACH STEP

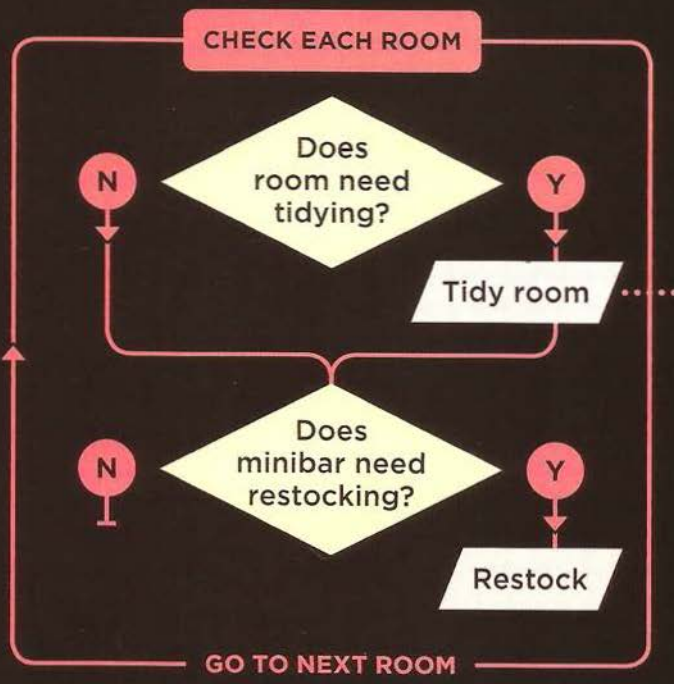
Each of the steps needs to be written in a programming language that the computer understands. In our case, this is JavaScript.

As tempting as it can be to start coding straight away, it pays to spend time designing your script before you start writing it.

DESIGNING A SCRIPT: TASKS

Once you know the **goal** of your script, you can work out the individual tasks needed to achieve it. This high-level view of the tasks can be represented using a flowchart.

FLOWCHART: TASKS OF A HOTEL CLEANER



DESIGNING A SCRIPT: STEPS

Each individual task may be broken down into a sequence of steps. When you are ready to code the script, these steps can then be translated into individual lines of code.

LIST: STEPS REQUIRED TO TIDY A ROOM

STEP 1	Remove used bedding
STEP 2	Wipe all surfaces
STEP 3	Vacuum floors
STEP 4	Fit new bedding
STEP 5	Remove used towels and soaps
STEP 6	Clean toilet, bath, sink, surfaces
STEP 7	Place new towels and soaps
STEP 8	Wipe bathroom floor

As you will see on the next page, the steps that a computer needs to follow in order to perform a task are often very different from those that you or I might take.

FROM STEPS TO CODE

Every step for every task shown in a flowchart needs to be written in a language the computer can understand and follow.

In this book, we are focussing on the JavaScript language and how it is used in web browsers.

Just like learning any new language, you need to get to grips with the:

- **Vocabulary:** The words that computers understand
- **Syntax:** How you put those words together to create instructions computers can follow

Along with learning the language itself, if you are new to programming, you will also need to learn how a computer achieves different types of goals using a **programmatic** approach to problem-solving.

Computers are very logical and obedient. They need to be told every detail of what they are expected to do, and they will do it without question. Because they need different types of instructions compared to you or I, everyone who learns to program makes lots of mistakes at the start. Don't be disheartened; in Chapter 10 you will see several ways to discover what might have gone wrong – programmers call this **debugging**.

You need to learn to "think" like a computer because they solve tasks in different ways than you or I might approach them.

Computers solve problems **programmatically**; they follow series of instructions, one after another. The type of instructions they need are often different to the type of instructions you might give to another human. Therefore, throughout the book you will not only learn the vocabulary and syntax that JavaScript uses, but you will also learn how to write instructions that computers can follow.

For example, when you look at the picture on the left how do you tell which person is the tallest? A computer would need explicit, step-by-step instructions, such as:

1. Find the height of the first person
2. Assume he or she is the "tallest person"
3. Look at the height of the remaining people one-by-one and compare their height to the "tallest person" you have found so far
4. At each step, if you find someone whose height is greater than the current "tallest person", he or she becomes the new "tallest person"
5. Once you have checked all the people, tell me which one is the tallest

So the computer needs to look at each person in turn, and for each one it performs a test ("Are they taller than the current tallest person?"). Once it has done this for each person it can give its answer.

DEFINING A GOAL & DESIGNING THE SCRIPT

Consider how you might approach a different type of script. This example calculates the cost of a name plaque. Customers are charged by the letter.

The first thing you should do is detail your goals for the script (what you want it to achieve):

Customers can have a name added to a plaque; each letter costs $5. When a user enters a name, show them how much it will cost.

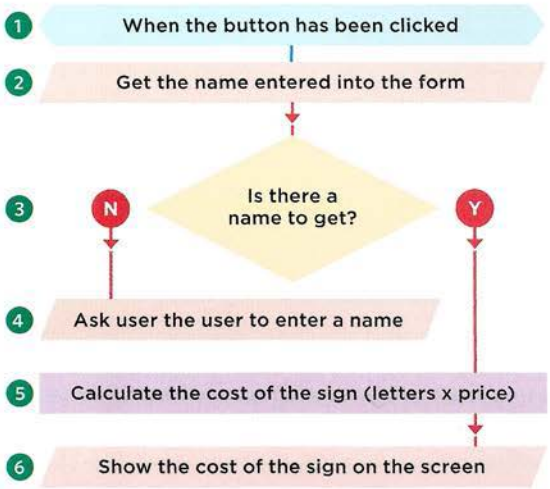
Next, break it into a series of tasks that have to be performed in order to achieve the goals:

1. The script is triggered when the button is clicked.
2. It collects the name entered into the form field.
3. It checks that the user has entered a value.
4. If the user has not entered anything, a message will appear telling them to enter a name.
5. If a name has been entered, calculate the cost of the sign by multiplying the number of letters by the cost per letter.
6. Show how much the plaque costs.

(These numbers correspond with the flowchart on the right-hand page.)

SKETCHING OUT THE TASKS IN A FLOWCHART

Often scripts will need to perform different tasks in different situations. You can use flowcharts to work out how the tasks fit together. The flowcharts show the paths between each step.



Arrows show how the script moves from one task to the next. The different shapes represent different types of tasks. In some places there are decisions which cause the code to follow different paths.

You will learn how to turn this example into code in Chapter 2. You will also see many more examples of different flowcharts throughout the book, and you will meet code that helps you deal with each of these types of situations.

Some experienced programmers use more complex diagram styles that are specifically designed to represent code - however, they have a steeper learning curve. These informal flowcharts will help you understand how scripts work while you are in the process of learning the language.

FLOWCHART KEY

Generic step	Event
Input or output	Decision

SUMMARY

THE ABC OF PROGRAMMING

A: What is a script and how do I create one?

- A script is a series of instructions that the computer can follow in order to achieve a goal.
- Each time the script runs, it might only use a subset of all the instructions.
- Computers approach tasks in a different way than humans, so your instructions must let the computer solve the task programmatically.
- To approach writing a script, break down your goal into a series of tasks and then work out each step needed to complete that task (a flowchart can help).

1/b

HOW DO COMPUTERS FIT IN WITH THE WORLD AROUND THEM?

COMPUTERS CREATE MODELS OF THE WORLD USING DATA

Here is a model of a hotel, along with some model trees, model people, and model cars. To a human, it is clear what kind of real-world object each one represents.

A computer has no predefined concept of what a hotel or car is. It does not know what they are used for. Your laptop or phone will not have a favorite brand of car, nor will it know what star rating your hotel is.

So how do we use computers to create hotel booking apps, or video games where players can race a car? The answer is that programmers create a very different kind of model, especially for computers.

Programmers make these models using data. That is not as strange or as scary as it sounds because the data is all the computer needs in order to follow the instructions you give it to carry out its tasks.

OBJECTS & PROPERTIES

If you could not see the picture of the hotel and cars, the data in the information boxes alone would still tell you a lot about this scene.

OBJECTS (THINGS)

In computer programming, each physical thing in the world can be represented as an **object**. There are two different **types** of objects here: a hotel and a car.

Programmers might say that there is one **instance** of the hotel object, and two **instances** of the car object.

Each object can have its own:

- Properties
- Events
- Methods

Together they create a working model of that object.

PROPERTIES (CHARACTERISTICS)

Both of the cars share common characteristics. In fact, all cars have a make, a color, and engine size. You could even determine their current speed. Programmers call these characteristics the **properties** of an object.

Each property has a **name** and a **value**, and each of these name/value pairs tells you something about each individual instance of the object.

The most obvious property of this hotel is its `name`. The value for that property is `Quay`. You can tell the number of rooms the hotel has by looking at the value next to the `rooms` property.

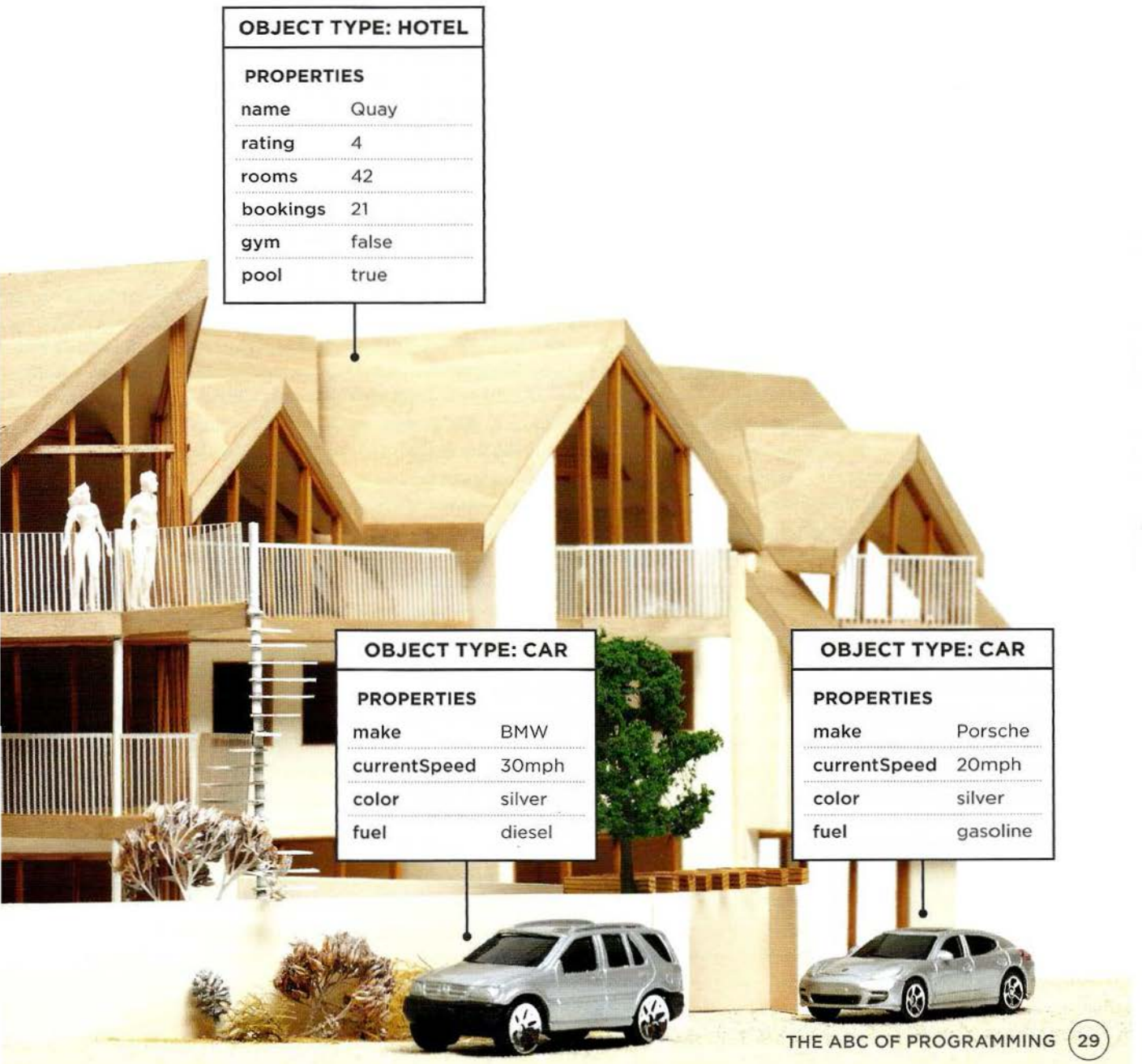
The idea of name/value pairs is used in both HTML and CSS. In HTML, an attribute is like a property; different attributes have different names, and each attribute can have a value. Similarly, in CSS you can change the color of a heading by creating a rule that gives the `color` property a specific value, or you can change the typeface it is written in by giving the `font-family` property a specific value. Name/value pairs are used a lot in programming.

HOTEL OBJECT

The hotel object uses property names and values to tell you about this particular hotel, such as the hotel's name, its rating, the number of rooms it has, and how many of these are booked. You can also tell whether or not this hotel has certain facilities.

CAR OBJECTS

The car objects both share the same properties, but each one has different values for those properties. They tell you the make of car, what speed each car is currently traveling at, what color it is, and what type of fuel it requires.



EVENTS

In the real world, people interact with objects. These interactions can change the values of the properties in these objects.

WHAT IS AN EVENT?

There are common ways in which people interact with each type of object. For example, in a car a driver will typically use at least two pedals. The car has been designed to respond differently when the driver interacts with each of the different pedals:

- The accelerator makes the car go faster
- The brake slows it down

Similarly, programs are designed to do different things when users interact with the computer in different ways. For example, clicking on a contact link on a web page could bring up a contact form, and entering text into a search box may automatically trigger the search functionality.

An **event** is the computer's way of sticking up its hand to say, "Hey, this just happened!"

WHAT DOES AN EVENT DO?

Programmers choose which events they respond to. When a specific event happens, that event can be used to trigger a specific section of the code.

Scripts often use different events to trigger different types of functionality.

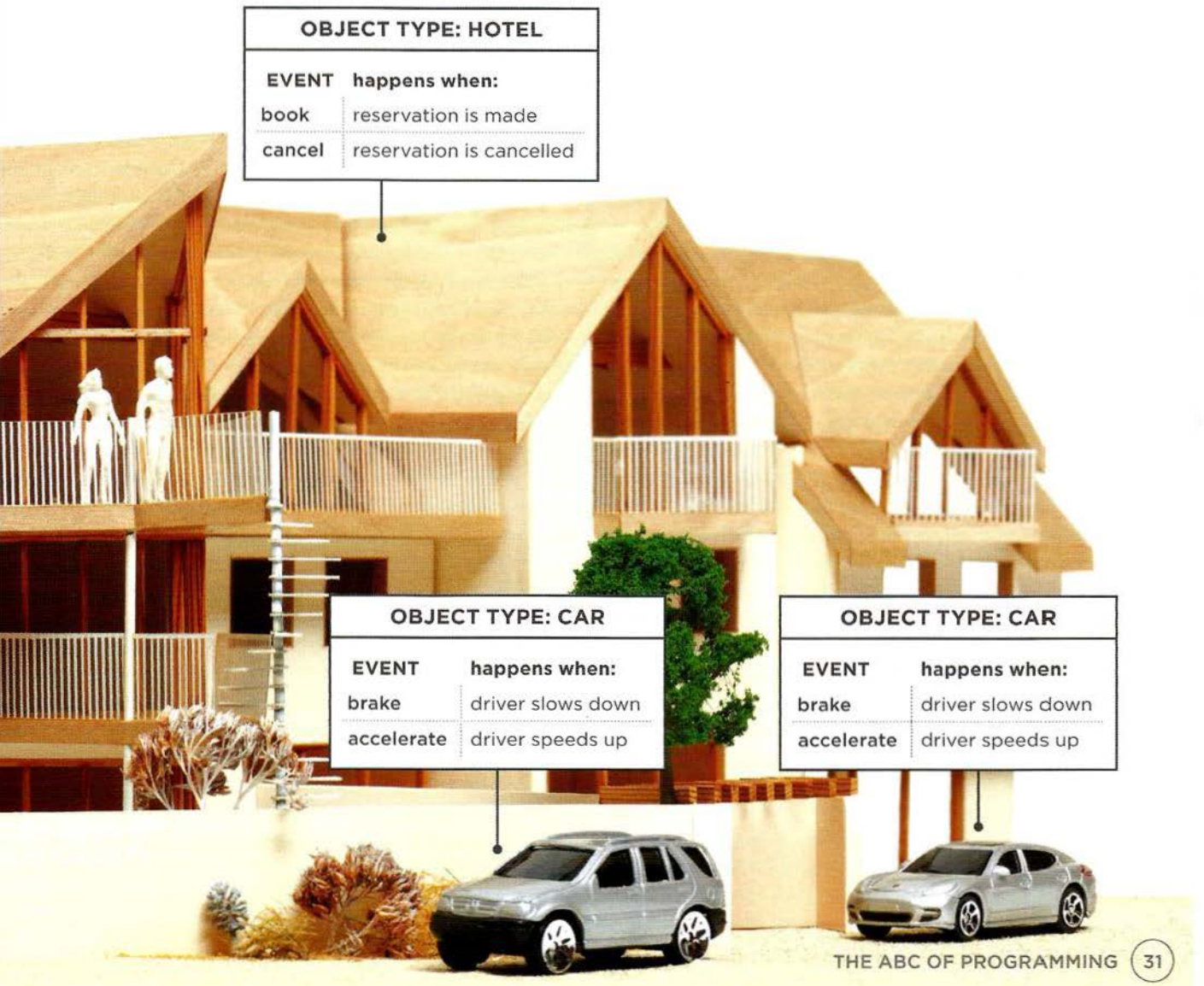
So a script will state which events the programmer wants to respond to, and what part of the script should be run when each of those events occur.

HOTEL OBJECT

A hotel will regularly have bookings for rooms. Each time a room is reserved, an event called `book` can be used to trigger code that will increase the value of the `bookings` property. Likewise, a `cancel` event can trigger code that decreases the value of the `bookings` property.

CAR OBJECTS

A driver will accelerate and brake throughout any car journey. An `accelerate` event can trigger code to increase the value of the `currentSpeed` property and a `brake` event can trigger code to decrease it. You will learn about the code that responds to the events and changes these properties on the next page.



METHODS

Methods represent things people need to do with objects. They can retrieve or update the values of an object's properties.

WHAT IS A METHOD?

Methods typically represent how people (or other things) interact with an object in the real world.

They are like questions and instructions that:

- Tell you something about that object (using information stored in its properties)
- Change the value of one or more of that object's properties

WHAT DOES A METHOD DO?

The code for a method can contain lots of instructions that together represent one task.

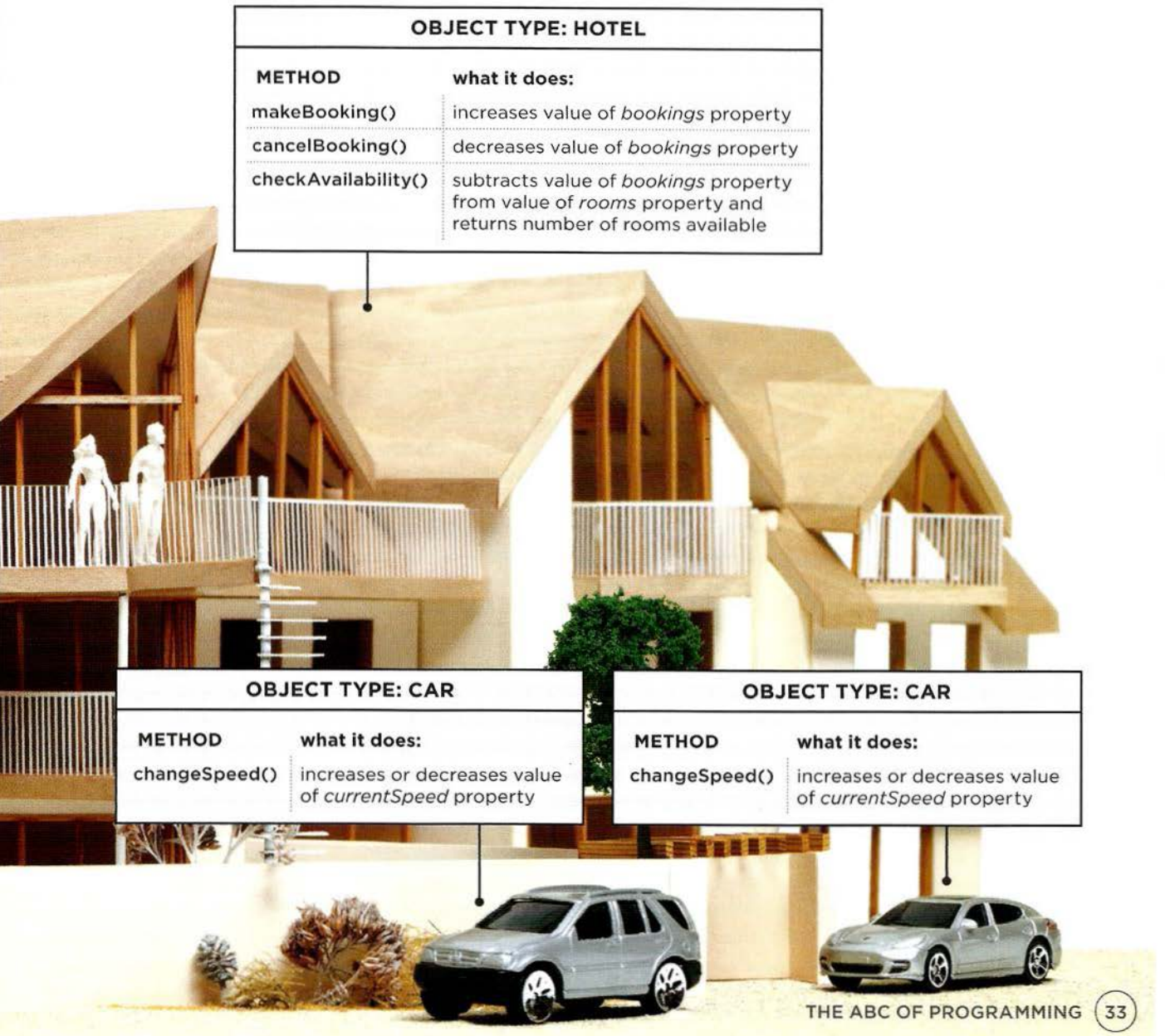
When you use a method, you do not always need to know *how* it achieves its task; you just need to know how to ask the question and how to interpret any answers it gives you.

HOTEL OBJECT

Hotels will commonly be asked if any rooms are free. To answer this question, a method can be written that subtracts the number of bookings from the total number of rooms. Methods can also be used to increase and decrease the value of the `bookings` property when rooms are booked or cancelled.

CAR OBJECTS

The value of the `currentSpeed` property needs to go up and down as the driver accelerates and brakes. The code to increase or decrease the value of the `currentSpeed` property could be written in a method, and that method could be called `changeSpeed()`.



OBJECT TYPE: HOTEL

METHOD	what it does:
makeBooking()	increases value of *bookings* property
cancelBooking()	decreases value of *bookings* property
checkAvailability()	subtracts value of *bookings* property from value of *rooms* property and returns number of rooms available

OBJECT TYPE: CAR

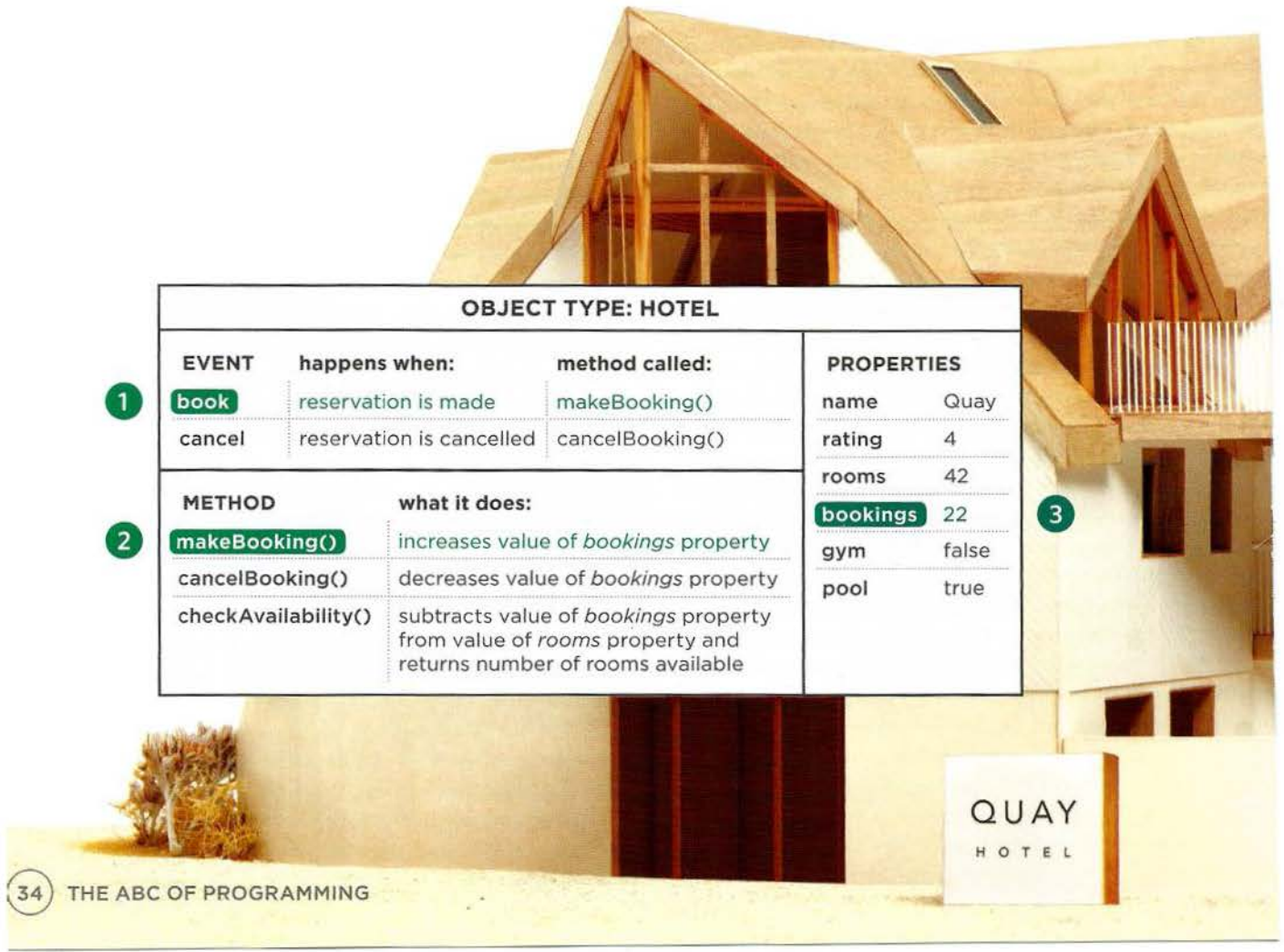
METHOD	what it does:
changeSpeed()	increases or decreases value of *currentSpeed* property

OBJECT TYPE: CAR

METHOD	what it does:
changeSpeed()	increases or decreases value of *currentSpeed* property

PUTTING IT ALL TOGETHER

Computers use data to create models of things in the real world. The events, methods, and properties of an object all relate to each other: Events can trigger methods, and methods can retrieve or update an object's properties.

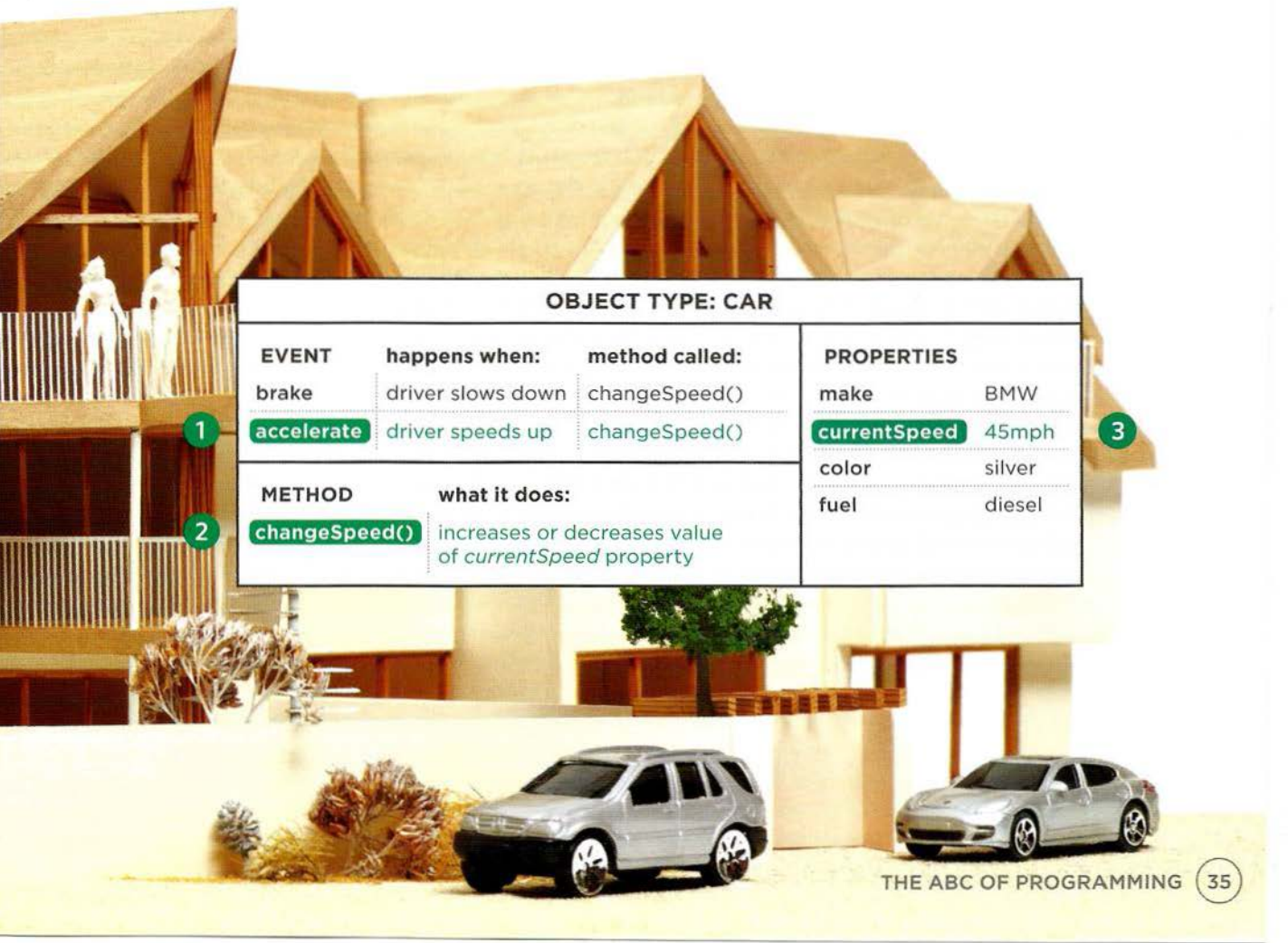


HOTEL OBJECT

1. When a reservation is made, the `book` event fires.
2. The `book` event triggers the `makeBooking()` method, which increases the value of the `bookings` property.
3. The value of the `bookings` property is changed to reflect how many rooms the hotel has available.

CAR OBJECTS

1. As a driver speeds up, the `accelerate` event fires.
2. The `accelerate` event calls the `changeSpeed()` method, which in turn increases the value of the `currentSpeed` property.
3. The value of the `currentSpeed` property reflects how fast the car is traveling.



WEB BROWSERS ARE PROGRAMS BUILT USING OBJECTS

You have seen how data can be used to create a model of a hotel or a car. Web browsers create similar models of the web page they are showing and of the browser window that the page is being shown in.

WINDOW OBJECT

On the right-hand page you can see a model of a computer with a browser open on the screen.

The browser represents each window or tab using a `window` object. The `location` property of the `window` object will tell you the URL of the current page.

DOCUMENT OBJECT

The current web page loaded into each window is modelled using a `document` object.

The `title` property of the `document` object tells you what is between the opening `<title>` and closing `</title>` tag for that web page, and the `lastModified` property of the `document` object tells you the date this page was last updated.


OBJECT TYPE: WINDOW
PROPERTIES
location http://www.javascriptbook.com/
OBJECT TYPE: DOCUMENT
PROPERTIES
URL http://www.javascriptbook.com/
lastModified 09/04/2014 15:33:37
title Learn JavaScript & jQuery - A book that teaches you in a nicer way

THE DOCUMENT OBJECT REPRESENTS AN HTML PAGE

Using the document object, you can access and change what content users see on the page and respond to how they interact with it.

Like other objects that represent real-world things, the document object has:

PROPERTIES

Properties describe characteristics of the current web page (such as the title of the page).

METHODS

Methods perform tasks associated with the document currently loaded in the browser (such as getting information from a specified element or adding new content).

EVENTS

You can respond to events, such as a user clicking or tapping on an element.

Because all major web browsers implement the document object in the same way, the people who create the browsers have already:

- Implemented properties that you can access to find out about the current page in the browser
- Written methods that achieve some common tasks that you are likely to want to do with an HTML page

So you will be learning how to work with this object. In fact, the document object is just one of a set of objects that all major browsers support. When the browser creates a model of a web page, it not only creates a document object, but it also creates a new object for each element on the page. Together these objects are described in the **Document Object Model**, which you will meet in Chapter 5.

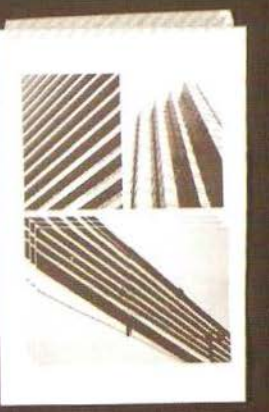
OBJECT TYPE: DOCUMENT

PROPERTIES

URL	http://www.javascriptbook.com/
lastModified	09/04/2014 15:33:37
title	Learn JavaScript & jQuery - A book that teaches you in a nicer way

EVENT	happens when:
load	page and assets have finished loading
click	user clicks the mouse over the page
keypress	user presses down on a key

METHOD	what it does:
write()	adds new content to the document
getElementById()	accesses an element when you state its id attribute



HOW A BROWSER SEES A WEB PAGE

In order to understand how you can change the content of an HTML page using JavaScript, you need to know how a browser interprets the HTML code and applies styling to it.

1: RECEIVE A PAGE AS HTML CODE

Each page on a website can be seen as a separate **document**. So, the web consists of many sites, each made up of one or more documents.

2: CREATE A MODEL OF THE PAGE AND STORE IT IN MEMORY

The model shown on the right hand page is a representation of one very basic page. Its structure is reminiscent of a family tree. At the top of the model is a **document object**, which represents the whole document.

Beneath the **document** object each box is called a **node**. Each of these nodes is another object. This example features three types of nodes representing elements, text within the elements, and attribute.

3: USE A RENDERING ENGINE TO SHOW THE PAGE ON SCREEN

If there is no CSS, the rendering engine will apply default styles to HTML elements. However, the HTML code for this example links to a CSS style sheet, so the browser requests that file and displays the page accordingly.

When the browser receives CSS rules, the rendering engine processes them and applies each rule to its corresponding elements. This is how the browser positions the elements in the correct place, with the right colors, fonts, and so on.

All major browsers use a JavaScript interpreter to translate your instructions (in JavaScript) into instructions the computer can follow.

When you use JavaScript in the browser, there is a part of the browser that is called an **interpreter** (or scripting engine).

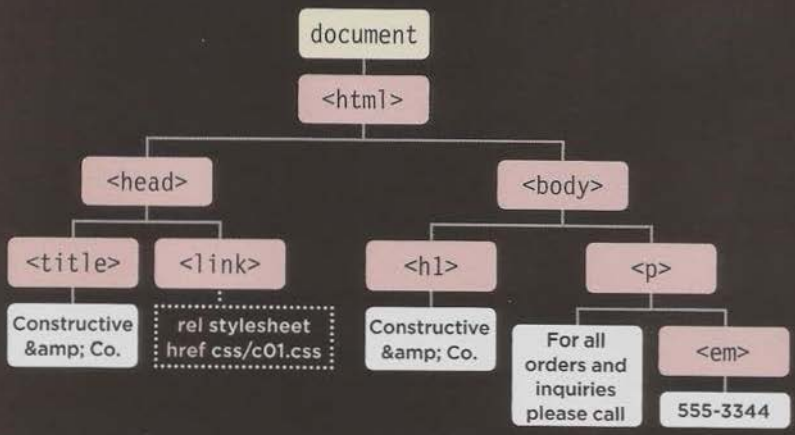
The interpreter takes your instructions (in JavaScript) and translates them into instructions the browser can use to achieve the tasks you want it to perform.

In an **interpreted programming language**, like JavaScript, each line of code is translated one-by-one as the script is run.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Constructive &amp;amp; Co.</title>
    <link rel="stylesheet" href="css/c01.css" />
  </head>
  <body>
    <h1>Constructive &amp;amp; Co.</h1>
    <p>For all orders and inquiries please call
      <em>555-3344</em></p>
  </body>
</html>
```

1

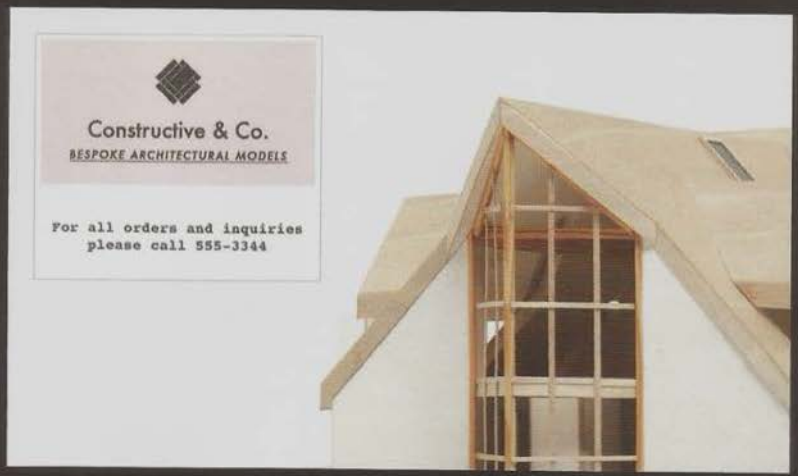
The browser receives an HTML page.



2

It creates a model of the page and stores it in memory.

- OBJECT
- ELEMENT
- TEXT
- ATTRIBUTES



3

It shows the page on screen using a rendering engine.

SUMMARY

THE ABC OF PROGRAMMING

B: How do computers fit in with the world around them?

- Computers create models of the world using data.
- The models use objects to represent physical things. Objects can have: properties that tell us about the object; methods that perform tasks using the properties of that object; events which are triggered when a user interacts with the computer.
- Programmers can write code to say "When this event occurs, run that code."
- Web browsers use HTML markup to create a model of the web page. Each element creates its own node (which is a kind of object).
- To make web pages interactive, you write code that uses the browser's model of the web page.

1/c

HOW DO I WRITE A SCRIPT FOR A WEB PAGE?

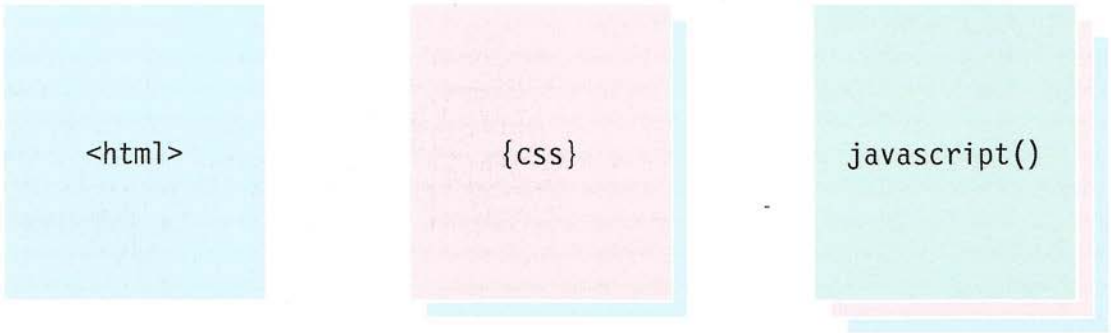
HOW HTML, CSS, & JAVASCRIPT FIT TOGETHER

Before diving into the JavaScript language, you need to know how it will fit together with the HTML and CSS in your web pages.

Web developers usually talk about three languages that are used to create web pages: HTML, CSS, and JavaScript.

Where possible, aim to keep the three languages in separate files, with the HTML page linking to CSS and JavaScript files.

Each language forms a separate **layer** with a different purpose. Each layer, from left to right, builds on the previous one.



CONTENT LAYER

`.html` files

This is where the content of the page lives. The HTML gives the page structure and adds semantics.

PRESENTATION LAYER

`.css` files

The CSS enhances the HTML page with rules that state how the HTML content is presented (backgrounds, borders, box dimensions, colors, fonts, etc.).

BEHAVIOR LAYER

`.js` files

This is where we can change how the page behaves, adding interactivity. We will aim to keep as much of our JavaScript as possible in separate files.

Programmers often refer to this as a **separation of concerns**.

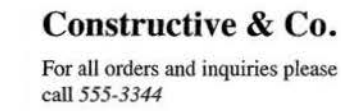
PROGRESSIVE ENHANCEMENT

These three layers form the basis of a popular approach to building web pages called progressive enhancement.

As more and more web-enabled devices come onto the market, this concept is becoming more widely adopted.

It's not just screen sizes that are varied - connection speeds and capabilities of each device can also differ.

Also, some people browse with JavaScript turned off, so you need to make sure that the page still works for them.



HTML ONLY

Starting with the HTML layer allows you to focus on the most important thing about your site: its content.

Being plain HTML, this layer should work on all kinds of devices, be accessible to all users, and load quite quickly on slow connections.

HTML+CSS

Adding the CSS rules in a separate file keeps rules regarding how the page looks away from the content itself.

You can use the same style sheet with all of your site, making your sites faster to load and easier to maintain. Or you can use different style sheets with the same content to create different views of the same data.

HTML+CSS+JAVASCRIPT

The JavaScript is added last and enhances the usability of the page or the experience of interacting with the site.

Keeping it separate means that the page still works if the user cannot load or run the JavaScript. You can also reuse the code on several pages (making the site faster to load and easier to maintain).

CREATING A BASIC JAVASCRIPT

JavaScript is written in plain text, just like HTML and CSS, so you do not need any new tools to write a script. This example adds a greeting into an HTML page. The greeting changes depending on the time of day.

1 Create a folder to put the example in called `c01`, then start up your favorite code editor, and enter the text to the right.

A JavaScript file is just a text file (like HTML and CSS files are) but it has a `.js` file extension, so save this file with the name `add-content.js`

Don't worry about what the code means yet, for now we will focus on how the script is created and how it fits with an HTML page.

```
var today = new Date();
var hourNow = today.getHours();
var greeting;

if (hourNow > 18) {
    greeting = 'Good evening!';
} else if (hourNow > 12) {
    greeting = 'Good afternoon!';
} else if (hourNow > 0) {
    greeting = 'Good morning!';
} else {
    greeting = 'Welcome!';
}

document.write('<h3>' + greeting  + '</h3>');
```

2 Get the CSS and images for this example from the website that accompanies the book: `www.javascriptbook.com`

To keep the files organized, in the same way that CSS files often live in a folder called `styles` or `css`, your JavaScript files can live in a folder called `scripts`, `javascript`, or `js`. In this case, save your file in a folder called `js`

Here you can see the file structure that you will end up with when you finish the example. Always treat file names as being case-sensitive.

LINKING TO A JAVASCRIPT FILE FROM AN HTML PAGE

When you want to use JavaScript with a web page, you use the HTML `<script>` element to tell the browser it is coming across a script. Its `src` attribute tells people where the JavaScript file is stored.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Constructive &amp;amp; Co.</title>
    <link rel="stylesheet" href="css/c01.css" />
  </head>
  <body>
    <h1>Constructive &amp;amp; Co.</h1>
    <script src="js/add-content.js"></script>
    <p>For all orders and inquiries please call
      <em>555-3344</em></p>
  </body>
</html>
```

3 In your code editor, enter the HTML shown on the left. Save this file with the name `add-content.html`

The HTML `<script>` element is used to load the JavaScript file into the page. It has an attribute called `src`, whose value is the path to the script you created.

This tells the browser to find and load the script file (just like the `src` attribute on an `<img>` tag).

4 Open the HTML file in your browser. You should see that the JavaScript has added a greeting (in this case, *Good Afternoon!*) to the page. (These greetings are coming from the JavaScript file; they are not in the HTML file.)

Please note: Internet Explorer sometimes prevents JavaScript running when you open a page stored on your hard drive. If this affects you, please try Chrome, Firefox, Opera, or Safari instead.

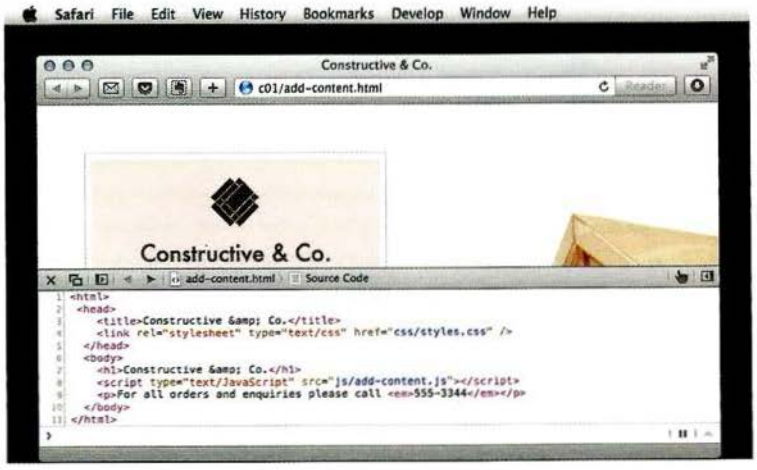
THE SOURCE CODE IS NOT AMENDED

If you look at the source code for the example you just created, you will see that the HTML is still exactly the same.

5 Once you have tried the example in your browser, view the source code for the page. (This option is usually under the *View*, *Tools* or *Develop* menu of the browser.)

6 The source of the web page does not actually show the new element that has been added into the page; it just shows the link to the JavaScript file.

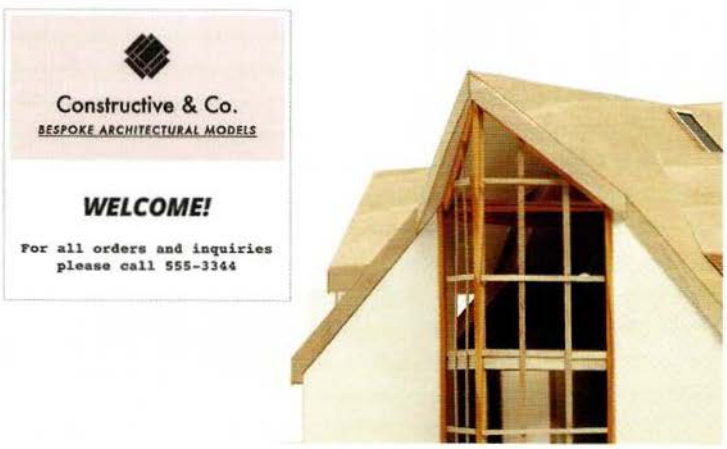
As you move through the book, you will see most of the scripts are added just before the closing </body> tag (this is often considered a better place to put your scripts).



PLACING THE SCRIPT IN THE PAGE

You may see JavaScript in the HTML between opening <script> and closing </script> tags (but it is better to put scripts in their own files).

```
<!DOCTYPE html>
<html>
  <head>
    <title>Constructive &amp;amp; Co.</title>
    <link rel="stylesheet" href="css/c01.css" />
  </head>
  <body>
    <h1>Constructive &amp;amp; Co.</h1>
    <script>document.write('<h3>Welcome!</h3>');
    </script>
    <p>For all orders and inquiries please call
      <em>555-3344</em></p>
  </body>
</html>
```



7 Finally, try opening the HTML file, removing the src attribute from the opening <script> tag, and adding the new code shown on the left between the opening <script> tag and the closing </script> tag. The src attribute is no longer needed because the JavaScript is in the HTML page.

As noted on p44, it is better not to mix JavaScript in your HTML pages like this, but it is mentioned here as you may come across this technique.

8 Open the HTML file in your web browser and the welcome greeting is written into the page.

As you may have guessed, `document.write()` *writes* content into the *document* (the web page). It is a simple way to add content to a page, but not always the best. Chapter 5 discusses various ways to update the content of a page.

HOW TO USE OBJECTS & METHODS

This one line of JavaScript shows how to use objects and methods. Programmers refer to this as **calling** a method of an object.

The **document** object represents the entire web page. All web browsers implement this object, and you can use it just by giving its name.

The **write()** method of the **document** object allows new content to be written into the page where the **<script>** element sits.

OBJECT

METHOD

```
document.write('Good afternoon!');
```

MEMBER OPERATOR

PARAMETERS

The **document** object has several methods and properties. They are known as **members** of that object.

You can access the members of an object using a dot between the object name and the member you want to access. It is called a **member operator**.

Whenever a method requires some information in order to work, the data is given inside the parentheses.

Each piece of information is called a **parameter** of the method. In this case, the **write()** method needs to know what to write into the page.

Behind the scenes, the browser uses a lot more code to make the words appear on the screen, but you don't need to know how the browser does this.

You only need to know how to call the object and method, and how to tell it the information it needs to do the job you want it to. It will do the rest.

There are lots of objects like the **document** object, and lots of methods like the **write()** method that will help you write your own scripts.

JAVASCRIPT RUNS WHERE IT IS FOUND IN THE HTML

When the browser comes across a <script> element, it stops to load the script and then checks to see if it needs to do anything.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Constructive &amp; Co.</title>
    <link rel="stylesheet" href="css/c01.css" />
  </head>
  <body>
    <h1>Constructive &amp; Co.</h1>
    <p>For all orders and inquiries please call <em>555-3344</em></p>
    <script src="js/add-content.js"></script>
  </body>
</html>
```

Note how the **<script>** element can be moved below the first paragraph, and this affects where the new greeting is written into the page.

This has implications for where **<script>** elements should be placed, and can affect the loading time of pages (see p356).

SUMMARY

THE ABC OF PROGRAMMING

C: How do I write a script for a web page?

- It is best to keep JavaScript code in its own JavaScript file. JavaScript files are text files (like HTML pages and CSS style sheets), but they have the `.js` extension.
- The HTML `<script>` element is used in HTML pages to tell the browser to load the JavaScript file (rather like the `<link>` element can be used to load a CSS file).
- If you view the source code of the page in the browser, the JavaScript will not have changed the HTML, because the script works with the model of the web page that the browser has created.

2

BASIC JAVASCRIPT INSTRUCTIONS

In this chapter, you will start learning to read and write JavaScript. You will also learn how to give a web browser instructions you want it to follow.

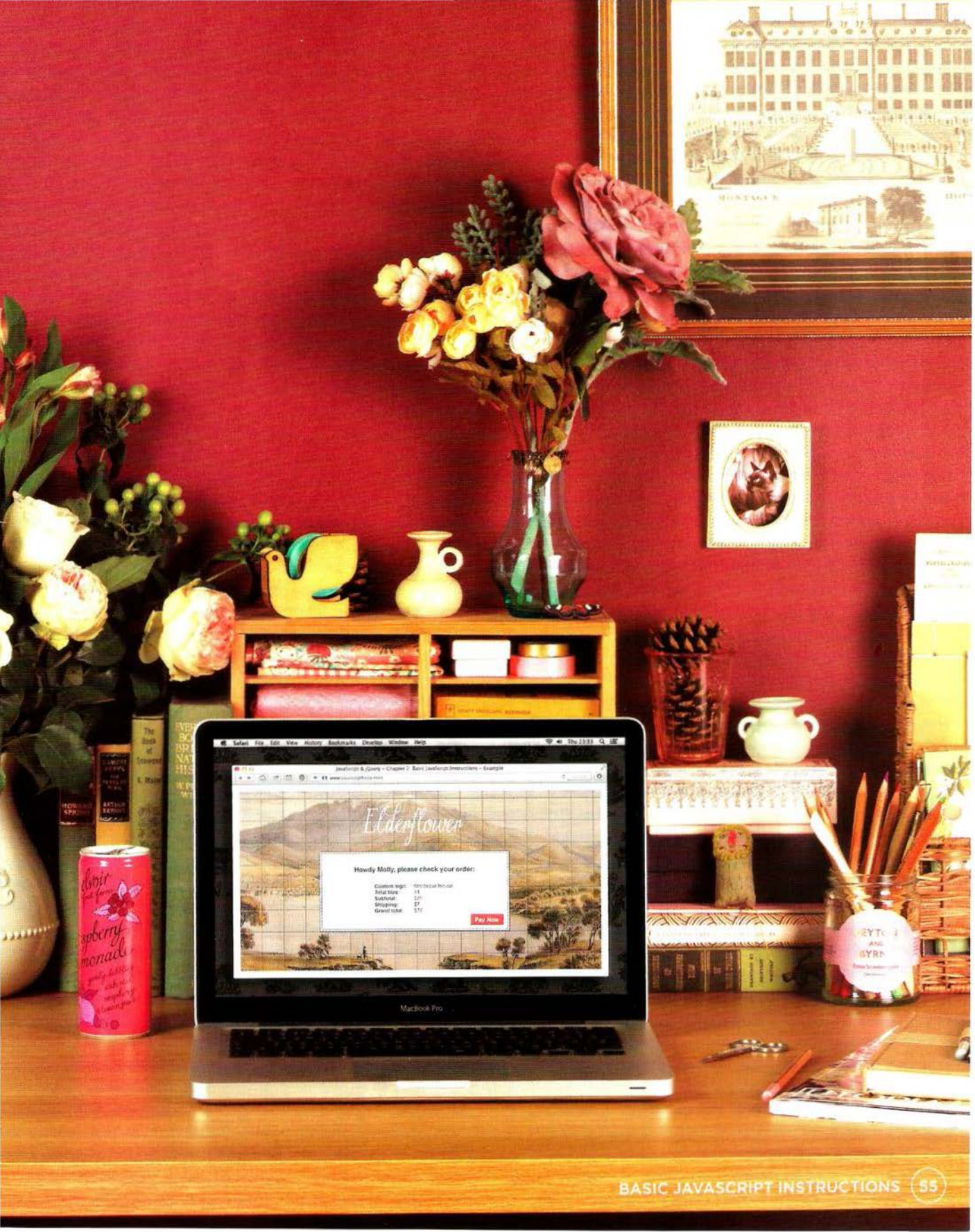
THE LANGUAGE: SYNTAX AND GRAMMAR

Like any new language, there are new words to learn (the vocabulary) and rules for how these can be put together (the grammar and syntax of the language).

GIVING INSTRUCTIONS: FOR A BROWSER TO FOLLOW

Web browsers (and computers in general) approach tasks in a very different way than a human might. Your instructions need to reflect how computers get things done.

We will start with a few of the key building blocks of the language and look at how they can be used to write some very basic scripts (consisting of a few simple steps) before going on to look at some more complex concepts in subsequent chapters.


Safari File Edit View History Bookmarks Develop Window Help
JavaScript & jQuery - Chapter 2: Basic JavaScript Instructions - Example
ElderFlower
Howdy Molly, please check your order:
Pay Now
MacBook Pro

STATEMENTS

A script is a series of instructions that a computer can follow one-by-one. Each individual instruction or step is known as a **statement**. Statements should end with a semicolon.

We will look at what the code on the right does shortly, but for the moment note that:

- Each of the lines of code in **green** is a **statement**.
- The **pink** curly braces indicate the start and end of a **code block**. (Each code block could contain many more statements.)
- The code in **purple** determines which code should run (as you will see on p149).

```
var today = new Date();
var hourNow = today.getHours();
var greeting;

if (hourNow > 18) {
  greeting = 'Good evening';
} else if (hourNow > 12) {
  greeting = 'Good afternoon';
} else if (hourNow > 0) {
  greeting = 'Good morning';
} else {
  greeting = 'Welcome';
}
document.write(greeting);
```

JAVASCRIPT IS CASE SENSITIVE

JavaScript is case sensitive so `hourNow` means something different to `HourNow` or `HOURNOW`.

STATEMENTS ARE INSTRUCTIONS AND EACH ONE STARTS ON A NEW LINE

A statement is an individual instruction that the computer should follow. Each one should start on a new line and end with a semicolon. This makes your code easier to read and follow.

The semicolon also tells the JavaScript interpreter when a step is over, indicating that it should move to the next step.

STATEMENTS CAN BE ORGANIZED INTO CODE BLOCKS

Some statements are surrounded by curly braces; these are known as **code blocks**. The closing curly brace is not followed by a semicolon.

Above, each code block contains one statement related to what the current time is. Code blocks will often be used to group together many more statements. This helps programmers organize their code and makes it more readable.

COMMENTS

You should write **comments** to explain what your code does. They help make your code easier to read and understand. This can help you and others who read your code.

```
/* This script displays a greeting to the user based upon the current time.
   It is an example from JavaScript & jQuery book */

var today = new Date();          // Create a new date object
var hourNow = today.getHours();  // Find the current hour
var greeting;

// Display the appropriate greeting based on the current time
if (hourNow > 18) {
  greeting = 'Good evening';
} else if (hourNow > 12) {
  greeting = 'Good afternoon';
} else if (hourNow > 0) {
  greeting = 'Good morning';
} else {
  greeting = 'Welcome';
}
document.write(greeting);
```

JavaScript code is **green**
Multi-line comments are **pink**
Single-line comments are **gray**

MULTI-LINE COMMENTS

To write a comment that stretches over more than one line, you use a **multi-line** comment, starting with the /* characters and ending with the */ characters. Anything between these characters is not processed by the JavaScript interpreter.

Multi-line comments are often used for descriptions of how the script works, or to prevent a section of the script from running when testing it.

SINGLE-LINE COMMENTS

In a **single-line** comment, anything that follows the two forward slash characters // on that line will not be processed by the JavaScript interpreter. Single-line comments are often used for short descriptions of what the code is doing.

Good use of comments will help you if you come back to your code after several days or months. They also help those who are new to your code.

WHAT IS A VARIABLE?

A script will have to temporarily store the bits of information it needs to do its job. It can store this data in **variables**.

When you write JavaScript, you have to tell the interpreter every individual step that you want it to perform. This sometimes involves more detail than you might expect.

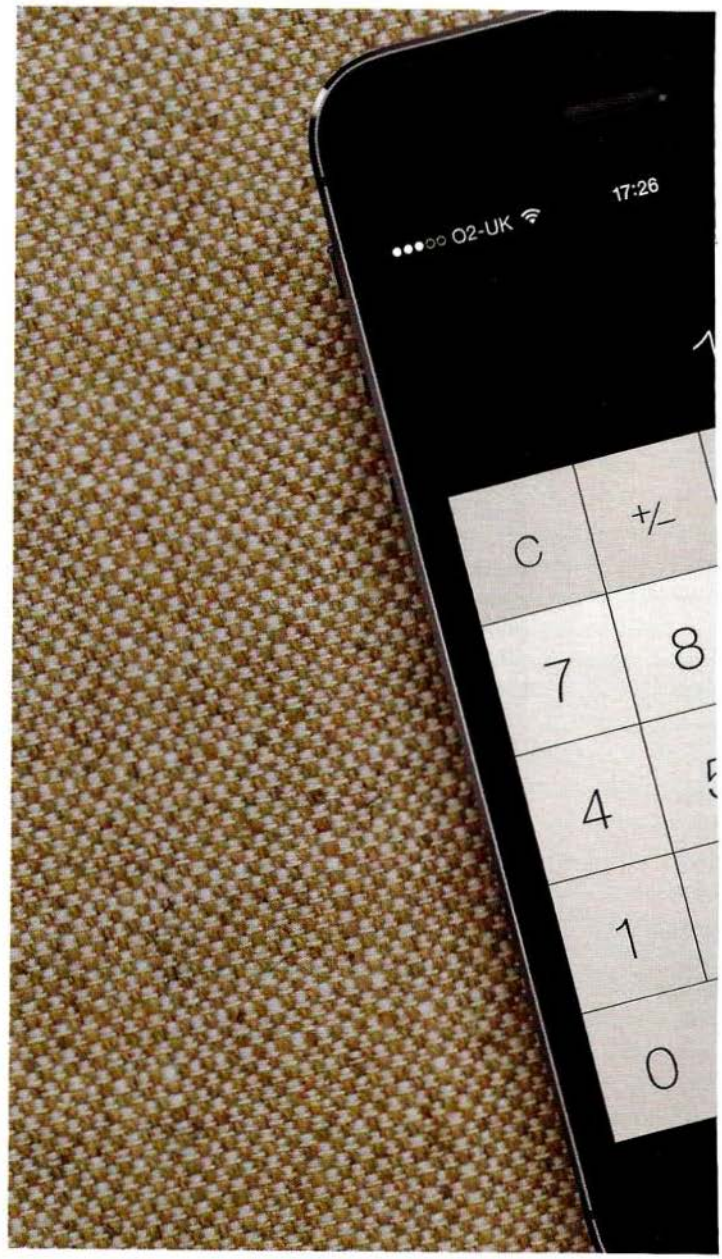
Think about calculating the area of a wall; in math the area of a rectangle is obtained by multiplying two numbers:

width × *height* = *area*

You may be able to do calculations like this in your head, but when writing a script to do this calculation, you need to give the computer very detailed instructions. You might tell it to perform the following four steps in order:

1. Remember the value for *width*
2. Remember the value for *height*
3. Multiply *width* by *height* to get the *area*
4. Return the result to the user

In this case, you would use variables to "remember" the values for *width* and *height*. (This also illustrates how a script contains very explicit instructions about exactly what you want the computer to do.)
You can compare variables to short-term memory, because once you leave the page, the browser will forget any information it holds.



A variable is a good name for this concept because the data stored in a variable can change (or vary) each time a script runs.

No matter what the dimensions of any individual wall are, you know that you can find its *area* by multiplying the *width* of that wall by its *height*. Similarly, scripts often need to achieve the same goal even when they are run with different data, so variables can be used to represent values in your scripts that are likely to change. The result is said to be **calculated** or **computed** using the data stored in the variables.

The use of variables to represent numbers or other kinds of data is very similar to the concept of algebra (where letters are used to represent numbers). There is one key difference, however. The equals sign does something very different in programming (as you will see on the next two pages).

VARIABLES: HOW TO DECLARE THEM

Before you can use a variable, you need to announce that you want to use it. This involves creating the variable and giving it a name. Programmers say that you **declare** the variable.

```
var quantity;
```

VARIABLE KEYWORD

VARIABLE NAME

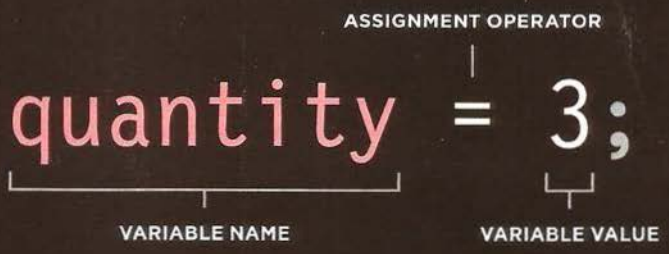
var is an example of what programmers call a **keyword**. The JavaScript interpreter knows that this keyword is used to create a variable.

In order to use the variable, you must give it a name. (This is sometimes called an identifier.) In this case, the variable is called **quantity**.

If a variable name is more than one word, it is usually written in camelCase. This means the first word is all lowercase and any subsequent words have their first letter capitalized.

VARIABLES: HOW TO ASSIGN THEM A VALUE

Once you have created a variable, you can tell it what information you would like it to store for you. Programmers say that you **assign a value** to the variable.



You can now use the variable by its name. Here we set a value for the variable called **quantity**. Where possible, a variable's name should describe the kind of data the variable holds.

The equals sign (=) is an **assignment operator**. It says that you are going to assign a value to the variable. It is also used to update the value given to a variable (see p68).

Until you have assigned a value to a variable, programmers say the value is **undefined**.

Where a variable is declared can have an effect upon whether the rest of the script can use it. Programmers call this the **scope** of a variable and it is covered on p98.

DATA TYPES

JavaScript distinguishes between numbers, strings, and `true` or `false` values known as Booleans.

NUMERIC DATA TYPE

The numeric data type handles numbers.

```
0.75
```

For tasks that involve counting or calculating sums, you will use numbers 0-9. For example, five thousand, two hundred and seventy-two would be written 5272 (note there is no comma between the thousands and the hundreds). You can also have negative numbers (such as -23678) and decimals (three quarters is written as 0.75).

STRING DATA TYPE

The strings data type consists of letters and other characters.

```
'Hi, Ivy!'
```

Note how the string data type is enclosed within a pair of quotes. These can be single or double quotes, but the opening quote must match the closing quote.

Strings can be used when working with any kind of text. They are frequently used to add new content into a page and they can contain HTML markup.

BOOLEAN DATA TYPE

Boolean data types can have one of two values: `true` or `false`.

```
true
```

It might seem a little abstract at first, but the Boolean data type is actually very helpful.

You can think of it a little like a light switch – it is either on or off. As you will see in Chapter 4, Booleans are helpful when determining which part of a script should run.

Numbers are not only used for things like calculators; they are also used for tasks such as determining the size of the screen, moving the position of an element on a page, or setting the amount of time an element should take to fade in.

In addition to these three data types, JavaScript also has others (arrays, objects, `undefined`, and `null`) that you will meet in later chapters.

Unlike some other programming languages, when declaring a variable in JavaScript, you do not need to specify what type of data it will hold.

USING A VARIABLE TO STORE A NUMBER

JAVASCRIPT c02/js/numeric-variable.js

```
var price;
var quantity;
var total;

price = 5;
quantity = 14;
total = price * quantity;

var el = document.getElementById('cost');
el.textContent = '$' + total;
```

HTML c02/numeric-variable.html

```
<h1>Elderflower</h1>
<div id="content">
  <h2>Custom Signage</h2>
  <div id="cost">Cost: $5 per tile</div>
  <img src="images/preview.jpg" alt="Sign" />
</div>
<script src="js/numeric-variable.js"></script>
```

RESULT

Here, three variables are created and values are assigned to them.

- `price` holds the price of an individual tile
- `quantity` holds the number of tiles a customer wants
- `total` holds the total cost of the tiles

Note that the numbers are not written inside quotation marks. Once a value has been assigned to a variable, you can use the variable name to represent that value (much like you might have done in algebra). Here, the total cost is calculated by multiplying the price of a single tile by the number of tiles the customer wants.

The result is then written into the page on the final two lines. You see this technique in more detail on p194 and p216.
The first of these two lines finds the element whose `id` attribute has a value of `cost`, and the final line replaces the content of that element with new content.

Note: There are many ways to write content into a page, and several places you can place your script. The advantages and disadvantages of each technique are discussed on p226. This technique will not work in IE8.

USING A VARIABLE TO STORE A STRING

For the moment, concentrate on the first four lines of JavaScript. Two variables are declared (`username` and `message`), and they are used to hold strings (the user's name and a message for that user).

The code to update the page (shown in the last four lines) is discussed fully in Chapter 5. This code selects two elements using the values of their `id` attributes. The text in those elements is updated using the values stored in these variables.

Note how the string is placed inside quote marks. The quotes can be single or double quotes, but they must match. If you start with a single quote, you must end with a single quote, and if you start with a double quote, you must end with a double quote:

- ✔ `"hello"` ✘ `"hello'`
- ✔ `'hello'` ✘ `'hello"`

Quotes should be straight (not curly) quotes:

- ✔ " " ✘ “ ”
- ✔ ' ' ✘ ‘ ’

Strings must always be written on one line:

- ✔ `'See our upcoming range'`
- ✘ `'See our upcoming range'`

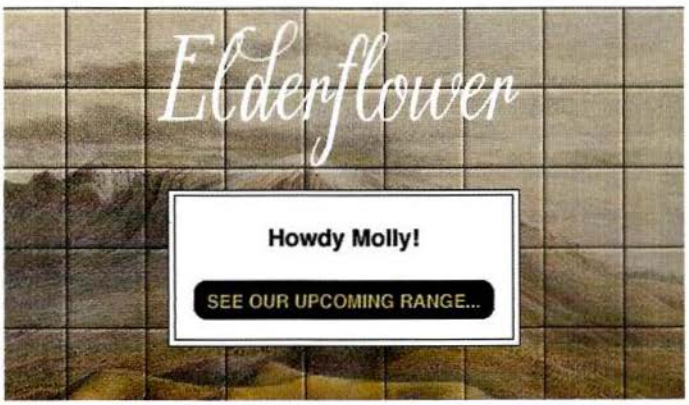
c02/js/string-variable.js — JAVASCRIPT

```
var username;
var message;
username = 'Molly';
message = 'See our upcoming range';

var elName = document.getElementById('name');
elName.textContent = username;
var elNote = document.getElementById('note');
elNote.textContent = message;
```

c02/string-variable.html — HTML

```
<h1>Elderflower</h1>
<div id="content">
  <div id="title">Howdy
    <span id="name">friend</span>!</div>
  <div id="note">Take a look around...</div>
</div>
<script src="js/string-variable.js"></script>
```

RESULT



USING QUOTES INSIDE A STRING

JAVASCRIPT c02/js/string-with-quotes.js

```
var title;
var message;
title = "Molly's Special Offers";
message = '<a href=\"sale.html\">25% off!</a>';

var elTitle = document.getElementById('title');
elTitle.innerHTML = title;
var elNote = document.getElementById('note');
elNote.innerHTML = message;
```

HTML c02/string-with-quotes.html

```
<h1>Elderflower</h1>
<div id="content">
  <div id="title">Special Offers</div>
  <div id="note">Sign-up to receive personalized
    offers!</div>
</div>
<script src="js/string-with-quotes.js"></script>
```

RESULT

Sometimes you will want to use a double or single quote mark *within* a string.

Because strings can live in single or double quotes, if you just want to use double quotes in the string, you could surround the entire string in single quotes.

If you just want to use single quotes in the string, you could surround the string in double quotes (as shown in the third line of this code example).

You can also use a technique called **escaping** the quotation characters. This is done by using a **backwards slash** (or "backslash") *before* any type of quote mark that appears within a string (as shown on the fourth line of this code sample). The backwards slash tells the interpreter that the following character is part of the string, rather than the end of it.

Techniques for adding content to a page are covered in Chapter 5. This example uses a property called `innerHTML` to add HTML to the page. In certain cases, this property *can* pose a security risk (discussed on p228 - p231).

USING A VARIABLE TO STORE A BOOLEAN

A Boolean variable can only have a value of `true` or `false`, but this data type is very helpful.

In the example on the right, the values `true` or `false` are used in the `class` attributes of HTML elements. These values trigger different CSS class rules: `true` shows a check, `false` shows a cross. (You learn how the `class` attribute is set in Chapter 5.)

It is rare that you would want to write the words `true` or `false` into the page for the user to read, but this data type does have two very popular uses:

First, Booleans are used when the value can only be `true`/`false`. You could also think of these values as on/off or 0/1: `true` is equivalent to on or 1, `false` is equivalent to off or 0

Second, Booleans are used when your code can take more than one path. Remember, different code may run in different circumstances (as shown in the flowcharts throughout the book).

The path the code takes depends on a test or condition.

c02/js/boolean-variable.js — JAVASCRIPT

```
var inStock;
var shipping;
inStock = true;
shipping = false;

var elStock = document.getElementById('stock');
elStock.className = inStock;

var elShip = document.getElementById('shipping');
elShip.className = shipping;
```

c02/boolean-variable.html — HTML

```
<h1>Elderflower</h1>
<div id="content">
  <div class="message">Available:
    <span id="stock"></span></div>
  <div class="message">Shipping:
    <span id="shipping"></span></div>
</div>
<script src="js/boolean-variable.js"></script>
```

RESULT

SHORTHAND FOR CREATING VARIABLES

JAVASCRIPT c02/js/shorthand-variable.js

```
① var price = 5;
  var quantity = 14;
  var total = price * quantity;
```

```
② var price, quantity, total;
  price = 5;
  quantity = 14;
  total = price * quantity;
```

```
③ var price = 5, quantity = 14;
  var total = price * quantity;
```

```
④ // Write total into the element with id of cost
  var el = document.getElementById('cost');
  el.textContent = '$' + total;
```

RESULT

Programmers sometimes use shorthand to create variables. Here are three variations of how to declare variables and assign them values:

1. Variables are declared and values assigned in the same statement.

2. Three variables are declared on the same line, then values assigned to each.

3. Two variables are declared and assigned values on the same line. Then one is declared and assigned a value on the next line.

(The third example shows two numbers, but you can declare variables that hold different types of data on the same line, e.g., a string and a number.)

4. Here, a variable is used to hold a reference to an element in the HTML page. This allows you to work directly with the element stored in that variable. (See more about this on p190.)

While the shorthand might save you a little bit of typing, it can make your code a little harder to follow. So, when you are starting off, you will find it easier to spread your code over a few more lines to make it easier to read and understand.

CHANGING THE VALUE OF A VARIABLE

Once you have assigned a value to a variable, you can then change what is stored in the variable later in the same script.

Once the variable has been created, you do not need to use the `var` keyword to assign it a new value. You just use the variable name, the equals sign (also known as the assignment operator), and the new value for that attribute.

For example, the value of a `shipping` variable might start out as being `false`. Then something in the code might change the ability to ship the item and you could therefore change the value to `true`.

In this code example, the values of the two variables are both swapped from being `true` to `false` and vice versa.

c02/js/update-variable.js

JAVASCRIPT

```
var inStock;
var shipping;

inStock = true;
shipping = false;

/* Some other processing might go here and, as
a result, the script might need to change these
values */

inStock = false;
shipping = true;

var elStock = document.getElementById('stock');
elStock.className = inStock;
var elShip = document.getElementById('shipping');
elShip.className = shipping;
```

RESULT

RULES FOR NAMING VARIABLES

Here are six rules you must always follow when giving a variable a name:

1

The name must begin with a letter, dollar sign ($), or an underscore (_). It must **not** start with a number.

2

The name can contain letters, numbers, dollar sign ($), or an underscore (_). Note that you must not use a dash (-) or a period (.) in a variable name.

3

You cannot use **keywords** or **reserved** words. Keywords are special words that tell the interpreter to do something. For example, `var` is a keyword used to declare a variable. Reserved words are ones that may be used in a *future* version of JavaScript.

ONLINE EXTRA

View a full list of keywords and reserved words in JavaScript.

4

All variables are case sensitive, so `score` and `Score` would be different variable names, but it is bad practice to create two variables that have the same name using different cases.

5

Use a name that describes the kind of information that the variable stores. For example, `firstName` might be used to store a person's first name, `lastName` for their last name, and `age` for their age.

6

If your variable name is made up of more than one word, use a capital letter for the first letter of every word *after* the first word. For example, `firstName` rather than `firstname` (this is referred to as camel case). You can also use an underscore between each word (you cannot use a dash).

ARRAYS

An array is a special type of variable. It doesn't just store one value; it stores a list of values.

You should consider using an array whenever you are working with a **list** or a set of values that are **related** to each other.

Arrays are especially helpful when you do not know how many items a list will contain because, when you create the array, you do not need to specify how many values it will hold.

If you don't know how many items a list will contain, rather than creating enough variables for a long list (when you might only use a small percentage of them), using an array is considered a better solution.

For example, an array can be suited to storing the individual items on a shopping list because it is a list of related items.

Additionally, each time you write a new shopping list, the number of items on it may differ.

As you will see on the next page, values in an array are separated by commas.

In Chapter 12, you will see that arrays can be very helpful when representing complex data.

CREATING AN ARRAY

JAVASCRIPT c02/js/array-literal.js

```
var colors;
colors = ['white', 'black', 'custom'];

var el = document.getElementById('colors');
el.textContent = colors[0];
```

RESULT

JAVASCRIPT c02/js/array-constructor.js

```
var colors = new Array('white',
                       'black',
                       'custom');

var el = document.getElementById('colors');
el.innerHTML = colors.item(0);
```

The array literal (shown in the first code sample) is preferred over the array constructor when creating arrays.

You create an array and give it a name just like you would any other variable (using the `var` keyword followed by the name of the array).

The values are assigned to the array inside a pair of square brackets, and each value is separated by a comma. The values in the array do not need to be the same data type, so you can store a string, a number and a Boolean all in the same array.

This technique for creating an array is known as an **array literal**. It is usually the preferred method for creating an array. You can also write each value on a separate line:

```
colors = ['white',
          'black',
          'custom'];
```

On the left, you can see an array created using a different technique called an **array constructor**. This uses the `new` keyword followed by `Array();` The values are then specified in parentheses (not square brackets), and each value is separated by a comma. You can also use a method called `item()` to retrieve data from the array. (The index number of the item is specified in the parentheses.)

VALUES IN ARRAYS

Values in an array are accessed as if they are in a numbered list. It is important to know that the numbering of this list starts at zero (not one).

NUMBERING ITEMS IN AN ARRAY

Each item in an array is automatically given a number called an **index**. This can be used to access specific items in the array. Consider the following array which holds three colors:

```
var colors;
colors = ['white',
          'black',
          'custom'];
```

Confusingly, index values start at 0 (not 1), so the following table shows items from the array and their corresponding index values:

INDEX	VALUE
0	'white'
1	'black'
2	'custom'

ACCESSING ITEMS IN AN ARRAY

To retrieve the third item on the list, the array name is specified along with the index number in square brackets.

Here you can see a variable called `itemThree` is declared. Its value is set to be the third color from the `colors` array.

```
var itemThree;
itemThree = colors[2];
```

NUMBER OF ITEMS IN AN ARRAY

Each array has a property called `length`, which holds the number of items in the array.

Below you can see that a variable called `numColors` is declared. Its value is set to be the number of the items in the array.

The name of the array is followed by a period symbol (or full stop) which is then followed by the `length` keyword.

```
var numColors;
numColors = colors.length;
```

Throughout the book (especially in Chapter 12) you meet more features of arrays, which are a very flexible and powerful feature of JavaScript.

ACCESSING & CHANGING VALUES IN AN ARRAY

JAVASCRIPT c02/js/update-array.js

```
// Create the array
var colors = ['white',
              'black',
              'custom'];

// Update the third item in the array
colors[2] = 'beige';

// Get the element with an id of colors
var el = document.getElementById('colors');

// Replace with third item from the array
el.textContent = colors[2];
```

RESULT

The first lines of code on the left create an array containing a list of three colors. (The values can be added on the same line or on separate lines as shown here.)

Having created the array, the third item on the list is changed from `'custom'` to `'beige'`.

To access a value from an array, after the array name you specify the index number for that value inside square brackets.

You can change the value of an item an array by selecting it and assigning it a new value just as you would any other variable (using the equals sign and the new value for that item).

In the last two statements, the newly updated third item in the array is added to the page.

If you wanted to write out *all* of the items in an array, you would use a loop, which you will meet on p170.

EXPRESSIONS

An **expression** evaluates into (results in) a single value. Broadly speaking there are two types of expressions.

1

EXPRESSIONS THAT JUST ASSIGN A VALUE TO A VARIABLE

In order for a variable to be useful, it needs to be given a value. As you have seen, this is done using the assignment operator (the equals sign).

```
var color = 'beige';
```

The value of `color` is now `beige`.

When you first declare a variable using the `var` keyword, it is given a special value of `undefined`. This will change when you assign a value to it. Technically, undefined is a data type like a number, string, or Boolean.

2

EXPRESSIONS THAT USE TWO OR MORE VALUES TO RETURN A SINGLE VALUE

You can perform operations on any number of individual values (see next page) to determine a single value. For example:

```
var area = 3 * 2;
```

The value of `area` is now 6.

Here the expression 3 * 2 evaluates into 6. This example also uses the assignment operator, so the result of the expression 3 * 2 is stored in the variable called `area`.

Another example where an expression uses two values to yield a single value would be where two strings are joined to create a single string.

OPERATORS

Expressions rely on things called **operators**; they allow programmers to create a single value from one or more values.

Covered in this chapter:

ASSIGNMENT OPERATORS

Assign a value to a variable

```
color = 'beige';
```

The value of `color` is now `beige`.
(See p61)

ARITHMETIC OPERATORS

Perform basic math

```
area = 3 * 2;
```

The value of `area` is now `6`.
(See p76)

STRING OPERATORS

Combine two strings

```
greeting = 'Hi ' + 'Molly';
```

The value of `greeting` is now `Hi Molly`.
(See p78)

Covered in Chapter 4:

COMPARISON OPERATORS

Compare two values and return `true` or `false`

```
buy = 3 > 5;
```

The value of `buy` is `false`.
(See p150)

LOGICAL OPERATORS

Combine expressions and return `true` or `false`

```
buy = (5 > 3) && (2 < 4);
```

The value of `buy` is now `true`.
(See p156)

ARITHMETIC OPERATORS

JavaScript contains the following mathematical operators, which you can use with numbers. You may remember some from math class.

NAME	OPERATOR	PURPOSE & NOTES	EXAMPLE	RESULT
ADDITION	+	Adds one value to another	`10 + 5`	15
SUBTRACTION	-	Subtracts one value from another	`10 - 5`	5
DIVISION	/	Divides two values	`10 / 5`	2
MULTIPLICATION	*	Multiplies two values using an asterisk (Note that this is not the letter x)	`10 * 5`	50
INCREMENT	++	Adds one to the current number	`i = 10; i++;`	11
DECREMENT	--	Subtracts one from the current number	`i = 10; i--;`	9
MODULUS	%	Divides two values and returns the remainder	`10 % 3`	1

ORDER OF EXECUTION

Several arithmetic operations can be performed in one expression, but it is important to understand how the result will be calculated. Multiplication and division are performed *before* addition or subtraction. This can affect the number that you expect to see. To illustrate this effect, look at the following examples.

Here the numbers are calculated left to right, so the total is 16:

```
total = 2 + 4 + 10;
```

But in the following example the total is 42 (not 60):

```
total = 2 + 4 * 10;
```

This is because multiplication and division happen *before* addition and subtraction.

To change the order in which operations are performed, place the calculation you want done *first* inside parentheses. So for the following, the total is 60:

```
total = (2 + 4) * 10;
```

The parentheses indicate that the 2 is added to the 4, and *then* the resulting figure is multiplied by 10.

USING ARITHMETIC OPERATORS

JAVASCRIPT c02/js/arithmetic-operator.js

```
var subtotal = (13 + 1) * 5;      // Subtotal is 70
var shipping = 0.5 * (13 + 1);    // Shipping is 7

var total = subtotal + shipping; // Total is 77

var elSub = document.getElementById('subtotal');
elSub.textContent = subtotal;

var elShip = document.getElementById('shipping');
elShip.textContent = shipping;

var elTotal = document.getElementById('total');
elTotal.textContent = total;
```

RESULT

This example demonstrates how mathematical operators are used with numbers to calculate the combined values of two costs.

The first couple of lines create two variables: one to store the subtotal of the order, the other to hold the cost of shipping the order; so the variables are named accordingly: `subtotal` and `shipping`.

On the third line, the total is calculated by adding together these two values.

This demonstrates how the mathematical operators can use variables that represent numbers. (That is, the numbers do not need to be written explicitly into the code.)

The remaining six lines of code write the results to the screen.

STRING OPERATOR

There is just one string operator: the + symbol. It is used to join the strings on either side of it.

There are many occasions where you may need to join two or more strings to create a single value. Programmers call the process of joining together two or more strings to create one new string **concatenation**.

For example, you might have a first and last name in two separate variables and want to join them to show a full name. In this example, the variable called `fullName` would hold the string `'Ivy Stone'`.

```
var firstName = 'Ivy ';
var lastName = 'Stone';
var fullName = firstName + lastName;
```

MIXING NUMBERS AND STRINGS TOGETHER

When you place quotes around a number, it is a string (not a numeric data type), and you cannot perform addition operations on strings.

```
var cost1 = '7';
var cost2 = '9';
var total = cost1 + cost2;
```

You would end up with a string saying `'79'`.

If you try to add a numeric data type to a string, then the number becomes part of the string, e.g., adding a house number to a street name:

```
var number = 12;
var street = 'Ivy Road';
var add = number + street;
```

You would end up with a string saying `'12Ivy Road'`.

If you try to use any of the other arithmetic operators on a string, then the value that results is usually a value called `NaN`. This means "not a number."

```
var score = 'seven';
var score2 = 'nine';
var total = score * score2;
```

You would end up with the value `NaN`.

USING STRING OPERATORS

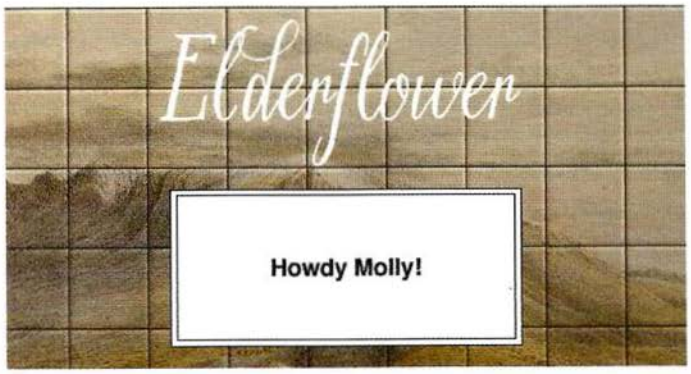
JAVASCRIPT c02/js/string-operator.js

```
var greeting = 'Howdy ';
var name = 'Molly';

var welcomeMessage = greeting + name + '!';

var el = document.getElementById('greeting');
el.textContent = welcomeMessage;
```

HTML c02/string-operator.html

```
<h1>Elderflower</h1>
<div id="content">
  <div id="greeting" class="message">Hello
    <span id="name">friend</span>!
  </div>
</div>
<script src="js/string-operator.js"></script>
```

RESULT



This example will display a personalized welcome message on the page.

The first line creates a variable called `greeting`, which stores the message for the user. Here the greeting is the word `Howdy`.

The second line creates a variable that stores the name of the user. The variable is called `name`, and the user in this case is `Molly`.

The personal welcome message is created by concatenating (or joining) these two variables, adding an exclamation mark, and storing them in a new variable called `welcomeMessage`.

Look back at the `greeting` variable on the first line, and note how there is a space after the word `Howdy`. If the space was omitted, the value of `welcomeMessage` would be `"HowdyMolly!"`

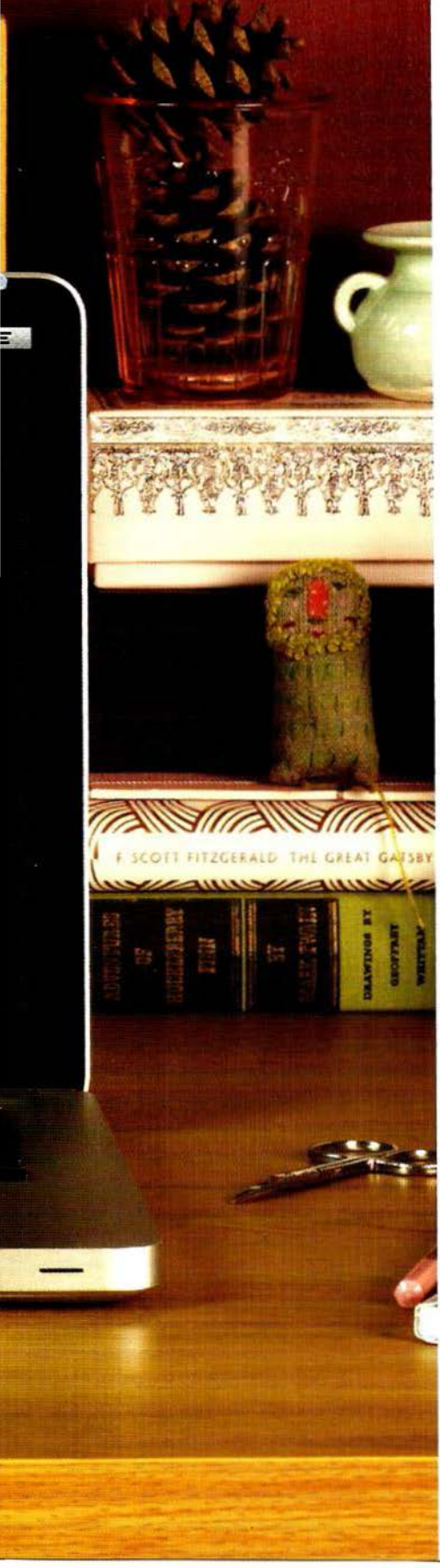

KRAFT ENVELOPE 85576006
The Book of Seaweed
A. Major
Safari File Edit View History Bookmarks Develop Window Help
JavaScript & jQuery - Chapter 2: Basic JavaScript Instructions - Example
www.javascriptbook.com
Elderflower
Howdy Molly, please check your order:
Custom sign: Montague House
Total tiles: 14
Subtotal: $70
Shipping: $7
Grand total: $77
Pay Now
MacBook Pro

EXAMPLE
BASIC JAVASCRIPT INSTRUCTIONS

c02/example.html

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript &amp;amp; jQuery - Chapter 2: Basic JavaScript Instructions -
      Example</title>
    <link rel="stylesheet" href="css/c02.css" />
  </head>
  <body>
    <h1>Elderflower</h1>
    <div id="content">
      <div id="greeting" class="message">Hello!</div>
      <table>
        <tr>
          <td>Custom sign: </td>
          <td id="userSign"></td>
        </tr>
        <tr>
          <td>Total tiles: </td>
          <td id="tiles"></td>
        </tr>
        <tr>
          <td>Subtotal: </td>
          <td id="subTotal">$</td>
        </tr>
        <tr>
          <td>Shipping: </td>
          <td id="shipping">$</td>
        </tr>
        <tr>
          <td>Grand total: </td>
          <td id="grandTotal">$</td>
        </tr>
      </table>
      <a href="#" class="action">Pay Now</a>
    </div>
    <script src="js/example.js"></script>
  </body>
</html>
```



EXAMPLE
BASIC JAVASCRIPT INSTRUCTIONS

This example combines several techniques that you have seen throughout this chapter.

You can see the code for this example on the next two pages. Single line comments are used to describe what each section of the code does.

To start, three variables are created that store information that is used in the welcome message. These variables are then concatenated (joined together) to create the full message the user sees.

The next part of the example demonstrates how basic math is performed on numbers to calculate the cost of a sign.

- A variable called `sign` holds the text the sign will show.
- A property called `length` is used to determine how many characters are in the string (you will meet this property on p128).
- The cost of the sign (the subtotal) is calculated by multiplying the number of tiles by the cost of each one.
- The grand total is created by adding $7 for shipping.

Finally, the information is written into the page by selecting elements and then replacing the content of that element (using a technique you meet fully in Chapter 5). It selects elements from the HTML page using the value of their `id` attributes and then updates the text inside those elements.

Once you have worked your way through this example, you should have a good basic understanding of how data is stored in variables and how to perform basic operations with the data in those variables.

EXAMPLE
BASIC JAVASCRIPT INSTRUCTIONS

c02/example.html

HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>JavaScript &amp;amp; jQuery - Chapter 2: Basic JavaScript Instructions -
      Example</title>
    <link rel="stylesheet" href="css/c02.css" />
  </head>
  <body>
    <h1>Elderflower</h1>
    <div id="content">
      <div id="greeting" class="message">Hello!</div>
      <table>
        <tr>
          <td>Custom sign: </td>
          <td id="userSign"></td>
        </tr>
        <tr>
          <td>Total tiles: </td>
          <td id="tiles"></td>
        </tr>
        <tr>
          <td>Subtotal: </td>
          <td id="subTotal">$</td>
        </tr>
        <tr>
          <td>Shipping: </td>
          <td id="shipping">$</td>
        </tr>
        <tr>
          <td>Grand total: </td>
          <td id="grandTotal">$</td>
        </tr>
      </table>
      <a href="#" class="action">Pay Now</a>
    </div>
    <script src="js/example.js"></script>
  </body>
</html>
```

SUMMARY

BASIC JAVASCRIPT INSTRUCTIONS

- A script is made up of a series of statements. Each statement is like a step in a recipe.
- Scripts contain very precise instructions. For example, you might specify that a value must be remembered before creating a calculation using that value.
- Variables are used to temporarily store pieces of information used in the script.
- Arrays are special types of variables that store more than one piece of related information.
- JavaScript distinguishes between numbers (0-9), strings (text), and Boolean values (true or false).
- Expressions evaluate into a single value.
- Expressions rely on operators to calculate a value.

EXAMPLE
BASIC JAVASCRIPT INSTRUCTIONS

JAVASCRIPT

c02/js/example.js

```
// Create variables for the welcome message
var greeting = 'Howdy ';
var name = 'Molly';
var message = ', please check your order:';
// Concatenate the three variables above to create the welcome message
var welcome = greeting + name + message;

// Create variables to hold details about the sign
var sign = 'Montague House';
var tiles = sign.length;
var subTotal = tiles * 5;
var shipping = 7;
var grandTotal = subTotal + shipping;

// Get the element that has an id of greeting
var el = document.getElementById('greeting');
// Replace the content of that element with the personalized welcome message
el.textContent = welcome;

// Get the element that has an id of userSign then update its contents
var elSign = document.getElementById('userSign');
elSign.textContent = sign;

// Get the element that has an id of tiles then update its contents
var elTiles = document.getElementById('tiles');
elTiles.textContent = tiles;

// Get the element that has an id of subTotal then update its contents
var elSubTotal = document.getElementById('subTotal');
elSubTotal.textContent = '$' + subTotal;

// Get the element that has an id of shipping then update its contents
var elSubTotal = document.getElementById('shipping');
elSubTotal.textContent = '$' + shipping;

// Get the element that has an id of grandTotal then update its contents
var elGrandTotal = document.getElementById('grandTotal');
elGrandTotal.textContent = '$' + grandTotal;
```

SUMMARY

BASIC JAVASCRIPT INSTRUCTIONS

- A script is made up of a series of statements. Each statement is like a step in a recipe.
- Scripts contain very precise instructions. For example, you might specify that a value must be remembered before creating a calculation using that value.
- Variables are used to temporarily store pieces of information used in the script.
- Arrays are special types of variables that store more than one piece of related information.
- JavaScript distinguishes between numbers (0-9), strings (text), and Boolean values (true or false).
- Expressions evaluate into a single value.
- Expressions rely on operators to calculate a value.

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.


杉本博司
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.)

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...
```

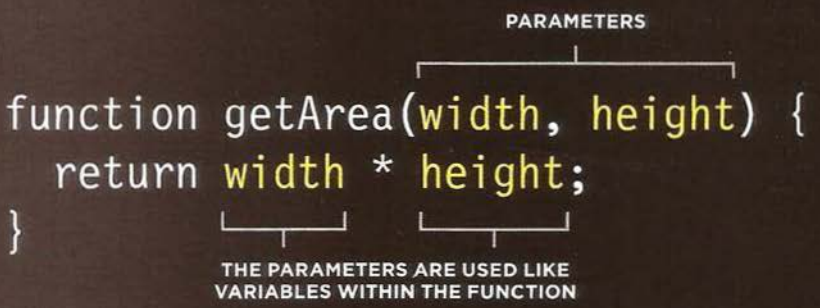
Sometimes you will see a function called *before* it has been declared. This still works because the interpreter runs through a script before executing each statement, so it will know that a function declaration appears later in the script. But for the moment, we will declare the function before calling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



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

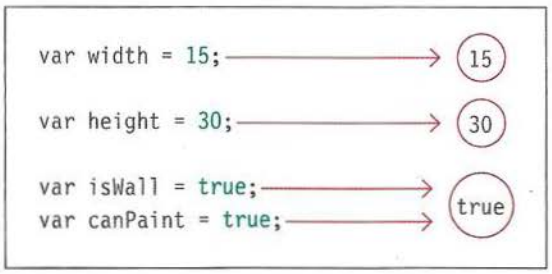
Global variables use more memory. The browser has to remember them for as long as the web page using them is loaded. Local variables are only remembered during the period of time that a function is being executed.

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:



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

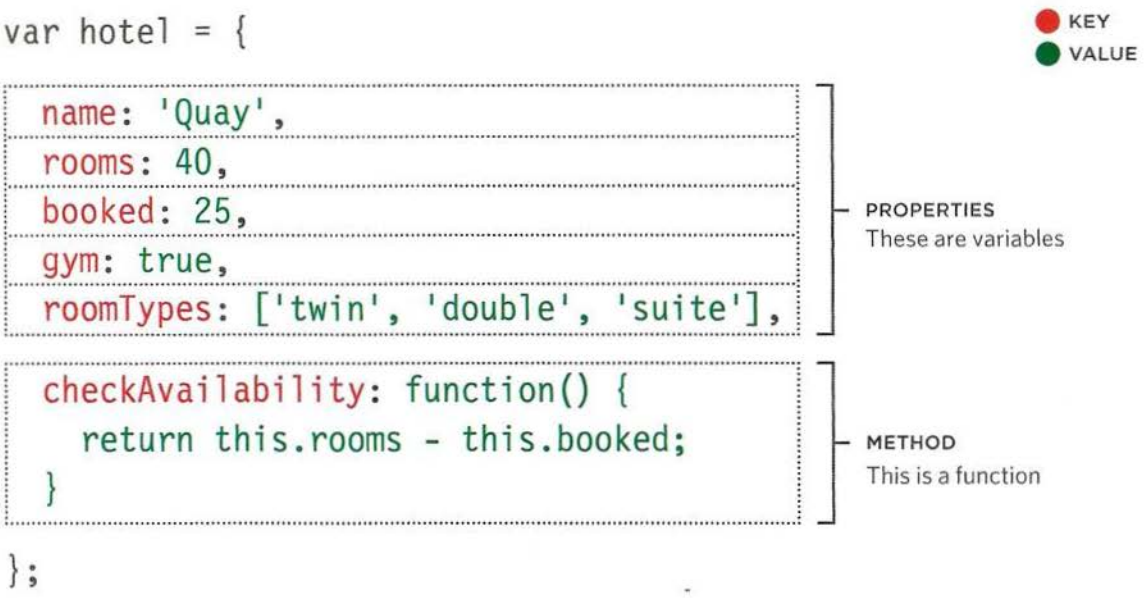
If a function is part of an object, it is called a **method**. Methods represent tasks that are associated with the object. For example, you can check how many rooms are available by subtracting the number of booked rooms from the total number of rooms.

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.



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

	KEY	VALUE
PROPERTIES:	`name`	string
	`rooms`	number
	`booked`	number
	`gym`	Boolean
	`roomTypes`	array
METHODS:	`checkAvailability`	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 = {
  name: 'Quay',
  rooms: 40,
  booked: 25,
  checkAvailability: function() {
    return this.rooms - this.booked;
  }
};
```

PROPERTIES

METHOD

OBJECT
KEY
VALUE

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.

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

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

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();

hotel.name = 'Quay';
hotel.rooms = 40;
hotel.booked = 25;

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

OBJECT
KEY
VALUE

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.

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

KEY
VALUE

PROPERTIES

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).

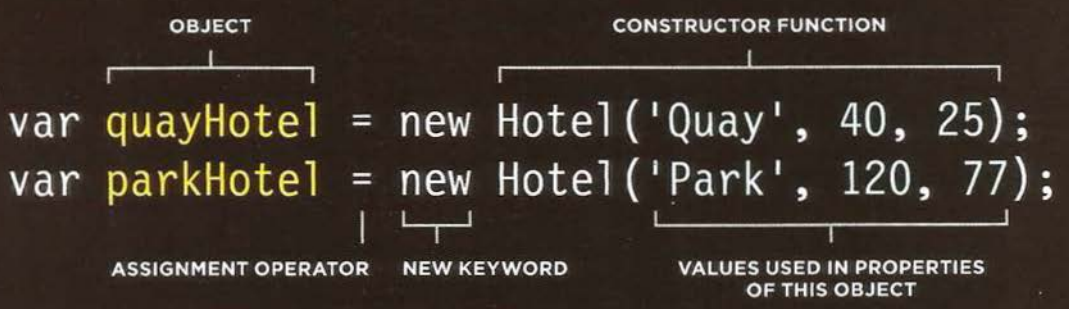
The name of a constructor function usually begins with a capital letter (unlike other functions, which tend to begin with a lowercase character).

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

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

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 `class` 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.

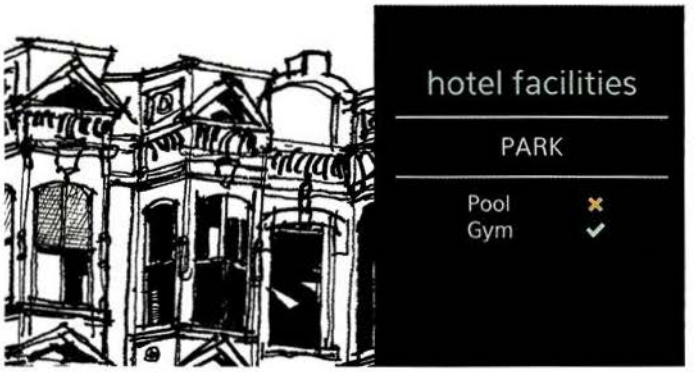
In this case, the `booked` property is removed from the object.

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



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.

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

LITERAL NOTATION

```
var hotel = {}

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

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.

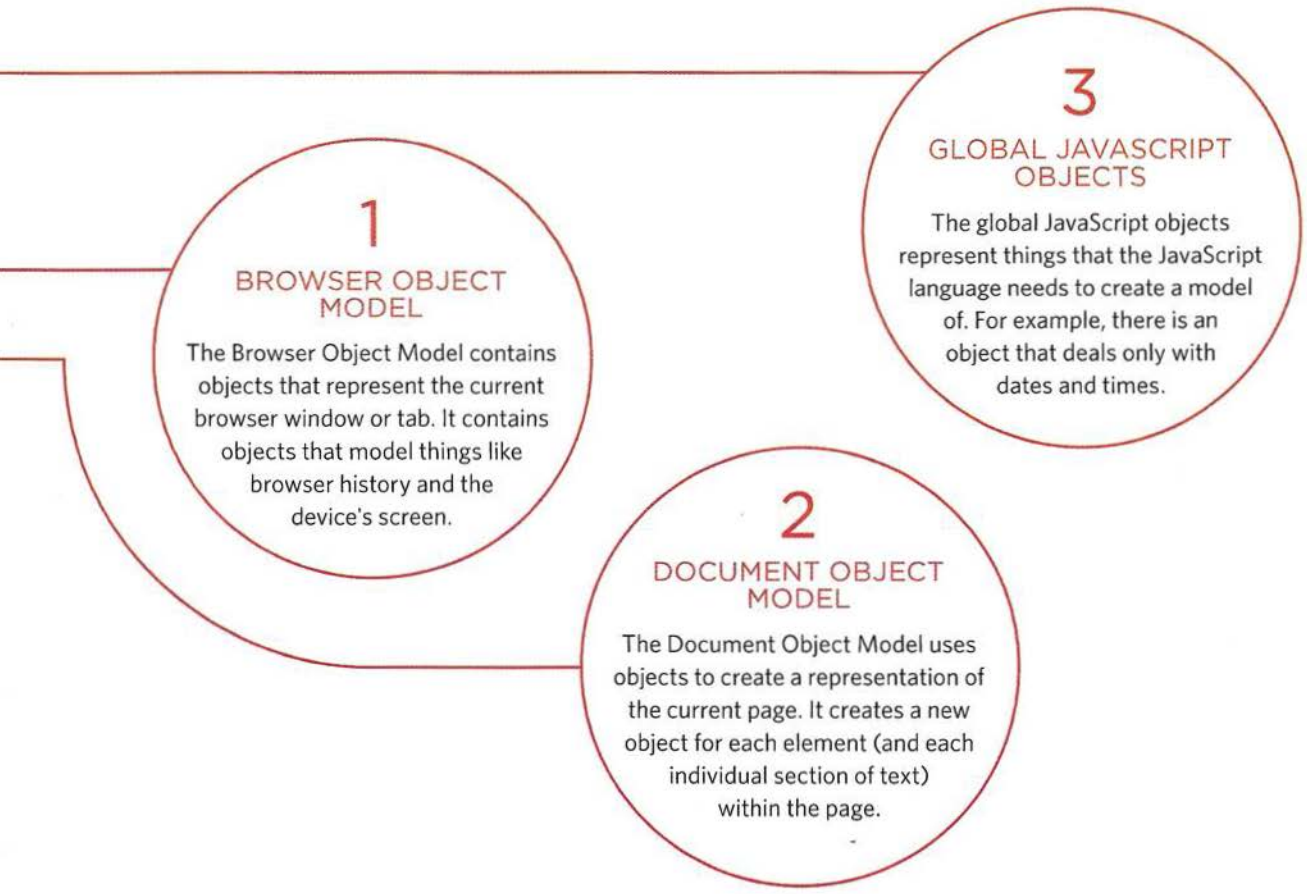
The objects *you* create will usually be specifically written to suit *your* needs. They model the data used within, or contain functionality needed by, your script. Whereas, the built-in objects contain functionality commonly needed by many scripts.

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:



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.

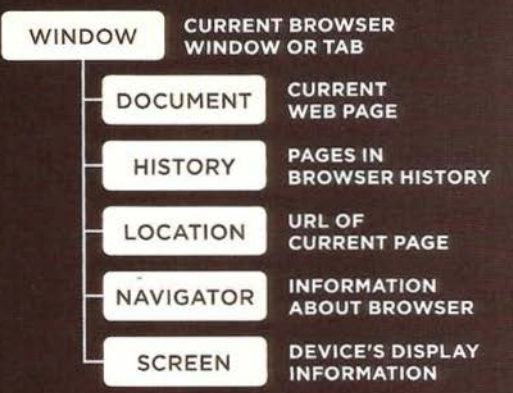
Chapter 5 is dedicated to the Document Object Model because it is needed to access and update the contents of a web page.

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.

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.



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**

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:

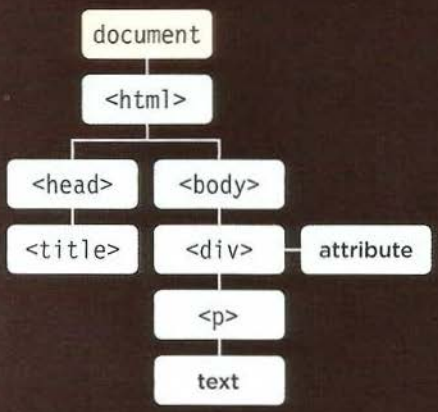
```
window.screen.width;
```

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
`window.innerHeight`	Height of window (excluding browser chrome/user interface) (in pixels)
`window.innerWidth`	Width of window (excluding browser chrome/user interface) (in pixels)
`window.pageXOffset`	Distance document has been scrolled horizontally (in pixels)
`window.pageYOffset`	Distance document has been scrolled vertically (in pixels)
`window.screenX`	X-coordinate of pointer, relative to top left corner of screen (in pixels)
`window.screenY`	Y-coordinate of pointer, relative to top left corner of screen (in pixels)
`window.location`	Current URL of `window` object (or local file path)
`window.document`	Reference to `document` object, which is used to represent the current page contained in window
`window.history`	Reference to `history` object for browser window or tab, which contains details of the pages that have been viewed in that window or tab
`window.history.length`	Number of items in `history` object for browser window or tab
`window.screen`	Reference to `screen` object
`window.screen.width`	Accesses `screen` object and finds value of its `width` property (in pixels)
`window.screen.height`	Accesses `screen` object and finds value of its `height` property (in pixels)

METHOD	DESCRIPTION
`window.alert()`	Creates dialog box with message (user must click OK button to close it)
`window.open()`	Opens new browser window with URL specified as parameter (if browser has pop-up blocking software installed, this method may not work)
`window.print()`	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
`document.title`	Title of current document
`document.lastModified`	Date on which document was last modified
`document.URL`	Returns string containing URL of current document
`document.domain`	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
`document.write()`	Writes text to document (see restrictions on p226)
`document.getElementById()`	Returns element, if there is an element with the value of the `id` attribute that matches (full description see p195)
`document.querySelectorAll()`	Returns list of elements that match a CSS selector, which is specified as a parameter (see p202)
`document.createElement()`	Creates new element (see p222)
`document.createTextNode()`	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.

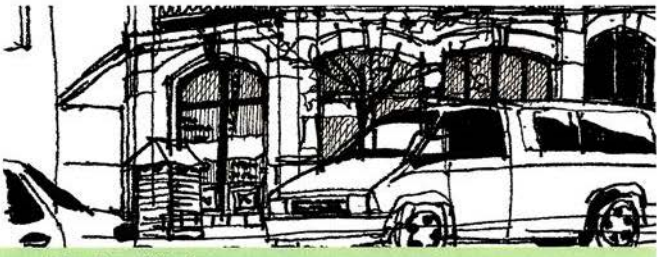
These details are stored inside a variable called `msg`, along with HTML markup to display the information. Again, the += operator adds the new value onto the existing content of the `msg` variable.

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
`length`	Returns number of characters in the string in most cases (see note bottom-left)

METHOD	DESCRIPTION
`toUpperCase()`	Changes string to uppercase characters
`toLowerCase()`	Changes string to lowercase characters
`charAt()`	Takes an index number as a parameter, and returns the character found at that position
`indexOf()`	Returns index number of the first time a character or set of characters is found within the string
`lastIndexOf()`	Returns index number of the last time a character or set of characters is found within the string
`substring()`	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
`split()`	When a character is specified, it splits the string each time it is found, then stores each individual part in an array
`trim()`	Removes whitespace from start and end of string
`replace()`	Like find and replace, 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).

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 ';
② var msg = '<h2>length</h2><p>' + saying.length + '</p>';
  msg += '<h2>uppercase</h2><p>' + saying.toUpperCase() + '</p>';
  msg += '<h2>lowercase</h2><p>' + saying.toLowerCase() + '</p>';
  msg += '<h2>character index: 12</h2><p>' + saying.charAt(12) + '</p>';
③ msg += '<h2>first ee</h2><p>' + saying.indexOf('ee') + '</p>';
  msg += '<h2>last e</h2><p>' + saying.lastIndexOf('e') + '</p>';
  msg += '<h2>character index: 8-14</h2><p>' + saying.substring(8, 14) + '</p>';
  msg += '<h2>replace</h2><p>' + saying.replace('me', 'w') + '</p>';

④ var el = document.getElementById('info');
  el.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
`isNaN()`	Checks if the value is not a number
`toFixed()`	Rounds to specified number of decimal places (returns a string)
`toPrecision()`	Rounds to total number of places (returns a string)
`toExponential()`	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 3.75×10^9 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>toPrecision()</h2><p>' + originalNumber.toPrecision(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. `toPrecision(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
`Math.PI`	Returns pi (approximately 3.14159265359)

METHOD	DESCRIPTION
`Math.round()`	Rounds number to the nearest integer
`Math.sqrt(n)`	Returns square root of positive number, e.g., `Math.sqrt(9)` returns 3
`Math.ceil()`	Rounds number up to the nearest integer
`Math.floor()`	Rounds number down to the nearest integer
`Math.random()`	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

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.

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

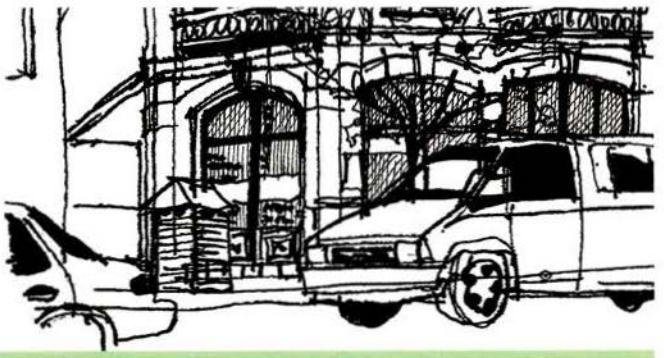
Once you have an instance of the `Date` object (holding the current date and time), you can use any of its properties or methods.

JAVASCRIPT

c03/js/date-object.js

```
① var today = new Date();
② var year = today.getFullYear();

③ var el = document.getElementById('footer');
  el.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:

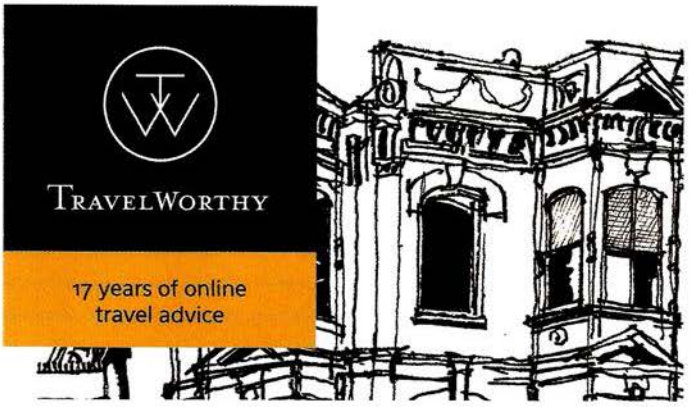
```
MMM DD, YYYY HH:MM:SS
Apr 16, 1996 15:45:55
```

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

JAVASCRIPT c03/js/date-object-difference.js

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

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

RESULT

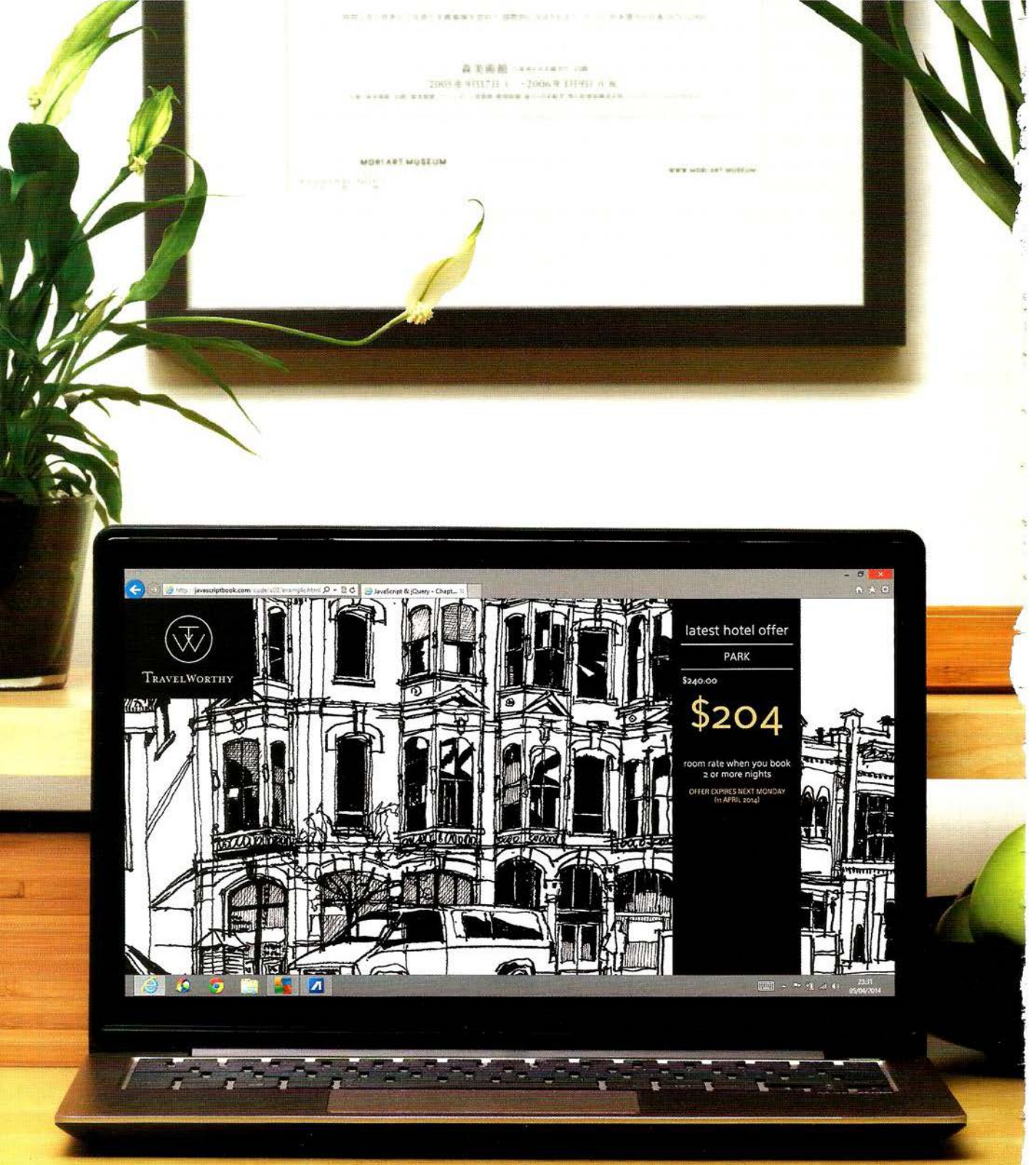


1. In this example, you can see a date being set in the past.

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).


MORI ART MUSEUM
javascriptbook.com
JavaScript & jQuery - Chapt...
TRAVELWORTHY
latest hotel offer
PARK
$240.00
$204
room rate when you book
2 or more nights
OFFER EXPIRES NEXT MONDAY
(11 APRIL 2014)
23:31
05/04/2014

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
}());
```

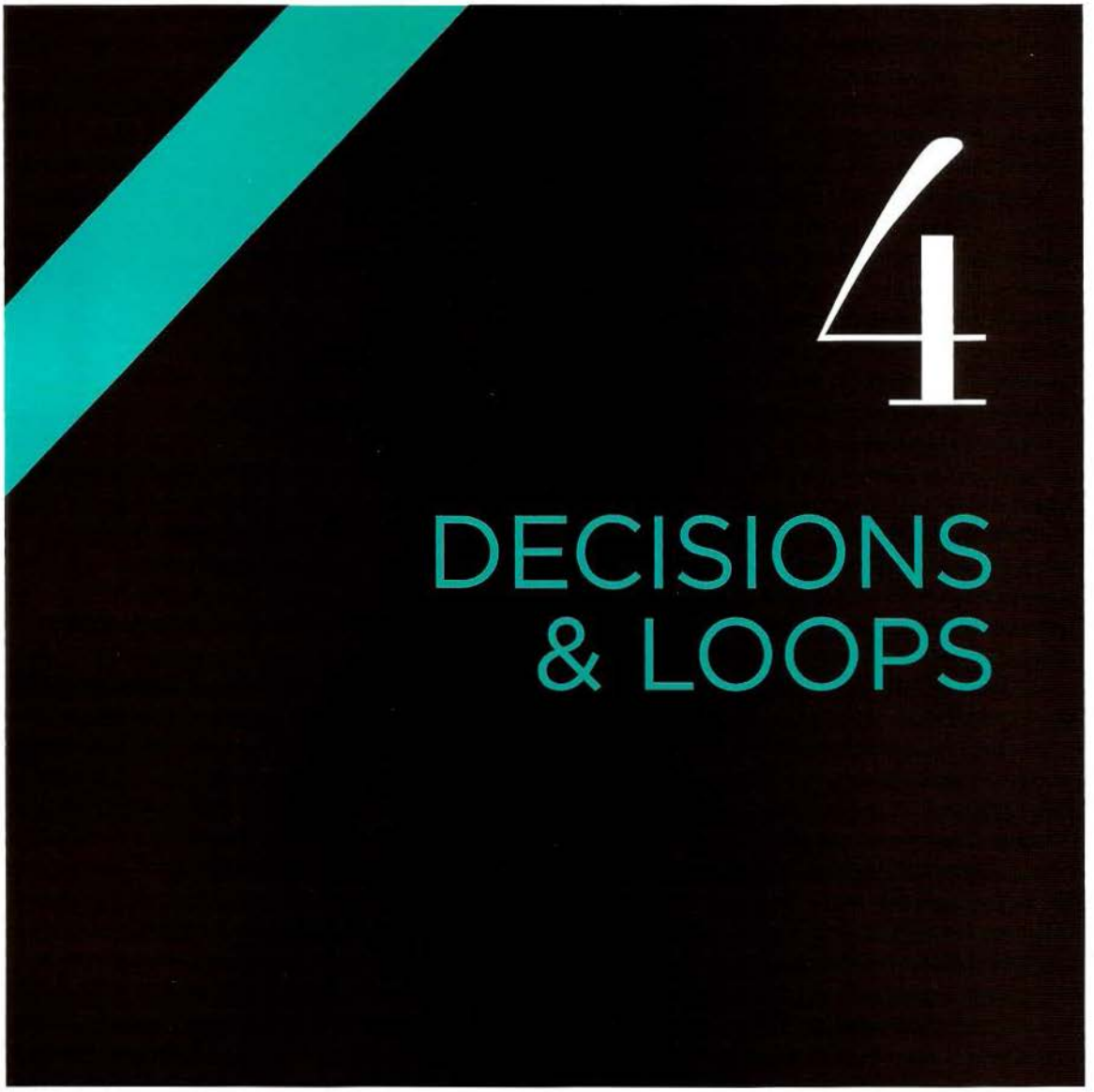
➔ This symbol indicates that the code is wrapping from the previous line and should not contain line breaks.

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).


4
DECISIONS
& LOOPS

Looking at a flowchart (for all but the most basic scripts), the code can take more than one path, which means the browser runs different code in different situations. In this chapter, you will learn how to create and control the flow of data in your scripts to handle different situations.

Scripts often need to behave differently depending upon how the user interacts with the web page and/or the browser window itself. To determine which path to take, programmers often rely upon the following three concepts:

EVALUATIONS

You can analyze values in your scripts to determine whether or note they match expected results.

DECISIONS

Using the results of evaluations, you can decide which path your script should go down.

LOOPS

There are also many occasions where you will want to perform the same set of steps repeatedly.

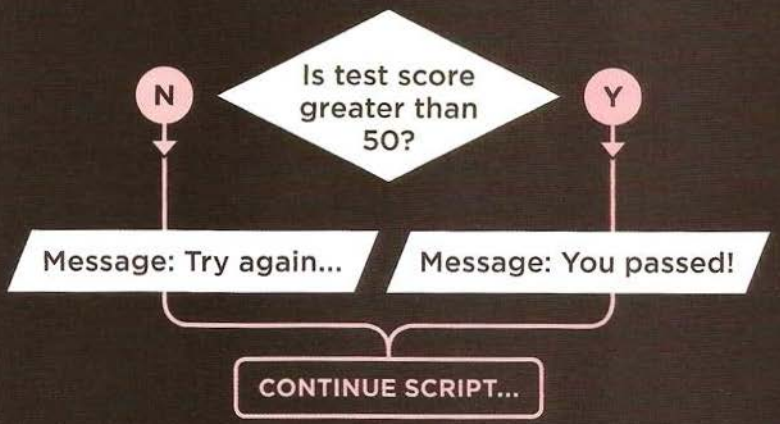

MR PEEK
AT THE ZOO
fischertec
L'HEURE H!
MY LUCKY TRIP
DARTH VADER and son
la mer
Grands Magasins
BULLSEYE!
TARGET PRACTICE FOR YOUR MIND

DECISION MAKING

There are often several places in a script where decisions are made that determine which lines of code should be run next. Flowcharts can help you plan for these occasions.

In a flowchart, the diamond shape represents a point where a decision must be made and the code can take one of two different paths. Each path is made up of a different set of tasks, which means you have to write different code for each situation.

In order to determine which path to take, you set a **condition**. For example, you can check that one value is equal to another, greater than another, or less than another. If the condition returns `true`, you take one path; if it is `false`, you take another path.



In the same way that there are operators to do basic math, or to join two strings, there are **comparison operators** that allow you to compare values and test whether a condition is met or not.

Examples of comparison operators include the greater than (`>`) and less than (`<`) symbols, and double equals sign (`==`) which checks whether two values are the same.

EVALUATING CONDITIONS & CONDITIONAL STATEMENTS

There are two components to a decision:
1: An expression is evaluated, which returns a value
2: A conditional statement says what to do in a given situation

EVALUATION OF A CONDITION

In order to make a decision, your code checks the current status of the script. This is commonly done by comparing two values using a comparison operator which returns a value of **`true`** or **`false`**.

CONDITIONAL STATEMENTS

A conditional statement is based on a concept of if/then/else; *if* a condition is met, *then* your code executes one or more statements, *else* your code does something different (or just skips the step).

CONDITION

```
if (score > 50) {
  document.write('You passed!');
} else {
  document.write('Try again...');
}
```

WHAT THIS IS SAYING:

if the condition returns **true**
execute the statements between the **first** set of curly brackets
otherwise
execute the statements between the **second** set of curly brackets

(You will also meet truthy and falsy values on p167. They are treated as if true or false.)

You can also multiple conditions by combining two or more comparison operators. For example, you can check whether two conditions are both met, or if just one of several conditions is met.

Over the next few pages, you will meet several permutations of the **`if...`** statements, and also a statement called a **switch** statement. Collectively, these are known as **conditional** statements.

COMPARISON OPERATORS: EVALUATING CONDITIONS

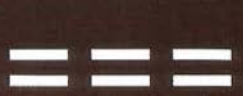
You can evaluate a situation by comparing one value in the script to what you expect it might be. The result will be a Boolean: `true` or `false`.

IS EQUAL TO

This operator compares two values (numbers, strings, or Booleans) to see if they are the same.

`'Hello' == 'Goodbye'` returns `false` because they are *not* the same string.
`'Hello' == 'Hello'` returns `true` because they *are* the same string.

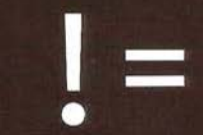
It is usually preferable to use the strict method:



STRICT EQUAL TO

This operator compares two values to check that both the data type and value are the same.

`'3' === 3` returns `false` because they are *not* the same data type or value.
`'3' === '3'` returns `true` because they *are* the same data type and value.



IS NOT EQUAL TO

This operator compares two values (numbers, strings, or Booleans) to see if they are *not* the same.

`'Hello' != 'Goodbye'` returns `true` because they are *not* the same string.
`'Hello' != 'Hello'` returns `false` because they *are* the same string.

It is usually preferable to use the strict method:

STRICT NOT EQUAL TO

This operator compares two values to check that both the data type and value are *not* the same.

`'3' !== 3` returns `true` because they are *not* the same data type or value.
`'3' !== '3'` returns `false` because they *are* the same data type and value.

Programmers refer to the testing or checking of a condition as **evaluating** the condition. Conditions can be much more complex than those shown here, but they usually result in a value of `true` or `false`.

There are a couple of notable exceptions:
i) Every value can be *treated* as true or false even if it is not a Boolean `true` or `false` value (see p167).
ii) In short-circuit evaluation, a condition might not need to run (see p169).

GREATER THAN

This operator checks if the number on the left is *greater than* the number on the right.

`4 > 3` returns `true`
`3 > 4` returns `false`

LESS THAN

This operator checks if the number on the left is *less than* the number on the right.

`4 < 3` returns `false`
`3 < 4` returns `true`

GREATER THAN OR EQUAL TO

This operator checks if the number on the left is *greater than or equal to* the number on the right.

`4 >= 3` returns `true`
`3 >= 4` returns `false`
`3 >= 3` returns `true`

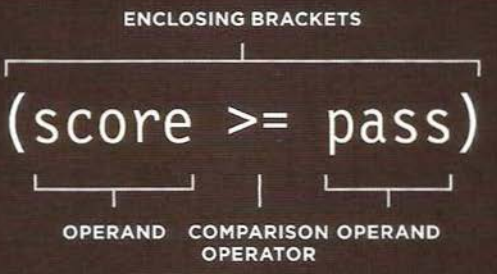
LESS THAN OR EQUAL TO

This operator checks if the number on the left is *less than or equal to* the number on the right.

`4 <= 3` returns `false`
`3 <= 4` returns `true`
`3 <= 3` returns `true`

STRUCTURING COMPARISON OPERATORS

In any condition, there is usually one operator and two operands. The operands are placed on each side of the operator. They can be values or variables. You often see expressions enclosed in brackets.



If you remember back to Chapter 2, this is an example of an **expression** because the condition resolves into a single value: in this case it will be either `true` or `false`.

The enclosing brackets are important when the expression is used as a condition in a comparison operator. But when you are assigning a value to a variable, they are not needed (see right-hand page).

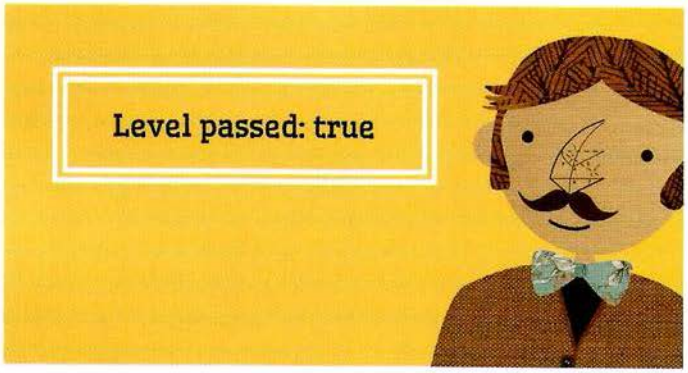
USING COMPARISON OPERATORS

JAVASCRIPT

c04/js/comparison-operator.js

```
var pass = 50;   // Pass mark
var score = 90;  // Score

// Check if the user has passed
var hasPassed = score >= pass;

// Write the message into the page
var el = document.getElementById('answer');
el.textContent = 'Level passed: ' + hasPassed;
```

RESULT



At the most basic level, you can evaluate two variables using a comparison operator to return a `true` or `false` value.

In this example, a user is taking a test, and the script tells the user whether they have passed this round of the test.

The example starts by setting two variables:

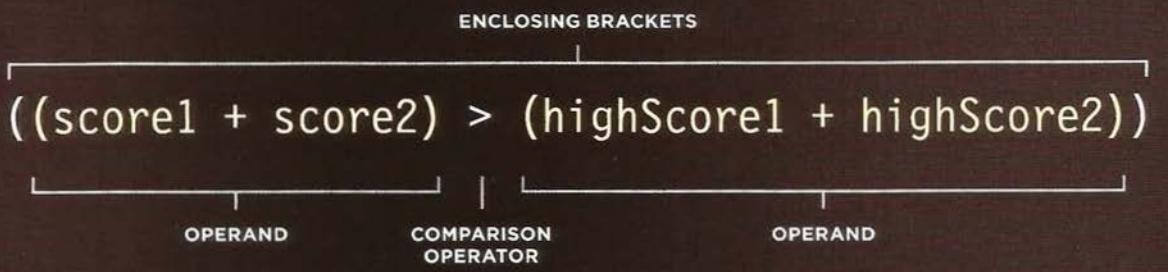
1. `pass` to hold the pass mark
2. `score` to hold the users score

To see if the user has passed, a comparison operator checks whether `score` is greater than or equal to `pass`. The result will be `true` or `false`, and is stored in a variable called `hasPassed`. On the next line, the result is written to the screen.

The last two lines select the element whose `id` attribute has a value of `answer`, and then updates its contents. You will learn more about this technique in the next chapter.

USING EXPRESSIONS WITH COMPARISON OPERATORS

The operand does not have to be a single value or variable name. An operand can be an *expression* (because each expression evaluates into a single value).



COMPARING TWO EXPRESSIONS

In this example, there are two rounds to the test and the code will check if the user has achieved a new high score, beating the previous record.

The script starts by storing the user's scores for each round in variables. Then the highest scores for each round are stored in two more variables.

The comparison operator checks if the user's total score is greater than the highest score for the test and stores the result in a variable called `comparison`.

JAVASCRIPT c04/js/comparison-operator-continued.js

```
var score1 = 90;      // Round 1 score
var score2 = 95;      // Round 2 score
var highScore1 = 75; // Round 1 high score
var highScore2 = 95; // Round 2 high score

// Check if scores are higher than current high scores
var comparison = (score1 + score2) > (highScore1 + highScore2);

// Write the message into the page
var el = document.getElementById('answer');
el.textContent = 'New high score: ' + comparison;
```

RESULT

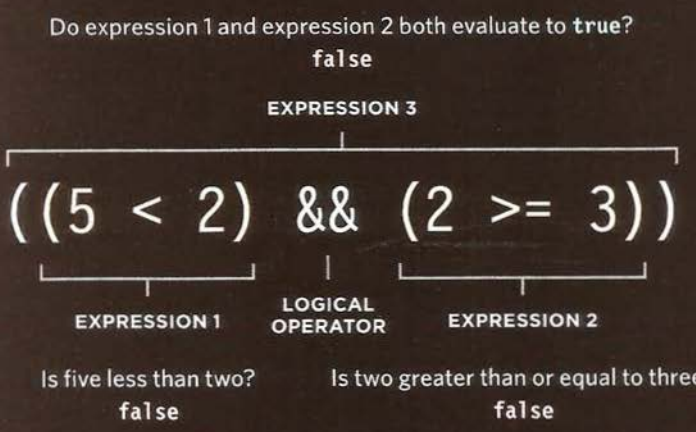
In the comparison operator, the operand on the left calculates the user's total score. The operand on the right adds together the highest scores for each round. The result is then added to the page.

When you assign the result of the comparison to a variable, you do not strictly need the containing parentheses (shown in white on the left-hand page).

Some programmers use them anyway to indicate that the code evaluates into a single value. Others only use containing parentheses when they form part of a condition.

LOGICAL OPERATORS

Comparison operators usually return single values of **true** or **false**. Logical operators allow you to compare the results of more than one comparison operator.



In this one line of code are three expressions, each of which will resolve to the value **true** or **false**.

The expressions on the left and the right both use comparison operators, and both return **false**.

The third expression uses a logical operator (rather than a comparison operator). The logical AND operator checks to see whether both expressions on either side of it return **true** (in this case they do not, so it evaluates to **false**).

&&

LOGICAL AND

This operator tests more than one condition.

`((2 < 5) && (3 >= 2))`
returns **true**

If both expressions evaluate to **true** then the expression returns **true**. If just one of these returns **false**, then the expression will return **false**.

```
true && true  returns true
true && false returns false
false && true  returns false
false && false returns false
```

||

LOGICAL OR

This operator tests at least one condition.

`((2 < 5) || (2 < 1))`
returns **true**

If either expression evaluates to **true**, then the expression returns **true**. If both return **false**, then the expression will return **false**.

```
true || true  returns true
true || false returns true
false || true  returns true
false || false returns false
```

!

LOGICAL NOT

This operator takes a single Boolean value and inverts it.

`!(2 < 1)`
returns **true**

This reverses the state of an expression. If it was **false** (without the ! before it) it would return **true**. If the statement was **true**, it would return **false**.

```
!true  returns false
!false returns true
```

SHORT-CIRCUIT EVALUATION

Logical expressions are evaluated **left** to **right**.
If the first condition can provide enough information to get the answer, then there is no need to evaluate the second condition.

```
false && anything
   ^
```

it has found a **false**

There is no point continuing to determine the other result. They cannot both be **true**.

```
true || anything
   ^
```

it has found a **true**

There is no point continuing because at least one of the values is **true**.

USING LOGICAL AND

In this example, a math test has two rounds. For each round there are two variables: one holds the user's score for that round; the other holds the pass mark for that round.

The logical AND is used to see if the user's score is greater than or equal to the pass mark in both of the rounds of the test. The result is stored in a variable called `passBoth`.

The example finishes off by letting the user know whether or not they have passed both rounds.

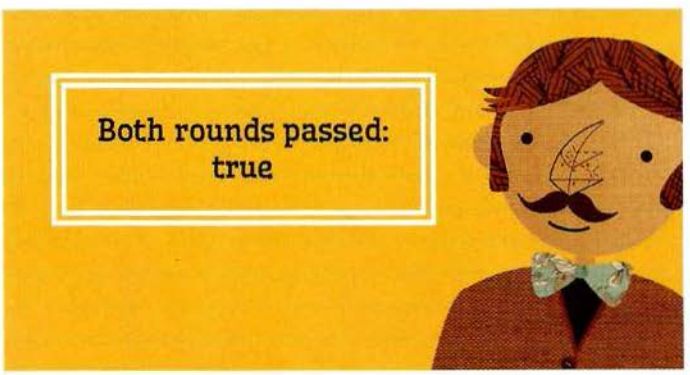
c04/js/logical-and.js

JAVASCRIPT

```
var score1 = 8;    // Round 1 score
var score2 = 8;    // Round 2 score
var pass1 = 6;     // Round 1 pass mark
var pass2 = 6;     // Round 2 pass mark

// Check whether user passed both rounds, store result in variable
var passBoth = (score1 >= pass1) && (score2 >= pass2);

// Create message
var msg = 'Both rounds passed: ' + passBoth;

// Write the message into the page
var el = document.getElementById('answer');
el.textContent = msg;
```

It is rare that you would ever write the Boolean result into the page (like we are doing here). As you will see later in the chapter, it is more likely that you would check a condition, and if it is true, run other statements.

RESULT



USING LOGICAL OR & LOGICAL NOT

Here is the same test but this time using the logical OR operator to find out if the user has passed at least one of the two rounds. If they pass just one round, they do not need to retake the test.

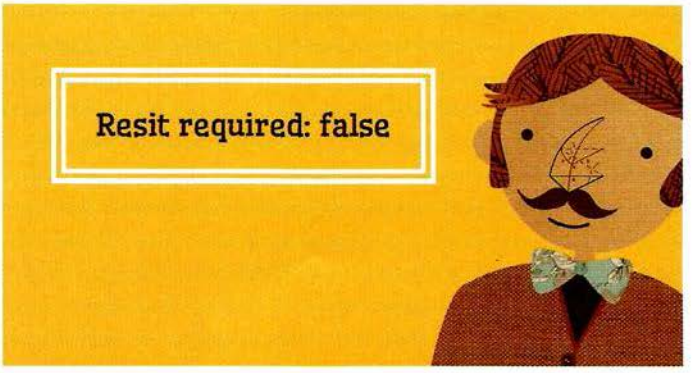
Look at the numbers stored in the four variables at the start of the example. The user has passed both rounds, so the `minPass` variable will hold the Boolean value of `true`.

Next, the message is stored in a variable called `msg`. At the end of the message, the logical NOT will invert the result of the Boolean variable so it is `false`. It is then written into the page.

JAVASCRIPT c04/js/logical-or-logical-not.js

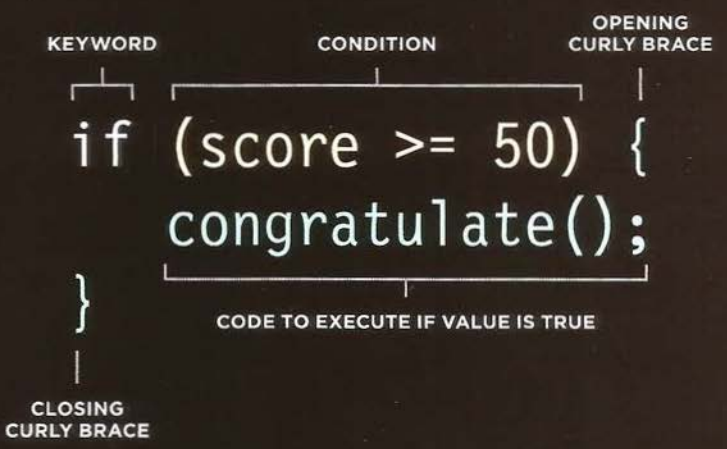
```
var score1 = 8;    // Round 1 score
var score2 = 8;    // Round 2 score
var pass1 = 6;     // Round 1 pass mark
var pass2 = 6;     // Round 2 pass mark

// Check whether user passed one of the two rounds, store result in variable
var minPass = ((score1 >= pass1) || (score2 >= pass2));

// Create message
var msg = 'Resit required: ' + !(minPass);

// Write the message into the page
var el = document.getElementById('answer');
el.textContent = msg;
```

RESULT



IF STATEMENTS

The **if** statement evaluates (or checks) a condition. If the condition evaluates to **true**, any statements in the subsequent code block are executed.



If the condition evaluates to **true**, the following code block (the code in the next set of curly braces) is executed.

If the condition resolves to **false**, the statements in that code block are *not* run. (The script continues to run from the end of the next code block.)

USING IF STATEMENTS

JAVASCRIPT c04/js/if-statement.js

```
var score = 75;      // Score
var msg;             // Message

if (score >= 50) {  // If score is 50 or higher
  msg = 'Congratulations!';
  msg += ' Proceed to the next round.';
}
var el = document.getElementById('answer');
el.textContent = msg;
```

RESULT

In this example, the `if` statement is checking if the value currently held in a variable called `score` is 50 or more.

In this case, the statement evaluates to `true` (because the score is 75, which is greater than 50). Therefore, the contents of the statements within the subsequent code block are run, creating a message that congratulates the user and tells them to proceed.

After the code block, the message is written to the page.

If the value of the `score` variable had been less than 50, the statements in the code block would not have run, and the code would have continued on to the next line after the code block.

JAVASCRIPT c04/js/if-statement-with-function.js

```
var score = 75;     // Score
var msg = '';       // Message

function congratulate() {
    msg += 'Congratulations! ';
}

if (score >= 50) {  // If score is 50 or more
  congratulate();
  msg += 'Proceed to the next round.';
}
var el = document.getElementById('answer');
el.innerHTML = msg;
```

On the left is an alternative version of the same example that demonstrates how lines of code do not always run in the order you expect them to. ***If*** the condition is met then:

1. The first statement in the code block calls the `congratulate()` function.

2. The code within the `congratulate()` function runs.

3. The second line within the `if` statement's code block runs.

IF...ELSE STATEMENTS

The **`if...else`** statement checks a condition.
If it resolves to **`true`** the first code block is executed.
If the condition resolves to **`false`** the second code block is run instead.

```
if (score >= 50) {
    congratulate();
}
else {
    encourage();
}
```

CODE TO EXECUTE IF VALUE IS TRUE

CODE TO EXECUTE IF VALUE IS FALSE

- CONDITIONAL STATEMENT
- CONDITION
- IF CODE BLOCK
- ELSE CODE BLOCK

USING IF...ELSE STATEMENTS

JAVASCRIPT c04/js/if-else-statement.js

```
var pass = 50;        // Pass mark
var score = 75;       // Current score
var msg;              // Message

// Select message to write based on score
if (score >= pass) {
  msg = 'Congratulations, you passed!';
} else {
  msg = 'Have another go!';
}

var el = document.getElementById('answer');
el.textContent = msg;
```

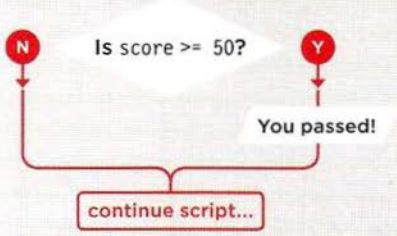
RESULT

Here you can see that an `if...else` statement allows you to provide two sets of code:
1. one set if the condition evaluates to `true`
2. another set if the condition is `false`

In this test, there are two possible outcomes: a user can either get a score equal to or greater than the pass mark (which means they pass), or they can score less than the pass mark (which means they fail). One response is required for each eventuality. The response is then written to the page.

Note that the statements inside an `if` statement should be followed by a semicolon, but there is no need to place one after the closing curly brace of the code blocks.

An `if` statement only runs a set of statements if the condition is `true`:



An `if...else` statement runs one set of code if the condition is `true` or a different set if it is `false`: