

# Begin to Code with JavaScript

## Rob Miles



This is a pre-release section of the work “Begin to Code with JavaScript”. Everything is subject to change, especially the jokes. You can find the “Begin to Code with JavaScript” podcast page here:

[www.robmiles.com/ispodcast](http://www.robmiles.com/ispodcast)

The code samples for the book and links to the screencasts can be found here:

[www.begintocodewithjavascript.com](http://www.begintocodewithjavascript.com)

Feel free to send constructive comments to [writing@begintocodewithjavascript.com](mailto:writing@begintocodewithjavascript.com)

# Introduction

Programming is the most creative thing you can learn how to do. Why? If you learn to paint, you can create pictures. If you learn to play the violin, you can make music. But if you learn to program, you can create entirely new experiences (and you can make pictures and music too, if you wish). Once you've started on the programming path, there's no limit to where you can go. There are always new devices, technologies, and marketplaces where you can use your programming skills.

Think of this book as your first step on a journey to programming enlightenment. The best journeys are undertaken with a destination in mind, and the destination of this journey is "usefulness." By the end of this book, you will have the skills and knowledge to write useful programs and make them available to anyone in the world.

But first, a word of warning. I would not say that learning to write programs is easy. This is for two reasons:

- If I tell you it's easy, and you still can't do it you might feel bad about this (and rather cross with me)
- If I tell it's easy and you manage to do it, you might think that it isn't worth doing.

Learning to program is not easy. It's a kind of difficult that you might not have seen before. Programming is all about detail and sequencing. You must learn how the computer does things and how to express what you want it to do.

Imagine that you were lucky enough to be able to afford your own personal chef. At the start you would have to explain things like "If it is sunny outside I like orange juice and a grapefruit for breakfast, but if it is raining I'd like a bowl of porridge and a big mug of coffee". Occasionally your chef would make mistakes, perhaps you would get a black coffee rather than the latte that you wanted. However, over time you would add more detail to your instructions until your chef knew exactly what to do.

A computer is like a chef that doesn't even know how to cook. Rather than saying "make me a coffee" you would have to say, "Take the brown powder from the coffee bag and add it to hot water". Then you would have to explain how to make hot water, and how you must be careful with the kettle and so on. This is hard work.

It turns out that the key to success as a programmer is much the same as for many other endeavors. To become a world-renowned violin player, you will have to practice a lot. The same is true for programming. You must spend a lot of time working on your programs to acquire code-writing skills. However, the good news is that just as a violin player really enjoys making the instrument sing, making a computer do exactly what you want turns out to be a very rewarding experience. It gets even more enjoyable when you see other people using programs that you've written and finding them useful and fun to use.

# How this book fits together

I've organized this book in three parts. Each part builds on the previous one with the aim of turning you into a successful programmer. We start off discovering the environment in which JavaScript programs run. Then we learn the fundamentals of programming and we finish by making some properly useful (and fun) programs.

## Part 1: The JavaScript world

The first part gets you started. You'll discover the environment in which JavaScript programs run and learn how to create web pages containing JavaScript programs.

## Part 2: Coding with JavaScript

Part 2 describes the features of the JavaScript that you use to create programs that work on data. You will pick up some fundamental programming skills that apply to a wide range of other languages, and get you thinking about what it is that programs actually do. You'll find out how to break large programs into smaller elements and how you can create custom data types that reflect the specific problem being solved.

## Part 3: Useful JavaScript

Now that you can make JavaScript programs it's time to have some fun with them. You'll discover how to create good looking applications, how to make programs that are secure and reliable and finish off with a bit of game development.

# How you will learn

In each chapter, I will tell you a bit more about programming. I'll show you how to do something, and then I'll invite you to make something of your own by using what you've learned. You'll never be more than a page or so away from doing something or making something unique and personal. After that, it's up to you to make something amazing!

You can read the book straight through if you like, but you'll learn much more if you slow down and work with the practical parts along the way. Like learning to ride a bicycle, you'll learn by *doing*. You must put in the time and practice to learn how to do it. But this book will give you the knowledge and confidence to try your hand at programming, and it will also be around to help you if your programming doesn't turn out as you expected. Here are some elements in the book that will help you learn by doing:

### Make Something Happen

Yes, the best way to learn things is by doing, so you'll find "Make Something Happen" elements throughout the text. These elements offer ways for you to practice your programming skills. Each starts with an example and

then introduces some steps you can try on your own. Everything you create will run on Windows, macOS, or Linux.

## Code Analysis

A great way to learn how to program is by looking at code written by others and working out what it does (and sometimes why it doesn't do what it should). The book contains over 150 sample programs for you to look at. In this book's "Code Analysis" challenges, you'll use your deductive skills to figure out the behavior of a program, fix bugs, and suggest improvements.

## What Could Go Wrong?

If you don't already know that programs can fail, you'll learn this hard lesson soon after you begin writing your first program. To help you deal with this in advance, I've included "What Could Go Wrong?" elements, which anticipate problems you might have and provide solutions to those problems. For example, when I introduce something new, I'll sometimes spend some time considering how it can fail and what you need to worry about when you use the new feature.

### Programmer's Points

I've spent a lot of time teaching programming. But I've also written many programs and sold a few to paying customers. I've learned some things the hard way that I really wish I'd known at the start. The aim of "Programmer's Points" is to give you this information up front so that you can start taking a professional view of software development as you learn how to do it.

"Programmer's Points" cover a wide range of issues, from programming to people to philosophy. I strongly advise you to read and absorb these points carefully—they can save you a lot of time in the future!

# What you will need

You'll need a computer and some software to work with the programs in this book. I'm afraid I can't provide you with a computer, but in the first chapter you'll find out how you can get started with nothing more than a computer and a web browser. Later you'll discover how to use the Visual Studio Code editor to create JavaScript programs.

## Using a PC or laptop

You can use Windows, macOS, or Linux to create and run the programs in the text. Your PC doesn't have to be particularly powerful, but these are the minimum specifications I'd recommend:

- A 1 GHz or faster processor, preferably an Intel i5 or better.
- At least 4 gigabytes (GB) of memory (RAM), but preferably 8 GB or more.

- 256 GB hard drive space. (The JavaScript frameworks and Visual Studio Code installations take about 1 GB of hard drive space.)

There are no specific requirements for the graphics display, although a higher-resolution screen will enable you to see more when writing your programs.

## Using a mobile device

You can run JavaScript programs on a mobile phone or tablet by visiting the web pages in which the programs are held. There are also some applications that can be used to create and run JavaScript programs but my experience has been that a laptop or desktop computer is a better place to work.

## Using a Raspberry Pi

If you want to get started in the most inexpensive way possible you can use a Raspberry Pi running the Raspbian Operating System. This has a Chromium compatible browser and is also capable of running Visual Studio Code.

## Sample Code

In every chapter in this book, I'll demonstrate and explain programs that teach you how to begin to program—and that you can then use to create programs of your own. You can download this book's sample code from GitHub by following the link here:

<http://www.begintocodewithjavascript.com/code>

GitHub was developed as a software development platform but it has turned out to be much more than that. It is a place where anyone can store files and share them. All the sample programs used in the book are held on GitHub.

At the start of the book you'll discover how to use GitHub to make your own copy of the sample programs. You can then use GitHub to publish JavaScript enabled web pages for anyone in the world to view.

You will need to connect to the internet and create a GitHub account (it is free) to do this.

## Electronic media

For the important content elements, I've made some videos. The book text will contain screenshots that you can use, but these can go out of date. Follow the links to the walkthroughs to get the latest steps to follow. There are also some audio recordings you can listen to if you feel brave. You can find all these here:

<http://www.begintocodewithjavascript.com/media>

## Acknowledgments

Thanks to everyone for giving me a chance to do it all again.

Pre-release

# 1

# The world of JavaScript

We are going to start our journey by looking at the world of JavaScript. We'll begin by considering just what it is that a programming language does. Then we'll investigate the JavaScript programming language and discover how JavaScript programs get to run on your computer. We'll learn how web pages provide an environment for JavaScript and how to use HyperText Markup Language (HTML) and Cascading StyleSheets (CSS) to create containers for our JavaScript programs. We'll discover just how powerful modern web browsers are as software development tools and how to have a conversation with JavaScript from within a browser. We'll also learn how to manage our software source code and share it with others.

# 1

# Running JavaScript

## What you will learn

Programmers have a set of tools and techniques they use when they create programs. In this chapter, you're going to discover how JavaScript programs run on a computer. You'll also have your first of many conversations with the JavaScript command prompt and investigate your first JavaScript program. Finally, you'll download the Git and Visual Studio Code tools and the example programs for this book and do some simple editing.

## What is JavaScript?

Before we go off and look at some JavaScript it's worth considering just what we are running. JavaScript is a *programming language*. In other words, it's a language that you use to write programs. A program is a set of instructions that tells a computer how to do something. We can't use a "proper" language like English to do this because "proper English" is just too confusing for a computer to understand. As an example, I give you the doctor's instructions:

```
"Drink your medicine after a hot bath."
```

We would probably have a hot bath and then drink our medicine. A computer would probably drink the hot bath and then drink its medicine. You can interpret the above instructions either way because the English language allows you to write ambiguous statements. Programming languages must be designed so that instructions written using them are not open to interpretation, they must tell the computer precisely what to do. This usually means breaking actions down into a sequence of simple steps:

```
Step1: Take a hot bath  
Step2: Drink your medicine
```

We can get this effect in English (as you can see above) but a programming language forces us to write instructions in this way. JavaScript is one of many programming languages which have been invented to provide humans with a way of telling the computer what to do.

In my programming career, I've learnt many different languages over the years and I confidently expect to have to learn even more in the future. None of them are perfect, and I see each of them as a tool that I would use in a particular situation, just like I would choose a different tool depending on whether I was making a hole in a brick wall, a pane of glass or a piece of wood.

Some people get very excited when talking about the "best" programming language. I'm quite happy to discuss what makes the best programming languages, just like I'm happy to tell you all about my favorite type of car, but I don't see this as something to get worked up about. I love JavaScript for its power and the ease with which I can distribute my code. I love Python for its expressiveness and how I can create complex solutions with tiny bits of code. I love the C# programming language for the way it pushes me to produce well-structured solutions. I love the C++ programming language for the way that it gives me absolute control of hardware underneath my program. And so on. JavaScript does have things about it which make me want to tear my hair out in frustration. But that's true of the other languages too. And all programming languages have things about them I love. But most of all I love JavaScript for the way that I can use it to pay my bills.

#### Programmer's Point

##### The best programming language for you is the one that pays you the most

I think it is very fitting that the first programmer's point is one that has a strong commercial focus. Whenever I get asked which is the "best" programming language I always say that my favorite language is the one that I get paid the most to use. It turns out that I'll write in any programming language if the price is right.

I strongly believe that you can enjoy programming well in any language, and that includes JavaScript. Conversely, you can have a horrible time writing bad programs in any language. The language is just the medium which you use to express your ideas.

So, if you tell someone that you're writing JavaScript programs and they tell you that it's not a very good programming language for reasons that you don't understand, just show them how many jobs there are out there for people

who can write JavaScript code.

## JavaScript origins

You might think that programming languages are a bit like space rockets, in that they are designed by white-coated scientists with mega-brains who get everything right first time and always produce perfect solutions. However, this is not the case. Programming languages have been developed over the years for all kinds of reasons, including ones like "it seemed a good idea at the time".

JavaScript was invented by Brendan Eich of Netscape Communications Corporation and first appeared in a Netscape web browser that was released at the end of 1995. The language had a variety of names before the company decided on JavaScript. It turned out to be a poor choice of name because it makes it easy to confuse JavaScript with the Java programming language, which is actually quite different from JavaScript.

JavaScript was intended as a simple way of making web pages interactive. Its name reflects the way that it was supposed to be used alongside Java applications (called *applets*) running in a web browser. However, JavaScript was extended beyond all the expectations of its creator and is now one of the most popular programming languages in the world. Whenever you visit a web site it is almost certain that you will be talking to a JavaScript program.

This book will teach you JavaScript, but actually I'm trying to turn you into a programmer. The fundamentals of program creation are the same for JavaScript and pretty much all programming languages. Once you've learned how to write JavaScript you'll be able to transfer this skill into many other languages, including C++, C#, Visual Basic and Python. It's a bit like the way that once you have learned to drive you can drive any vehicle. When you are using a strange car you just need to find out where the various switches and controls are, and then you can set off on your journey.

## JavaScript and the web browser

The inventor of JavaScript intended for it to be used in a web browser and that is where we are going to start using it. It is possible to create JavaScript programs that run outside the browser, we will consider how to do this in the third part of this text. You can use any modern browser, but the exercises in this text use a browser based on the Chromium framework. I'm using Microsoft Edge Chromium which is available for Windows PC and Apple Macintosh. You can use Google Chrome or the Chromium browser for Linux if you prefer.

## Our first brush with JavaScript

You've reached a significant point in the process of learning how to program. You're about to begin exploring how programs work. This is a bit like opening the front door of a new apartment or house or getting into a shiny new car you've bought. It's an exciting time, so take a deep breath, find a nice cup (or glass) of something you like to drink and settle down comfortably.

You are going to start by doing something that you've done thousands of times in the past. You are going to visit a site on the World Wide Web. But then, with a single press of a key, you're going to explore a world behind the web

page and get a glimpse of the role that JavaScript plays in making it work.

## Make Something Happen

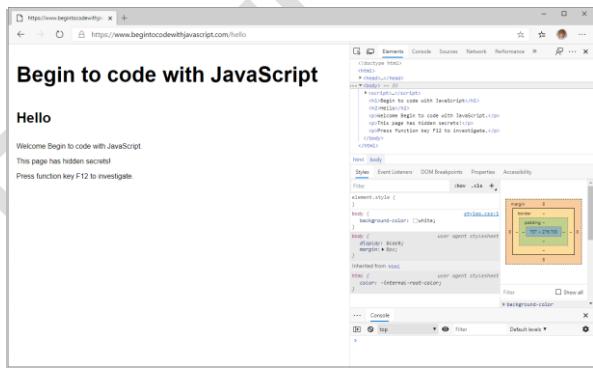
### A web page with secrets

First you need to open your browser. Then visit the web page:

<http://www.begintocodewithjavascript.com/hello>

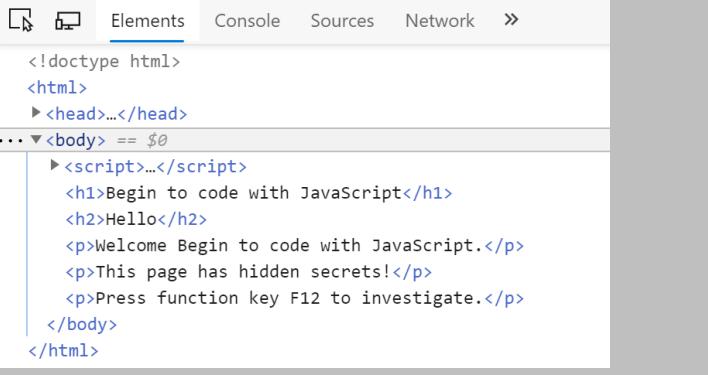


This looks like a very ordinary web page. But it holds a secret behavior that you can find by pressing the F12 key on your keyboard.



1.2 Ch01\_inset01\_02 Developer View

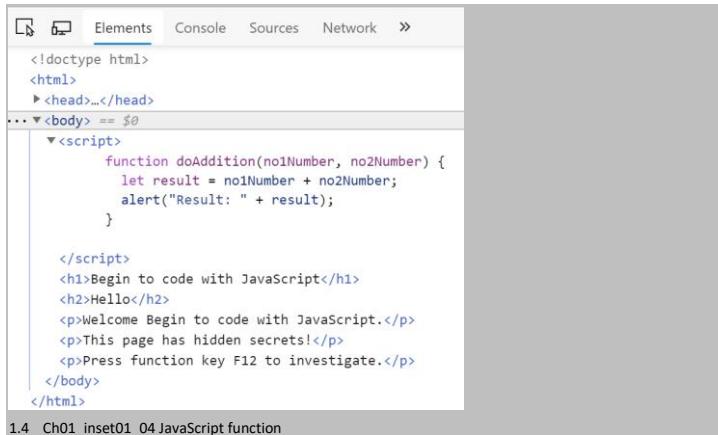
This is called *Developer View*. It shows all the elements that make up the web page. A complete description of everything you can do in this view would not fit in this book. Don't be worried about how complicated it all looks, we are only going to use a couple of the features. We are going to start by looking at the elements that make up the text on the page. Make sure that the Elements tab is selected as you can see in the figure above. Then look at the text underneath.



The figure shows a screenshot of a browser's developer tools. The 'Elements' tab is selected, indicated by a blue underline. Below the tabs, the page's HTML structure is displayed. The code includes a doctype declaration, an HTML element containing a head and body. The body contains a script element, two heading elements (h1 and h2), and three paragraph elements. The text within these elements is visible, demonstrating how the browser parses and displays HTML.

1.3 Ch01\_inset01\_03 Page elements

The Figure above shows the elements on this page. You can see that the text that appears on the page is here. Parts of the text are enclosed in what look like formatting instructions, for example some is marked as `<h1>` and some as `<p>`. If you look back at the web page as displayed you will notice that the `<h1>` text is in a large heading font, whereas the `<p>` text is in smaller text. This is how pages are formatted. You can see how the page works, but where is the secret? To answer this, click the right-pointing arrowhead at the left of the word script to open this part of the view.



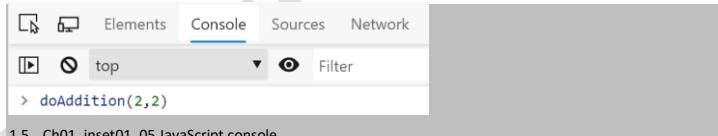
```
<!doctype html>
<html>
  <head></head>
  ... <body> == $0
    <script>
      function doAddition(no1Number, no2Number) {
        let result = no1Number + no2Number;
        alert("Result: " + result);
      }

    </script>
    <h1>Begin to code with JavaScript</h1>
    <h2>Hello</h2>
    <p>Welcome Begin to code with JavaScript.</p>
    <p>This page has hidden secrets!</p>
    <p>Press function key F12 to investigate.</p>
  </body>
</html>
```

#### 1.4 Ch01\_inset01\_04 JavaScript function

Clicking the arrow opens that part of the listing. The hidden feature is a function called `doAddition`. This takes two numbers, adds them together and displays the result using an alert. Later in the text we will go into detail of how this JavaScript works, but even at this stage it is quite clear what is going on.

However, this function is never actually used in the webpage. We can use it ourselves by entering it into the JavaScript console which is built into the browser. This performs JavaScript statements that you type in. You can open the console by selecting the tab at the top of the window.



```
> doAddition(2,2)
```

#### 1.5 Ch01\_inset01\_05 JavaScript console

This is the Console window. I've typed in the name of the function and given it two numbers to work on. When the function runs it display an alert with the result it has calculated.



#### 1.6 Ch01\_inset01\_06 Result alert

We can use the JavaScript console to type in other JavaScript commands. You can use JavaScript to perform calculations by just entering them. When you press the Enter key the answer will be displayed. In fact, the console is often keen to give you an answer even before you press Enter. The console will also try to help as you type in by

suggesting what you might be typing. You can accept any suggestion by using the cursor key to select the suggestion and then pressing the Tab key.

## Code Analysis

We can learn a little about the way JavaScript works by giving the JavaScript console some commands and considering the responses.

```
> 2+3
```

This looks like a sum, and as you might expect, you get a number for an answer

```
<- 5
```

We can repeat this with something other than a number:

```
> "Rob"+" Miles"
```

Some text enclosed in double quotes is interpreted by JavaScript as a string of text and it is perfectly happy to use + to add two strings together. Note that if you want to have a space between the two words in the sum you have to actually put it into the strings that you add together (in my example above there is a space in front of the 'M' in the second word).

```
<- "Rob Miles"
```

We can do other kinds of sums, for example we can subtract using minus (-).

```
> 6-5
```

This produces the result that you would expect.

```
<- 1
```

JavaScript seems quite happy when we ask it to do sensible things. Now, let's try asking it to do something stupid. What do you think would happen if we tried to subtract one string from another using the following statement?

```
> "Rob"- " Miles"
```

While it seems sensible to regard + as meaning "add these strings together" there doesn't seem to be a sensible interpretation of minus when you are working with strings. In other words, it is meaningless to try and subtract one string from another. If you enter this into the console you get a strange response from JavaScript

```
<- NaN
```

The JavaScript console is not calling for grandma to come and sort the problem out. The value "NaN" means "not a number". It is a way that JavaScript indicates the result of a calculation has no meaning. Some programming

languages would display an error message and stop a program if you tried to use them to subtract one string from another. JavaScript does not work like this. It just generates a result value that means “this result is not a number” and keeps going. We will consider how a program can manage errors like this later in the text.

And since we are talking about errors, how about asking JavaScript to do some silly math:

```
> 1/0
```

When I got my first pocket calculator the first I tried to do with it was calculate one divided by zero. I was richly rewarded by a result that just kept counting upwards. What do you think JavaScript will do?

```
<- Infinity
```

JavaScript says that the result of the calculation is the value “Infinity”. This is another special value that is generated by JavaScript when it does calculations. Talking of calculations, how about asking JavaScript to do another one for us.

```
> 2/10+1/10
```

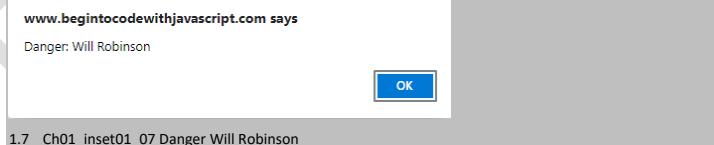
This calculation involves real numbers (i.e. ones with a fractional part). The calculation is adding 0.2 to 0.1 (a fifth to a tenth). This should produce the result 0.3 but what we get is interesting:

```
<- 0.30000000000000004
```

This number is very, very close to 0.3 (the correct answer) but is ever so slightly larger than it should be. This illustrates an important aspect of the way that computers work. Some values that we can express very easily on paper are not held exactly by the machine. This is only usually a problem if we start performing tests with the values that we calculate, for example a check to see if the calculated result above was equal to the value 0.3 might fail because of the tiny difference. Let’s see if we can use JavaScript to do some things for us. How about this:

```
> alert("Danger: Will Robinson")
```

This statement doesn’t calculate a result. Instead it calls a function called `alert`. The function is provided with a string of text. It asks the browser to display the string as a message in an alert box.



#### 1.7 Ch01\_inset01\_07 Danger Will Robinson

This is how the `doAddition` function displays the result it has calculated. Finally, let’s try another function called `print`:

```
> print()
```

What you will see next depends on the computer and the browser that you are using. But you should see a print window appear which offers you the chance to print the web page. If you've ever wondered what happens when you press the print button on a web page; you know now.

Congratulations. You now know how web pages work. Your browser fetches a file from the server and then follows the instructions in that file to build a page for you to look at. The file contains text that is to be displayed on the page along with formatting instructions. A page file can also contain JavaScript program code.

The instructions that the browser follows are expressed in a language called Hyper-Text Markup Language (HTML). In the next chapter we'll take a detailed look at HTML. But before we do that, we need to get some tools that will let us fetch the sample code for this book onto our computer and work with HTML and JavaScript.

## Tools

You will need some software tools to get the best out of the exercises in this book. We are going to start with two, a program called "git" that will manage the program files that we work on and a program called "Visual Studio Code" which we will use to work on the files. Neither of these will cost you any money, and they are available for Windows, Macintosh and Linux based computers. You can follow through the printed instructions below, or you can use one of my Video Walkthroughs that you can find here:

<https://www.begintocodewithjavascript.com/media>

### Programmer's Points

#### Git and Visual Studio Code are professional tools

When you learned to ride a bike you probably had one with training wheels. And people learning to drive a car usually start in something small and easy to handle. You are learning programming with the tools that professionals work with. This is a bit like learning to drive using a Formula 1 racing car. However, there is nothing to worry about here. A Formula 1 car might look a bit scary, but it still has a steering wheel and the usual set of pedals. You don't have to drive it fast if you don't want to and the consequences of a crash are much less.

GitHub and Visual Studio code have a huge range of features, but you don't have to use them. Just like there are buttons on my car dashboard that I don't press because I'm not sure what they do, you don't have to know about every feature of these tools to make good use of them.

It is very sensible to start developing with "proper" tools as recruiters are often as interested in the tools that you are familiar with as they are with the programming languages that you can work with.

## Getting Git

The source code of programs that you write is stored on your computer as files of text. You work on your programs

by changing the contents of these files. When I was starting out programming, I learned very quickly that you can go backwards as well as forwards when writing software. Sometimes I would spend a lot of time making changes to my programs that would turn out to be a bad idea and I would have to go back and undo them all. I solved this problem by making copies of my program code before I did any major edits. That way if anything went bad, I could go back to my original files.

Lots of other programmers noticed this problem too. They also noticed that if you release a program to users it is very useful to have a “snapshot” of that code so that you can keep track of any changes that you make. The best programmers are great at being “intelligently lazy” and so they created software to manage this. One of the most popular programs is called Git.

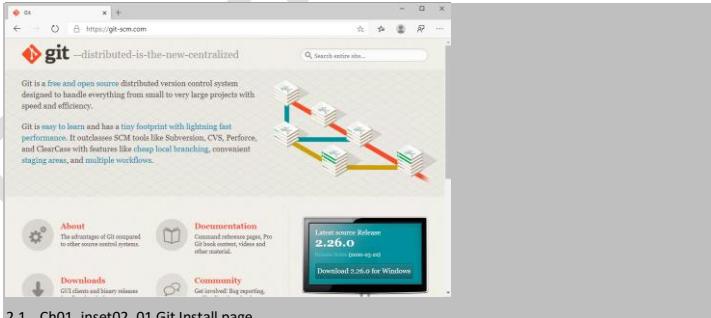
Git was created in 2005 by Linus Torvalds who was writing the Linux operating system at the time. He needed a tool that could track what he was doing and make it easy for him to work with other people. So he created his own. Git is a professional tool and very powerful. It lets large numbers of developers work together on a single project. Different teams can work on their own versions of the code which can then be merged. There is no need for you to use of all these powerful features though. You’re just going to use Git to keep track of our work and as a way of obtaining the example programs.

## Make Something Happen

### Install Git

I'm going to give you instructions for Windows 10. The instructions for macOS are very similar. First you need to open your browser and visit the web page:

<https://git-scm.com>



2.1 Ch01\_inset02\_01 Git Install page

Follow the installation process selecting all the defaults.

## Getting Visual Studio Code

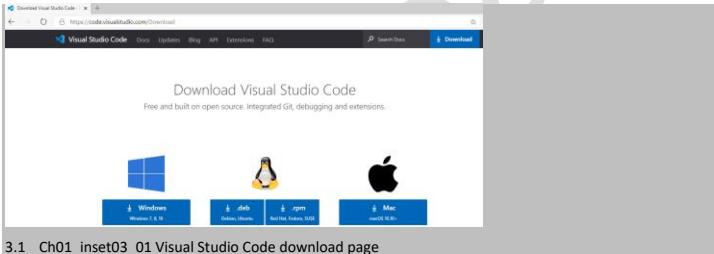
If you want to write a letter you would use a Word processor. To perform calculations, you might use a spreadsheet. Visual Studio Code is a tool that you can use to edit your program files. It can do a lot more than this, as we shall see later. But for now, we are going to use it as a super powerful program editor. Visual Studio Code is free.

Make Something Happen

Install Visual Studio Code

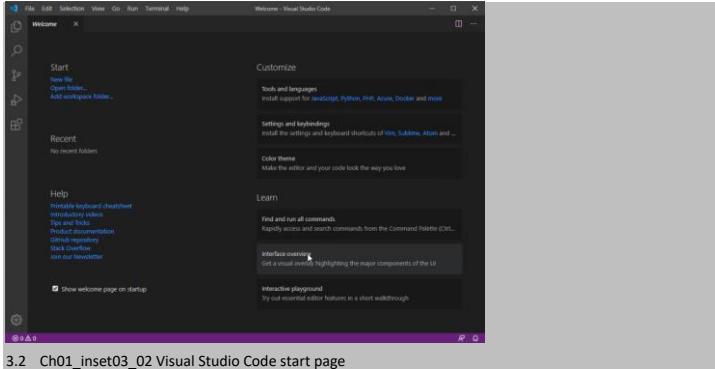
I'm going to give you instructions for Windows 10. The instructions for macOS are very similar. First you need to open your browser and visit the web page:

<https://code.visualstudio.com/Download>



3.1 Ch01\_inset03\_01 Visual Studio Code download page

Click the version of Visual Studio Code that you want and follow the instructions to install it. Once it is installed you will see the start page.



3.2 Ch01\_inset03\_02 Visual Studio Code start page

Now that you have Visual Studio installed the next thing you need to do is fetch the sample files to work on.

## Getting the Sample Files

The sample programs, along with a lot of other stuff, are stored on *GitHub*. GitHub is a service that is underpinned by the Git system. You can store your own files on GitHub (and not just programs). You can also use GitHub to host web pages containing JavaScript programs that you create. This makes programs that you write accessible by anyone in the world. To do all this you will need to create a GitHub username and download some software onto your computer. We will create your username later. For now, we are going to just download the sample repository and edit `hello.html`, the file that we worked with at the start of this chapter.

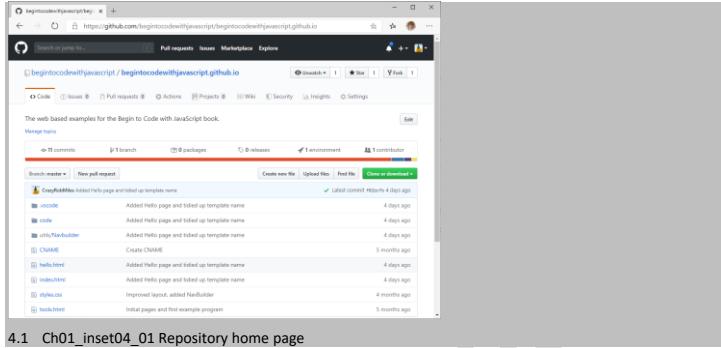
Make Something Happen

Clone the sample repository

A *repository* in Git is a collection of files. Whenever I start working on something new I create a repository to hold all the files I'm going to create. I've got a private repository that contains all the text of this book. And I've made a public repository to hold the sample files. Repositories on GitHub can be accessed directly from the browser. The sample files for this book are at the repository with this url (uniform resource locator):

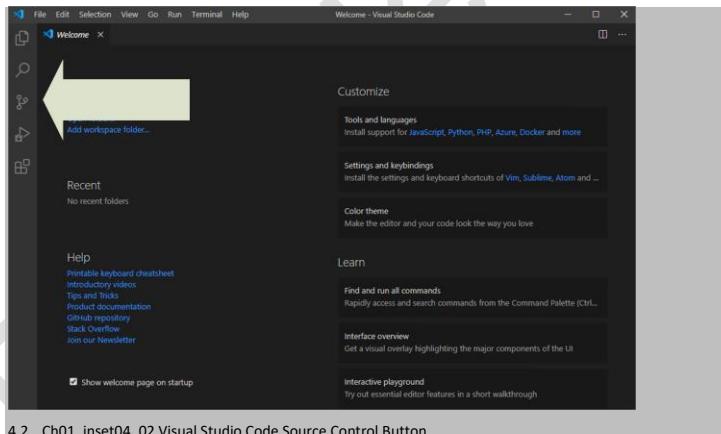
<https://github.com/begintocodewithjavascript/begintocodewithjavascript.github.io>

If you visit this url with your browser you will find that you can navigate all the files, including the file "hello.html" that we investigated earlier and take a look at what is inside them.



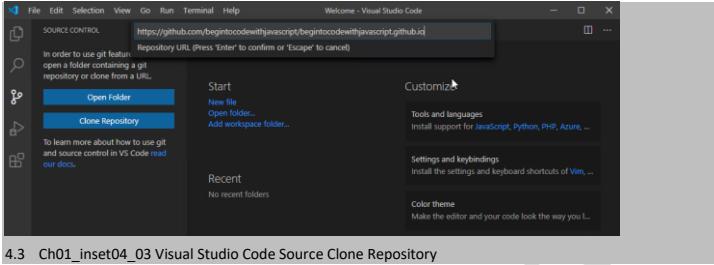
4.1 Ch01\_inset04\_01 Repository home page

You can see that GitHub is keeping track of the changes that I have made to the example programs. We are going to use Visual Studio Code to clone this repository.



4.2 Ch01\_inset04\_02 Visual Studio Code Source Control Button

Start Visual Studio Code and click the Source Control button as shown on the figure above. This opens the Source Control dialog as shown below. Next click the Clone Repository button to begin the process of fetching a repository from GitHub.

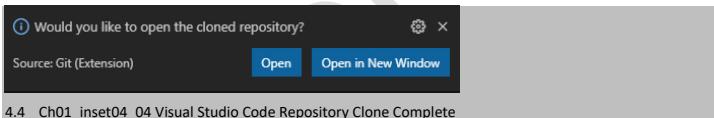


#### 4.3 Ch01\_inset04\_03 Visual Studio Code Source Clone Repository

Visual Studio code is going to download the contents of the repository and store them on your machine. Enter the url of the repository in the dialog that appears. The url you want to use is:

<https://github.com/beginntocodewithjavascript/beginntocodewithjavascript.github.io>

When you press enter at the end of the url you will be asked where on your computer you want to put the files that are about to be copied. I suggest that you create a folder called GitHub in your "documents" folder and use that, but you can put the repository anywhere you like. Once you've selected the folder Visual Studio will copy all the files in the repository from the GitHub site onto your computer.



#### 4.4 Ch01\_inset04\_04 Visual Studio Code Repository Clone Complete

When all the files have been copied Visual Studio Code will ask if you want to open the repository. Click Open to open it.

Congratulations, you have cloned your first repository! Later in the text you will discover how to create your own repositories to store your programs. Remember that you can use GitHub to store anything that you might want to work on, not just program files. If you have an assignment to write you could create a repository to hold the documents and images. This would be an even better idea if you were working on the assignment with other people as GitHub is a great collaboration tool.

## Working on files with Visual Studio Code

We can round off this chapter by working the JavaScript program that we saw at the very start. The process we are going to follow will look like this:

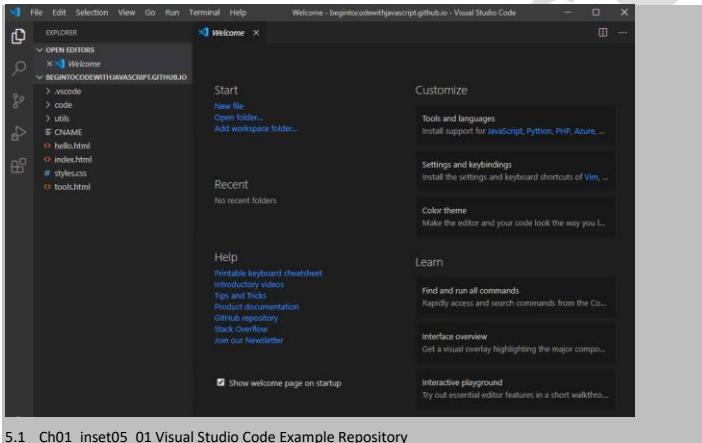
1. Edit the program in the HTML file.
2. Save the file back to disk.
3. Use a web browser to view the HTML file and see what it does.

This is the process you will be using for a lot of the rest of the book. In the next chapter you will discover how to make your programs public so that anyone in the world can view them.

## Make Something Happen

### Edit the secret program

At the end of the last session you opened the example repository that you'd downloaded from GitHub. Now you get to edit the hello.html file that we saw contains the secret program.



5.1 Ch01\_inset05\_01 Visual Studio Code Example Repository

The Explorer window at the left hand side of the Visual Studio Code window provides a view of all the files and folders in the repository. You can click the “>” in front of folders in the Explorer view to open them and view their contents. For now, you are just going to look in the hello.html file, so click the filename hello.html in the Explorer to open it.

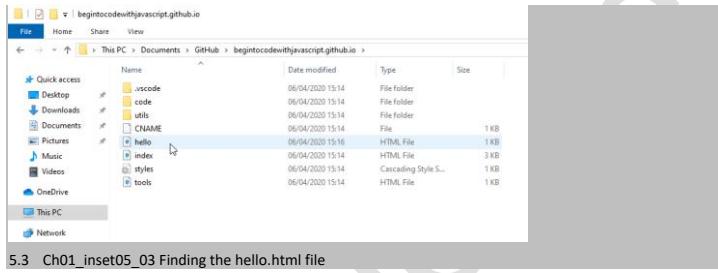
A screenshot of the Visual Studio Code editor. The title bar says 'hello.html - begintocodewithjavascript.github.io - Visual Studio Code'. The left sidebar shows the 'hello.html' file is selected. The main editor area contains the following HTML and JavaScript code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <title>Hello World</title>
  </head>
  <body>
    <script>
      let result = 10 + 20;
      console.log(result);
    </script>
  </body>
</html>
```

5.2 Ch01\_inset05\_02 Visual Studio Code Editing hello.html

Once the page has opened you can make some changes to the text in the file. I've changed one heading so that it says "Hello from Rob". You can save the file by holding down the control key and pressing S (CTRL+S). Or you could use the Save command. Either way, you now want to view the changed file in a browser to see if the changes have worked.

When you cloned the repository, you told Visual Studio Code where to put the files, so now is the time to open File Explorer and navigate to that folder. If you've forgotten where you put the files you can find out just by resting your mouse pointer over the filename in Explorer. Visual Studio Code will then show you the path to that file.



### 5.3 Ch01\_inset05\_03 Finding the hello.html file

If we double-click this file it will be opened by the browser.



### Ch01\_inset05\_04 Browsing the hello.html file

Now you will see the file in all its edited glory. Note that the address being browsed is now a file on your local storage, rather than on the web. Note also that you can press F12 if you like and view the contents of the file just like we did at the start of this chapter.

## What you have learned

You might feel that you've spent a lot of this chapter just following instructions, but actually you've learned rather a

lot. You've discovered that JavaScript is a programming language, providing a means by which you can tell a computer how to do something. You've had a conversation with JavaScript itself. You've learned that looking after the source files of your programs is important, although great programmers sometimes think of very silly names (for example "git") for their programs sometimes. You've installed the git system and your program editor, Visual Studio Code. Finally, you've copied all the example code onto your machine by "cloning" the repository held on GitHub and even managed to edit one file and view the effects in your browser.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions" about JavaScript, computers, programs, and programming.

What does the word "script" mean in the name JavaScript?

The word "script" in the name refers to the way that JavaScript programs were intended to run. The browser would read each JavaScript statement and then perform it; just like an actor would act out the script of a play. This is not how all programming languages work. Some programming languages are designed to be *compiled*. This means that the source code of the program is converted into the low-level instructions that are run by the computer hardware. These low-level instructions are then directly obeyed by the hardware to make the program run.

Compiled languages run faster than scripts because when the compiled program runs the computer doesn't have to put any effort into working out what the program source is doing, it can just obey the low-level instructions. However, you need to make a different version of the compiled code for each different type of computer. For example, a compiled file for a Windows PC would not run on a Raspberry Pi.

JavaScript was intended to perform simple tasks inside a browser, so it was created as a scripting language. However, it has now become so popular that modern browsers compile JavaScript before running it so that it runs as quickly as possible.

Does my JavaScript programs run on the Web Server?

No. The job of a web server is just to serve up files. The browser (the program running on the user's computer) is responsible for actually creating the display of a web page and running any JavaScript programs in that page.

Do JavaScript programs run at the same speed on all computers?

No. The faster the host computer, the faster the browser (and the JavaScript programs it is hosting) will run.

Do JavaScript programs run faster if I have a faster network connection?

No. A faster network connection will improve the speed at which the JavaScript programs will be loaded into the browser, but the actual speed the JavaScript program runs is determined by the speed of the host computer. Having said that, if the JavaScript program uses the host computer network connection these actions will of course happen more quickly.

Can we view the JavaScript programs in every page we visit?

Yes, you can. The F12 trick (pressing F12 when viewing a web page in a browser) will open the development view of the page. You can use this to view the JavaScript source code in the page. If you are concerned about

someone copying the JavaScript code you can use a tool called an *obfuscator* which is a piece of software that changes the appearance (but not the behavior) of a program so that it is very hard to understand. Take a look at <https://www.javascriptobfuscator.com/> for more details.

#### How big can a JavaScript program be?

A JavaScript program can be very large indeed. Modern web browsers are very good at handling large programs and the speed of modern networks means that the code can be downloaded very quickly. Some people have even created complete computer emulations in JavaScript that you can run in a browser.

#### Can you run JavaScript outside a web browser?

Yes you can. Some web pages can be converted into applications which then run on the local computer. There are also ways in which a computer can be made to host JavaScript applications in the same way that a browser does. We will look at these later in the text.

#### Why is "Git" called "Git"?

This is probably the hardest question in this book. In the UK the word "git" is a form of mild abuse. You would call someone a git if they spilled your drink on purpose. It seems that Linus Torvalds called his first version of the program "His stupid content tracker" and then hit upon the word git as a shorter version of this.

#### Can I do private work on GitHub?

Yes. GitHub is very popular with programmers who are working on Open Source projects, but you can also make a GitHub repository private so that only you can see it. If you use the free subscription the number of private repositories you can create is limited, as is the number of people you can work with on a shared project.

#### What do I do if I "break" my program?

Some people worry that things they do with a program on the computer might "break" it in some way. I used to worry about this too, but I've conquered this fear by making sure that whenever I do something, I always have a way back. Git and GitHub are very useful in this respect. Later in the text we will discover how to use GitHub to take "snapshots" of projects which we can return to if we break our program. We can also use the Git desktop program to search for changes that we have made to the files in a project.

#### Why is the Visual Studio Code display of the hello.html file in different colors?

This is called *source code highlighting*. Visual Studio Code has a list of words that are "special" as far as JavaScript and HTML are concerned. These special words are called *keywords*. For each keyword Visual Studio has a characteristic color, in the case of Visual Studio Code keywords are displayed in blue, functions are displayed in yellow, strings of text are orange and everything else is white. The intention is to make it easier for programmers to understand the structure of the program. Note that there is nothing in the program file that specifies the color of each element, this is something that Visual Studio Code does.

#### Will "artificial intelligence" mean that one day we won't have to write programs?

This is a very deep question. To me, artificial intelligence is a field where lots of people are working very hard to make a computer really good at guessing. It turns out that by giving computers lots of information, and telling them how the information is related, a program can then use all this stuff to make a pretty good guess as to the context of a statement.

I suppose that all humans do is "guess" at the meaning of things. Maybe one day a doctor really will want me to drink a hot bath before I take my medicine (see the instructions above), in which case I'll do the wrong thing. However, humans have a much greater capacity to store experiences and link them together, which puts the computer at a distinct disadvantage when it comes to showing intelligence. Maybe in time this will change. We are already seeing that in specific fields of expertise, for example finance and medical diagnosis, artificial intelligence can do very well.

However, in my opinion, when it comes to telling the computer exactly what we want them to do, we'll be needing programmers for quite a long time. Certainly, long enough for you to pay off your mortgage.

# 2

# HyperText Markup Language (HTML)

## What you will learn

In the previous chapter you learned that a JavaScript program can live inside a web page. You saw that the file `hello.html` had a secret script inside it. In this chapter we will find out more about the HTML standard that tells the browser program what a web page should look like. Then we'll discover how to link JavaScript to elements on a web page to allow our programs to interact with the user.

## HTML and the World Wide Web

The first version of a *HyperText Markup Language* (HTML) was created in 1989 by Tim Berners-Lee. He wanted to make it easier for researchers to share information. At the time research reports were written as individual documents. If a document you were reading contained a *reference* to another you would have to go and find the other one. Tim Berners-Lee designed a system of computer *servers* that share electronic copies of documents. A document

could contain *hyper links* to other documents. Readers used a *browser* program to read the document from the servers and follow the links from one document to another. These documents are called *HyperText* documents and the language that described their contents is called the *HyperText Markup Language* or *HTML*.

Tim Berners-Lee also designed a protocol to manage the transfer of *HTML* formatted documents from the server into the browser. This standard is called the *Hyper Text Transfer Protocol* or *HTTP*. There is now also secure version of this protocol called *“HTTPS”* which add security to the web. *HTTPS* allows a browser to confirm the identity of a server and it also protects messages sent between the server and the browser to prevent eavesdropping. The *HTTPS* protocol is what makes it possible for us to use the world wide web for banking and e-commerce.

In 1990 the first system was released as the “World Wide Web”. The documents that you could download were called “web pages”. The sever hosting the web pages was called a “web site”. In 1993 Marc Andreeson added the ability to display images in web pages and the web became extremely popular.

The World Wide Web was designed to be extensible and for many years different browser manufacturers added their own enhancements to the standards, leading to problems with compatibility where web sites would only work with specific browser programs. Recently the situation has stabilized. The World Wide Web Consortium (W3C) now sets out standards which are implemented by all browser manufacturers. The latest standard, *HTML 5*, is now very stable and is the version used in this book.

## Fetching web pages

The location of the page on a server is given using a *uniform resource locator* or url. This has three elements:

- The *protocol* to be used to talk to the site. This sets out how a browser asks for a web page, and how the server replies. Web pages use *HTTP* and *HTTPS*. *HTTP* stands for “*HyperText Transport Protocol*” and *HTTPS* is the secure version of this protocol.
- The *host*. This gives the address of the server on the network. The world wide web sits on top of a networking protocol called *TCP/IP* (*Transport Control Protocol/Internet Protocol*) and this is the address on that network of the system that holds the web site you want to connect to.
- The *path*. This is the path on the host to the item that the browser wants to read.

You can see all these elements in the url that we used to access the hello page in Chapter 1.



1. Figure 2.1 Ch02\_Fig\_01 URL structure

This url specifies that the site uses the secure version of the hypertext transfer protocol, that the address of the host server is “*begintocodewithjavascript.com*” and that the path to the file containing the web page is “*hello.html*”. If you leave off the path the browser will automatically request a file with the path “*index.html*”. Most browsers will

now automatically fill out the “`https://`” when you type in a web address. If the path is omitted from the url the server will send the contents of a file called “`index.html`”, which is called the *index page* of a site.

When the user requests a web site the browser sends a message to the server to request the page. This message is formatted according to the HyperText Transport Protocol (HTTP) and is often called a “get” request (because it starts with the word “GET”). The server then sends a response which includes status code and then, if it is available, the text of the web page itself. If the page cannot be found on the server (perhaps because the url was not given correctly) the HTTP status code results in the familiar “404 page not found” message. We will learn more about this process later in the book when we write some JavaScript code that gets web pages.

## What is HTML?

This is not a guide to HTML. You can buy whole books that do very thorough job of describing the language and how it is used. But you should finish this section with a good understanding of the fundamentals. HTML is a *markup language*. That is what the “M” in HTML stands for. The word “markup” comes from the printing profession. Printers would be given text that had been “marked up” with instructions such as “print this part in large font” and “print this part in italic”.



2. Figure 2.2 Ch02\_fig02 Please Leave Blank

Figure 2.2 above shows what happens if you don’t use a markup language properly. The customer wanted a cake with no writing on it. They said “Please Leave Blank” when asked what they wanted written on the cake. Unfortunately, the baker took this instruction literally. This kind of miss-understanding is impossible with HTML. The language has a rigid separation between the text that is to be displayed and the formatting instructions. In HTML, if I want something to be *emphasized*, I will use an HTML markup command to request this:

```
<em>This text is emphasized.</em> This text is not.
```

The sequence `<em>` is recognized by the browser as meaning “make the text that follows this instruction look slightly different from the other text”. It is called a *tag*. The browser will display emphasized text until it sees the sequence `</em>` which marks the end of the emphasized text. Most browsers emphasize text by displaying it as *italic*. If we viewed the above HTML in a Microsoft Edge we would see something that looks like this:

```
This text is emphasized. This text is not.
```

Once you understand the fundamentals of HTML it you can use it to format text. The HTML below shows a few more tags.

```
This is <em>emphasized</em><br>
This is <i>italic</i><br>
This is <strong>strong</strong><br>
This is <b>bold</b><br>
This is <small>small</small><br>
This is <del>deleted</del><br>
This is <ins>inserted</ins><br>
This is <u>underlined</u><br>
This is <mark>marked</mark><br>
```

#### Ch02-01 Text format tags

The example HTML above uses a tag, `<br>`, which means “take a new line”. The `<br>` tag does not need to be matched by a `</br>` element to “close it off”. This is because it has an immediate effect on the layout, it is not “applied” to any specific items on the page. When I pass this text into a browser, I get the following output.

This is *emphasized*  
This is *italic*  
This is **strong**  
This is **bold**  
This is small  
This is ~~deleted~~  
This is inserted  
This is underlined  
This is marked

#### 3. Ch02\_fig03 HTML text modification

If you look closely at the text in Figure 2 you will notice that some of the requests have similar results. For example, the emphasized and italic formats both produced italic output. The bold, italic and underline tags are regarded as

slightly less useful than the more general ones such as "emphasized" or "strong". The reasoning behind this is that if a display has no way of producing italic characters a request to display something in italic is not going to work.

However, if the display is asked to "emphasize" something it may be able to do this in a different way, perhaps by changing the color of the text. Output produced by HTML is intended to be displayed in a useful way on a huge range of output devices. When you use a markup language you should be thinking about the effect you want to add to a piece of text. You should think "I need to make this stand out; I'll use the 'strong' format" rather than just making the text bold.

You can write the commands using upper or lower case, or any combination. In other words the tags `<em>`, `<EM>` and `<Em>` are all regarded as the same thing by the browser.

## Display symbols

By now you should have a good idea how HTML works. A tag `<blah>` marks the start of something. The sequence `</blah>` marks the end. The tags can be nested, (i.e. placed inside each other).

```
<em>This is emphasized <strong>This is strong and emphasized</strong></em>
```

This HTML would generate:

```
This is emphasized This is strong and emphasized
```

For every start tag (`<blah>`) that marks a formatted area of text there should be a matching end tag (`</blah>`). Most browsers are quite tolerant if you get this wrong, but the display that you get might not be what you want.

The question you are probably asking now is "How I can ever get to display the `<` (less than) and `>` (greater than) symbols in my web pages?" The answer is that HTML uses another character to mark the start of a *symbol entity*. The `&` character marks the start of a symbol. Symbols can be identified by their name:

```
This is a less than: &lt; symbol and this is a greater than &gt; symbol
```

The name of the less than character (`<`) is "lt" and that of greater than (`>`) is "gt". Note that the end of a symbol name is marked by a semi-colon (`;`). If you are now wondering how we display an ampersand (`&`) the answer is that it has the symbol name amp.

```
This is an ampersand: &
```

You can find a handy list of symbols and their names here: <https://dev.w3.org/html5/html-author/charref> Note that when you give a symbol name the case of the names is significant.

```
&Eacute;<br>
&eacute;<br>
```

The HTML above would display the upper (É) and lower (é) case versions of “e acute”. If you like emojis (and who doesn’t) you can add these to your web pages by using a symbol that includes the number of the emoji that you want.

```
Happy face: &#128540;<br>
```

#### *Ch02-02 HTML Symbols*

This will display a happy face.

Happy face: 😊

#### 4. Ch02\_fig04 Happy face

If you want to discover all the numbers that you can use to put emojis in your web pages, take a look here:  
<https://emojiguide.org/>

## Lay out text in paragraphs

We now know how to format text. Next we must consider how we can lay this text out on the page. When HTML text is displayed the original layout of the text input is ignored. In other word, consider the text.

```
Hello  
world  
  
from Rob
```

The layout of this text is a bit of a mess. However, when this text is displayed by a browser you see the following:

```
Hello world from Rob
```

The browser takes in the original text, splits it into words and then displays the words with single spaces between them. Any layout information in the source text is discarded. This is a good idea because the designer of a web page can't make any assumptions about the display that will be used. The same page needs to work on large and small displays, from smartphones to large LCD panels.

We've seen that the `<br>` sequence asks the browser to take a new line during the display of text. Now we are going to consider some more commands that control how text is laid out when it is displayed. The `<p>` and `</p>` commands enclose text that should appear in a paragraph.

```
<p>This is the first paragraph</p>
<p>This is the second paragraph</p>
```

This HTML will display two paragraphs.

```
This is the first paragraph
This is the second paragraph
```

The `<br>` command is not the same as the `<p>` command; it does not space the lines out like a paragraph would.

## Create headings

We can use other tags to mark up text as headings at different levels:

```
<h1>Heading 1</h1>
<h2>Heading 2</h2>
<h3>Heading 3</h3>
<h4>Heading 4</h4>
<p>A normal paragraph</p>
```

We can use these in documents to create headings.

# Heading 1

## Heading 2

### Heading 3

#### Heading 4

A normal paragraph

5. Ch02\_fig05 Headings

You can use headings to create structure in a document.

## Use pre-formatted text

But sometimes you might have something that you have already formatted. In this case you can use the `<pre>` tag to tell the browser not to perform any layout:

```
<pre>
This text
is rendered
exactly how I wrote it.
</pre>
```

The text enclosed by the `<pre>` tags is displayed by the browser without any changes to the formatting.

```
This text
is rendered
exactly how I wrote it.
```

The browser uses a *monospaced* font when displaying pre-formatted text. In a monospaced font all the characters have the same width. Many fonts, including the one used to print this paragraph, are *proportional*. This means that each character has a particular width, for example the "l" character is much smaller than the "m" character. However, for some text, for example ASCII art, it is important that all the characters line up. This logo would not look

correct if it was not displayed with a monospaced font.

```
<p> My Logo</p>
<pre>
| _ \ __ | | _ | \ V \ O | _ _ 
| | \ / _ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ 
| _ &lt; ; ( ) | | _ | | | | | _ \ 
| | \ \ \ / | . / | | | | | | | | | |
</pre>
```

#### Ch02-03 A pre-formatted logo

Note that the ASCII art above contains a “<” character. I’ve had to convert this to a symbol (&lt;) so that it is displayed correctly. This is important. Remember that the browser will not format pre-formatted text, but it still observes the character conventions that you must use to display characters and symbols. You can add tags to the pre-formatted text to make parts of it emphasized. You can even put `<p>` tags inside preformatted blocks of and they might work, but this is not advised because it makes your html *badly formed*.

#### My Logo



6. Ch02\_fig06 My logo

#### Programmer's Point

#### Don't abuse the browser

Browsers are generally very tolerant of badly formatted HTML. The browser will try to display something even if the HTML it receives is badly formatted. This means you can get away with HTML like this:

`<em>Emphasized <strong>strongly emphasised</em> just strong </strong>`

The browser will display what you would expect:

***Emphasized strongly emphasized just strong***

#### Ch02\_Readeraid\_01\_Fig\_01 emphasized text

However, is *malformed HTML*. You might be wondering why. Let's take a look at the sequence of tags in the text.



#### Ch02\_Readeraid\_01\_Fig\_01 Bad Nesting

This is an example of what is called *bad nesting*. This is because the `<em>` tag “ends” inside the `<strong>` tag. In properly formed HTML a tag that is created inside another will end before the enclosing tag ends. The complete sequence of a start tag, text and end tag is called an *element*. While one element can completely contain another, it is not correct for elements to *overlap* like the ones in the above figure. The correct version of this HTML is shown below. Note that each element is complete.



#### Ch02\_Readeraid\_01\_Fig\_02 Good Nesting

The above figure shows what correct nesting looks like. Each element ends before its enclosing one. The reason why I'm stressing this is that most browsers can work out the meaning of the badly nested HTML and display it correctly, but some might not. This could lead to your web pages looking wrong to some people. I'm sure you've had the experience of having to switch browser because a particular web site doesn't look right. Now you know how this can happen.

You can identify incorrectly nested HTML when you see an end tag that doesn't match the most recent start tag. If you want to use a program to make sure that your HTML is correct you can use the official validator site here <https://validator.w3.org/>. You can point the validator at a site you have created or paste HTML text into it for checking.

## Add comments to documents

You can add comments to an HTML document by enclosing the comment text in the sequences `<!--` and `-->` as follows:

```
<!-- Document Version 1.0 created by Rob Miles -->
```

The author credit would not be displayed by the browser, but you could view it in the source code by pressing the F12 key to open the developer view. As we go through this text I'll be telling you regularly how useful it is to add comments to your work, so I think it is a good idea to start doing this now.

## Add images to web pages

For the first few years of its life the World Wide Web didn't have any pictures at all. The image tag was added by Marc Andreessen, one of the authors of Mosaic, the most popular browser in the early days of the web. The image tag contains the name of a file that contains an image:

```

```

The image tag uses an *attribute* to specify the file that contains the image to be displayed. An attribute is given inside the tag as a name and value pair, separated by the equals character. When the browser finds an img tag it looks for the src attribute and then looks for an image file with that name. In the case of the above HTML the browser would look for an image called "seaside.JPG". It would look in the sample place on the server from which it loaded the web page. We must make sure that file exists on the server, otherwise the image will not be displayed.

### What Could Go Wrong? – Beware of faulty filenames

The src attribute in an img tag is followed by the name of the file that is to be fetched from the server. While HTML doesn't care about the capitalization of tags (you can write IMG, img or Img for the tag name) the computer fetching the image file might. Some computers will deliver a file stored as "seaside.jpg" if you ask for one called "seaside.JPG". Others will complain that the file is not available.

I normally encounter this problem when I take a web site off my PC (where it has been working perfectly) and place it on the server (when all the image files suddenly vanish).

You can add another attribute to an img tag that gives alternative text that is displayed if the image cannot be found.

```

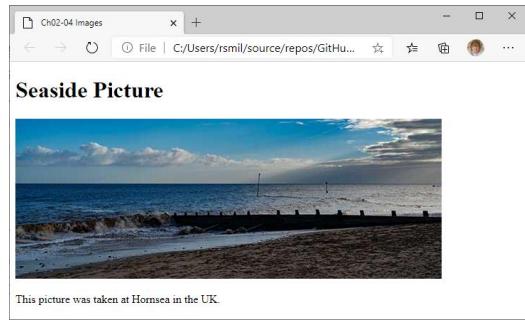
```

Now the browser will display the text "Maybe Rob got the filename wrong" if the image can't be located.

The image will be displayed in line with the text on the page. We can use the HTML layout tags to lay an image out sensibly with the surrounding text.

```
<h1>Seaside Picture</h1>
<p></p>
<p>This picture was taken at Hornsea in the UK.</p>
```

#### Ch02-04 Images



#### 7. Ch02\_fig07 Image

This image is 600 pixels wide. A pixel (short for *picture cell*) is one of the dots that make up the picture. The more pixels that you have the better looking the picture is. However, this can cause problems if the picture is too large to fit on the device being used to display the image. The `image` tag supports `width` and `height` attributes that can be used to set the displayed size of an image. So, if I want to display the image as 400 pixels wide I can do this:

```
<p>
```

Note that I didn't specify the height, in which case the browser will automatically calculate the height that matches a width of 400 pixels. You can specify both height and width if you like, but you need to be careful not to make the pictures distorted. Setting the absolute width of an image using height and width attributes looks like a good idea at first but it can be restricting. Remember that an underlying principle of the World Wide Web is that a page should display in a useful way on any device. An image size of 400 pixels might be fine for a small device, but it will appear very small if viewed on a large TV display. In the next chapter we will discover how we can use stylesheets to allow items on a webpage to be automatically scaled for the target device.

## The HTML document

We now know that we can use tags to mark regions of text as needing to be formatted in a particular way; for example, `<em>` for emphasized text. We can also mark regions of text as being in paragraphs or levels of headings. We can apply several tags to a given piece of text to allow formatting instructions to be layered on top of each other, but we need to make sure that these instructions are properly “nested” inside each other. Now we can consider how to create a properly formatted HTML document. This is comprised of several sections:

```
<!DOCTYPE HTML>1
<html lang="en">2
  <head>3
    <!-- Heading here --!
  </head>4
  <body>5
    <!-- Body text here --!
  </body>6
</html>7
```

The browser looks for the sequence `<!DOCTYPE HTML>` at the start to make sure that it is reading an HTML file. All the HTML that describes the page is given between `<html>` and `</html>` tags. The `</html>` tag contains a `lang` attribute that specifies the language of the page. The language “`en`” is English. The `<head>` and `</head>` tags mark the start and end of the *heading* of the document. The heading contains information about the content of the page including styling information (of which more next chapter). The text in-between the `<body>` and `</body>` tags is what is to be displayed. In other words; everything we have learned up to now goes into the body part of the web page file.

## Linking HTML Documents together

An HTML document can contain elements that link to another document. The other document can be on the same server or it can be on a different server entirely. A link is created by using an “`a`” tag which has an `href` attribute that

---

<sup>1</sup> Indicates that this is an HTML document

<sup>2</sup> HTML tag with a language attribute

<sup>3</sup> Start of the heading of the web page

<sup>4</sup> End of the heading

<sup>5</sup> Start of the body text of the web page

<sup>6</sup> End of the body text

<sup>7</sup> End of the HTML text

contains the url of the destination page.

```
Click on <a href="otherpage.html">this link</a> to open another page.
```

The text in the body of the `<a>` tag is the text that the browser will highlight as the link. In the example HTML above the words "this link" will be the linkable text. This will result in text on the page that looks like this:

```
Click on this Link to open another page.
```

If the reader clicks the link the browser will open a local file, in this case called "`otherpage.html`", which will be displayed. The destination of the link can refer to a page on a completely different site:

```
<p>Click on <a href="https://www.robmiles.com"> this link</a> to go to my blog.</p>
```

#### *Ch02-05 References*

## Making active Web Pages

There are lots of other things that I could tell you about HTML. The language can be used to create numbered and un-numbered lists and tables. However, this is not a book about HTML, it is about programming. What we want is a way of getting JavaScript code to run inside our web page. Then we can start exploring the language.

You already know that a JavaScript program can sit alongside an HTML page design. You saw that in Chapter 1 when you used the Developer View (obtained with F12) in the browser to take look at the hidden program inside the web page `hello.htm`. That HTML file contained a `script` element holding some JavaScript code. We used the console to run a JavaScript function. Now we are going to trigger a function by pressing a button.

### Using a button

```
<button onclick="doSayHello()">Say Hello</button>
```

One way to create an active web page is by using a button. The HTML above creates a button that contains the text "Say Hello". The button is displayed in the normal flow of the text in the page.

## Say Hello

### 8. Ch02\_fig08 Say Hello button

The button has an `onclick` attribute. One of the great things about JavaScript is that most of the time the names make sense. The `onclick` attribute specifies a function that is to be used when the button is clicked. In this case the attribute specifies a JavaScript function called "`doSayHello`". A JavaScript function is a sequence of JavaScript statements that have been given a name. We will take a detailed look at functions in Chapter 8.

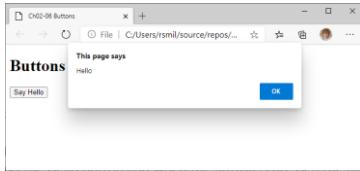
```
function doSayHello() {  
    alert("Hello");  
}
```

This function only performs a single action, it displays an alert that says "Hello" to the user when it is called. The line of JavaScript that displays the alert is called a *statement*. The end of a statement is marked by a semi-colon character. A function can contain many statements, each of which is ended with a ;

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
<title>Ch02-06 Buttons</title>  
</head>  
  
<body>  
<h1>Buttons</h1>  
<p>  
<button onclick="doSayHello()">Say Hello</button>  
</p>  
  
<script>  
    function doSayHello() {  
        alert("Hello");  
    }  
  
</script>  
</body>  
</html>
```

## Ch02-06 Buttons

This is the complete HTML text of the web page. The `<script>` element is at the bottom of the body of the document. The page displays the Say Hello button and when the button is pressed the alert is displayed.



9. Ch02\_fig09 Say Hello alert

## Reading input from a user

```
<input type="text" id="alertText" value="Alert!">
```

The `button` tag lets us create an element in a web page that responds to a user action. Next, we need a way of getting input from a user. The `input` tag lets us do just that. It has three attributes:

The `type` attribute tells the browser the type of input that is being read. In the code above we are reading text, so the attribute `type` is set to `text`. If you set the `type` attribute to `password` the contents of the input are hidden as they are typed. This is how JavaScript programs read passwords in web pages.

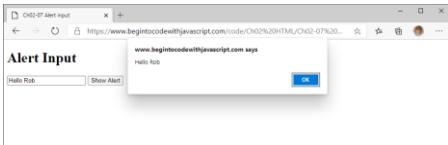
The `id` attribute gives an element a unique name. This name can be used in the JavaScript code to locate the element. If we had two input elements, we would use a different name for each. I've called the element `alertText` because this nicely reflects what the element is being used for.

The `value` attribute specifies the value in this element tag. This is how we can pre-populate an input with text. When this input element is displayed it will have the text "Alert!" in it. If we want the input to be blank when it is displayed we can set the value to an empty string.

```
<p>
  <input type="text" id="alertInputText" value="Alert!">
  <button onclick="doShowAlert()">Show Alert</button>
</p>
<script>
  function doShowAlert() {
```

```
var element = document.getElementById("alertInputText");
alert(element.value);
}
</script>
```

The web page contains the input element, a button element that will call a function to display the alert and the function that uses the input text in an alert. The user can type their own text into the input and then press the Show Alert button to have the text displayed in an alert. Figure 2.10 below shows what the program looks like when it is used.



10. Ch02\_fig10 Customizable alert

#### *Ch02-07 Alert Input*

If you run the example you will notice that when you press the “Show Alert” button the text you have entered in the input area is displayed in the alert.

## HTML and JavaScript

It's worth spending some time discovering how HTML and JavaScript work together, as this underpins almost all of the programs that we are going to write. The JavaScript program needs a way of interacting with the HTML document it is part of. This interaction is provided by *methods* which are part of the *document object*. The document object is a container that holds all the elements on the page. A method is a behavior provided by an object. The document object contains methods alongside the HTML elements that make up the page. Our program uses the `getElementsByID` method to get a reference to the element on the page. It then gets the text out of this element and then displays that text in an alert.

If you're not sure about this, how about an analogy. Think of the HTML document as “Rob's Car Rental”. When someone comes to pick up a car they will say “I've come to pick up car registration 'ABC 123'” and I will hand them the keys and reply “It's over in bay E6”. They can then go and find the car. I don't hand the customer the car over the counter (I'm not strong enough for that). I just tell them where the car is so they can go and find it.

In the case of the HTML document each of the elements in the document is like a car in the parking lot for my rental business. An element can be given an ID just like a car has a registration plate. In our document the ID is `alertInputText`. The method `getElementByID` is the means by which a JavaScript program can ask the document where an element is.

```
var element = document.getElementById("alertInputText");
```

On the right-hand side of the statement above you can see the use of `getElementById` to get the location of the text element with the id `alertInputText`. The left hand side of the statement creates a *variable* to hold this location. The word `var` creates a JavaScript *variable*. A variable is a named location that stores some information that the program wants to remember.

In "Rob's Car Rental" I would offer to write down the location of a car for a customer who was afraid of forgetting where their car was. I'd give them a piece of paper with "Car Location" (the name of the "variable") and "Bay E6" (the value of the variable) written on it. In the JavaScript above the variable we are creating is called `element` (because it refers to an element in the document) and the value is the location of the text input element. This operation is called an *assignment* because the program is assigning a value to a variable. An assignment operation is denoted using the equals (=) character. We will discuss variables in detail in chapter 4. Now that the program has a variable called `element` that contains a reference to the input we can extract the text value from this element and store it in a variable called `message`.

```
var message = element.value;
```

The variable called `message` now contains the text that was typed into the input by the user (remember that we set this to "Alert!" in the HTML). The program can now display this text in an alert.

```
alert(message);
```

It is very important that you understand what is going on here. Up until now everything has seemed quite reasonable, and then suddenly you've been hit with something really complicated. I'm sorry about that. Just go through the code and try to map the statements back to what the program is trying to do. And remember that the equals character means "set this variable to the value". It does not mean that the program is testing to see if one thing is equal to another.

If you are confused about how the various parts of the program fit together, consider that the program is doing exactly the same thing as if I had given a car hire customer the location of their car and then asked them to come back and tell me how much fuel there was in that car. That sequence would go as follows:

4. Get the location of the car.
5. Go to the car.
6. Get the value from the fuel level display.

7. Bring that value back to me.

In the case of the JavaScript program that is displaying the message, the sequence is:

1. Use `getElementById` method to get a reference to the input element.
2. Follow the reference to the element.
3. Get the text value from the input element.
4. Display that text in an Alert.

## CODE ANALYSIS

### The `doShowAlert` function

```
function doShowAlert() {  
    var element = document.getElementById("alertInputText");  
    var message = element.value;  
    alert(message);  
}
```

We can build on our understanding of this important aspect of JavaScript by looking at this function and considering some questions.

What would happen if you got the id of the text element wrong?

The `doShowAlert` method uses an unspoken “contract” between the HTML and the JavaScript code. The `doShowAlert` function asks the `getElementById` to find an element with the id “`alertInputText`”. If this contract is broken because the element has the name “`alertInputText`” the `getElementById` method will not be able to deliver a result. This is a bit like me telling a car rental customer to look for their car in a location that doesn’t exist. In this case the `getElementById` method will return a special value called “`null`” which means “I couldn’t find anything”. This would cause the rest of the `doShowAlert` function to fail. In the case of my car rental customer they would come back and tell me that the location does not exist. In the case of the `doShowAlert` function there would be no error reported to the user, but the alert would not be displayed. Later in this book we’ll look at how you can write code that will test for methods returning results that mean “I couldn’t find what you wanted”.

In JavaScript you tend to have to go hunting for the errors that you make. In some programming languages you are told about errors when they occur. In JavaScript things tend to fail silently, or just do something wrong.

What would happen if the user didn’t type any text into the text area on the web page before pressing the button?

If you look at the HTML at the very start of this section you will see that the value attribute of the text tag is set to “`Alert!`”. If the user doesn’t replace this with their message the word “`Alert!`” will be displayed.

What would happen if I pressed the button several times?

The browser will block any activity on the web page until you clear the alert that is displayed. When you press the

button again the `doShowAlert` function would be called again. It would make two new variables called `element` and `message` and uses them to display the appropriate message text.

What does `var` do?

The word `var` is a command to JavaScript to create a *variable*. The name of the variable follows the word `var`. The variable holds a value that the program wants to make use of. A program can assign values to variables by using `=` to tell JavaScript to perform an assignment.

## Display text output

In the previous section we used a JavaScript program to read data from a web page by getting a reference to an element on the page and then reading information from that element. Displaying text on the screen is a similar process. A JavaScript program can use a reference to an object to change attributes of the element. We are going to write a program that changes the text in a paragraph into a string of text that we have entered. The complete HTML file looks like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Ch02-08 Paragraph Update</title>
</head>

<body>
  <h1>Paragraph Update</h1>
  <p>
    <input type="text" id="inputText" value="">8
    <button onclick="doUpdateParagraph()">Update the Paragraph</button>9
    <p id="outputParagraph"></p>10
  </p>

  <script>
    function doUpdateParagraph() {11
      var inputElement = document.getElementById("inputText");12
```

<sup>8</sup> Input text element

<sup>9</sup> Button that calls `doUpdateParagraph`

<sup>10</sup> Paragraph for the output

<sup>11</sup> Function that updates the paragraph

<sup>12</sup> Gets a reference to the input

```
var outputElement = document.getElementById("outputParagraph");13
var message = inputElement.value;14
outputElement.textContent = message;15
}
</script>
</body>
</html>
```

#### Ch02-08 Paragraph Update

This example is an extension of the previous one. Instead of displaying text using an alert this example sets the `textContent` attribute of a paragraph to the text that the user enters into the dialog box. The fundamental behavior of this program is given in these four lines.

```
var inputElement = document.getElementById("inputText");
var message = inputElement.value;
outputElement.textContent = message;
```

The first two lines set up variables that refer to the input and output elements. The third line gets the message to be displayed and the fourth puts this message onto the web page.



11. Ch02\_fig11 Paragraph Update

### MAKE SOMETHING HAPPEN

<sup>13</sup> Gets a reference to the output

<sup>14</sup> Reads the text from the input

<sup>15</sup> Writes the text into the output

## Work with object properties

You may be wondering what the `textContent` property does and how the program uses it. It might be worth investigating this. We can use the JavaScript console in the Developer Tools to do this. Find the folder on your PC that contains the sample code for the book. (If you haven't downloaded the sample code you can find the instructions in Chapter 01). Find the folder **Ch02 HTML/Ch02-08 Paragraph Update**. Double click the file **index.html** in that folder. This should open your browser and you should see a page that looks like the one in Figure 11 above. Now do the following:

Press F12 to open the Developer Tools view. Select the Console tab. Enter the following JavaScript statement:

```
var outputElement = document.getElementById("outputParagraph")
```

This is the statement in our program that gets a reference to the `outputParagraph` in the document. We now have a variable called `outputElement` that refers to the output paragraph. We can prove this by using our new variable.

```
outputElement.textContent = "fred"
```

Take a look back at the web page. You should see that the word "fred" has appeared. By setting the value of the `innerText` property of the paragraph we can change the text in the paragraph. A JavaScript program can read properties as well as write them. Enter the following statement:

```
alert(outputElement.textContent)
```

This will make an alert box appear with "fred" in it (because that is the `textContent` of the element referred to by `outputElement`). Now let's see what happens if we make a mistake. Try this:

```
outputElement.tetContent="test"
```

This statement looks sensible, but I've miss-typed "`textContent`" as "`tetContent`". The paragraph element does not have a `tetContent` property. However, this statement doesn't cause an error, but the word test is not displayed either. What happens is that JavaScript creates a new property for the `outputElement` variable. The new property is called "`tetContent`" and it is set to the value "test". You can prove this by entering the following:

```
alert(outputElement.tetContent)
```

This will display an alert showing the value in the `tetContent` property, which is the string "test". We will discover more about creating properties in objects in Chapter 7.

See if you can change the web page so that the name is displayed as a heading (`<h1>`) rather than a paragraph. You can use Visual Studio Code to edit the html for the web page. Will you have to change the JavaScript or the HTML?

# Egg Timer

We now know enough to be able to create a properly useful program. We are going to create an egg timer. The user will press a button and then be told when five minutes (the perfect time for a boiled egg) have elapsed. We know how to connect a JavaScript function to an HTML button. The next thing we need to know is how to measure the passage of time. We can do this by using a JavaScript function called `setTimeout`. We have used functions already. The `alert` function accepts a string that it displays. The `setTimeout` function accepts two things: a function that will be called when the timer expires and the length of the timeout. The timeout length is given in thousandths of a second. The statement below will cause the function `doEndTimer` to run one second after `setTimeout` was called.

```
setTimeout(doEndTimer, 1000);
```

Our egg timer will use two functions. One function will run when the user presses a button to start the timer. This function will set a timer that will run the second function after five minutes. The second function will display an alert that indicates that the timer has completed.

```
function doStartTimer() {  
    setTimeout(doEndTimer, 5*60*1000);  
}  
  
function doEndTimer() {  
    alert("Your egg is ready");  
}
```

## Ch02-09 Egg Timer

The `doStartTimer` function is connected to a button so that the user can start the timer. The `doEndTimer` will be called when the timer completes. I've added a calculation that works out the delay value. I want a five minute delay. There are 60 seconds in a minute and a value of 1000 would give me a one second delay. This makes it easier to change the delay. If we want to make a hard-boiled egg that takes seven minutes I just have to change the 5 to a 7. Note that the \* character is used in JavaScript to mean *multiply*. You will find out more about doing calculation in Chapter 4.

MAKE SOMETHING HAPPEN

Investigate the egg timer

```
<!DOCTYPE html>
```

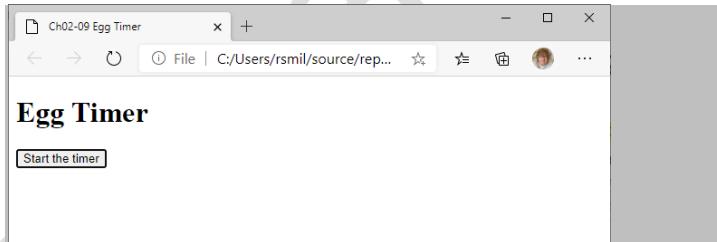
```
<html lang="en">
<head>
  <title>Ch02-09 Egg Timer</title>
</head>

<body>
  <h1>Egg Timer</h1>
  <p>
    <button onclick="doStartTimer()">Start the timer</button>
  </p>

  <script>
    function doStartTimer() {
      setTimeout(doEndTimer, 5*60*1000);
    }

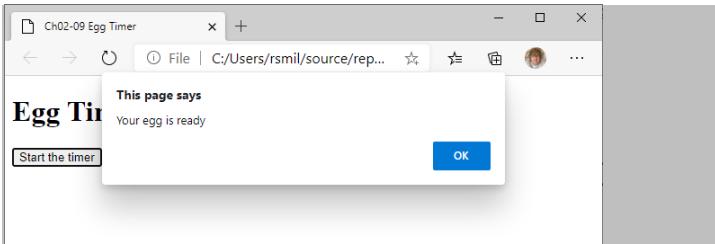
    function doEndTimer() {
      alert("Your egg is ready");
    }
  </script>
</body>
</html>
```

Let's take a look at how the egg timer works. Find the folder **Ch02 HTML/Ch02-09 Egg Timer**. Double click the file **index.html** in that folder to open the page.



Ch02\_Readeraid\_03\_Fig\_01 Egg timer

Click the "Start the timer" button once. This version of the code only has a delay for 10 seconds so after 10 seconds you would see the alert appear.



### Ch02\_Readeraid\_O3\_Fig\_02 Egg timer Ready

Click the “Start the timer” three times in succession. Wait and see what happens. Was this what you expected? It turns out that each time you press the button a new timeout is created. Now press F12 so to open up the Developer Tools. Enter the following and press Enter. What would you expect to see?

```
doEndTimer()
```

This is a call of the function that runs to display the end message. You should see the alert appear telling you that your egg is ready. Click OK in the alert to close it. Enter the following and press enter. What would you expect to happen?

```
setTimeout(doEndTimer, 3*1000)
```

After three seconds the alert appears, because that is the length of the timeout. You should also have seen something else appear when the function runs. You will also see an integer displayed. If you repeat the call of `setTimeout` you will see another value displayed. It is usually one bigger than the previous one. This number is the “id” of timer. This can be used to identify a timeout so that it can be canceled. We are not going to do that, so you can ignore this value.

See if you can change the web page so that it supports multiple cooking times. I’d like buttons for “Soft” (four minutes) “Normal” (five minutes) and “Bullet” (ten minutes). You will have to add two more buttons and two more JavaScript functions to the program.

If you want to see how I did it open up the file in **Ch02-09 Selectable Egg Timer**. My solution even displays the status of the timer.

## Adding sound to the egg timer

Our egg timer works fine, but it would be nice if it could do a little more than just display an alert when our egg is ready. A web page can contain an `audio` element that can be used to play sounds.

```
<audio id="alarmAudio">
  <source src="everythingSound.mp3" type="audio/mpeg">
    Your browser does not support the audio element.
</audio>
```

The `audio` element includes another element called `src` that specifies where the audio data is going to come from. In this case the audio is held in an MP3 file called `everythingSound.mp3` which is held on the server. The text inside the `audio` element is displayed if the browser does not support the `audio` element. I've given this element an `id` so that the code in the `doEndTimer` function can find the `audio` element and ask it to play the mp3 file.

```
function doEndTimer() {
  alarmSoundElement = document.getElementById("alarmAudio");
  alarmSoundElement.play();
}
```

#### Ch02-10 Alarm Egg Timer

This code looks like that in the **Ch02-08 Paragraph Update** example. In that case the `getElementById` method was fetching a paragraph element to be updated. In the function above `getElementById` is fetching an audio element to be played. An audio element provides a `play` method that starts it playing. The rest of the file is exactly the same as the original egg timer. If you try this program you will quite an impressive sound when your egg is ready.

## Controlling Audio Playback

The egg timer page does not display anything to represent the audio element. It is "hidden" inside the HTML. You can modify an `Audio` element so that a player control is shown on the web page. To do this you just have to add the word `control` into the element tag:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Ch02-11 Sound playback</title>
</head>

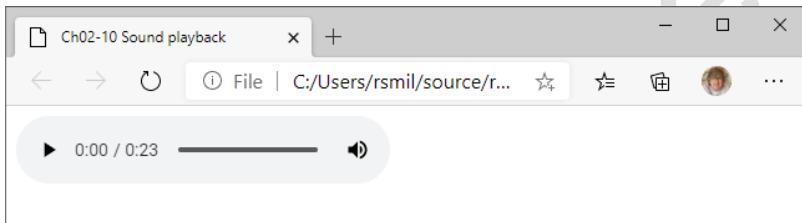
<body>
  <audio controls>
    <source src="everythingSound.mp3" type="audio/mpeg">
    Your browser does not support the audio element.

```

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

#### Ch02-11 Sound playback

This is the complete source of an mp3 file playback page. If you visit the page you will see a simple playback control.



12. Ch02\_fig12 Sound playback

This is how the playback control looks when using the Edge browser. Other browsers will look slightly different, but the fundamental controls will be the same. A viewer of the page can start the playback by pressing the play control at the far left.

## An Image Display Program

The final example program in this chapter shows how JavaScript can change the content of an image displayed on the screen. You can use this technique to implement “slide shows” and also allow the user to select images for display. The image to be updated must be given an id:

```
</p>
```

This `img` element displays the picture in the file `seaside1.JPG`. A JavaScript program can change the displayed image by modifying the `src` attribute of the image and making it refer to a different image file:

```
var pic = document.getElementById("pageImage");  
pic.src="fairground.JPG";
```

These two statements get a reference to the image and then set the src attribute of the img to refer to the image `fairground.jpg`. This will update the image displayed by the browser. Note that this is a repetition of a pattern that you've seen several times now. A program obtains a reference to a display element and then makes changes to it. You can find a complete image picker program in the example **Ch02-12 Image Picker**

## MAKE SOMETHING HAPPEN

### Create your own pages

You now know enough to create your own pages that contain timers, images, and buttons. Here are some ideas for you to think about:

- Make a “mood page”. The page will display buttons labelled, “Happy”, “Sad”, “Worried” etc. When the user presses a button, the page will display an appropriate message and play a piece of appropriate music.
- Make a “fitness” page. Users will press a button to select an exercise length and the page will display exercise instructions and start a timer for that exercise.
- Make a slide show. Users press a button and the page will show a sequence of images. To do this you can use a number of calls to `setTimeout` to trigger picture changes at different times in the future; perhaps one at 2 minutes, one at four minutes and so on.

## What you have learned

This chapter has given you a good understanding of what the world wide web is and how it works. Here are the major points you've covered in this chapter:

- The HTTP (HyperText Transport Protocol) is used by *browsers* to request pages of data from web servers.
- Data arriving at the browser is *formatted* using HTML (HyperText Markup Language) and that a web page contains commands to the browser (for example *emphasize* this word) using tags (for example `<em>`) to mark out *elements* in the text. Elements can contain text, images, audio and pre-formatted text. Elements can also contain links to other web pages which can be local to a page or on distant servers.
- HTML text can contain *symbol* definitions. Symbols include characters such as '<' and '>' which are used to mark tags) and can also be used to incorporate emojis into web pages.
- An HTML document is comprised of a line that identifies the document as HTML, followed by Header and Body elements enclosed in an HTML element. The body of the document can contain a `<script>` element that holds JavaScript code.

- A web page can contain a button element that runs a JavaScript function when the button is activated.
- JavaScript code interacts with the HTML document via a document object containing all of the elements of the page. The document provides methods that a program can call to interact with it. The document method getElementById can be used to obtain a reference to the page element with a particular id.
- A JavaScript program can contain variables. These are named storage locations. A variable can be assigned a value which it will store for later use. The assignment operation is denoted by the equals (=) character.
- A JavaScript function can locate elements in a document by their id attribute and then use element behaviors to change the attributes on the elements. This is how a JavaScript program could update the text in a paragraph or change the source file for an image.
- The setTimeout method can be used to call a JavaScript function at a given time in the future.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions" about JavaScript, computers, programs, and programming.

#### What is the difference between the internet and the world wide web?

The internet is mechanism for connecting large numbers of computers together. The world wide web is just one thing that we can use the internet for. If the internet was a railway the world wide web would be one type of passenger train providing a particular service to customers.

#### What is the difference between HTML and HTTP?

HTTP is the Hyper Text Transport protocol. This is used to structure the conversation between a web browser and a web server. The browser uses HTTP to ask "Get me a page". The server then gives a response, along with the page if it is found. The format of the question and the response is defined by HTTP. The design of the content of the page is expressed using HyperText Markup Language. This tells the browser things like "put a picture here" or "make this part of the text a paragraph".

#### What is a url?

A url is the address of a resource that a browser wants to read. It starts with something that identifies what kind of thing is being requested. If it starts with http it means that the browser would like a web page. The middle part of the url is the network address of the server that holds the web page to be read. The final part of the url is the address on the server of the web page. This is a path to a file. If the address is omitted the server will return the contents of a file called index.html which is called the index page of a web site.

#### What is special about the file index.html?

The index.html file is called the *index page*. It may contain links to other web pages on the same site along with links to pages on other sites on the world wide web.

Where do I put things like image and audio files when I build a web site?

The simplest place to put images and audio files is in the same folder as the web site. So the folder that contains index.html can also contain these images and sounds. However, a path to a resource can include folders, so it is possible to organize a web site so all the images and sound files are held separately from the web pages. We will do this in the next chapter.

Why should I not use pre-formatted text for all my web pages?

The `<pre>..</pre>` element allows page designers to tell the browser that a block of text has already been formatted and that the browser is not to perform any additional layout. This can be useful for displaying such things as program listings which have a fixed format but it does not allow the browser to make any allowance for the target device. One of the fundamentals of web page design is that the browser should be responsible for laying out the page. The page itself should contain hints such as "take a new paragraph here" and allow the browser to sort out the final appearance.

Why should I use `<em>` rather than `<i>`?

The `<i>` (italic) tag means "use italic text". The `<em>` tag means "make this text stand out". If the browser is running on a device that does not support italic text it is much more useful for it to be asked to emphasize text (which it could do by changing color or inverting black and white) rather than select a character type that it is not able to display.

How are HTML tags and elements related?

A tag is the `<p>` marker that denotes that this text is an instruction to the browser rather than something to be displayed on a page. A complete sequence of tags (perhaps with a start and end tag) marks a complete element in a web page.

Does every HTML tag have to have a start `<p>` and an end `</p>` element?

No. Lots of tags do, for example `<p>` marks the start of a paragraph and `</p>` marks the end. But some, for example `<br>` (take a new line) do not.

Can you put one element inside another?

Yes. A paragraph element may contain elements of emphasized text. And an audio element contains an element that identifies the source of the audio to be played.

What is the difference between an attribute and a property?

The HTML source of a web page contains elements with *attributes*. For example `` would create an image element with a `src` attribute set to the image in the file "seaside1.JPG". Within JavaScript the web page the program is part of is represented by a document object that contains a collection of objects. Each object represents one of the elements on the page. Each element object has a *property* which maps onto a particular page attribute. A JavaScript program could change the `src` attribute of an image element to make it display a different picture. In short; attributes are the original values that are set in the HTML and properties are the

representation of these values that can be manipulated in a JavaScript program.

#### What is a reference?

In real life a reference can be something that you follow to get somewhere. In a JavaScript program a reference is used by a program to find a particular object. An object is a collection of data and behaviors which represent something our program is working with. JavaScript uses objects to represent elements on a web page. Each element is represented by an object. A reference is a lump of data that holds the location of a particular object.

#### What is the difference between a function and a method?

JavaScript contains functions which are blocks of JavaScript code that have a name. We have written functions with names like doEndTimer. Methods are functions that are held inside objects. We have used the method getElementById which is provided by the document object.

# 3

# Cascading Style Sheets (CSS)

## What you will learn

In the last two chapters you've learned how to use HTML to design a web page and JavaScript to add some behaviors to it. However, you've probably also noticed that the web pages that you've been creating don't look much like ones on the web pages you usually visit. They are lacking in design and color. In this chapter you'll discover how to manage the appearance of a web page to make it more appealing. Along the way you'll pick up some important points about JavaScript programming and make some nifty applications and games.

## Putting on the style

We can start by considering what style is. Apparently, it is something that you either have or have not got. And I've been told many times that I don't have any. But in the case of web pages design I have a little bit. And I can tell you how to create and apply styles, at which point how a web page looks is entirely down to you. The *style* of an element

on a web page covers such things as the foreground and background color it has, the type of character design (font), size of the text and things like margins. You can think of a style as a lump of data that you want to apply to something. We are going to start by styling some text. Once we've worked out how to apply style to a single element in a document we will move on to consider how we can make it easier for us to change the style of all the elements in a document.

## Splashing some color

We can start by adding a bit of color to a page. The definition of an HTML element on a web page can contain attributes that describe the element. We can add a *style* attribute to an element to set the foreground color for that element.

```
<p>This is an ordinary paragraph</p>
<p style="color:red">This is a red paragraph.</p>
```

Above you can see how style is applied to a single paragraph. The first paragraph is ordinary. The second has been styled with red text.

This is an ordinary paragraph

This is a red paragraph.

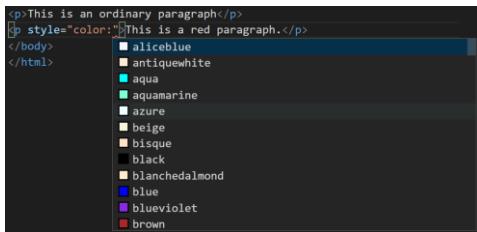
13. Figure 3.1 Styling text

The HTML standard contains a set of color definitions that you can refer to by name. Visual Studio Code will show you a tiny preview of the color when you are editing the text of your code.

```
<p>This is an ordinary paragraph</p>
<p style="color: red">This is a red paragraph.</p>
```

14. Figure 3.2 Visual Studio Color Preview

Visual Studio Code will also produce a color menu when you start typing a color value into a style when you are editing your HTML source.



15. Figure 3.3 Visual Studio Color Selector

There are very many style settings. The interactive help in Visual Studio will show you all the possible settings when you start typing a style command. Style settings can be combined in a single style description by separating them with the semi-colon character. The style below would result in a tasteful display of red text on a yellow background.

```
<p style="color:red; background: yellow;">Red on yellow.</p>
```

#### Ch03-01 Styling HTML Elements

### MAKE SOMETHING HAPPEN

#### Color highlighting on mouse rollover

By now you should be used to a pattern of JavaScript program that works like this:

1. Attach a JavaScript function to an event.
2. When the function runs it gets a reference to an element in the document.
3. The function then changes a property on the element to change the appearance of the document.

We've used this to make a program that respond to button clicks and another one that runs a function after a time interval has elapsed. Now we are going to use exactly same pattern to make web page that highlights text when you roll the mouse over it. You must have seen this used on web pages that you have visited. Take a look at the code below:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Ch03-02 Color Change on Mouse Over</title>
</head>
```

```
<body>
<p onmouseover="doMouseOver()" id="mouseOverPar">Roll your mouse over this paragraph.</p>
<script>
    function doMouseOver()
    {
        var par = document.getElementById("mouseOverPar");
        par.style="color: red";
    }
</script>
</body>
</html>
```

Can you work out which event we are using, and how the function `doMouseOver` changes the color of the text?

The event that we are using is called `onmouseover` and the program is using the `style` property of the paragraph to make it turn red. The browser will call the function `doMouseOver` when it detects that the mouse pointer is over the paragraph text. The function `doMouseOver` obtains a reference to the paragraph and then sets the style of that paragraph to have the color red.

Let's take a look at how the code works in practice. Find the folder [Ch03 HTML/ Ch03-02 Color Change on Mouse Over](#). Double click the file `index.html` in that folder to open the page. You will see a simple page with the message "Roll your mouse over this paragraph." in the top left-hand corner. Roll your mouse over the text and note what happens. You should see the text turn red, which is exactly what you want, but you should notice something else too. Is the behavior what you want? I think you would prefer it if the text turned back to black when the mouse was not over the paragraph.

We have found our first *bug*. A bug in a program is a behavior that you don't want. Finding and fixing bugs is big chunk of software development. In this case the bug came about because we didn't think through what rollover actually means. Perhaps we assumed that the browser would restore the original color of the text when the mouse left the paragraph. But that is not how the program works. How do you think we can fix this?

We can fix it by using another event. It turns out that an element on a web page can generate events when a mouse leaves it as well as when the mouse moves over it. The event that we need to use is called `onmouseout` and we need to connect it to a function that sets the text color of the paragraph back to black. See if you can fix your copy of the program so that it works correctly. Edit the `index.html` file using Visual Studio Code, save it and then test it with the browser.

If you want to see my fixed solution, take a look in [Ch03 HTML/ Ch03-03 Color Change on Mouse Over Working](#). This works, but later in the text we will discover a much easier way to create a rollover effect.

#### PROGRAMMER'S POINT

#### Bugs are a fact of life for a programmer

You are going to have to get used to creating bugs. I've been programming for many years and I still write code with bugs in. And I expect to create many more bugs as I write more programs. The thing to remember is

that creating a bug is fine. Creating a *fault* is not. You get a fault when a customer finds a bug in your program. Programmers spot bugs by testing. Programmers make faults by not testing enough, so that bugs make it into the final product. Whenever you make something you need to work out how to test it. In the case of our “paragraph highlight” program the testing is obvious – just move the mouse over and see what happens. When we make some bigger programs, we’ll discover that testing can be more complex.

## Work with fonts

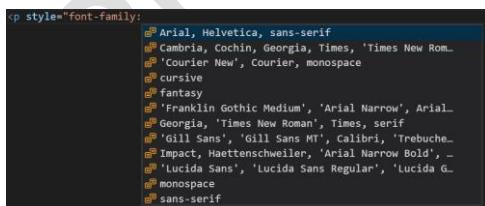
Fonts are one of many things about computers where computer manufacturers have “agreed to differ”. They all agree that they need a font that looks like this (this is called a *serif* font because it has rounded corners) and a font that looks like this (this is called a *sans-serif* font because it doesn’t have rounded corners and “sans” is French for “without”). However, the manufacturers have not agreed on the names for them. For example, the “serif” font is called “Times New Roman” on Windows PCs and “Times” on Apple Macs.

### Select a text font

This means that when we specify the font to use in a web page, we can’t request a specific font because we don’t know what type of computer is being used to view the page. Instead we will specify a *font family*. This is specified in the style for an element. We can specify a list of fonts that we would like to use, and the browser will work through them in order looking for a font that it can use:

```
<p style="font-family:Arial, Helvetica, sans-serif">This is in sans-serif font</p>
```

The style above asks for **Arial**, followed by **Helvetica** and finally **sans-serif**. Most computers have a sans-serif font, so the last entry in the list acts as a “catch all”. The first item on the list is a specific font, the last item on the list will be a more abstract font-type. If I were picking vegetables to have for dinner the first item would be “chips” (a specific dish) and the last would be “potatoes” (a general catch-all for the particular vegetable I want). Font selection works in the same way. Note that when I select a font family I get the designs for bold and italic versions of the characters in that font too.



16. Figure 3.4 Visual Studio Code Font Selector

We have already seen how helpful Visual Studio Code is when selecting style colors. It also pops up suggestions for

font families which have been arranged in a sensible way. The suggested families give a good range of typefaces that you can use in your pages, as shown in Figure 5 below. You can find the HTML code that generates this page in the example pages at [Ch03 HTML/ Ch03-04 Fonts in JavaScript](#).

This is standard text.

Arial, Helvetica, sans-serif

Cambria, Cochin, Georgia, Times, Times New Roman, serif

cursive

fantasy

**Franklin Gothic Medium, Arial Narrow, Arial, sans-serif**

Georgia, Times New Roman, Times, serif

Gill Sans, Gill Sans MT, Calibri, Trebuchet MS, sans-serif

**Impact, Haettenschweiler, Arial Narrow Bold, sans-serif**

Lucida Sans, Lucida Sans Regular, Lucida Grande

monospace

17. Figure 3.5 Font examples

## WHAT COULD GO WRONG

### Fonts can be a minefield

I must admit that I find it difficult to spot the difference between the Cambria and Georgia fonts in Figure 3.5 above. But some people can, and they may have strong opinions. Make sure that you agree with your customer about the fonts that you are going to use. I don't use many fonts in my pages. I usually use one "serif" font for headings and a "sans-serif" for normal text (or vice versa). Just because you can use lots of fonts in a page doesn't mean that you should.

You should make sure that if a font name contains spaces (for example 'Times New Roman') you should enclose this name in single quotes in the font family setting. You should also note from Figure 3.5 above that different fonts have quite different sizes, which can affect your page layout.

## Select a font size

As I write this book, I'm not really worrying too much about the particular font size of the text. I know that the headings must be larger than "normal" text but I don't concern myself too much with specific dimensions. When designing web pages you should take a similar approach. In other words; if you want to display large text in a heading, select the h1 format for the heading, don't change the size of the font in the heading.

If you want to specify the size of text in a web pages there are a number of units you can use. You can express the size in inches, cm, pixels, points or as a percentage of the size of the display. But I would advise you to use the unit *em*. An em value of 1 means "normal" sized text. An em value of 0.5 would mean half normal sized. This makes all the font sizes *relative* rather than *absolute*. This is usually what you want. As the creator of a page you want to make

sure that the text will readable on all devices. Setting an absolute size would make text that looked perfect on one device and wrong on every other.

If you want text twice as big as the normal size for that font you ask for "2 em" and so on. If you want smaller text, we use a value of em which is smaller than 1. You set the size of the text on the screen by setting a `font-size` in a style:

```
<p>This is normal text.</p>
<p style="font-size:1em">This is 1 em.</p>
<p style="font-size:2em">This is 2 em.</p>
<p style="font-size:0.5em">This is 0.5 em.</p>
```

You can find the HTML code that shows these examples in [Ch03 HTML/Ch03-05 Font Sizes](#)

This is normal text.

This is 1 em.

This is 2 em.

This is 0.5 em.

#### 18. Figure 3.6 Font sizes

Figure 3.6 above shows the different sizes in use. Note that 1 em text is the same size as "normal" text, which is just what you would expect.

## Text alignment

```
<p style="text-align: left;">This text is aligned at the left hand margin of the page and the words  
will wrap with a ragged edge.  
This is the normal format of text</p>
<p style="text-align: center;">This text is aligned in the center.  
Useful for headings and quotations.</p>
<p style="text-align: right;">This text is aligned at the right margin of the page.</p>
<p style="text-align:justify;">This text is aligned at the left and right margins of the page.  
This makes the text look like the pages of a book or a column in a newspaper.</p>
```

You can also add an element to a style command to tell the browser how to lay out the text. By default (i.e. unless you specify otherwise) your text will be laid out with each line starting at the left-hand margin. You can add a `text-align` setting to a style as shown.

This text is aligned at the left hand margin of the page and the words will wrap with a ragged edge. This is the normal format of text

This text is aligned in the center. Useful for headings and quotations.

This text is aligned at the right margin of the page.

This text is aligned at the left and right margins of the page. This makes the text look like the pages of a book or a column in a newspaper.

19. Figure 3.7 Text alignment

The text in Figure 3.7 above shows how the sample HTML is laid out.

## Make a ticking clock

We can use our ability to display large text on the screen to create a large ticking clock. However, to do this we need to know how a JavaScript program can obtain the current time (the value of hours, minutes and seconds to be displayed by the clock). We also need to create a program that runs every second to update the clock display.

### MAKE SOMETHING HAPPEN

#### Get the time for display

The programs that we have created so far have interacted with the `document` object which represents the HTML that makes up the web page. To get the time we need to create another type of object, the `Date` object. Let's see how this works. Find the folder **Ch03 HTML/ Ch03-07 Clock Display**. Double click the file `index.html` in that folder to open the page. You should see a clock displaying "0:0:0". We are going to get the time and display it on the clock. Press F12 to open the Developer Tools view and move to the console window. Enter the following JavaScript statement:

```
var currentDate = new Date()
```

We've seen `var` before, it is how a program creates variables. The variable that is created is called `currentDate` and it is made to refer to a newly created `Date` object. The word here `new` means "make a new object". We've not had to create any objects before, our programs have used objects that already exist when the program runs.

The `Date` object is provided to allow JavaScript programs to work with dates and times.

When a new `Date` object is created it is set with the current date and time. The variable `currentDate` is a reference that refers to the newly created `Date` object. We've used references before, when we created a reference to a paragraph element in a web page that we wanted to update. Take a look at example **Ch02-08 Paragraph Update** if you need to refresh your understanding of this. We can ask an object questions by calling methods on the reference to it. Type in the following:

```
alert(currentDate.getHours())
```

This statement uses the `getHours` method which is part of a `Date` object. This method returns a number that contains the hours value of the date. The statement displays this in an alert. There are also methods to get the minutes and seconds values. They have sensible names. See if you can use them to display these time values as well. Note that you must put the open and close brackets () after each method name so that JavaScript knows you want to call the method. We will find out more about making method calls in Chapter 8.

Now type in the following statements to set variables with the hours, minute and second values that we need for the clock:

```
var hours = currentDate.getHours()  
var mins = currentDate.getMinutes()  
var secs = currentDate.getSeconds()
```

We now have the hours, mins and secs values that we can use to build up a string to display as the time. The string will contain the time values separated by the ":" character. Type in the following statement to create the time string.

```
var timeString = hours+":"+mins+":"+secs
```

This statement creates a string which contains the time that we want to display. The values in the hours, mins and secs variables are converted into text. You can view this string using an alert:

```
alert(timeString)
```

If you check the time that appears in the alert you will find that it is out of date because it will have taken you a few seconds to type all these statements. The value in `currentDate` is a *snapshot* of the date and time. This will not be a problem in our clock program because the time values will be displayed immediately after they have been read. We now know how to get the time value into a string ready for display. The HTML for the clock display program contains a paragraph which has the id `timePar`. Type in the following statements:

```
var outputElement = document.getElementById("timePar")  
outputElement.textContent=timeString
```

We've used this pattern of statements before. The first one creates a variable that refers to the output element we want to use to display the time. The second statement sets the `textContent` property of that element to the contents of the `timeString` variable. This will cause the time to be displayed. The HTML for this page contains a function that does everything we have just typed in. Type in the following statement to call this function:

```
doClockTick()
```

You should see the date and time update to show the current date and time. Press F12 to close the Developer Tools in the browser.

## Create a ticking clock

We now know how to create a string that contains the current time and then display that string in large text. Now we need a way to make the clock “tick”. When we created the egg timer programs in Chapter 2 we used a function called `setTimeout` to call a JavaScript function after a specified timeout value. Another useful function in JavaScript is called `setInterval`. This calls a function at regular intervals. As with the `setTimeout` function, the interval is specified in milliseconds.

```
setInterval(doClockTick,1000);
```

The statement above would cause a method called `doClockTick` to be called every second. This is the function that will update our clock. The final piece of our application is a way of starting the interval timer. We could ask the user to press a button to start the clock (this is how the egg timer works) but it would be best if the clock starts ticking as soon as the page is opened. There is an event called `onload` which can be made to call a function when an element on a web page is loaded. You can find the HTML code below in **Ch03 HTML/ Ch03-08 Ticking Clock**. It displays a three digit clock.



20. Figure 3.8 Ticking Clock

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Ch03-08 Ticking Clock</title>
</head>
```

```
<body onload="doStartClock()"16
<p id="timePar" style="font-size:10em;font-family: 'Courier New', Courier, monospace;
text-align: center;">>00:00:00</p>17
<script>
    function doStartClock()18
    {
        setInterval(doClockTick,1000);
    }

    function doClockTick()20
    {
        var timeString = hours+":"+mins+":"+secs;

        var outputElement = document.getElementById("timePar");
        outputElement.textContent=timeString;
    }
</script>
</body>
</html>
```

## CODE ANALYSIS

### Ticking clock

You may have some questions about this code. And if you haven't, I have. It's important that you understand the answers as they underpin some important aspects of JavaScript programming.

What is the difference between `var` and `new`?

The commands `var` and `new` look very similar, in that they seem to be associated with the creation of something, but you may be confused about exactly what is going on. In the case of `var`, a program is creating new variable:

---

Δ

<sup>16</sup> Calls `doStartClock` when the page is loaded.

<sup>17</sup> This is the paragraph that contains the clock display

<sup>18</sup> Function that starts the clock

<sup>19</sup> Call `doClockTick` every second

<sup>20</sup> Update the clock with the current time

```
var age = 21;
```

This would create a variable called `age` which is set to the number 21 (which is a bit optimistic in my case).

```
var outputElement = document.getElementById("timePar");
```

This would create a variable called `outputElement` which is set to the result delivered by the `getElementsByID` from the `document` object. You can refresh your understanding of how this works by reading the section "HTML and JavaScript" in chapter 2. So, every time we want to create a variable (i.e. a named location into which we can store something) we use `var`. In the next chapter we will consider variables in much more detail.

A program uses `new` when it wants to make a new object. The word `new` is followed by the name of the type of object that is to be created:

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

This creates a new variable called `currentDate` (that's what `var` does) and then sets this variable to refer to a `Date` object created using `new`. So, `var` is used to create variables and `new` is used to create objects.

Why do I have to put semi-colons (;) after each statement in a program, but not when I use the console?

When you type JavaScript statements into the console you end each one by pressing the Enter key. This means that the JavaScript console knows when you have finished typing a command, because you have pressed enter. However, statements in a JavaScript program held in an HTML page can span several lines of the page, so you must put in the semi-colons after each statement so that JavaScript can see when one statement ends and another begins.

Why is one function name enclosed in quotes, whereas the other is not?

The clock program uses two functions. One is called `doStartClock` and starts the clock running. The other function is called `doClockTick` and updates the clock every second. The function `doStartClock` is called when the page is loaded by using the `onload` attribute of the `body` element:

```
<body onload="doStartClock()">
```

The `doClockTick` function is called every second using the `setInterval` function:

```
setInterval(doClockTick, 1000);
```

You may be wondering why the name of `doStartClock` is enclosed in double quotes, whereas `doClockTick` is not? At least, I hope you are, because appreciating the distinction will help you a lot in understanding how JavaScript and HTML work together. In the case of the `onload` event in the HTML, the action to be performed is a string containing JavaScript statements that are obeyed when the element is loaded. The string that we are using calls the `doStartClock` method, but it could be any sequence of JavaScript statements:

```
<body onLoad="var x=99;alert(x);">
```

This is completely legal HTML. The JavaScript that is performed on loading creates a variable called `x`, sets the

value to 99 and then displays an alert showing this value. However, the `setInterval` function is given a reference to a function to be called every second, not a string containing some JavaScript code. You may be wondering why the two things work in different ways. This is because the `onload` event is part of an HTML element, whereas the `setInterval` function is called from within the JavaScript program code.

#### Why does the time take a second to appear when the clock starts?

If you load up the example page you will find that it takes a second for the time to appear. For a second after the page has loaded the time is displayed as "0:0:0". If you think about it, this is exactly how the program works. The `setInterval` function calls the function `doClockTick` at one second intervals, but this means that program must wait for this interval to elapse before the clock is displayed. Can you think of a way to solve this problem?

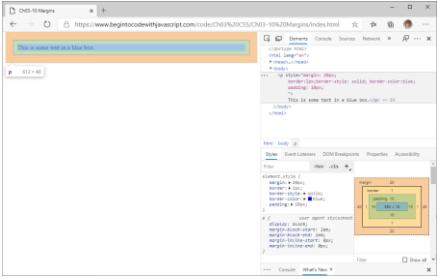
The solution is simple. The program must call `doClockTick` from the `doStartClock` function. Since the `doStartClock` method is called when the page is loaded, this will cause the display to be updated at the start. You can find my version of this in the folder **Ch03 HTML/Ch03-09 Clock Quick Start**.

## Margins around text

Text on a printed page does not extend right to the edge of the paper. This book has *margins* around the paragraphs. Some paragraphs (for example the "Code Explained" section above) have different margins from the rest of the text. This makes the paragraphs stand out. A style can express the size of margins around a paragraph. It can also describe a border for a paragraph. The paragraph above has an outer margin, a border and then "padding" around the text inside the border.

```
<p style="margin: 20px;  
padding: 10px;  
border:1px; border-style: solid; border-color:blue;">  
This is some text in a blue box </p>
```

The dimensions of the margins and the border are expressed in units we have not seen before they are called "px", which is short for *pixel*

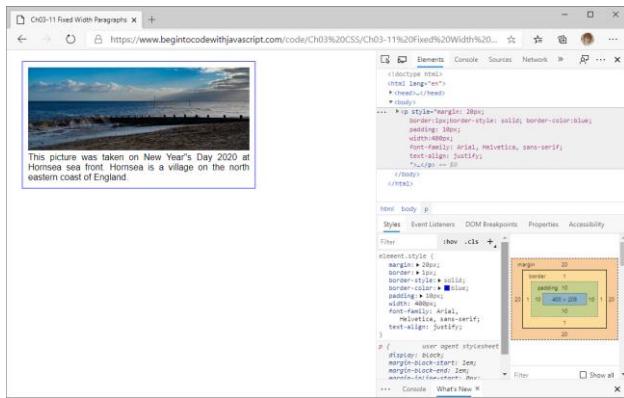


21. Figure 3.9 Margin Display

I could have spent some time drawing a diagram to show how the margin, border and padding values are used to control the layout of text on a page, but it turns out that the Edge browser will do this for me. In Figure 3.8 above you can see the developer view of the HTML above, which you can find at page [Ch03 HTML/ Ch03-10 Margins](#). The diagram on the bottom right shows how the margin, border and padding elements all fit inside each other.

I've specified the margin, border and padding dimensions in a unit we have not seen before. The px unit (short for *pixels*) is an *absolute* unit that equates to a single dot on the target display. We have used these units before, when we specified the size of an image to be drawn on the screen. If you are concerned with precise layout you can use these to lay out text and graphics exactly. This size matches that used to set the size of images, so you can combine text and graphics to make some very displays. The HTML below, which you can find at page [Ch03 HTML/ Ch03-11 Fixed Width Paragraphs](#), puts an image and a descriptive paragraph inside a blue box. This

```
<p style="margin: 20px;  
border:1px;  
border-style: solid;  
border-color:blue;  
padding: 10px;  
width:400px;  
font-family: Arial, Helvetica, sans-serif;  
text-align: justify;  
">  
  
This picture was taken on New Year's Day 2020 at Hornsea sea front.  
Hornsea is a village on the north eastern coast of England.</p>
```



22. Figure 3.10 Text and Graphics

There are lots more things you can do with styles. The best way to find out about them is to use the pop-up help in Visual Studio Code to get ideas for command options and then try them out.

## Making a stylesheet

An HTML document can use the `style` attribute to add style to any element in a document. However, it seems like hard work to have to add style elements to everything. Fortunately, HTML has a way that simplifies applying style to a document. We can add a *stylesheet* to an HTML document to apply styles to elements in the document. The stylesheet is added to the head of the document in between the `<style>` and `</style>` tags as shown below.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Ch03-12 Changing styles</title>
  <style>21
    p {22
      color: blue;
      font-family: Arial, Helvetica, sans-serif;
    }
  </style>
</head>
<body>
  This picture was taken on New Year's Day 2020 at
  Hornsea sea front. Hornsea is a village on the north
  eastern coast of England.
</body>
</html>
```

<sup>21</sup> Style element

<sup>22</sup> Selector for the p style

```
        }23  
    </style>  
</head>  
  
<body>  
    <p>  
        This is a modified paragraph.</p>  
</body>  
  
</html>
```

We can use a stylesheet to set style properties on elements. A style setting starts with a *selector* which specifies the element that is being styled. We want to style the *p* element, so that is what we specify in the stylesheet above. The changes the stylesheet makes result in all *p* elements in the document being displayed in blue text using the Arial, Helvetica or sans-serif fonts. A stylesheet can provide style settings for many elements.

## Creating a stylesheet file

When a web site is created the designers usually come up with a standard “house style” which is to be applied to all the pages. The house style settings could be included in the *<head>* section of the each page as shown in the sample above, but this would make it hard to change the style of the site because each document would need to be edited. To make this easier the style settings can be stored in a separate file. A *<link>* element in the HTML header then specifies the stylesheet file to be added.

```
<head>  
    <title>Ch03-13 Stylesheet File</title>  
    <link rel="stylesheet" href="styles.css">  
</head>
```

The HTML above shows how this works. The *link* element contains a *rel* attribute that tells the browser the type of resource that the link relates to. In this case the link relates to a *stylesheet*. The link to the file containing the style information is specified in the *href* attribute. In the case of the HTML page above this is a local file called *styles.css* which is held in the same folder as the html page. However, this file could be in a different folder, or even on another server. The actual stylesheet file contains the style instructions:

---

<sup>23</sup> End of settings for *p*

```
p {  
    color: blue;  
    font-family: Arial, Helvetica, sans-serif;  
}
```

## PROGRAMMER'S POINT

### It is a good idea to separate style from layout

A web page is made up of layout (what is on the page and where it is) and style (how the page looks). It is very sensible to separate these elements. I could have filled this entire book with a description of how to style web pages. But it would not be a very good read because I'm not very good at design. I'd much rather find a designer who is good with fonts and colors and ask them to sort those things out. Being able to put the styling information into a separate file means that I can design the layout and the behavior of the application entirely apart from the style.

Computer scientists talk about “separation of concerns” in a project, where different people work on different parts. At the start of the project everyone agrees how the different components will work together and then they can work on just their parts. In the case of HTML and style, I would tell the designer I was using p, h1 and h2 for different levels of text and then they could work on the look of these styles. Later when we start writing larger JavaScript programs we will discover a way of separating the program code from the HTML, which allows another level of separation.

## Creating style classes

If we are making a simple web application, we might be able to express all our formatting requirements using the paragraph and heading styles. However, a more complex one would need to contain other styles. For example, we may want to have a different format for displaying an address. It might need to be red, and in a monospaced font and aligned to the right-hand margin. We can add a new style *class* called address to the stylesheet for a document:

```
.address {  
    color: red;  
    font-family: 'Courier New', Courier, monospace;  
    text-align: right;  
}
```

Notice that this looks like the way that we modified the styling of p, except that the name has a leading dot “.” to indicate that this is a newly created style. We can then specify that this class provides the style of a paragraph:

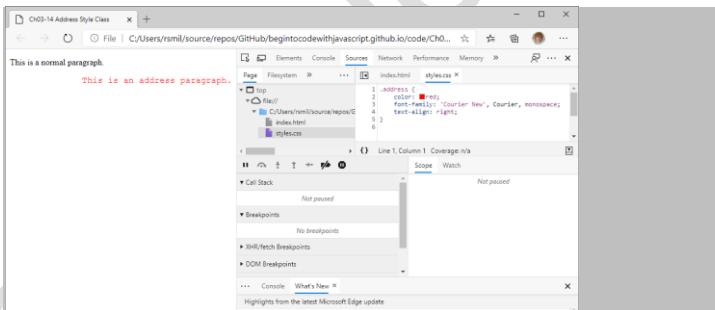
```
<p class="address"> This is an address paragraph.</p>
```

The `class` attribute of an element specifies a CSS (Cascading Style Sheet) style that is to be used to style the element. This means that the text in the paragraph above would be drawn in a red, monospaced font at the right-hand side of the page. You can see this in action by [Ch03 HTML/ Ch03-14 Address Style](#). You can create as many styles as you need. If you were working with a designer you would both agree on the style names to be used for the various elements and then you could generate HTML files elements tagged with the classes that should be used to format them.

## MAKE SOMETHING HAPPEN

### Exploring stylesheets

We can use the Developer Menu in the browser to take a look at how the stylesheets work. Find the folder **Ch03 HTML/Ch03-14 Address Style Class** and open index.html with your browser. This shows you the address paragraph formatted with a red monospaced font. Press **F12** to open the Developer menu. Then select the **Sources** tab in the dialog at the top of the developer menu.



Ch03\_Readeraid\_01\_Fig\_01 Viewing stylesheets in the browser

You can now see the `index.html` and `style.css` files. If you click in the files you can view their contents. When we start making more complex web pages you'll find this view very useful. You can use it to view the sources of any web page you visit, but don't get discouraged by how complicated they seem.

## Formatting parts of a document using div and span

We can add a class attribute to any element of text that we want to format, but sometimes it is useful to apply a style class to multiple items. In the case of the `address` style above we would like to mark all the paragraphs in the address that we are displaying. We could mark each paragraph individually, but it would be nice if we could mark

them all at once. The `<div>` and `<span>` elements make this possible. They are used to create regions to which classes can be assigned.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Ch03-14 Address Style</title>
  <link rel="stylesheet" href="styles.css">
</head>

<body>
  <div class="address">
    <p>Rob Miles</p>
    <p>18 Pussycat Mews</p>
    <p>London</p>
    <p>NE14 10S</p>
  </div>
</body>

</html>
```

The HTML above would format all the paragraphs of the address using the address class because they are enclosed in a division. The `div` element specifies a *division* in the document. Each `div` element marks the start and end of a paragraph of text. This means that we can't use a `div` element to set the style of some words in a paragraph. If we want to just format part of a paragraph using a particular style we can use the `span` element.

```
<p>
An HTML document can use the <span class="codeInText">style</span> attribute to add style to
any element in a document.
</p>
```

In the example above I want the word `style` to stand out from the text as it is an HTML element name. I've created a class called `codeInText` and used it to apply a style to the word `style` in the sentence. I can't use `div` in this situation because this would break the sentence over several lines. The example **Ch03 HTML/ Ch03-16 Code Style with span** shows how it works. You can put `span` and `div` elements inside each other but it doesn't make much sense to put a `div` inside a `span` because this would cause paragraph breaks in your line of text. You must of course make sure that any "nesting" is properly formatted. Take a look at the Programmer's point "Don't abuse the browser" in Chapter 2 for more details on this issue.

# “Cascading” styles

You may be wondering why they are called *cascading* style sheets. The name refers to the way that styles “cascade” down from one element to all the ones that it encloses. Consider the following HTML:

```
<!DOCTYPE html>
<html lang="en">
<head>
<title>Ch03-17 Cascading Styles</title>
</head>
<body style="color: blue;">
<p>This is an ordinary paragraph.</p>
<p style="color:red">This is a red paragraph.</p>
<p style="background: yellow;">This has a yellow background.</p>
</body>
</html>
```

The document contains three style elements. The first is applied to the body of the document. The next two are applied to elements inside the body of the text. The `body` element has been styled with the color blue. The `body` element encloses the two paragraph elements so this setting *cascades* down onto these paragraphs as shown below.

This is an ordinary paragraph.

This is a red paragraph.

This has a yellow background.

23. Figure 3.11 Cascading styles in action

The “ordinary” paragraph has blue text because of the style applied to the enclosing body element. The red paragraph is red because the color setting in the style for the paragraph *overrides* the cascading style. However, the text with a yellow background has blue text because the style on this paragraph does not modify the color in the enclosing one, instead it modifies the color of the text background. The rule is that style settings are “inherited” from enclosing elements unless they are “overridden” in them.

PROGRAMMER'S POINT

Managing styles is all about design

As you go through this book I want to you learn that a lot of solution development is about making your life easier. For example, you might have a problem if you showed your customer the solution you had built and they tell you they want shocking pink text on an orange background (it has been known). If you'd applied the `text` style to each individual HTML element this would mean you would have to work through the entire document and make the requested changes. If you've used stylesheets properly you should be able to change just one file to make a change like this.

A huge amount of programming is about organization and planning. This chapter has introduced some excellent tools that you can use to create and manage the appearance of an HTML page.

## Using selectors

Earlier in this chapter, in the section "Color highlighting on mouse rollover" we used the `onmouseover` event to trigger a JavaScript function to change the color of text in a paragraph. At the time I said that there is a much easier way to make text highlighted when the user rolls their mouse over it.

```
.rollover {  
    color:black;  
}  
  
.rollover:hover {  
    color:red;  
}
```

The stylesheet above creates a style called `rollover`. There are two definitions for the style. The first definition just sets a color of black. The second definition of `rollover` has an additional `selector` which is `hover`. A selector indicates the circumstance in which this variant of the style should be used. The `hover` selector specifies that this style is to be used when the mouse is hovering over an element with the class set to `rollover`. This style sets the color to red. If you view this page at [Ch03 CSS\Ch03-18 Rollover](#) with CSS you will see how it works.

There is a lot more you can do with selectors. You can use them to set styles for unvisited links, selected items, even the first letter of a paragraph. You can find a full reference here. You will have to work hard to understand all of it (I did) but it is powerful stuff: [https://developer.mozilla.org/en-US/docs/Web/CSS/CSS\\_Selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Selectors)

## What you have learned

This chapter has given you a good understanding of what style is and how to manage it. Here are the major points you have covered in the chapter:

- The definition of an element in an HTML page can include a `style` attribute that describes how the element should be displayed, for example to make the text in a paragraph red.

- The definition of an element can also include event attributes that execute a piece of JavaScript code when a particular event occurs, for example the `onmouseover` event triggers code when the user moves the mouse pointer over the element.
- The style information for a text element can include the font to be used to draw the text. Fonts are specified as *families* which include all the variants of the text to be displayed (for example *italic* and **bold**). It is conventional to provide a number of different `font-family` values when specifying a font options because systems use different names for popular font designs.
- The size of elements in an HTML document can be specified in multiple ways. If the intention is to change the size of text relative to other text the size should be expressed in `em` units, where an `em` value of 1 is “normal sized” text.
- Text can be aligned across an HTML page using the `text-align` attribute.
- The `setInterval` function can be used to call a JavaScript function at regular intervals. We used this to create a ticking clock.
- The style of an element can include definitions for margin, padding and border items. The margin is the outer margin around the element. The border can be drawn in a variety of styles and colors and the border has a set thickness. The padding value gives the amount to inset the item inside the bordered area.
- When specifying the dimensions of borders and margins around HTML elements the `px` unit can be used. A value expressed in `px` units represents a number of *pixels* and equates to the size of a pixel in an image. Using `px` units means that you can get absolute control over the layout of items at the expense of portability (i.e. the pages may look too large or too small on some devices).
- A stylesheet contains style settings that can be assigned to elements in an HTML document. Style settings can modify styles such as `p`, `h1` and `h2` or create completely new styles which can be assigned to elements using the `class` attribute.
- A stylesheet can be held in a separate file from an HTML page. The page would contain a `link` element in the page heading which identifies the stylesheet file to be used. This allows complete separation between the content and the styling of a page.
- Styles are applied to elements in a way that cascades down from enclosing elements. For example, if the body of a document is styled with red text that setting will cascade down into any elements in the document. However, a color style attribute in a paragraph inside the document will override this cascading setting.
- The `div` and `span` elements act as containers around other elements. Style attributes applied to the `div` and `span` elements cascade down into all the elements contained in them. When a `div` element is rendered the browser will insert line breaks at the start and the end of the `div`. This does not happen with a `span` element, making the `span` element useful for applying styles to words in sentences.

To reinforce your understanding of this chapter, you might want to consider the following "profound questions" about styles and stylesheets.

Can I add a style attribute to any element?

Yes. It might not be sensible to set the font of the script element in an HTML file, but it will not cause an error if you do this.

What happens if I add an irrelevant style to an item?

Nothing. If you want to set a font for the <script> part of your HTML file you can do and you will not get any errors. But it would not be a sensible thing to do.

What happens if I set conflicting style settings?

The most "recently" applied style will be the one that is enforced. In other words, if the text color of the body of the document is set to black, but the text color of the paragraph in the body is set to red, the text will be red. This is how "cascading" works.

Is a CSS file a program?

No. The JavaScript programs that we have written set out a sequence of actions to be performed. A CSS file contains a number of style settings items which are applied to HTML elements that are assigned to them.

Is redText a good name for a CSS class?

I don't think so. If your customer decides that they really want to change to blue text for that style you can't easily change the name of the class. The name of a class should reflect what you want a style to achieve, for example you could have a class called "displayName" which was used to display names and another called "displayAddress" for addresses. If a style is being displayed as red that's something that would be reflected in the settings for that style class, not in the name of the class itself.

Can I store CSS files on a different server from the HTML file that contains the web page?

Yes you can. The link element that gets the CSS file contains a url that can refer to a file on a distant machine.

When should I use div, and when should I use span?

Div is used if you want to control the style of some elements that make up a *division* in your page. By division I mean something like an entire address, or order, or report. The division will be separated from the rest of the page by line breaks. Span is used when you want to style a small portion of a paragraph, for example to highlight the word `code` in this sentence. In this case you don't want any line breaks in the text.

What is the difference between id and class?

An element on a web page can have a setting for an id value and a class value. The id value is unique for that

element in the html document. JavaScript code working with the document can locate an element by its id. The class value specifies the style to be applied to this element. A large number of elements in a document can have the same class value. This would make changing the style of these elements very easy, because the designer would just have to change the definition of the class in the stylesheet and it would be applied to all the elements.

#### What is the difference between the em and px units?

This is confusing. Not least because `em` also means “emphasized” when applied to text. An `em` value of 1 refers to the size of a “normal” character in the current font. So I can specify a size in `em` values so that I can make my text larger or smaller relative to the rest of the text. I don’t want to use an absolute value for my text size because, as we have seen, different fonts have different “standard” text sizes and I don’t want everything to become too large or too small if the font changes. So `em` is very useful if you are want to express that some things are bigger than others, but you don’t want tie things to specific sizes.

A `px` value refers to a number of “pixels” on the screen. Modern high-resolution screens have pixels that are so small that this might not be an accurate mapping, but the idea is that `px` values give you the ability to place things more precisely, particularly in relation to images which are also sized using pixel dimensions. So `em` is used if you want to lay elements out very precisely in relation to each other.

#### Is this all there is to CSS?

Absolutely not. We have barely scratched the surface in this chapter. You can perform animation, fade images in and out and do all manner of other things too. The great thing is that the interactive help from tools like Visual Studio Code and the wonderful Developer Console in the browser allow you to experiment with styles to find out more. It is certainly worth doing this. As we have seen when we looked at the rollover style selector, we can replace JavaScript code with stylesheet behavior. One of the rules of HTML development is that you should always check to see if a stylesheet could be used to get the effect that you want before you write any JavaScript code. Just because you could write a program do get a particular effect doesn’t mean that you should.

# 4

# Working with data

## What you will learn

In this chapter, you'll build more JavaScript programs. You'll discover that a computer is fundamentally a data processor and that a program tells the computer what to do with the data. You'll see how programs store data using *variables*, and you'll learn how JavaScript manages diverse kinds of data that can be stored by a program. You'll also learn how JavaScript manages the *visibility* of variables within a program. By the end of this chapter, you'll be able to create useful programs.

## Computers as data processors

Humans are a race of toolmakers. We invent things to make our lives easier, and we've been doing it for thousands of years. We started with mechanical devices, such as the plow, which made farming more efficient, but in the last century we've moved into electronic devices and, more recently, into computers.

As computers became smaller and cheaper, they found their way into things around us. Many devices (for example, the smartphone) are possible only because we can put a computer inside to make them work. However, we need to remember what the computer does; it automates operations that formerly required brain power. There's nothing particularly clever about a computer; it simply follows the instructions that it's been given.

A computer works on data in the same way that a sausage machine works on meat: something is put in one end,

some processing is performed, and something comes out the other end. You can think of a program as similar to the instructions a coach gives to a football or soccer team before a play. The coach might say something like, "If they attack on the left, I want Jerry and Chris to run back, but if they kick the ball down the field, I want Jerry to chase the ball." Then, when the game unfolds, the team will respond to events in a way that should let them outplay their opponents.

However, there is one important distinction between a computer program and the way a team might behave in a football game. A football player would know when given some senseless instructions. If the coach said, "If they attack on the left, I want Jerry to sing the first verse of the national anthem and then run as fast as he can toward the exit," the player would raise an objection.

Unfortunately, a program is unaware of the sensibility of the data it is processing, in the same way that a sausage machine is unaware of what meat is. Put a bicycle into a sausage machine, and the machine will try to make sausages out of it. Put meaningless data into a computer, and it will do meaningless things with it. As far as computers are concerned, data is just a pattern of signals coming in that must be manipulated in some way to produce another pattern of signals. A computer program is the sequence of instructions that tell a computer what to do with the input data and what form the output data should have.

Examples of typical data-processing applications include the following (as shown in Figure 4.1):

- Smartphone—A microcomputer in your phone takes signals from a radio and converts them into sound. At the same time, it takes signals from a microphone and makes them into patterns of bits that will be sent out from the radio.
- Car—A microcomputer in the engine takes information from sensors telling it the current engine speed, road speed, oxygen content of the air, accelerator setting, and so on. The microcomputer produces voltages that control the carburetor settings, the timing of the spark plugs, and other things to optimize engine performance .
- Game console—A computer takes instructions from the controllers and uses them to manage the artificial world that it is creating for the gamer.



**Figure 4.1** Computers in devices

\*\*\*Production: Based on figure 2.3 from page 25 of *Begin to Code with C#*, ISBN 9781509301157, but will need to be updated with more contemporary devices - although I'd love to keep the Windows phone ☺

Most reasonably complex devices created today contain data-processing components to optimize their performance, and some exist only because we can build in such capabilities. The growth of "The Internet of Things" is introducing computers into a huge range of areas. It's important to think of data processing as much more than working out the company payroll—calculating numbers and printing out results (the traditional uses of computers). As software engineers, we will inevitably spend a great deal of our time fitting data-processing components into other devices to drive them. These embedded systems mean many people will be using computers even if they're not even aware of it!

#### Programmer's Point

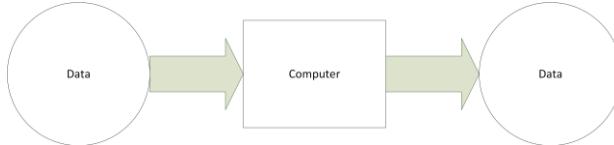
#### Software might be a matter of life and death

Remember that seemingly innocuous programs can have life-threatening capabilities. For example, a doctor may use a spreadsheet you have written to calculate doses of drugs for patients. In this case, a defect in the program could result in physical harm. (I don't think doctors do this—but you never know.) For a deeply scary description of what can go wrong when programmers don't pay attention to the fundamentals, search for Therac-25 on the web.

## Programs as data processors

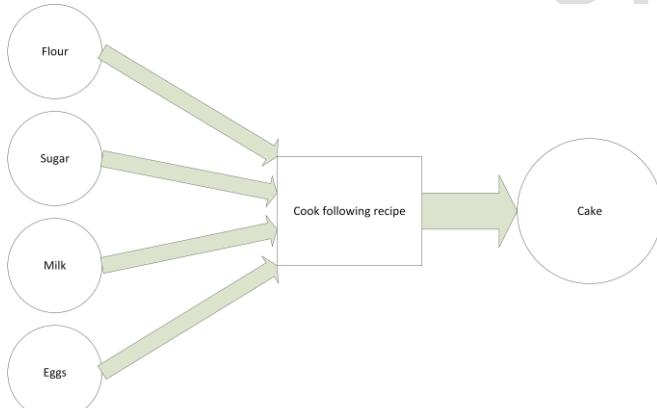
Figure 4.2 shows what every computer does. Data goes into the computer, which does something with it, and then

data comes out of the computer. What form the data takes and what the output means is entirely up to us, as is what the program does.



**Figure 4.2** A computer as a data processor

Another way to think of a program is like a recipe, which is illustrated in Figure 4.5.



**Figure 4.3** Recipes and programs

In this example, the cook plays the role of the computer and the recipe is the program that controls what the cook does with the ingredients. A recipe can work with many different ingredients, and a program can work with many different inputs, too. For example, a program might take your age and the name of a movie you want to see and provide an output that determines whether you can go see that movie based on its suitability rating.

## JavaScript as a data processor

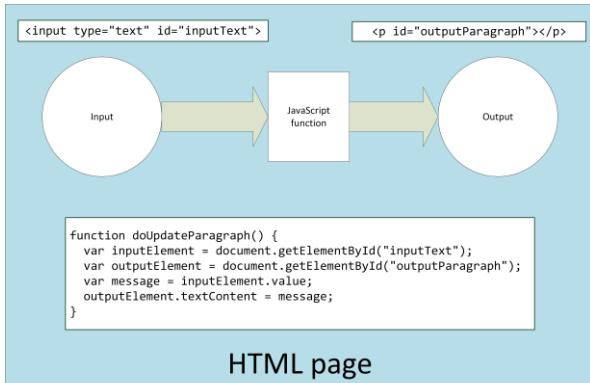


Figure 4.4 JavaScript as a data processor

Figure 4.4 shows the workings of one of the early example programs, Ch02-08 Paragraph Update, from a data processing perspective. The input to the program is an input box on the HTML page, and the output from the program is the text in a paragraph in the HTML page. In this case the JavaScript program code is in the function `doUpdateParagraph` which runs when the user presses a button on the page. This function doesn't perform any processing on the data input, it simply transfers text from the input element to the output element. This is a program structure that we will be using a lot in the next few chapters. We can change what a program does by changing the instructions in the functions that run inside the web page. Statements in a function will act on data and generate new values by evaluating *expressions*.

## Process data with expressions

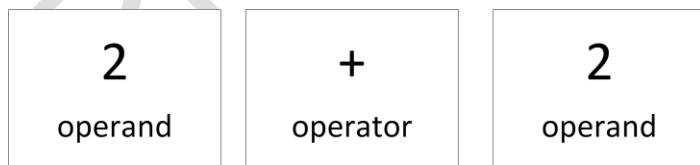


Figure 4.5 Ch04\_Fig\_05 Expression

An expression can be as simple as a single value (for example 2 or "Rob Miles") or it can contain *operators* and

*operands*. Figure 4.5 above shows a simple expression with two operands and one operator. Things that do the actual work are called *operators*. In the case of `2+2`, there are two operands (the two values of 2) and one operator (the plus). When you feed an expression into the JavaScript command prompt, it identifies the operators and operands and then works out the answer. In chapter 1 in **Our first brush with JavaScript** we entered some expressions and saw the results. Now let's enter some more.

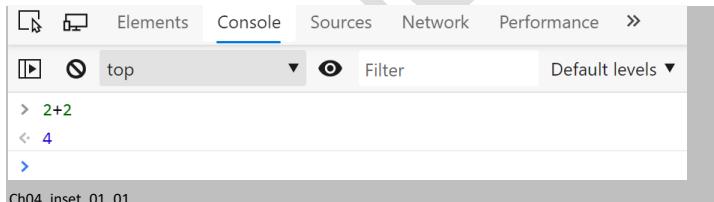
## Code Analysis

### Evaluate expressions with JavaScript

The function in Figure 4.4 doesn't process any data, it simply transfers the text from the input to the output. Data processing in JavaScript is performed by the evaluation of *expressions*. An expression can be as simple as a single value (which is called a *literal* because it is taken exactly as it is). Alternatively, an expression could perform a complicated calculation. Let's do some expression evaluation to find out more.

Question: Can JavaScript work out `2+2`?

Answer: I hope so. Go to the example page **Ch04 Working with data\Ch04-01 Empty Page**. Press F12 to open the Developer View of the web page and select the Console tab. Type the expression `2+2` and press Enter.



The JavaScript console always attempts to evaluate any expression you give it and then return the result. In this case it has worked out that `2+2` is 4. We can do some more experiments using the console to investigate expressions. From now on, rather than showing you screenshots of the browser, I'll just show the output that you'll see in the console. In other words, the previous expression would look like this.

```
> 2+2  
<- 4
```

The typed text is shown in **black**, the output from JavaScript is shown in **blue**, and the command prompts are shown in **brown**.

Question: What do you think would happen if you tried to evaluate `2+3*4`?

Answer: The `*` (asterisk) operator means multiply. JavaScript uses the asterisk in place of the `x` (multiplication symbol) used in math. In math, we always perform higher-priority operations like multiply and divide before addition, so I'd expect the expression above to display the value `14`. The calculation `3*4` would be worked out first, giving an answer of `12`, and this would be added to the value `2`. If you try this in the console, you should see what

you would expect:

```
> 2+3^4  
<- 14
```

Question: What do you think would happen if you tried to evaluate `(2+3)*4`?

Answer: The parentheses enclose calculations that should be worked out first, so in the above expression, I'd expect to see the value **5** calculated `(2+3)` and then this value to be multiplied by **4**, giving a result of **20**.

```
> (2+3)*4  
<- 20
```

Question: What do you think would happen if you tried to evaluate `(2+3*4`?

Answer: This one is quite interesting. You should try it with the console. What happens is that JavaScript says to itself, "The expression I'm trying to work out is incomplete. I need a closing parenthesis." So, the console waits for more input from you. If you type in the closing parenthesis and complete the expression, the value is calculated and the result is displayed. You can even add more sums on the second line if you want.

```
> (2+3*4  
)  
<- 14
```

Question: What do you think would happen if you tried to evaluate `)2+3*4`?

Answer: If JavaScript sees a closing parenthesis before it sees an opening one, it instantly knows that something is wrong and displays an error.

```
> )2+3*4  
(x) Uncaught SyntaxError: unexpected token ')'
```

Note that the command shell is trying to help you work out where the error is by identifying the incorrect character.

## Scripting languages

We can use the console for having conversations like this because JavaScript is a “scripting” programming language (the clue is in the name). You can think of the console as a kind of “robot actor” who will perform whatever JavaScript statements you give it. In other words, you tell the console what you want your program to do using the JavaScript language. If the instructions don’t make sense to the “robot actor,” it tells us it can’t understand them (usually with red text). The process of taking a program and then acting on the instructions in it is called *interpreting* the program. Actors earn a living interpreting the words of a play; computers solve problems for us by interpreting program instructions.

Programmer’s Point

## Not all programming languages run like JavaScript

Not all programming languages are “scripting” languages, which are interpreted in the same way as JavaScript. Sometimes program instructions are converted into the very low-level instructions that the hardware of your computer understands. This process is called *compilation* and the program that performs this conversion is called a *compiler*. The compiled instructions can then be loaded into the computer to be executed. This technique produces programs that can run very fast, because when the compiled low-level instructions are performed, the computer doesn’t have to figure out what the instructions mean; they can just be obeyed.

You might think this means that JavaScript is a “slow” computer language, because each time a JavaScript program runs, the “robot actor” must work out the meaning of each command before performing it. However, this is not really a problem because modern computers run very, very fast, and JavaScript uses some clever trickery to compile your program as it runs.

# Data and information

Now that we understand computers as machines that process data, and we understand that programs tell computers what to do with the data, let’s delve a little bit deeper into the nature of data and information. People use the words *data* and *information* interchangeably, but it’s important to make a distinction between the two, because the way that computers and humans consider data is completely different. Look at Figure 4.6, which shows the difference.

## What the Computer Sees

```
07 104 100 100 100 100 100 100 100 100 100 100 100 100 100 100  
32 67 111 111 111 111 111 111 111 111 111 111 111 111 111 111  
115 42 92 105 116 52 90 101 99 111 109 101  
32 110 100 100 100 100 100 100 100 100 100 100 100 100 100 100  
112 112 108 103 52 116 111 32 300 105 115  
113 113 113 113 113 113 113 113 113 113 113 113 113 113 113 113  
100 32 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100  
111 108 105 105 105 105 99 97 108 52 98 97 109  
100 32 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100  
110 101 29 99 111 101 100 101 98 114 93  
100 32 116 104 101 108 32 119 95 116 106  
32 99 111 111 111 111 111 111 111 111 111 111 111 111 111 111 111  
100 32 116 111 32 97 115 115 117 107 101  
112 111 119 103 114 115 115 112 111 102 32 116  
104 101 37 109 97 114 116 116 48 105 101  
104 101 37 109 97 114 116 116 48 105 101  
32 97 110 100 32 101 113 17 97 108 32 115  
110 110 110 110 110 110 110 110 110 110 110 110 110 110 110 110  
104 109 99 104 32 116 106 101 32 76 97 119  
100 32 116 111 111 111 111 111 111 111 111 111 111 111 111 111 111  
97 111 100 32 111 102 32 76 97 116 117 114  
101 39 119 32 71 111 100 23 109 110 110  
103 103 103 103 103 103 103 103 103 103 103 103 103 103 103 103  
97 32 100 101 99 101 110 116 32 114 101  
111 111 111 111 111 111 111 111 111 111 111 111 111 111 111 111  
101 21 111 112 109 110 105 105 111 110 111 32  
111 102 32 109 97 110 107 105 110 100 32  
113 113 113 113 113 113 113 113 113 113 113 113 113 113 113 113  
97 116 32 116 104 101 121 12 104 111  
113 113 113 113 113 113 113 113 113 113 113 113 113 113 113 113  
32 116 104 101 32 99 97 117 115 101 115  
113 113 113 113 113 113 113 113 113 113 113 113 113 113 113 113  
32 116 100 100 100 100 100 100 100 100 100 100 100 100 100 100 100  
101 32 115 101 112 97 114 114 97 116 105 111  
110 110
```

## What We See



Figure 4.6 Data and information

\*\*\*Production: Please pick up figure 2.6 from page 36 of Begin to Code with C#, ISBN 9781509301157. Thanks, - Rob

The two items in Figure 4.6 contain the same data, except that the image on the left more closely resembles how the document would be stored in a computer. The computer uses a numeric value to represent each letter and space in the text. If you work through the values, you can figure out each value, beginning with the value 87, which represents an uppercase W (in the "When" that begins the first regular paragraph in the document on the right).

Because of the way computers hold data, yet another layer lies beneath the mapping of numbers to letters. Each number is held by the computer as a unique pattern of on and off signals, or **1s** and **0s**. In the realm of computing, each **1** or **0** is known as a *bit*. (For a wonderful explanation of how computers operate at this level and of how these workings form the basis for all coding, see Charles Petzold's *Code: The Hidden Language of Computer Hardware and Software*.) The value **87**, which we know means "uppercase W," is held as the following way:

1010111

This is the *binary* representation of the value. I don't have the space to go into precisely how this works (and Charles Petzold already did this!), but you can think of this bit pattern as meaning "87 is made up of a 1 plus a 2 plus a 4 plus a 16 plus a 64."

Each of the bits in the pattern tells the computer hardware whether a particular power of two is present. Don't worry too much if you don't fully understand this but do remember that as far as the computer is concerned, data is a collection of 1s and 0s that computers store and manipulate. That's *data*.

*Information*, on the other hand, is the interpretation of the data by people to mean something. Strictly speaking, computers process data and humans work on information.

For example, the computer could hold the following bit pattern somewhere in memory:

11111111 11111111 11111111 00000000

You could regard this as meaning "You are \$256 overdrawn at the bank" or "You are 256 feet below the surface of the ground" or "Eight of the thirty-two light switches are off." The transition from data to information is usually made when a human reads the output.

I am being so pedantic because it is vital to remember that a computer does not "know" what the data it is processing means. As far as the computer is concerned, data is just patterns of bits; it is the user who gives meaning to these patterns. Remember this when you get a bank statement that says you have \$8,388,608 in your account when you really have only \$83!

## Data processing in JavaScript

We now know that JavaScript is a data processor. A script containing JavaScript statements is interpreted by the browser, which then produces some output. We also know that within the computer running a JavaScript program, data values are represented by patterns of bits (ons and offs). Now we need to discover how variables let our programs store and manipulate the data being processed.

# Variables in programs

We have already used quite a few variables in our JavaScript programs. *Variables* are how programs remember things. You can think of a variable as a storage location you can refer to by name. What you store in the location, and the name you give it, are up to you. You can create a variable in a JavaScript program by thinking of a name for the variable and then putting a value in the variable. Perhaps you know someone with a pressing need to add up some numbers. In this case the first statement in your program might look like this:

```
var24 total25;
```

If we want to add up a bunch of numbers, we will need something to store the total value. This statement creates a variable with the name `total`. The program will set the initial value of `total` to 0 and then add each number to it. To set the initial value of the `total` variable we use an *assignment* statement:

```
total = 0;
```

The '`=`' in the statement tells JavaScript that the program is performing an assignment. The variable name on the left-hand side of the equals specifies the destination of the assignment (i.e. the place to put the result). The expression on the right-hand side provides the result to store in the variable. When JavaScript sees a variable name in an expression it fetches the value out of the variable and uses that value. Which means that this statement should make perfect sense to you:

```
total = total + 1;
```

The item on the right-hand side of the statement is an *expression* which will generate a result. JavaScript gets the value of `total` and adds 1 to it. It then puts this result into the variable `total`. In other words, the effect of the statement above is to increase the value in `total` by 1.

[Make Something Happen](#)

---

<sup>24</sup> Tells JavaScript that the program is creating a variable.

<sup>25</sup> The name of the variable.

## Working with variables

Let's have a look at variables in JavaScript by using the Developer Tools. Start your browser and navigate to the page in **Ch04 Working with data\Ch04-01 Empty Page** Open up the Developer View by pressing F12. You can begin by creating a **total** variable.

```
> var total
```

When you press enter JavaScript will create the variable. However, it gives a rather strange response:

```
> var total  
<- undefined
```

The response is because every time the console performs a statement it then displays the value generated by that statement. Creating a new variable doesn't create a value, so the console displays the value "undefined". You can see this in action if you enter a statement that assigns a value to total. Enter the statement below and look at the result:

```
> total = 0  
<- 0
```

In JavaScript the result of an assignment is the value that is assigned, so this statement will generate the value 0. Let's try performing the addition we saw in the text:

```
> total = total + 1  
<- 1
```

This statement works out the value of **total + 1** (which will be 0 + 1) and then assigns this to the **total** variable. The effect of this is that **total** now contains the value 1. Perform the statement again:

```
> total = total + 1  
<- 2
```

Each time we perform the addition statement, the value in total gets one larger. This expression might appear confusing. If you've worked with mathematical equations, you'll remember that the equals character means that one value is equal to another. From a mathematical point of view, the statement is obviously wrong, because the **total** cannot equal the **total plus 1**. However, it's important to remember that in JavaScript, the equals operator means "assign." So, the expression **total+1** is evaluated on the right side and then is assigned to the variable on the left side. In JavaScript it is perfectly OK to create a variable and assign it a value immediately:

```
> var total=0
```

If you do this you will notice that the result displayed the console is "undefined". This is because, as we saw at the start of this section, a statement that creates a variable does not return a value. If you haven't set a value into a variable JavaScript marks that variable as holding the value "undefined". Enter this statement to create a new variable called **test**

```
> var emptyTest
```

Now you can investigate the value that the newly created variable `emptyTest` holds. If you just enter the name `emptyTest` variable the console will hold the value stored in the variable `emptyTest`:

```
> emptyTest  
<- undefined
```

Note that this does **not** mean that the variable `emptyTest` does not exist. Instead it means that the variable `emptyTest` does not hold a value. The contents of `emptyTest` are undefined. We can pass the undefined value around as we would any other:

```
> emptyCopy=emptyTest  
<- undefined
```

Now I have a variable called `emptyCopy` that holds a copy of `emptyTest`. Both of these variables hold the undefined value. If I use an undefined value in a calculation I won't get an error, but I will get a result that is not a number:

```
> emptySum=emptyTest + 1  
<- NaN
```

This kind of calculation will not cause a problem with your program, except that it will produce silly results. JavaScript makes a distinction between things that are not defined and things that do not exist. If you try to view the value in a variable that does not exist something different happens. Enter the name `notDefined` and press Enter:

```
>notDefined  
Uncaught ReferenceError: notDefined is not defined at <anonymous>:1:1
```

The variable `notDefined` does not exist, so JavaScript gives us an error.

## JavaScript identifiers

Names of things in JavaScript are called *identifiers*. We used the identifier `total` for the first variable that we created. When you write a program, you must come up with identifiers for the variables in that program. JavaScript has rules about the way you can form identifiers:

An identifier must start with a letter, the dollar character (\$) or the underscore character (\_) and can contain letters, numerals (digits), or the underscore character.

The name `total` is a perfectly legal identifier, as is `xyz`. However, the identifier `2_be_or_not_2_be` would be rejected with an error, because it starts with a digit. Also, JavaScript views uppercase and lowercase letters differently; for example, `FRED` is regarded by JavaScript as different from `fred`.

Programmer's Point

## Create meaningful identifiers

I find it terribly surprising that some programmers use identifiers such as `X21` or `silly` or `hello_mum`. I don't. I work very hard to make my programs as easy to read as possible. So, I'll use ones such as `length` or, perhaps even better, `windowLengthInches`. My window length identifier uses a format where the first letter of each word is a capital letter. This is called *camel case* because the capital letters in the name stick up like humps on the back of a camel. Another convention uses the underscore character to split up the words in an identifier: `window_length_in_inches`. I reckon either of these is OK, although camel case is more common in JavaScript. I don't care which one you use, but I do care that you use it consistently throughout your program.

I don't care which method you choose to make up your identifiers; I only care that you strive to create ones with meaning. If the identifier applies to a particular thing, then identify that thing. And if that thing has particular units of measurement, then then add those, too. For example, if I was storing the age of a customer, I'd create a variable called `customerAgeInYears`.

JavaScript allows identifiers of any length, and longer identifiers don't slow down a program. However, very long names can be a bit hard for humans to read, so you should try to keep them down to the lengths shown in the examples.

## Code Analysis

### Code errors and testing

By now you should be getting used to the idea that if you give a JavaScript program the wrong instructions, the wrong thing will happen. However, consider the JavaScript below, which is supposed to add 1 to the value in the variable with the name `total`:

```
Total = total + 1
```

Question: There is an error in the statement above, which is supposed to add 1 to the variable `total`. What is the error?

Answer: Earlier in this chapter we used a statement that looks like this to add 1 to the value in a variable called `total`. It looks like we are doing the same thing here but that's not the case. There is a crucial difference between this statement and the one we saw earlier. This statement assigns the result of the calculation to a newly created variable called `Total`. The error can happen because, although we have used the word `var` to tell JavaScript that we are declaring a variable, this is not something that JavaScript insists on. If you assign something to a variable that JavaScript has not seen before, JavaScript will just create a new variable with the specified name.

The statement would not generate any complaints from JavaScript when it runs, but it would also not update the

value of `total` correctly. Instead it would create a new variable called `Total`. This is a *logic* error. The statement is completely legal as far as JavaScript is concerned, but it will do the wrong thing when it runs.

I mentioned at the start of this book that JavaScript has some features that make me want to tear my hair out. This is one of them, because it means that your punishment for a simple typing mistake is not an error or warning message. Instead you get a program that runs but doesn't work properly.

#### Question: How do we prevent logic errors?

Answer: The only way to attack logic errors is test. We need to run a program with some known values (values for which we know the total) and then verify that the answers agree with the test total. If the answers make sense, we can start to build confidence in our code. However, even if a program passes all the tests, it could still be faulty because there might be a fault that is not picked up by those tests.

Tests don't prove that a program is good; tests simply prove that a program is not as bad as it would be if it had failed the tests.

Tests work best if they are added at the time the program is created. We'll talk about testing strategies every time we make a new program.

```
total = Total + 1
```

#### Question: The statement above also contains a miss-spelling of a variable named `total`. However, this time the name on the right-hand side of the equals is miss-spelt. What will happen when this program runs?

Answer: JavaScript will refuse to run this statement. It will tell you that you are using a variable with the name `Total`, which it hasn't seen yet. Sometimes typing mistakes will be detected before your program runs, but other times they might not.

You might be thinking that you've been set up to fail because I've suggested that you use long, meaningful names, and typing those long, meaningful names creates more opportunities for mistakes. For now, one way around this problem is to use the text copy feature of your editor to copy names from one part of the program to another. There are also programs that can scan your JavaScript looking for variables that haven't been created using `var`. These can alert you to possible errors.

## Performing calculations

We know that JavaScript expressions are made up of operators and operands. The operators identify actions to be performed, and the operands are worked on by the operators. Now we can add a bit more detail to this explanation. Expressions can be as simple as a single value or as complex as a large calculation. Here are a few examples of numeric expressions:

```
2 + 3 * 4  
-1 + 3  
(2 + 3) * 4
```

These expressions are evaluated by JavaScript working from left to right, just as you would read them yourself. Again, just as in traditional math, multiplication and division are performed first, followed by addition and subtraction. JavaScript achieves this order by giving each operator a priority. When it works out an expression, it finds all the operators with the highest priority and applies them first. It then looks for the operators next in priority, and so on, until the result is obtained. The order of evaluation means that the expression  $2 + 3 * 4$  will calculate to 14, not 20.

If you want to force the order in which an expression evaluates, you can put parentheses around the elements of the expression you want to evaluate first, as in the final example above. You can also put parentheses inside parentheses if you want—provided you make sure that you have as many opening parentheses as closing ones. Being a simple soul, I tend to make things clear by putting parentheses around everything.

It is probably not worth getting too worked up about expression evaluation. Generally speaking, things tend to be evaluated how you would expect. Here is a list of some other operators, with descriptions of what they do, and their precedence (priority). The operators are listed with the highest priority first.

Operator	How it's used
-	Unary minus, the minus that JavaScript finds in negative numbers,
*	Multiplication. Note the use of the * rather than the more mathematically correct but confusing x.
/	Division, because of the difficulty of drawing one number above another during editing, we use this character instead.
+	Addition
-	Subtraction. Note that we use the same character as for unary minus.

This is not a complete list of the operators available, but it will do for now. Because these operators work on numbers, they are often called *numeric operators*. However, one of them, the + operator, can be applied between strings, as you've already seen.

## Code Analysis

### Work out the results

Question: See if you can work out the values of a, b, and c when the following statements have been evaluated:

```
var a = 1;
var b = 2;
var c = a + b;

c = c * (a + b);
b = a + b + c;
```

Answer: a=1, b=12, c=9. The best way to work this out is to behave like a computer would and work through each statement in turn. When I do this, I write down the variable values on a piece of paper and then update each as I go along. Doing this means that you can predict what a program will do without having to run it.

# Whole numbers and real numbers

We know that JavaScript is aware of two fundamental kinds of data—text data and numeric data. Now we need to delve a little deeper into how numeric data works. There are two kinds of numeric data—whole numbers and real numbers. Whole numbers have no fractional part. Up until now, every program that we have written has made use of whole numbers. A computer stores the value of a whole number exactly as entered. Real numbers, on the other hand, have a fractional element that can't always be held accurately in a computer.

As a programmer, you need to choose which kind of number you want to use to store each value.

## Code Analysis

### Whole numbers versus real numbers

You can learn about the difference between whole numbers and real numbers by looking at a few situations in which they might be used.

Question: I'm building a device that can count the number of hairs on your head. Should I store this value as a whole number or as a real number?

Answer: This should be a whole number because there is no such thing as half a hair.

Question: I want to use my hair-counting machine on 100 people and determine the average number of hairs on all their heads. Should I store this value as a whole number or as a real number?

Answer: When we work out the result, we'll find that the average includes a fractional part, which means that we should use a real number to store it.

Question: I want to keep track of the price of a product in my program. Should I use whole numbers or real numbers?

Answer: This is very tricky. You might think that the price should be stored as a real number—for example, \$1.50 (one and a half dollars). However, you could also store the price as the whole number, 150 cents. The type of number you use in a situation like this depends on how that number is used. If you're just keeping track of the total amount of money you take in while selling your product, you can use a whole number to hold the price and the total. However, if you are also lending money to people to buy your product and you want to calculate the interest to charge them, you would need a fractional component to hold the number more precisely.

## Programmer's Point

### The way you store a variable depends on what you want to do with it

It seems obvious that you would use a whole number to count the number of hairs on your head. However, one could argue that we could also use a whole number to represent the average number of hairs on 100 people's heads. This is because the calculated average would be in the thousands. Fractions of a hair would

not add much useful information, so we could drop any fractional parts and round to the nearest whole number. When you consider how you are going to represent data in a program, you must take into account how it will be used.

## Real numbers and floating-point numbers

Real number types have a fractional part, which is the part of the number after the decimal point. Real numbers are not always stored exactly as they are entered into JavaScript programs. Numbers are mapped to computer memory in a way that stores a value that is as close as possible to the original. The stored data is often called a *floating-point* representation. You can increase the accuracy of the storage process by using larger amounts of computer memory, but you are never able to hold all real values precisely.

This is not a problem, however. Values such as pi can never be held exactly because they “go on forever.” (I’ve got a book that contains the value of pi to 1 million decimal places, but I still can’t say that this is the exact value of pi. All I can say is that the value in the book is many more times more accurate than anyone will ever need.)

When considering how numbers are stored, we need to think about *range* and *precision*. Precision sets out how precisely a number is stored. A particular floating-point variable could store the value 123456789.0 or 0.123456789, but it can’t store 123456789.123456789 because it does not have enough precision to hold 18 digits. The range of floating-point storage determines how far you can “slide” the decimal point around to store very large or very small numbers. For example, we could store the value 123,456,700, or we can store 0.0001234567. For a floating-point number in JavaScript, we have 15 to 16 digits of precision, and we can slide the decimal point 308 places to the right (to store huge numbers) or 324 places to the left (to store tiny numbers).

The mapping of real numbers to a floating-point representation does bring some challenges when using computers. It turns out that a simple fraction such as 0.1 (a tenth) cannot be held accurately by a computer because of the way values are held. The value stored to represent 0.1 will be very close to that value, but not the same. This has implications for the way we write programs.

### Code Analysis

#### Floating-point variables and errors

We can find out more about how floating-point values work by doing some experiments using the JavaScript Developer View in the browser. If we just type numeric expressions, we can view the results that JavaScript calculates.

Question: What happens if we try to store a value that can’t be held accurately as a floating-point value?

Answer: We know that the value 0.1 can’t be held accurately in a computer, so let’s enter that value into the Python Shell and see what comes back. Go to the console in browser and enter the following:

```
> 0.1
```

[<= 0.1](#)

At this point, you might think that I've been lying to you because I said that the value 0.1 can't be held accurately, and now this example shows JavaScript returning the value 0.1. However, I'm not lying to you—JavaScript is. The JavaScript print routine "rounds" values when it prints them. In other words, it says that if the number to be printed is `0.1000000000000000551115` or thereabouts (which it is), then it will just print 0.1.

**Question:** Does this "rounding" really happen?

**Answer:** At the moment, you've just got my word for it that values are rounded when printed and that errors are being hidden from us as a result. However, if we perform a simple calculation, we can introduce an error that is large enough to escape being rounded. Enter the following calculation into the JavaScript Shell and note what comes back.

```
> 0.1+0.2  
<- 0.30000000000000004
```

The result of adding 0.1 to 0.2 should be 0.3. But, because the values are held as binary floating-point values, the result of the calculation contains an error large enough to escape being rounded. It turns out that our highly expensive computer really can't get its sums right!

You might think that your all-powerful computer should be able to hold all values precisely. It comes as a bit of a shock to discover that this is not true and that a simple pocket calculator can outperform your powerful PC.

However, this lack of accuracy is not a problem in programming because we don't usually have incoming data that is particularly precise anyway. For example, if I refine my hair counting device to measure hair length, it would be difficult for me to measure hair length with more than a tenth of an inch (2.4 millimeters) of accuracy. For hair data analysis, we need only around three or four digits of accuracy. It is very unlikely that you will ever process any data that requires the 15 digits that JavaScript can give you.

It is also worth noting at this point that these issues have nothing to do with JavaScript. Most, if not all, modern computers store and manipulate floating-point values using a standard established by the Institute of Electronic and Electrical Engineers (IEEE) in 1985. All programs that run on a computer will manipulate values in the same way, so floating-point numbers in JavaScript are no different from those in any other language.

The only difference between JavaScript floating-point values and those in other languages is that a floating-point variable in JavaScript occupies 8 bytes of memory, which is twice the size of the float type in the languages C, C++, Java, and C# (but not Python). A JavaScript floating-point variable equates with a *double precision* value in those languages.

#### Programmer's Point

#### Don't confuse precision with accuracy

It is very important to remember that numbers don't become more accurate just because they are stored with more precision. Scientists in a laboratory measuring the length of ant legs will not be able to do this to

more than a few digits of accuracy (unless they have some amazing technology), so there is no point in them using much higher precision to store and process their results. Using higher precision slows down the program and means that the variables take up more space in memory.

## Creating a random dice

We can explore the difference between integer and floating-point values in JavaScript by creating a “Random dice” web page. This will display a value between 1 and 6 when the user presses a button. JavaScript has a built-in library of `Math` functions including one called `random` which generates a real number that ranges from 0 to 1 (but does **not** include the value 1). Let’s investigate how this works.

### Make Something Happen

#### Random numbers

Let’s have a look at random numbers in JavaScript. Start your browser and navigate to the page in **Ch04 Working with data\Ch04-02 Computer Dice**. Press the button and note that you get a different dice roll each time.

##### Digital Dice

Rolled: 4

Ch04\_inset\_02\_01

Open up the Developer View by pressing F12. You can begin by displaying a random number by calling the `random` function from the `Math` library:

```
> Math.random()
```

When you press enter JavaScript will calculate a random number and display the result.

```
> Math.random()  
<- 0.01479622790601498
```

I would be very, very surprised if you got the same number as the one printed above. You will see a number between 0 and 1. Call `random` a few more times and note that you get a different number each time. If you tried it enough times it is possible you could see a value 0 but you would **never** see a value of 1. That is important to us.

The random function returns a value between 0 and 1. We can expand the range of the random number by multiplying it by the range that we want, in this case that range is 6. Try entering this:

```
> Math.random()*6
```

This will generate a result that ranges from 0 up to, but not including 6. (because `random()` can never return 1). Try it.

```
> Math.random()*6  
<- 1.342641962710725
```

The next thing we need to do is get rid of that fractional part. JavaScript provides another `Math` function, called `floor` which chops the fractional off a number. No matter how the fractional part, it is always discarded. Try this:

```
> Math.floor(1.9999)
```

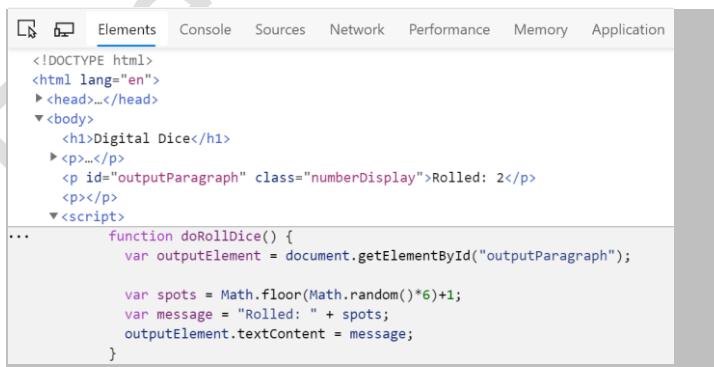
The value 1.9999 is very close to 2, but the `floor` function throws away the entire fractional part. We can apply the `floor` function to our random value. Try this.

```
> Math.floor(Math.random()*6)
```

This is an important thing to learn. I can feed an expression into a function call. The above statement gets a value from the `random` function, multiplies it by 6 and feeds the result into the `floor` function. If you repeat this statement lots of times you should see the values 0,1,2,3,4 and 5 appearing at random. We want a value between 1 and 6, so we just add 1 to this:

```
> Math.floor(Math.random()*6)+1
```

This is the “brains” of our dice program. Click the Elements tab in the Developer View and then expand the `<script>` element to view the `doRollDice` function.



```
Elements Console Sources Network Performance Memory Application

<!DOCTYPE html>
<html lang="en">
  <head>...</head>
  <body>
    <h1>Digital Dice</h1>
    <p>...</p>
    <p id="outputParagraph" class="numberDisplay">Rolled: 2</p>
  </body>
</html>

<script>
  function doRollDice() {
    var outputElement = document.getElementById("outputParagraph");

    var spots = Math.floor(Math.random()*6)+1;
    var message = "Rolled: " + spots;
    outputElement.textContent = message;
  }
</script>
```

#### Ch04\_inset\_02\_02

In the middle of the function there is a statement that sets the value of the spots variable to a random number between 1 and 6. How would you change the program to produce a number between 1 and 20?

```
> Math.floor(Math.random()*20)+1
```

This turns out to be very easy, the program must multiply the random value by 20 rather than 6.

## Working with text

We now know how to use variables that hold numbers. A program can also create a variable that holds a string of text.

```
var customerName = "Fred";
```

This statement looks exactly like the statement we used to create the `total` variable except that the value being assigned is a string of text. The string being assigned is enclosed in double quote characters that define the limits of the text. In the double quote characters are called *delimiters* because they define the limits of the text. The delimiters are not part of the string being stored so the `customerName` variable just contains the word Fred.

The variable `customerName` is different from the `total` variable in that it holds text rather than a number. We can use this variable anywhere we would use a string.

```
var message = "the name is " + customerName;
```

In the expression being assigned, the text in the variable `customerName` is added onto the end of the string `"the name is "`. As `customerName` currently holds the string `"Fred"` (we set this in the previous statement), the above assignment would create another string variable called `message` which contains `"the name is Fred"`.

## JavaScript string delimiters

The `"` character can act as a delimiter that marks the start and the end of a string in the program. But what if we want to enter a string that contains a `"` character? In that case we can use a single quote (`'`) to mark the start and end of the string:

```
var message = 'Read "Begin to code with JavaScript". It is an amazing book';
```

If you want to enter a string that contains both single and double quotes you can use backticks (`) to delimit the string:

```
var message = `Read "Begin to code with JavaScript". It's an amazing book`;
```

If you have a need to enter a string that contains both kinds of quotes and backticks then it must be an amazing string. You can enter that by using *escape sequences*.

## Escape sequences in strings

You can include quote characters in a string by using an *escape sequence*. Normally, each character in a string represents that character. In other words, an A in a string means 'A'. However, when JavaScript sees the escape character —the backslash (\) character — it looks at the text following the escape character to decide what character is being described. This is called an escape sequence. There are many different escape sequences you can use in a JavaScript string. The most useful escape sequences are shown in the following table.

Escape sequence	What it means	What it does
\\	Backslash character (\)	Enter a backslash into the string
\'	Single quote ('')	Enter a single quote into the string
\`	Backtick (`)	Enter a single backtick into the string
\"	Double quote (")	Enter a double quote into the string
\n	Unicode Line Feed/New Line	End this line and take a new one
\t	Unicode Tab	Move to the right to the next tab stop
\r	Unicode Carriage return	Return the printing position to the start of the line

If you're wondering what Unicode means, it is a mapping of numbers to character designs. We saw it in Chapter 2 in the section "Display Symbols".

## Working with strings and numbers

You can create expressions involving strings, but the only operator that can be used between two strings is the + operator that we saw earlier. You can also create expressions involving strings and numbers, but you need to be a bit careful when you do this.

### Code Analysis

#### Combining strings and numbers

We can find out more about how strings and numbers can be combined in a JavaScript program by using the Developer View to answer some questions.

Question: What happens if I add a number to a string?

Answer: We know that JavaScript regards numbers and strings as different types of data. Let's see what happens when we add them together:

```
> "hello" + 99  
<- "hello99"
```

This statement adds the numeric value 99 to the string "Hello". JavaScript automatically converts the number into a string giving the result that you see above. However, this can lead to strange behavior if you add lots of numbers to a string...

Question: What happens if I add lots of numbers to a string?

Answer: The way that JavaScript performs the conversion of numbers to strings can result in some interesting consequences:

```
> 1 + 2 + "hello" + 3 + 4  
<- "3he11o34"
```

JavaScript works along the expression. It adds the 1 and the 2 to produce the value 3. Then it sees a string and goes "Oh. I need to make this number into a string". It converts the value 3 into a string and adds it to "hello". Then it converts everything else it finds into strings too and adds them together. If you want to force the calculations to be performed before the values are converted into strings you can use brackets.

```
> (1+2) + "hello" + (3+4)  
<- "3he11o7"
```

Note that this is not the kind of programming I approve of, because it is a bit confusing. If I really wanted to create an output like the above I would break it down into a number of separate statements.

```
var calc1 = 1 + 2;  
var calc2 = 3 + 4;  
var result = calc1 + "hello" + calc2;
```

This makes it very clear to the reader of my program that I wanted the values to be calculated before I displayed them.

## Converting strings into numbers

We have seen that JavaScript regards numeric and text variables as different types of data. We have also seen that JavaScript will automatically convert from a number into a string when it thinks this is appropriate (although it may sometimes get this wrong). For example, we might want to create an adding machine web page. The user enters two values presses a button which causes the sum of the numbers to be displayed.

## A very simple Adding Machine

It can add two numbers together.

First number:

Second number:

4

**Figure 4.7** Adding machine

Figure 4.7 above shows how the page can be used to solve the age-old question “What is  $2+2$ ?” The page contains two input fields for the values to be entered, a button to request a calculation and a paragraph element that displays the result. We know how to create almost every part of this application except for one thing. The user will enter the numbers to be added as text strings. We need a way of converting these strings into numbers that can be added together.

### MAKE SOMETHING HAPPEN

#### Converting strings into numbers

JavaScript makes a distinction between numbers and values. Let's investigate how this works. Open the web page in the example folder **Ch04 Working with data\Ch04-03 Adding Machine**. This displays an adding machine. Enter two numbers and press the Add Numbers button. Note that the right answer appears. Now press F12 to open the Developer View. Click the console tab to view the console prompt. Type the following sum and press enter:

> `2+2`

The console always returns the result of any expression you enter, so you see the answer would expect:

<- `4`

Now enter a different sum:

> `"2"+"2"`

This is adding the string “2” to the string “2”.

<- `22`

We can see this distinction in action when we create variables. Create the two variables by entering the following:

```
> var stringTwo = "2"  
> var numberTwo = 2
```

JavaScript provides a function called `typeof` which will tell you the type of a variable you supply to it. Use it to investigate the type of the variables `name` and `age`.

```
> typeof(stringTwo)  
<- "string"  
> typeof(numberTwo)  
<- "number"
```

JavaScript provides a function called `Number` which will attempt to convert whatever you give it into a number. Let's use this to convert the string version of 2 into a number version:

```
> var convertedTwo = Number(stringTwo)
```

This statement creates a new numeric variable called `convertedTwo` which contains the number held in the variable `stringTwo`. The type of `convertedTwo` is a number, as you would expect

```
> typeof(convertedTwo)  
<- "number"
```

Leave the Developer Tools window open, you will be doing some more work this in a little while.

## Make an adding machine

Now that we know we can use the `Number`

function to convert from a string type into a numeric type we can create our adding machine. The HTML for the elements on the page is as follows:

```
<h1>A very simple Adding Machine</h1>  
<p>It can add two numbers together.</p>  
<p>  
First number: <input type="text"26 id="no1Text" value="0">  
</p>  
<p>  
Second number: <input type="text"27 id="no2Text" value="0">  
</p>  
<p>
```

---

<sup>26</sup> First number input

<sup>27</sup> Second number input

```
<input type="button" value="Add numbers"28 onclick="doAddition()"></button>  
</p>  
  
<p id="resultParagraph"29  
  Result displayed here.  
</p>
```

You can map each of the elements in the HTML onto items on Figure 4.1. There are two text input areas for the user to type in the values that are to be added. These areas have ids which are `n1text` and `n2text`. There is also an output paragraph which will be used to display the output. The output paragraph has the id `resultParagraph`. When the user clicks the button the function `doAddition` will be called to calculate the result and display it. This function takes the text out of the inputs, converts it into numbers and then does the calculation.

The `input` element reads a string of text from the user. This works if we just want to read names. However, it is not so useful if we want to read in numbers.

```
var no1Element = document.getElementById("n1Text");30  
var no1Text = no1Element.value;31  
var no1Number = Number(no1Text);32
```

This is the code that gets the user input for the first value and converts it into a number that can be used in calculations. The `Number` function performs the conversion, the first two statements get a reference to the input element holding the number text and then get the text from that element. The program uses a similar sequence of statements to get the value entered in the second input element.

```
var result = no1Number + no2Number;
```

This is the statement that calculates the addition and stores it in a variable called `result`. This is the part of the program where the data processing is performed. The rest of the HTML and JavaScript are there to provide a means of

---

<sup>28</sup> Button to trigger the calculation

<sup>29</sup> Paragraph to display the answer.

<sup>30</sup> Get a reference to the element holding the user input

<sup>31</sup> Get the text from the user input element

<sup>32</sup> Convert the text into a number

input and output. The last thing the function needs to do is display the result for the user. We have written several functions that do this; the result is displayed by modifying the text in a paragraph on the web page.

```
var resultElement = document.getElementById("resultParagraph");
resultElement.innerText = result;
```

The first statement gets a reference to the `resultParagraph` element. The second statement sets the `innerText` to the value of the `result` that was calculated. Note that the `result` variable holds a number, but JavaScript will automatically convert this to text for display. You have seen this automatic conversion of numbers to text before.

## WHAT COULD GO WRONG

### Entering invalid numbers

When you create a program you have to think of ways that it could go wrong. For example, after being asked to type 2 into a text entry in our adding machine it is perfectly possible for a user to type this instead:

First number:

Ch04\_inset\_03\_01

We are using the function `Number` to convert text into a numeric value. It would be very impressive if the `Number` function was able to convert “two” into the value 2, but unfortunately it can’t. Instead it decides that “two” is not a number, and so it returns a result of “not a number” or `Nan`. Any JavaScript calculations involving something which is not a number generate a result of “not a number” and so an attempt to use text like this would result in the display below:

First number:

Second number:

`Nan`

Ch04\_inset\_03\_02

The good news is that at least the program didn’t output a number like -8399608 when the user upset it like this, but it is still less than perfect. In the next chapter we will discover how a program can decide when a number is valid and display a suitable alert message. However, the best way to remove an error like this is to do something

that ensures it can never happen. We can tell the browser that a given input element is a number rather than text.

```
First number: <input type="number" id="no1Text" value="0">
```

The HTML above shows how we do this. The type of the input is now `number`. If we do this the input element will only accept numbers when the user types things into it. If you used this input area on your smartphone you'd be given a numeric keypad to enter the value, rather than a full keyboard. On a Windows PC the Edge browser even shows up and down buttons that you can use to change the numeric value of the element.

It can add two numbers together.

First number:

Second number:

Ch04\_inset\_03\_03

Note that even though we have specified that the type of the input field is "number", we still get a text string from the input field when we use it in our programs. However, we can be sure that the text that we get from the element only contains digits. You can try this version of the page in the examples: **Ch04 Working with data\Ch04-04 Number Adding Machine**

#### PROGRAMMER'S POINT

#### Error handling is a big part of programming

Professional programmers spend at least as much time thinking about how things can go wrong as they do writing program code. They also spend a lot of time deciding how they can prove that their program works by testing it. This is one reason why what look like simple programs can take a long time to create.

## Making applications

We now know enough programming to be able to make some useful applications. So let's make some.

### Calculating a pizza order

Rigorous scientific research conducted by me at many hackathons I've attended has arrived at a figure of exactly 1.5

people per pizza. In other words, if I get 30 students I'll need 20 pizzas, and so on. I decided to make a web page that works out how many pizzas I need to order for a given number of students. The user types in the number of students and presses the Calculate Pizzas button to display the result. This is my first version:

```
<!DOCTYPE html>
<html>

<head>
  <title>Ch04-05 Pizzcalc Vesion 1</title>
  <link rel="stylesheet" href="styles.css">33
</head>

<body>
  <h1> "� Pizza Calclater</h1>
  <p>Calculates the number of pizzas you' ll need.</p>
  <p>
    Number of students: <input type="number" id="noOfStudentsText" value="0">34
  </p>
  <p>
    <input type="button" value="Calculate pizzas" onclick="doPizzaCalc()"></button>35
  </p>

  <p id="resultParagraph">36
    Result displayed here.
  </p>

<script>
  function doPizzaCalc() {37
    var noOfStudentsElement = document.getElementById("noOfStudentsText");
    var noOfStudentsText = noOfStudentsElement.value;
    var noOfStudents = Number(noOfStudentsText);38
  }
</script>
```

---

<sup>33</sup> Using a stylesheet to add some style

<sup>34</sup> Input element for the number of students

<sup>35</sup> Button to press to display result

<sup>36</sup> Paragraph to display the result

<sup>37</sup> Function that calculates the result

<sup>38</sup> Get the number of students

```

var noOfPizzas = noOfStudents / 1.5;39
var resultElement = document.getElementById("resultParagraph");40
resultElement.innerText = "You need " + noOfPizzas + " pizzas.";41
}
</script>
</body>

</html>

```

## Pizza Calculator

Calculates the number of pizzas you'll need.

Number of students:

You need 20 pizzas.

**Figure 4.8** Pizza Calculator

This is my first version. I'm using the Pizza emoji symbol (`&#x1f355;`) to get a nice pizza slice for the heading. The page works fine with the data above. If I say there are 30 students, the program will tell me I need 20 pizzas. However, there are problems with some numbers of pizzas:

## Pizza Calculator

Calculates the number of pizzas you'll need.

Number of students:

You need 26.666666666666668 pizzas.

**Figure 4.9** Pizza Fractions

---

<sup>39</sup> Perform the calculation

<sup>40</sup> Get the paragraph that displays the result

<sup>41</sup> Display the result

I can't ask the pizza place for a fraction of a pizza, so I need a way of converting the number of pizzas to an integer. At this point, I also must decide what the conversion will do. If I just use the `floor` function we used in the dice program, this will result in an order for 26 pizzas because the `floor` function truncates the floating-point value. This effectively means that I'll have pizza for only 39 people rather than 40, leaving one hungry student. There are several ways to address this problem. I might think the best way to attack the problem is to add one extra pizza to the order to take care of any "spares."

```
var noOfPizzas = Math.floor(noOfStudents / 1.5) + 1;
```

## MAKE SOMETHING HAPPEN

### Fix the pizza program

Load the example program in the folder **Ch04 Working with data\Ch04-05 Pizzacalc Version 1** and modify it using the above statement so that when you tell it there are 40 students, the program suggests that you buy 27 pizzas. Then change the program to make it less generous. Make the program always round down to the nearest integer number of pizzas to order. You can find my generous version in **Ch04 Working with data\Ch04-05 Pizzacalc Version 2**

The program uses a stylesheet to modify the style of the `<h1>` and `<p>`. You could modify these styles to make it look even better.

## Programmer's Point

### Never assume that you know what a program is supposed to do

If you wrote the pizza calculator for a customer, you should *not* decide for yourself what the program should do if it must order a fraction of a pizza. Your customer might want to "round down" the number of pizzas to keep their costs down. If this is the case, they will complain when your program adds an extra pizza. Alternatively, they might want to establish a reputation as a generous person, in which case the program should be made to "round up" the value.

As a programmer, never assume that you know what the program should do. You must always ask the customer. Otherwise, you might find yourself paying for over-ordered pizzas.

## Converting between Fahrenheit and centigrade

An interesting thing about the adding machine program and the pizza calculator is that they have similar structures. The adding machine takes an additional input (the second number to be added) but the way that it works is just the same as the pizza calculator. We can use this same pattern to make a third program that converts temperature readings between Fahrenheit and centigrade. Have a go at creating a conversion web page. Here are some hints:

### Conversion formula

To convert a temperature from Fahrenheit to centigrade, you subtract 32 from the Fahrenheit value and then divide the result by 1.8. This statement below shows how this would work. It assumes that you have created a variable called `fahrenheit` that contains the temperature in Fahrenheit.

```
var centigrade = (fahrenheit-32)/1.8;
```

### Truncating the displayed temperature value

We saw in the pizza calculator that JavaScript likes to display lots of decimal places when it shows you a number. It would be distracting if the program displayed a 60 degrees Fahrenheit as 15.5555555 degrees Centigrade. The method `toFixed` can be used on a numeric variable to create a string with a particular fixed number of decimal places. The statement below would create a variable called `resultString` that contains a number string that only contains one decimal place. In other words, a temperature of 15.555555 would be converted to a string containing 15.6.

```
var resultString = centigrade.toFixed(1);
```

### Displaying a thermometer emoji

You might want to add a thermometer emoji to your web page. This requires you to add two symbols to your HTML source. The HTML below will display a heading with a thermometer emoji.

```
<h1>&#x1f321;&#xfe0f; Fahrenheit to Centigrade</h1>
```

You can find my version of the solution here: [Ch04 Working with data\Ch04-07 Fahrenheit to Centigrade](#). You can now write any kind of conversion program you like, converting feet to meters, grams to ounces, or liters to gallons.

## Adding Comments

As soon as you start to make useful programs, I think you should start adding comments to make it clearer what your program is doing. You don't write comments for the computer, you write comments for someone reading your program:

```
/* Based on each pizza feeding 1.5 students. We divide the number
   of students by 1.5 to get the number of pizzas. Then we drop
   the fractional part and add 1 to round up the number
   Note that this means we might buy slightly too much
   pizza for some numbers of students.
*/
var noOfPizzas = Math.floor(noOfStudents / 1.5) + 1;
```

The comment in the program above makes it very clear exactly how we are calculating the number of pizzas and the reasoning behind the statement. If the comment wasn't there you'd have to know that 1.5 was there because we have decided that is how many students each pizza will feed. You can write a comment that spreads over several lines by enclosing your comment text in the characters /\* and \*/. When the browser sees the character sequence /\* in a JavaScript program it ignores the following text, up to the point where it sees a \*/ that ends the comment text. You can put comments anywhere in your program. The browser will completely ignore them. You can also create single line comments:

```
var centigrade = (fahrenheit-32)/1.8; // using standard conversion formula
```

The comment above is a single line comment. It starts at the character sequence // and finishes at the end of that line. We can add these kinds of comments on the end of a statement, or on a line by themselves. If you use Visual Studio Code to write your programs, you'll find that comments are displayed in green to make them stand out.

It is important that programs are written in a way that makes it easy for humans to understand what is going on. We have seen that when choosing identifiers for variables we need to make sure that the name describes what the variable is being used for. We can also make programs clearer by adding comments.

Some people say that writing a program is a bit like writing a story. I'm not completely convinced that this is true. I have found that some computer manuals are works of fiction, but programs are something else. I think that while it is not a story as such, a good program text does have some of the characteristics of good literature:

- It should be easy to read. At no point should the hapless reader be forced to backtrack or brush up on knowledge that the writer assumes is there. All the names in the text should impart meaning and be distinct from each other.

- It should have good punctuation and grammar. The various components should be organized in a clear and consistent way.
- It should look good on the page. A good program is well laid out. The different parts should be indented, and the statements spread over the page in a well formed manner.
- It should be clear who wrote it, and when it was last changed. If you write something good you should put your name on it. If you change what you wrote you should add information about the changes that you made and why.

A big part of a well written program is the comments that the programmer puts there. A program without comments is a bit like an airplane which has an autopilot but no windows. There is a chance that it might take you to the right place, but it will be hard to tell where it is going from the inside.

Be generous with your comments. They help to make your program much easier to understand. You will be surprised to find that you quickly forget how you got your program to work. You can also use comments to keep people informed of the particular version of the program, when it was last modified and why, and the name of the programmer who wrote it – even if it was you. From now on the example code that you see will have what I consider an appropriate level of comments.

#### PROGRAMMER'S POINT

##### Don't add too much detail in your comments

Writing comments is a very sensible thing to do. But don't go mad. Remember that the person who is reading your program can be expected to know JavaScript and doesn't need things explained to them in too much detail:

```
goatCount = goatCount + 1; // add one to goatCount
```

This is plain insulting to the reader I reckon. If you chose sensible names you should find that quite a lot of your program will express what it does directly from the code itself.

## HTML Comments

Note that these comments only work in the `<script>` part of the program. You can add comments to the HTML, as we saw in Chapter 2, but you use a different character sequence to mark the start and end of the comments:

```
<!-- Rob's Pizza Calculator Page Version 1.0 -->
```

The start of the comment is marked by the sequence `<!--` and end of the comment by the sequence `-->`. As with JavaScript comments, the text between the two sequences is ignored by the browser.

# Global and local variables

At the start of this section we wanted to make a totalizer program which can be used to add up a bunch of numbers. We can now create this. Let's assume that we are creating a solution for a customer who really does want to totalize some numbers. You have sat down with her and agreed on the following design for the application.

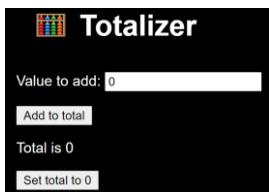


Figure 4.10 Totalizer

Your customer would like a stylish black background containing the Abacus emoji. We can create this by using a stylesheet that sets the color scheme for the application and finding the symbol number for the abacus (&#x1f9ee;).

She wants to be able to type in a value and press the “Add to total” button to add the value to the total. She also wants a button she can use to set the total back to zero when she has finished adding one set of values. You agree on the design shown in Figure 4.10 above.

## PROGRAMMER'S POINT

### Getting a good specification is vital

The sample page above is a good start for the specification of the Totalizer application. It is very important that you get a solid specification for any work that you perform, even (or perhaps especially) if you are working for someone you know. The nice thing about the screenshot of the application is that it sets out exactly what the solution should look like. However, there are some questions I'd want answering too.

I would like to know if there is any upper limit to the amount to be added to the total. I'd also like to know if the Totalizer should accept negative numbers to be subtracted from the total or whether the total should always be increased. The answers to these questions tell me whether the Totalizer should detect and reject invalid input values. The customer might be assuming that negative values should not be added (or might never have thought about this issue). Either way, as the builder of the solution you need to know how it should work. Otherwise you might end up having conversations with your customer which include phrases such as “It isn't supposed to do that...”.

## Global variables

The Totalizer program is interesting because it is the first program we have written that needs to “remember” something between function calls. Up until now every program that we have created takes data from the input elements, does something to it, and then displays the result. Any variables that we have created to store data in a function during data processing are discarded as soon as the process has finished. As an example, consider the temperature conversion program. This takes a temperature entered in Fahrenheit and converts it to Centigrade.

```
function doTempConvert() {  
  
    var fahrenheitElement = document.getElementById("fahrenheitText");42  
    var fahrenheitText = fahrenheitElement.value;43  
    var fahrenheit = Number(fahrenheitText);44  
  
    var centigrade = (fahrenheit-32)/1.8;45  
  
    var resultElement = document.getElementById("resultParagraph");46  
    var resultString = centigrade.toFixed() + " degrees Centigrade";47  
    resultElement.innerText = resultString;48  
}
```

All the variables in this function will be destroyed once the function has finished. They are described as *local* because they are local to the function. This is how the JavaScript manages variables created using var. Most of the time this is exactly what you want. We don’t want the program to use any values left over from a previous use of the function. However, the totalizer program needs to retain the total value for use in successive calls of the function that adds values to it. In other words; we want to write some code like this.

---

<sup>42</sup> Get a reference to the input element

<sup>43</sup> Get the text from the input element

<sup>44</sup> Convert the text into a numeric value

<sup>45</sup> Convert the value into Centigrade

<sup>46</sup> Get a reference to the output element

<sup>47</sup> Build the result string

<sup>48</sup> Put the result string on the output element

```
var total=0; // Global variable to hold the total

/* This function reads the value from the valueText element
   and adds it to the global total value */
function doAddToTotal() {

  var valueElement = document.getElementById("valueText");
  var valueText = valueElement.value;
  var value = Number(valueText);

  total = total + value; // update the global total value

  // Display the updated total
  var resultElement = document.getElementById("resultParagraph");
  var resultString = "Total is " + total;
  resultElement.innerText = resultString;
}
```

The variable `total` is special. It exists outside any JavaScript functions. We call it a *global* variable because it can be used by any function in my application. I've added a comment above the declaration of the `total` variable. This is because I like my global variables to stand out in the code.

#### PROGRAMMER'S POINT

#### Global variables are a necessary evil

If you talk to some programmers, they might tell you that your programs should never use global variables. This is because a global variable represents a possible failure point that is out of your control. I can be sure that all the variables in my functions contain correct values. This is because each time a function runs it makes clean new copies of every variable. But a global variable exists outside my functions. I can't regard it as "clean" because I don't know what other functions might have been doing with it. Mistakes by other programmers could make my functions do the wrong thing, and that is bad. If another function changes the contents of `total` my function could display a result which is incorrect. However, in the case of the Totalizer program, a global variable is the simplest way I can make it work.

Programmers talk about functions having *side-effects*. These are things that the function does which change the state of the system in which they are running. In the case of the Totalizer, the `doAddToTotal` function has a side-effect which increases the value of `total` by the amount entered by the user. This is a side-effect that is present by design. It is important to avoid unintended side-effects.

#### Code Analysis

## Global variables and side-effects

It is important that you understand the difference between local and global variables. So here are some questions you might have considered.

Question: When is a global variable created?

Answer: A global variable is created by the browser when the web page is loaded by the browser. The `total` variable is set to zero when it is created.

```
var total=0;
```

In Chapter 3, in the section “Create a ticking clock” we discovered a JavaScript function called `onload` which runs when a web page is loaded. A program could initialize (but of course not declare) global variables in that function.

Question: Is the value of a global variable retained if I re-load the web page?

Answer: No. When the page is reloaded the JavaScript environment is reset and new global variables are created.

Question: Is a single global variable shared between multiple tabs of the same page being viewed in a single browser? In other words, if I opened several views of the Totalizer program, would the value of `total` be shared between them?

Answer: No. Each web page runs a separate JavaScript environment.

Question: Do any other functions in the Totalizer have side-effects?

Answer: Yes. The function that is called to clear the total back to zero will set the value of `total` back to zero.

```
/* This function clears the total value and updates the display
*/
function doZeroTotal() {
    total=0; // set the global total to 0

    // update the display
    var resultElement = document.getElementById("resultParagraph");
    resultElement.innerHTML = "Total is 0";
```

This function sets the value of `total` to zero and then updates the display to reflect this.

Question: How could I create the Totalizer without using a global variable?

Answer: Later in the book we will discover a way of creating variables that “hang around” even when the function that created them has finished running. We will use this JavaScript feature to make a “total manager” that managed the total value in our solution.

You can find my version of the Totalizer program in the sample folder **Ch04 Working with data\Ch04-08 Totalizer**. This includes the stylesheet that sets up the requested color scheme.

**MAKE SOMETHING HAPPEN**

## Make some party games

There is no better way to show off your programming skills than by using them to make some silly party games. At least, that's what I think. We can create a good-looking party game using our skills with CSS and JavaScript. The basis of many games is randomness. We know how to use JavaScript to create random numbers, let's see if we can make some games using this.

### "Nerves of Steel"



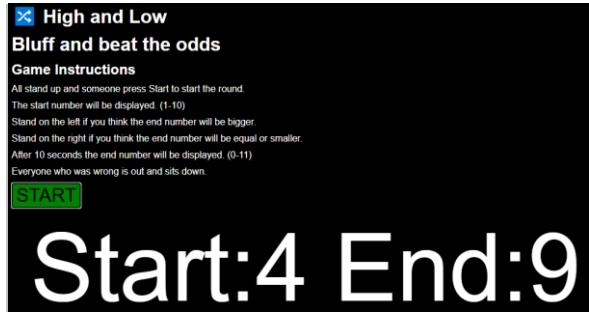
Ch04\_inset\_04\_01

We can use our ability to make random numbers, coupled with the `setTimeout` function we used to make egg timers in chapter 2 to create a "Nerves of Steel" party game. The game works like this:

4. One player presses the "Start Game" button.
5. The program displays "Players stand."
6. The program then pauses for a random time between 5 and 20 seconds. While the program is paused, players can sit down. The players need to keep track of the last person to sit down.
7. When the time interval expires the program displays "Last to sit down wins". Players still standing are eliminated and the winner is the last person to sit down.

This is a variant of the egg timer program from chapter 2. Rather than set a timeout for a fixed duration (5 minutes) the program selects a random time for the duration. You can make the game properly skillful if the game displays the selected time at the start of the game. You can also improve it using sound effects to mark the start and end of the timeout session.

## High Low



Ch04\_inset\_04\_02

Another way of using random numbers is to create a "High-Low" game. The game works like this:

1. One player presses the "Next Round" button.
2. The program displays a number between 1 and 10, inclusively.
3. The program then sleeps for 20 seconds. While the program is asleep, the players are invited to decide whether the next number will be higher or lower than the number just printed. Players who choose "high" stand on the right. Players who choose "low" stand on the left.
4. The program then displays a second number between 1 and 10, and anyone who was wrong is eliminated from the game. The program is then re-run with the players that are left until you have a winner.

This game can get very tactical, with players taking a chance on an unlikely number just so that they will be one of the people to go forward to the next round.

## Adding sound

You can add sound to the game. You can have a ticking clock sound effect while the players are waiting and a "ding" sound when the timer expires. To do this the program will have to start and stop the audio playback. It will also have to "reset" the ticking sound so that it plays from the beginning of the sound sample. A sound in HTML provides `play` and `pause` methods that can be used to control playback. It also provides a `currentType` property that a program can use to read or set. This is the `tickAudio` element in my version of the program:

```
<audio id="tickAudio">
  <source src="tick-track.mp3" type="audio/mpeg">
</audio>
```

I can start the playback as follows. The first statement sets the playback position at 0 so that the playback begins at the start of sound. The second statement plays the sample.

```
tickSoundElement.currentTime=0;  
tickSoundElement.play();
```

When I want the ticking sound to stop (when the timer has expired) I use the following:

```
tickSoundElement.pause();
```

Note that there is no command to stop the playback of a sound. You can find my versions of the games in the folders **Ch04 Working with data\Ch04-09 Nerves of Steel** and **Ch04 Working with data\Ch04-10 High and Low**. You can use these as the basis of any other games you might like to create.

## What you have learned

This chapter has given you a good understanding of how JavaScript programs store and manipulate data. Here are the major points you have covered in the chapter:

- A computer is fundamentally a data processor. A program receives data, does something with it and outputs more data. What the computer does with the data is determined by the program itself.
- The JavaScript programs that we have created take data in from input elements and display data by updating the text displayed by a paragraph element
- Data processing in a computer is performed by the evaluation of expressions. An expression contains operands (values to be worked with) and operators (which specify what is to be done with the operands). In the expression  $2+3$  the operands are the literal values 2 and 3. The operator is  $+$ .
- A program is unaware of the nature of the data that it is processing. Meaning is added when humans interpret the data as information.
- A variable is a named storage location which holds a value that a program is working with. Variables can be created using `var`.
- Variables can be used as operands in expressions. The assignment operator is used to set a value in a variable.
- Variables can be created before they are used, in which case they contain a value denoted as “`undefined`”. When assigned to a value a variable will acquire a type appropriate to the value. The two types we have used so far are `number` (a numeric value) and `string` (a string of text)
- Operators in an expression are evaluated according to their priority. This priority can be overridden by using brackets.

- Numbers in a program can be real numbers (with a fractional part) or whole numbers (integers). JavaScript provides Math functions for creating whole numbers from floating point ones. The `floor` function removes the fractional part, whereas the `round` function rounds up to the nearest whole number.
- Text in program source code can be delimited by the double quote ("), single quote (' ) or backtick (` ) characters. Escape characters are used in strings for the entry of delimiter characters and some non-printing characters.
- In an expression involves string and numeric values the numeric values are automatically converted into strings.
- The JavaScript function `Number` is used to convert a string into a numeric value. If this conversion is not possible the `Number` function returns the value "NaN" (not a number). All numeric calculations involving the value NaN will generate a result of NaN.
- Comments can be added to a program by delimiting them with /\* and \*/ character sequences. A single line comment is started by the character sequence // and extends to the end of the line containing the comment.
- A variable is made global by declaring it outside any JavaScript function. Global variables are useful because they allow values to be shared between functions, but they should be used with care as they represent a way in which a fault in one function could set a global variable to a value which could cause other functions to fail. Changes made by a function to the contents of a global variable are known as "side-effects".

To reinforce your understanding of this chapter, you might want to consider the following "profound questions" about variables.

Do all computer programs have to have an input?

Most programs have an input that they use to produce an output. However, not all programs have an input. The "Nerves of Steel" and "High Low" games get their input from random numbers before displaying them.

What is the difference between "undefined" and "not a number"?

A JavaScript variable normally holds a value. However, it can also hold the special values "undefined" and "not a number". A variable is "undefined" if it has been created but it has not been assigned a value yet. The value "NaN" (not a number) is used to represent a situation when a calculation did not create a numeric result. The `Number` function is used to convert a string of text into a number and will return a result of "NaN" if conversion was impossible because the supplied string does not contain digit characters.

Do long variable names slow my program down?

No. But they do speed up the process of understanding what the program does.

What is the difference between an operator and an operand?

An operator is a doing thing (in the English language) the verbs are a bit like the operators. An operand is the thing that is operated on. In the sentence "The cat sat on the mat" I would say that the operands are cat and mat, and the operator is "sat".

What is the difference between a real number and a whole number?

A real number has a fractional part. The value of Pi is a real number as it has a fractional component (3.1416). A whole number has no fractional part. Whole number values are used in programs for things like counting (how many sheep in a field). Real numbers are used for calculated values (the average weight of the sheep in a field).

What is the length of the longest string that a program can hold?

A string can be very, very, long. You could store an entire book in a single string if you wish.

Can I create a global variable inside a JavaScript function?

No. The important thing about a global variable is that it exists outside all functions. The variables created inside a function body are discarded when the function stops running. However, we sometimes need variables whose value is persisted between function calls. These are declared as global and can be accessed by all functions.

How can I create a string that contains the double quote characters that delimit it?

My programs usually use double quotes to mark the start and end of strings in the program text. So I would write "Rob Miles". If I want to include a double quote in the string there are two ways I can do this. The first is to use an escape sequence ("") in the string. The other way would be to use either single quote ('') or back tick (`) to delimit the string with the double quote characters in it.

5

# Making Decisions in Programs

## What you will learn

I've described a computer as like a sausage machine that accepts an input, does something with it, and then produces an output. This is a great way to start thinking about computers, but a computer does a lot more than that. Unlike a real-life sausage machine, which simply tries to create sausage from anything you put in it, a computer can respond to different inputs in different ways. In this chapter, you'll learn how to make your programs respond to different inputs. You'll also learn about the responsibility that comes with making the computer work in this way because you must be sure that the decisions your programs make are sensible.

## Boolean thinking

In Chapter 4, you learned that programs use variables to represent different types of data. I like to think that you will forever associate the number of hairs on your head with whole numbers (integers) and the average length of your hair with real numbers (floating-point values). Now it's time to meet another way of looking at data values: Boolean. Data that is Boolean can only have one of two values; true and false. You could use a Boolean value to represent whether a given person has any hair.

### Boolean values in JavaScript

A program can create variables that can hold Boolean values. As with other JavaScript data types, JavaScript will deduce the type of a variable from the context in which it is used.

```
var itIsTimeToGetUp = true
```

The above statement creates a variable called `itIsTimeToGetUp` and sets its value to `true`. In my world, it seems that it is always time to get up. In the highly unlikely event of me ever being allowed to stay in bed, we can change the

assignment to set the value to `false`:

```
var itIsTimeToGetUp = false
```

The words `true` and `false` are *keywords*. These are words that are “built in” to JavaScript. There are 63 different keywords. You’ve already seen a few of them, for example `function` is a keyword. When JavaScript sees the keywords `true` or `false` it thinks in terms of Boolean values.

JavaScript regards values that are numbers or text as being either “truthy” or “falsy”. Values are regarded as “truthy” unless they are zero, an empty string, Not a Number (NaN) or undefined – in which case they are “falsy”.

## Code Analysis

### Boolean values

Boolean values are a new type of data that a program can manipulate. But of course we have questions about them.

Question: What do you think would happen if you displayed the contents of a Boolean value?

```
alert(itIsTimeToGetUp)
```

Answer: When you display any value, JavaScript will try to convert that value into something sensible for us to look at. In the case of Boolean values, it will display “true” or “false”.

This page says

true

**ok**

Ch05\_inset\_01\_01

Question: Is there a JavaScript function called `Boolean` that will convert things into boolean values, just like there is a `Number` function that will convert things into a number?

Answer: Indeed, there is. The `Boolean` function applies the rules of “truthy” and “falsy” to the value supplied to it.

```
> Boolean(1)
<- true
> Boolean(0)
<- false
```

Applying `Boolean` to 0 gives the result `false`. Any other numeric value would be regarded as `true`.

Question: Are negative numbers regarded as `false`?

Answer: No. It is best to regard “truthy” as meaning “the presence of a value” rather than something which is positive or negative. Applying `Boolean` to a negative number will produce a result of `true`.

```
> Boolean(-1)
<- true
```

Question: Is the string “false” regarded as `true`?

Answer: Yes. If you understand this you can call yourself a “Truthy Ninja”. Any string other than an empty string is regarded as true.

```
> Boolean("false")
<- true
> Boolean("")
<- false
```

Question: Is the value “infinity” regarded as `true` or `false`?

Answer: In our first encounter with JavaScript in chapter 1 we tried dividing 1 by zero to see what happened. We discovered that this type of invalid calculation produces a value of “Infinity” as a result. The best way to discover whether Infinity is true or false is to ask JavaScript:

```
> Boolean(1/0)
<- true
```

The calculation `1/0` generates a result of `infinity`, which is regarded by the `Boolean` function as `true`.

Question: We know that if we ask JavaScript to perform a silly calculation, for example divide a number by a string, the result is a special value called “Not a number” (`NaN`). Is `NaN` regarded as `true` or `false`?

Answer: JavaScript regards “Not a number” as `false`. Dividing the number `1` by the string “`Rob`” produces `NaN` as a result. If we feed this into the `Boolean` function it decides that this value is `false`.

```
> Boolean(1/"Rob")
<- false
```

Question: What happens if a program combines Boolean values with other values?

Answer: In Chapter 4, in the section “Working with strings and numbers” we discovered that when we combine numbers with strings the numeric value is automatically “promoted” to a string. Something similar happens when logical values are combined with values of other types. A value which is true equates to the number `1` and the string “`true`”. A value of false equates to `0` and “`false`”.

```
> 1 + true
<- 2
> "hello" + true
<- "hellotrue"
```

Note that, as with numbers, this conversion does not work the other way. Just as we had to use the `Number` function to convert a string into a number, we have to use the `Boolean` function to convert values of other types into values that obey the “truthy” and “falsy” rules.

In Chapter 3 we created a ticking clock that displayed the time. The program in Chapter 3 used the `Date` object that JavaScript provides to get the current date and time.

```
var currentDate = new Date();
var hours = currentDate.getHours();
var mins = currentDate.getMinutes();
var secs = currentDate.getSeconds();
```

The statements above get the `hours`, `mins` and `secs` values for the current time. We could use these values to write some JavaScript that would decide whether or not I should get up.

## Boolean expressions

We've said that JavaScript expressions are made up of operators (which identify the operation) and operands (which identify the items being processed). **Figure 5-2** shows our first expression, which worked out the calculation,  $2+2$ .

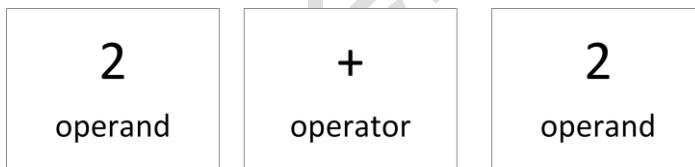


Figure 5.11 An arithmetic operator

An expression can contain a comparison operator (**Figure 5-3**):



Figure 5.12 A comparison operator

An expression containing a comparison operator evaluates to a result that is either `true` or `false`. The `>` operator in this expression means “greater than.” If you read the expression aloud, you say “hour greater than six.” In other

words, this expression is `true` if the hour value is greater than 6. I need to get up after 7 o'clock, so this is what I need for my alarm clock. Expressions which return either true or false are called *logical expressions*.

## Comparison operators

These are the comparison operators that you can use in JavaScript programs.

Operator	Name	Effect
<code>&gt;</code>	Greater than	True if the value on the left is greater than the value on the right
<code>&lt;</code>	Less than	True if the value on the left is less than the value on the right
<code>&gt;=</code>	Greater than or equals	True if the value on the left is greater than or equal to the value on the right
<code>&lt;=</code>	Less than or equals	True if the value on the left is less than or equal to the value on the right
<code>==</code>	Equals	True if the value on the left is equal to the value on the right
<code>!=</code>	Not equals	True if the value on the left is not equal to the value on the right

A program can use comparison operators in an expression to set a Boolean value.

```
itIsTimeToGetUp = hours > 6
```

This statement will set the variable `itIsTimeToGetUp` to the value `true` if the value in `hours` is greater than 6 and `false` if the value in `hours` is not greater than 6. If this seems hard to understand, try reading the statement and listen to how it sounds. “`itIsTimeToGetUp` equals hour greater than six” is a good explanation of the action of this statement.

### Code Analysis

## Examining comparison operators

Question: How does the equality operator work?

Answer: The equality operator evaluates to `true` if the two operands hold the same value.

```
> 1==1  
<- true
```

The equality operator can be used to compare strings and Boolean values too.

```
> "Rob"=="Rob"  
<- true  
> true == true  
<- true
```

Question: How do I remember which relational operator is which?

Answer: When I was learning to program, I associated the `<` in the `<=` operator with the letter L, which reminded

me that `<=` means “less than or equal to.”

Question: Can we apply relational operators between other types of expressions?

Answer: Yes, we can. If a relational operator is applied between two string operands, it uses an alphabetic comparison to determine the order.

```
> "Alice" < "Brian"  
<- true
```

JavaScript returned `true` because the name Alice appears alphabetically before Brian.

What could go wrong?

## Equality and floating-point values

In Chapter 4, we saw that a floating-point number is sometimes only an approximation of the real number value our program is using. In other words, some numbers are not stored precisely.

This approximation of real number values can lead to serious problems when we write programs that test to see whether two variables hold the same floating-point values. Consider the following statements, which I’ve typed into the Developer View in the Edge browser:

```
> var x = 0.3  
> var y = 0.1 + 0.2
```

These statements create two variables, `x` and `y`, which should both hold the value `0.3`. The variable `x` has the value `0.3` directly assigned, whereas the second variable, `y`, gets the value `0.3` as the result of a calculation that works out the result of `0.1 + 0.2`. What do you think we will see if we test the two variables for equality?

```
> x == y  
<- false
```

This expression uses the equality operator (`==`) which will produce a result of `true` if its two operands hold the same value. However, JavaScript decides that `x` and `y` are different because the variable `x` holds the value `0.3` and the variable `y` holds the value `0.30000000000000004`. This illustrates a problem with program code that compares floating-point values to determine whether they are equal. The tiny floating-point errors mean values we think are the same do not always evaluate that way.

If a program needs to compare two floating-point values for equality, the best approach is to decide they are equal if they differ by only a very small amount. If you don’t do this, you might find that your programs don’t behave as you might expect.

The date and time values returned from the JavaScript `Date` object are supplied as integers so you can test these for equality without problems.

## Logical operators

At the moment, my test to determine whether it is time to get up is only controlled by the hour value of the time.

```
itIsTimeToGetUp = hours > 6
```

The above statement sets the value of `itIsTimeToGetUp` to `true` if the hour is greater than 6 (i.e. from seven o'clock onwards) but we might want to get up at seven thirty. To be able to do this we need a way of testing for a time when the hour is greater than 6 and the minute is greater than 29. JavaScript provides three logical operators we can use to work with logical values. Perhaps they can help solve this problem.

Operator	Effect
!	Evaluates to True if the operand it is working on is False Evaluates to False if the operand it is working on is True
&&	Evaluates to True if the left-hand value and the right-hand value are True
	Evaluates to True if the left-hand value and the right-hand value are True

The `&&` (and) operator is applied between two Booleans value and returns `true` if both values are `true`. There is also an `||` (or) operator which is applied between two Boolean values and returns `True` if one of the values is `True`. The third operator is the operator `!` (not), which can be used to invert a Boolean value.

### Code Analysis

#### Logical operators

We can investigate the behavior of logical operators by using the Developer View in the Edge browser. We can just type in expressions and see how they evaluate. Please don't be confused by the way that the `<` and `>` characters are used in the developer console samples below.

Question: What does the following expression evaluate to?

```
<- !true
```

Answer: The effect of `!` is to invert a Boolean value, turning the `true` into a `false`.

```
> !true  
<- false
```

Question: How about this expression?

```
> true && true
```

Answer: The operands each side of the `&&` are `true`, so the result evaluates to `true`.

```
> true && true  
<- true
```

Question: What about the following expression?

```
> true && false
```

Answer: Because both sides (operands) of the `&&` (and) operator need to be `true` for the result to be `true`, you shouldn't be surprised to see a result of `false` here.

```
> true && false  
<- false
```

Question: What about the following expression?

```
> true || false
```

Answer: Because only one side of an `||` (or) operator needs to be `true` for the result to be `true`, the expression evaluates to `true`.

```
> true || false  
<- true
```

Question: So far, the examples have used only Boolean values. What happens when we start to combine Boolean and numeric values?

```
> true && 1
```

Answer: It turns out that JavaScript is quite happy to use and combine logical and numeric values. The above combination would not return `true`, however. Instead, it will return `1`.

```
> true && 1  
<- 1
```

This looks a bit confusing, but it gives us an insight into how JavaScript evaluates our logical expression. JavaScript will start at the beginning of a logical expression and then work along the expression until it finds the first value it can use to determine the result of the expression. It will then return that value.

In the above expression, when JavaScript sees the left-hand operand is `true`, it says to itself "Aha. The value of the `&&` (and) expression is now determined by the right-hand value. If the right-hand value is `true`, the result is `true`. If the right-hand value is `false`, the result is `false`." So, the expression simply returns the right-hand operand. We can test this behavior by reversing the order of the operands:

```
> 1 && true  
<- true
```

We know that any value other than `0` is `true`, so JavaScript will return the right-hand operand, which in this case is `True`. We can see this behavior with the `||` (or) operation, too. JavaScript only looks at the operands of a logical

`&&` operator until it can determine whether the result is `true` or `false`.

```
> 1 || false  
<- 1  
> 0 || True  
<- True
```

You might wish to experiment with other values to confirm that you understand what is happening.

We want to make some JavaScript that takes in hour and minute values and decides whether or not to sound the alarm. We could try to make an alarm that triggers after 7:30 by writing the following statement:

```
var itsTimeToGetUp= hours>6 && minutes>29
```

The `&&` (and) operator is applied between the result of two Boolean expressions and returns `true` if both of the expressions evaluate to true. The above statement would set the variable `itsTimeToGetUp` to `true` if the value in `hours` is greater than 6 and the value in `minutes` is greater than 29, which you might think is what we want. However, this statement is incorrect. We can discover the bug by designing some tests:

Hours	Minutes	Required Result	Observed Result
6	0	False	False
7	29	False	False
7	30	True	True
8	0	True	False

The table shows four times, along with the required result (i.e. what should happen) and the observed result (i.e. what does happen). One of the times has been observed to work incorrectly. When the time is 8:00 the value of `itsTimeToGetUp` is set to `false`, which is wrong.

The condition we are using evaluates to `true` if the `hours` value is greater than 6 and the `minutes` value is greater than 29. This means that the condition evaluates to `false` for any minute value which is less than 29, meaning it is `false` at 8:00. To fix the problem we need to develop a slightly more complex test:

```
var itsTimeToGetUp= (hours>7) || (hours==7 && minutes>29)
```

I've added parenthesis to show how the two tests are combined by the `||` (or) operator. If the value of `hours` is greater than 7 we don't care what the value of `minutes` is. If the `hours` is equal to 7 we need to test that the `minutes` is greater than 29. If you try the values in the table with the above statement you will find that it works correctly. This illustrates an important point when designing code intended to perform logic like this. You need to design tests which you can use to ensure that the program will do what you want.

# The if construction

Suppose I want to make a program that will display a message telling me if it's time to get out of bed. We can use the Boolean value we just created to control the execution of programs by using JavaScript's `if` construction.

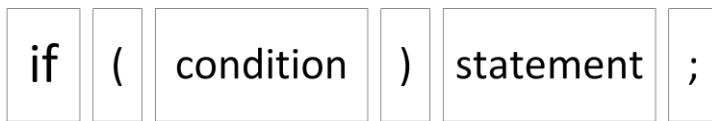


Figure 5.13 If construction

The figure above shows how an if construction fit together. The condition controls the execution of the statement. In other words, if the condition is "truthy" the statement is performed, otherwise it is not.

```
if (itIsTimeToGetUp) alert("It is time to get up!");
```

The statement above would display an alert if it were time to get up. You can see this in use in the example [Ch05 Making Decisions in Programs\Ch05-01 Alarm Alert](#) which displays the alert if you visit the page after 7:30 in the morning.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Ch05-01 Alarm Alert</title>
</head>

<body onload="doCheckAlarm()">
    <script>
        function doCheckAlarm() {
            var currentDate = new Date();
            var hours = currentDate.getHours();
            var mins = currentDate.getMinutes();
            var itIsTimeToGetUp = (hours>7) || (hours==7 && minutes>20);
        }
    </script>
</body>
</html>
```

```
        if (itIsTimeToGetUp) alert("It is time to get up!");49  
    }  
</script>  
</body>  
</html>
```

This is the full text of the alarm alert page. Note that the statement that implements the “intelligence” of the program is only one tiny part of the code.

The behavior of the if construction is controlled by the condition. The condition does not have to be a variable, it can also be a logical expression:

```
if ((hours>7) || (hours==7 && minutes>29)) alert("It is time to get up!");
```

This statement removes the need for the `itIsTimeToGetUp` variable. However, I quite like using the variable as it helps the user understand what the program is doing.

## Adding an else part

Many programs want to perform one action if a condition is `true` and another action if the condition is `false`. The `if` construction can have an `else` element added which identifies a statement to be performed if the condition is `false`.



Figure 5.14 If construction with else

The else part is added onto the end of a conditional statement. It comprises the keyword `else` followed by the statement to be performed if the condition is `false`. We could use it to make our alert program display a message when we can stay in bed.

---

<sup>49</sup> The if construction that controls the alert

```
if(itIsTimeToGetUp)
    alert("It is time to get up!");
else
    alert("You can stay in bed");
```

This program displays a different message depending on the time of day that the user runs the program. Note that although I've spread the statement over several lines, the content matches the structure in Figure 5.3

## CODE ANALYSIS

### If constructions

Question: Must an if construction have an else part?

Answer: No. They are very useful sometimes, but it depends on the problem that the program is trying to solve.

Question: What happens if a condition is never true?

Answer: If a condition is never true the statement controlled by the condition never gets to run.

Question: Why is the statement underneath the if condition in my example indented by a few spaces?

Answer: This statement doesn't need to be indented. JavaScript would be able to understand what we want the program to do even if we put everything on one line. The indentation is there to make the program easier to understand. It shows that the statement underneath the if construction is being controlled by the condition above it. Indenting like this is such common practice that you will find the behavior "baked in" to the Visual Studio Code editor. In other words, if you start typing an if construction and press the Enter key at the end of the condition part, Visual Studio Code will automatically indent the next line.

## Creating blocks of statements

The if condition controls the execution of a single statement. However, sometimes you might want to perform multiple statements if a condition is true. For example, we might want to play an alarm sound as well as displaying an alert message when it is time to get up. To do this, a program needs to control multiple statements from a single condition. You write code for a task like this by creating a block of statements.

A block of statements is a sequence of JavaScript statements enclosed in a pair of curly braces—the { and } characters. You have already seen blocks of statements in the programs we've examined and written; in those programs, the statements in all the functions are enclosed in a block. You can create a block anywhere in a program, and it is equivalent to a single statement.

```
if (itIsTimeToGetUp) {
    alarmAudio.play();
```

```
    outputElement.textContent = "It is time to get up!";
}
else {
    outputElement.textContent = "You can stay in bed";
}
```

The code above displays a message and plays an alarm sound. You can find the working program in the example **Ch05 Making Decisions in Programs\Ch05-03 Alarm Alert with sound block**. This uses the “everything sound” as an alarm, which some might feel a bit harsh, but it certainly wakes me up. Note that in the above code I’ve used curly braces to enclose the statements for both the if and else parts of the condition, even though there is no need to do this for the else part as it only contains one statement. I do this to make it clear what is going on. It also means that it is easier to add extra statements controlled by the else part as I can just put them inside the block.

## Use decisions to make an application

Now that you know how to make decisions in your programs, you can start to make more useful software. Let’s say your next-door neighbor is the owner of a theme park and has a job for you. Some rides at the theme park are restricted to people by age, and he wants to install some computers around his theme park so that people can find out which rides they may go on. He needs some software for the computers, and he’s offering a season pass to the park if you can come up with the goods, which is a very tempting proposition. He provides you with the following information about the rides at his park:

Ride Name	Minimum Age Requirement
Scenic River Cruise	None
Carnival Carousel	At least 3 years old
Jungle Adventure Water Splash	At least 6 years old
Downhill Mountain Run	At least 12 years old
The Regurgitator (a super scary roller coaster)	Must be at least 12 years old and less than 70

You discuss with him the design of the program *user interface*. The user interface is what people see when they use the program, and the steps that they go through when they are using it. In this application users will specify the ride they want to go on and enter their age. They then click a button and be told whether they can go on that ride:

# CRAZYADVENTUREWONDERFUNLAND

These are the rides that are available

- 1. Scenic River Cruise
- 2. Carnival Carousel
- 3. Jungle Adventure Water Splash
- 4. Downhill Mountain Run
- 5. The Regurgitator

Enter the number of the ride you want to go on:

Enter your age:

You can go on the Scenic River Cruise

Figure 5.15 Theme Park Rides



## Programmer's Point

### Design the user Interface with the customer

You might think that an interface like this would be simple to design and that the customer will have no strong opinions on how the user interface looks and functions. I've found this to be wrong. I've had the awful experience of proudly showing my finished solution to a customer only to be told that it was "Not what they wanted" and "Hard to use." I now understand that this was my fault. Rather than showing only my finished design, I should have created the design with the customer. That would have saved me a lot of work.

## Build the user interface

The first thing we need to do is create the HTML web page and the stylesheet for the application. In Chapter 3 we decided that it was a good idea to separate the stylesheet file which holds the style of the page elements from the page layout. It is also a good idea to separate the JavaScript program code from the HTML layout. We do this by putting the JavaScript into a file with the language extension ".js". We can then add an element in the head of the HTML file that specifies this filename:

```
<script src="themepark.js" ></script>
```

The HTML above is added to the `<head>` part of an HTML document and includes the contents of the JavaScript file "themepark.js" in an HTML page. You can see it in use in the HTML below.

```
<!DOCTYPE html>
<html lang="en">

<head>
    <title>Ch05-04 Theme Park Ride Selector </title>
    <link rel="stylesheet" href="styles.css">50
    <script src = "themepark.js" ></script>51
</head>

<body>
    <p class="menuHeading">CRAZYADVENTUREWONDERFUNLAND</p>
    <p class="menuText">These are the rides that are available</p>
    <ol class="menuRideList">52
        <li>Scenic River Cruise</li>
        <li>Carnival Carousel</li>
        <li>Jungle Adventure Water Splash</li>
        <li>Downhill Mountain Run</li>
        <li>The Regurgitator</li>
    </ol>
    <p class="menuText">Enter the number of the ride you want to go on:
        <input class="menuInput" type="number" id="rideNoText" value="1" min="1" max="5"> </p>
    <p class="menuText">Enter your age:
        <input class="menuInput" type="number" id="ageText" value="18" min="0" max="100"> </p>
    <button class="menuButton" onClick="doCheckAge()">Check your age</button>
    <p class="menuAnswer" id="menuAnswerPar"></p>
</body>
</html>
```

This is the HTML file for the ThemePark Ride Selector application. It uses some features of HTML that we've not seen before. We can create an ordered list of items by using the `<ol>` tag to enclose some `<li>` list elements. The browser

---

<sup>50</sup> Include the CSS file

<sup>51</sup> Include the JavaScript file

<sup>52</sup> Start of numbered list of rides

will automatically number the elements for us. Each element on the page is assigned a class that has a specific style. The settings for each of the styles are in a separate CSS stylesheet file called styles.css. A part of this file is given below:

```
.menuHeading {  
    font-size: 4em;  
    font-family: Impact, Haettenschweiler, 'Arial Narrow Bold', sans-serif;  
    color: red;  
    text-shadow: 3px 3px 0 blue, 10px 10px 10px darkblue;53  
}  
  
.menuText,.menuRideList, .menuButton, .menuInput, .menuYes, .menuNo54  
{  
    font-family: Arial, Helvetica, sans-serif;  
    font-size: 2em;  
    color: black;  
}  
  
.menuYes  
{  
    margin: 30px;  
    color: green;  
}  
... remainder of classes are defined here ...
```

The `menuHeading` class is in the HTML used to format the heading. It uses an impact font and adds two shadows to the text. The first shadow is blue and close to the text, providing a 3D effect for each character. The second shadow is more diffuse and darker blue so that it makes the characters appear to stand out from the page. You can see the effect in Figure 5.5. Each shadow is defined by a color value pre-ceeded by three values. The first two values give the x and y offsets of the shadow from the text. The third value gives how "diffuse" the shadow is. The first shadow is not diffused at all, whereas the second has a diffusion size of 10px, leading to the text as shown in Figure 5.5 above.

The stylesheet also applies some shared settings to all the menu input classes. This means that the font is set once for all those classes, making it easy to change the font if required. It is a good idea to group classes in this way if they all have a set of common characteristics. Remember that classes accumulate setting values which are then used on the HTML elements that are assigned to that class. So, for example, the `menuYes` class will bring together the following settings, some from those shared by other menu settings, and some specific to that class:

---

<sup>53</sup> This style adds shadows to the text

<sup>54</sup> These settings will be applied to all the classes.

```
font-family:Arial, Helvetica, sans-serif;  
font-size: 2em;  
  
color: green;
```

## Add the code

Now that we have the user interface complete we need to add the JavaScript that implements the behavior that the application needs. When the user presses the button to check their age the `doCheckAge` function runs. This function gets values for the number of the selected ride and the age of the person wishing to use it. The function then tests these values to see if the combination is valid or not. The first part of this function works in the same way as the add-ing machine that we created earlier. It fetches text from the input elements in the HTML and uses the `Number` func-tion to convert the text into a number.

```
var rideNoElement = document.getElementById("rideNoText");  
var rideNoText = rideNoElement.value;  
var rideNo = Number(rideNoText);55  
  
var ageElement = document.getElementById("ageText");  
var ageText = ageElement.value;  
var ageNo = Number(ageText);56
```

When these statements have completed the variables `rideNo` and `ageNo` contain the number of the ride and the age of the guest. The next thing the function does is get a reference to the paragraph that will be used to display the re-sult. If you look at the HTML for the user interface you'll see that this paragraph has the ID `menuAnswerPar`:

```
var resultElement = document.getElementById("menuAnswerPar");
```

Now that the program has the input data and somewhere to put the output it can make decisions about the use of the rides. If the user has selected ride number 1 there are no age restrictions for the Scenic River Cruise, so this code is a single test for a ride number of 1. If the user has selected this ride we set the style class for the result element to the `menuYes` class. This has the effect of changing the style of that element so that the text is now green. Then the

---

<sup>55</sup> Get the ride number that was entered

<sup>56</sup> Get the age of the person

inner text for the paragraph is set to "You can go on the Scenic River Cruise" so that this is displayed for the user.

```
if(rideNo==1) {  
    // This is the Scenic River Cruise  
    // There are no age limits for this ride  
    resultElement.className="menuYes";  
    resultElement.innerText = "You can go on the Scenic River Cruise";  
}
```

If the user has not selected ride number 1 the program must test to see if ride number 2 has been picked.

```
if(rideNo==2) {  
    // This is the Carnival Carousel  
    // riders must be 3 or over  
    if(ageNo<3) {  
        resultElement.className="menuNo";  
        resultElement.innerText = "You are too young for the Carnival Carousel";  
    }  
    else {  
        resultElement.className="menuYes";  
        resultElement.innerText = "You can go on the Carnival Carousel";  
    }  
}
```

The code above shows how this works. The program works by *nesting* one conditional statement inside another. Note how I've used the layout of the program to make it clear which statements are controlled by which condition.

Now that you have code that works for the Carnival Carousel, you can use it as the basis for the code that handles some of the other rides. To make the program work correctly for the Jungle Adventure Water Splash, you need to check for a different ride number and confirm or reject the user based on a different age value. Remember that for this ride, a visitor must be at least six years old.

```
if(rideNo==3) {  
    // This is the Jungle Adventure Water Splash  
    if(ageNo<6) {  
        resultElement.className="menuNo";  
        resultElement.innerText = "You are too young for the Jungle Adventure Water Splash";  
    }  
    else {
```

```
        resultElement.className="menuYes";
        resultElement.innerText = "You can go on the Jungle Adventure Water Splash";
    }
}
```

You can implement the Downhill Mountain Run very easily by using the same pattern as for the previous two rides. But the final ride, The Regurgitator, is the most difficult. The ride is so extreme that the owner of the theme park is concerned for the health of older people who use it and has added a maximum age restriction as well as a minimum age. The program must test for users who are older than 70 as well as those who are younger than 12. We must design a sequence of conditions to deal with this situation.

The code that deals with The Regurgitator is the most complex piece of the program that we've had to write so far. To make sense of how it needs to work, you need to know more about the way that `if` constructions are used in programs. Consider the following code:

```
if(rideNo==3) {
    // This is the Regurgitator
```

The condition is true when the user has selected ride number 3, and all the statements we add to the block of code controlled by this condition will run only if the selected ride is The Regurgitator. In other words, there is no need for any statement in that block to ask the question, “Is the selected ride The Regurgitator?” because the statements are run only if this is the case. The decisions leading up to a statement in a program determine the context in which that statement will run. I like to add comments to clarify the context:

```
if(rideNo==3) {
    // This is the Regurgitator
    if(ageNo<12) {
        resultElement.className="menuNo";
        resultElement.innerText = "You are too young for the Regurgitator";
    }
    else {
        // get here if the age is 12 or above
        if(ageNo>70) {
            resultElement.className="menuNo";
            resultElement.innerText = "You are too old for the Regurgitator";
        }
        else {
            resultElement.className="menuYes";
            resultElement.innerText = "You can go on the Regurgitator";
        }
    }
}
```

```
    }  
}
```

These comments make the program slightly longer, but they also make it a lot clearer. This code is the complete construction that deals with The Regurgitator. The best way to work out what it does is to work through each statement in turn with a value for the user's age. You can find all the sample code in the folder **Ch05 Making Decisions in Programs\Ch05-04 Theme Park**

## Using the switch construction

The program code for the ride selector is a sequence of if constructions controlled by the value of `rideNo`. This pattern appears frequently in programs, so JavaScript contains an additional construction to make it slightly easier. This is something we've not seen before. Up until now everything we have learned is about making something possible. However, the switch construction is all about making something easier. Programs can use a switch to select different behaviors based on the value in a single control variable. Take a look at this code, which implements the Theme Park ride selector.

```
switch(rideNo)  
{  
    case 1:  
        // This is the Scenic River Cruise  
        // There are no age limits for this ride  
        resultElement.className = "menuYes";  
        resultElement.innerText = "You can go on the Scenic River Cruise";  
        break;  
  
    case 2:  
        // This is the Carnival Carousel  
        // .. statements for Carnival Carousel go here  
        break;  
  
    case 3:  
        // This is the Jungle Adventure Water Splash  
        // .. statements for Jungle Adventure Water Splash go here  
        break;  
  
    case 4:  
        // This is the Downhill Mountain Run  
        // .. statements for Downhill Mountain Run go here  
        break;  
  
    case 5:
```

```
// This is the Regurgitator  
// ... statements for the Regurgitator go here  
}
```

The code above shows how the switch would be used. The value in `rideNo` is used as the control value for the `switch`, and the program will select the `case` which matches the control value. You can put as many statements as you like in a particular case, but you must make sure that the last statement in the case is the `break` keyword, which ends the execution of the code for that case. You can find my switch-powered solution for the Theme Park ride selector in the [Ch05 Making Decisions in Programs\Ch05-05 Switch Theme Park Ride Selector](#) folder.

You can use the `switch` statement with strings and integer values; it can be a convenient way of selecting an option. A `switch` construction can have a `default` selector, which is obeyed if none of the cases match the control value. You can also use multiple case elements to select a particular behavior. The switch below is controlled by a variable called `commandName`. The commands "Delete", "Del", and "Erase" all result in the `erase` behavior being selected:

```
var commandName ;  
  
switch(commandName)  
{  
    case "Delete":  
    case "Del":  
    case "Erase":  
        // Erase behaviour goes here  
        break;  
  
    case "Print":  
    case "Pr":  
    case "Output":  
        // Print behaviour goes here  
        break;  
  
    default:  
        // Invalid command behaviour goes here  
        break;  
}
```

#### WHAT COULD GO WRONG

Missing breaks in switches can cause mayhem

```

switch(rideNo)
{
    case 1:
        // This is the Scenic River Cruise
        // There are no age limits for this ride
        resultElement.className = "menuYes";
        resultElement.innerText = "You can go on the Scenic River Cruise";

    case 2:
        // This is the Carnival Carousel
        if (ageNo < 3) {
            resultElement.className = "menuNo";
            resultElement.innerText = "You are too young for the Carnival Carousel";
        }
        else {
            resultElement.className = "menuYes";
            resultElement.innerText = "You can go on the Carnival Carousel";
        }
        break;
}

```

The code above is part of my switch-controlled version of the Theme Park ride selector. It has a dangerous bug in it. The bug will not cause JavaScript to crash, but it will cause the program to do the wrong thing. The bug is caused by a missing `break` keyword between `case 1` and `case 2`. When the user selects ride number 1 the program will perform the statements for `case 1` and then go straight through and perform the statements for `case 2`. This means that if the user selects the Scenic River Cruise (option 1) the program will behave as if the Carnival Carousel (option 2) was selected. Bugs like this, which cause a program to “mostly work” are particularly dangerous and you should make sure to test for every input to make sure that you have no missing breaks in your switches.

## Make Something Happen

### Improve the ride selector

You can use the application in [Ch05 Making Decisions in Programs\Ch05-04 Theme Park](#) as the starting point for a really good ride selector program. But it is not perfect. It would benefit from some testing of the input values to prevent the user from entering invalid ride numbers or age values. You could even design custom graphics for each ride and then display them when the ride is selected. You could even add suitable sound effects for each ride too.

## Fortune Teller

The `Math.random` function can be used in `if` constructions to make programs that perform in a way that appears random.

```
var resultString = "You will meet a ";
```

```
if(Math.random()>0.5)
    resultString = resultString+"tall ";
else
    resultString = resultString+"short ";

if(Math.random()>0.5)
    resultString = resultString+"blonde ";
else
    resultString = resultString+"dark ";

resultString = resultString + "stranger.";
```

The `if` constructions test the value produced by a call to the `Math.random` function. This produces a value in the range 0 to 1. If the value is less than 0.5 the program selects one option, otherwise another option is picked. This is repeated to produce a random seeming program. You can build on this sequence of such conditions to make a fun fortune teller program. You could also create some graphical images to go along with the program predictions.

## What you have learned

This chapter has introduced you to the use of Boolean values in programs and showed you how to make code that can take decisions. Here are the major points we have covered:

- Boolean data has one of only two possible values, true and false. JavaScript contains the keywords `true` and `false` that can be used to represent these values in programs.
- JavaScript can regard variables of other types in terms of their “truthy” or “falsy” nature. Any numeric value other than 0 is regarded as true. Any string other than an empty string is regarded as true. The value that represents “Not a Number” is regarded as false, but the value that represents “infinity” is regarded as true.
- The JavaScript function `Boolean` can be used to convert a variable of any type into the Boolean value `true` or `false` according to the ways of “truthy” and “falsy”.
- Programs can generate Boolean values by using comparison operators (for example `<` - less than) between values of other types. Care should be exercised when comparing floating point values (numbers with a fractional part) for equality as they may not be held accurately.
- JavaScript also provides Boolean operators (for example `&&` - and) which can be used between Boolean values.
- The `if` construction is used in programs to select statements based on Boolean expression used as a condition. The `if` construction can have an `else` part which specifies a statement to be performed if the condition is false. Conditional statements can be nested.
- JavaScript statements can be enclosed in curly braces (`{` and `}`) to create blocks that can be controlled by a single condition in an `if` construction.

- It is possible to store the JavaScript component of an application in a separate code file which is included in an HTML page.
- The switch construction is an easier way to create code which selects a behavior based on the content of a single control variable.

Here are some questions that you might like to ponder about making decisions in programs:

**Question:** Can a program test two boolean values to see if they are equal?

**Answer:** Yes it can. The equality operator (`==`) will work between two value that are either true or false.

**Question:** Can JavaScript regard any value in terms of "truthy" and "falsy"?

**Answer:** Yes. Essentially, if there something there (a value other than zero, a non-empty string) then this will be regarded as true. Otherwise it will be falsy.

**Question:** Does every `if` construction have to have an `else` part?

**Answer:** No. If an `else` is present it will "bind" to the nearest `if` construction in the program. If you write a program in which only some conditions have `else` part you need to make sure that an `else` binds to the correct `if`.

**Question:** Is there a limit to how many `if` conditions you can nest inside each other?

**Answer:** No. JavaScript will be quite happy to let you put 100 `if` statements in a row (although you would have a problem editing them). If you find yourself doing this, you might want to step back from the problem a bit and see if there is a better way of attacking the problem.

**Question:** How long can a block be?

**Answer:** A block of code can be very long indeed. You could control thousand lines of JavaScript with a single `if` condition. However, very long blocks can make code hard to understand. If you want to control a large amount of code with a condition you should put the code into a function. We will learn about functions in chapter 8.

**Question:** Does the use of Boolean values mean that a program will always do the same thing given the same data inputs?

**Answer:** It is very important that, given the same inputs, the computer does the same thing each time. If the computer starts to behave inconsistently, this makes it much less useful. When we want random behavior from a computer (for example, when writing a fortune teller program), we have to obtain values that are explicitly random and make decisions based on those. Nobody wants a "moody" computer that changes its mind (although, of course, it might be fun to try to program one using random numbers).

**Question:** Will the computer always do the right thing when we write programs that make decisions?

**Answer:** It would be great if we could guarantee that the computer will always do the right thing. However, the computer is only ever as good as the program it is running. If something happens that the program was not expecting, it might respond incorrectly. For example, if a program was working out cooking time for a bowl of soup, and the user entered ten servings rather than one, the program would set the cooking time to be far too long (and probably burn down the kitchen in the process). In that situation, you can blame the user (because they input the wrong data), but there should probably also be a test in the program that checks to see if the value

entered is sensible. If the cooker can't hold more than three servings, it would seem sensible to perform a test that limits the input to three. When you write a program, you need to "second guess" what the user might do and create decisions that make your program behave sensibly in each situation.

Pre-release

# 6

# Repeating Actions in Programs

## What you will learn

In the last chapter you learned how a program can make decisions and change its behavior according to the data that it is given. In this chapter you are going to discover how to make a program repeat a sequence of actions using the JavaScript *loop* constructions. Along the way we're going to explore some new features of HTML and JavaScript. We are also going to discover some ways that programs can go wrong and how good design can reduce the chances of a program failing.

## App development

Our starting point is the Theme Park Ride program developed in Chapter 5. This program uses conditional (*if*) constructions to display whether a rider can go on their selected ride.

# CRAZYADVENTUREWONDERFUNLAND

These are the rides that are available

1. Scenic River Cruise
2. Carnival Carousel
3. Jungle Adventure Water Splash
4. Downhill Mountain Run
5. The Regurgitator

Enter the number of the ride you want to go on:

Enter your age:

You can go on the Scenic River Cruise

Figure 6.16 Theme Park Ride Selector

Figure 6.1 shows the solution in use. The rider entered their age as 18 and selected ride number 1; the “Scenic River Cruise” which they can go on. The program works but the owner of the theme park wants changes. She wants to remove the need for the ride selection and have the program display the ride names in red or green to indicate which ones can be used for a given age. She has created a new design and wants you to re-write the application to work match.

# CRAZYADVENTUREWONDERFUNLAND

These are the rides that are available

- Scenic River Cruise
- Carnival Carousel
- Jungle Adventure Water Splash
- Downhill Mountain Run
- The Regurgitator

Enter your age:

Figure 6.17 Proposed New Design

Figure 6.2 shows how she wants the solution to work. The rider has entered their age and pressed the “Check your rides” button. The rides in green are the ones that an eight-year-old guest can go on. You discuss the work with the

theme park owner and agree a price, which involves a lot of free ice-cream, and start work.

The first thing we need to do is remove the ride numbers from the ride display. The previous application used a numbered list, because the rider had entered the number of their chosen ride. However, the new application doesn't need ride numbers.

```
<ul class="menuRideList" id="rideList">
  <li id="scenicRiver">Scenic River Cruise</li>
  <li id="carnivalCarousel">Carnival Carousel</li>
  <li id="jungleAdventure">Jungle Adventure Water Splash</li>
  <li id="downhillMountain">Downhill Mountain Run</li>
  <li id="regurgitator">The Regurgitator</li>
</ul>
```

This is the HTML that we will use to display the names of the rides in the updated application. It uses an un-numbered list element, `<ul>`, that holds a collection of list elements `<li>` for rides. Now we need to modify the program in the JavaScript that controls the behavior of the application. The program will get the age of the rider and then use this to display to reflect whether they can go on each ride.

```
var jungleAdventureElement= document.getElementById("jungleAdventure");57
if (ageNo < 6) {58
  jungleAdventureElement.className = "menuNo";59
}
else {60
  jungleAdventureElement.className = "menuYes";61
}
```

This code shows how the program displays the results for the Jungle Adventure ride. It follows a pattern that we have seen many times before:

---

<sup>57</sup> Get a reference to the ride name display

<sup>58</sup> Test the age value

<sup>59</sup> Turn red if the ride can't be used

<sup>60</sup> Performed if the ride can be used

<sup>61</sup> Turn green if the ride can be used

Get a reference to the document element that will display the result for the rider to see.

Update a property of this document element to show the result.

The code above sets a variable called `jungleAdventureElement` to refer to the element holding the text being displayed for the Jungle Adventure ride (this is the list item with the ID `jungleAdventure`). It then uses a conditional statement controlled by the value in `ageNo` to update the `className` property of `jungleAdventureElement` to an appropriate style. The `ageNo` variable contains the age of the rider.

When the program runs the `className` property of `jungleAdventureElement` is set to "menuNo" if the value in `ageNo` is less than 6. If the value in `ageNo` is greater than or equal to 6 the `else` part of the conditional statement is performed and the `className` property of the element is set to "menuYes". The `className` property of an element gives the stylesheet class to be used to format this element. These two classes are defined in the `styles.css` file for this application, and they look like this:

```
.menuYes {  
    color: green;  
}  
  
.menuNo{  
    color: red;  
}
```

This combination of JavaScript and stylesheet will cause the text for the Jungle Adventure to change to red if the age of the ride user is less than 6 and to green for any other age. We must implement this construction for all the other rides in the theme park. A friend of ours is looking for some free access to the Theme Park, so we decide to sub-contract the work to him. He is an experienced programmer (or at least he has read a couple more books than we have). He does the work, gets his Theme Park tickets and hands over the finished program. The new program seems to work well, and the owner of the theme park is pleased with it at first. However, after a while she starts to get complaints from riders about incorrect displays for The Downhill Mountain ride. We need to find out what has gone wrong.

#### PROGRAMMER'S POINT

##### Always respond constructively to fault reports

I've had my share of bugs over the years. I learned quickly that customers appreciate a positive response to their fault reports. It is important to remember that when you are fixing a fault your loyalty is to solving the problem, not finding out who caused it. I never made a fuss when a bug turned out not to be my fault, and I always accepted responsibility when it was. By being constructive in my approach to fault reports I could turn a negative into a positive, so that my customers would end up praising my debugging skills rather moaning about my buggy code.

## CODE ANALYSIS

### Fixing faults

Programs contain bugs because they are created by fallible humans. We get a *bug* when, for whatever reason, a program does something that it should not. A bug becomes a *fault* when it affects the user in some way. It seems that there is a bug in the ride program that is causing the faulty display. Let's open the program in the folder **Ch06 Repeating actions\Ch06-01 Broken Ride Selector** and take a look.

Question: What age values don't work?

We've been told that the Downhill Mountain ride display is not working properly. Only riders who are 12 years or over can use this ride. Let's start with an age which is less than 12:

These are the rides that are available

- Scenic River Cruise
- Carnival Carousel
- Jungle Adventure Water Splash
- Downhill Mountain Run
- The Regurgitator

Enter your age: 8

This test shows that the program is working correctly for 8-year-old riders. They should not be allowed on the Downhill Mountain Run or the Regurgitator. Let's try 12, an age that should work.

These are the rides that are available

- Scenic River Cruise
- Carnival Carousel
- Jungle Adventure Water Splash
- Downhill Mountain Run
- The Regurgitator

Enter your age: 12

Aha! The bug is exposed. The Downhill Mountain Run is displayed in red, which is incorrect. A rider who is 12 years old should be allowed to go on this ride.

Question: What could cause the error?

Now we can see the bug we can investigate the cause. Use Visual Studio Code to open the program file **themepark.js** in the example folder and investigate the code. The part that you need to find is the part that handles the DownHill Mountain ride. There is a bug here. Can you see it?

```
var downhillMountainElement= document.getElementById("downhillMountain");
if (ageNo < 12) {
    downhillMountainElement.className = "menuNo";
}
else {
    jungleAdventureElement.className = "menuYes";
}
```

The program is supposed to set the Downhill Mountain element to red if the age is less than 12, or else set the element to green. The logic is the same as that used for the Jungle Adventure ride. But it is broken. Have you spotted the error yet?

```
var downhillMountainElement= document.getElementById("downhillMountain");
if (ageNo < 12) {
    downhillMountainElement.className = "menuNo";
}
else {
    jungleAdventureElement.className = "menuYes";
}
```

The error is in the **else** part. I've highlighted it above. Rather than setting the `className` on the `downhillMountainElement` it instead works on the `jungleAdventureElement`. The program is deciding correctly that the rider can go on the Downhill Mountain ride, but then displaying the result on the wrong element.

Question: How do we fix the bug?

This bug is easy to fix, we just need to change it so that the correct element is updated when the rider can go on the ride.

```
var downhillMountainElement= document.getElementById("downhillMountain");
if (ageNo < 12) {
    downhillMountainElement.className = "menuNo";
}
else {
    downhillMountainElement.className = "menuYes";
}
```

Question: How would you make a mistake like this?

It seems very strange to type a completely wrong name. But this kind of bug is quite likely. Consider how the code would be written. Our programming friend created the code for the Jungle Adventure ride and then copied it for the Downhill Mountain ride. He should have changed all the identifiers, but he missed one out and caused the bug.

Question: How do we prevent mistakes like this?

The primary cause of the fault was a lack of proper testing. If the program had been properly tested the fault

would have shown up. However, the practice of using block copy when writing the code made the bug possible. We should try to write our solutions in a way that reduces the number of ways they could go wrong. This includes not using block copy to repeat sections.

## The importance of well-designed code

A bridge made of random pieces of wood nailed together that shudders and creaks as you cross it is probably not built to a particularly good design. You can consider software in design terms too. The theme park ride application we have created is like a badly made bridge. It works but the internal structure is not good. As we have seen above, using multiple copies of the same if construction makes mistakes likely when writing the code. There is also another possible source of error. The names of the rides are held in the HTML file, but the age limits for the riders are held in the JavaScript file. If we want to add more rides or adjust the age limits for the rides we must edit both and make sure that their contents line up. Any mistakes will show up as more bugs.

### PROGRAMMER'S POINT

#### Design is important

You might think that worrying about the internal design of a program is a waste of time. If the program works, why do we need to care about how it fits together? However, it is important that you always strive to make your code well designed. If you are still not convinced, consider the implications of a bug in the Theme Park ride selector that allowed a three-year-old to go on "The Regurgitator" ride. The child might be injured, and the Theme Park owner would be in a lot of trouble. An investigation might conclude that we were at fault for not using best practice when creating the code.

When we make a solution, we should try to reduce the number of ways it could fail. Many bugs are produced when a working program is modified. The easier that program is to understand, and the smaller the number of changes that are required, the lower the likelihood of bugs being introduced. We will look at design techniques throughout the rest of this book.

## Adding data attributes to HTML elements

We can start to improve the structure of our application by putting all the data that controls it in the same place. This will remove one possible cause of errors. We are going to do this by putting the age limits inside the elements that display the ride information in a web page. This is a powerful feature of HTML and JavaScript that we will use a lot in later programs. The feature we are going to use is called a *data attribute*.

We have seen that an HTML element can contain attributes that modify it in some way. For example, to display red text I add a `style` attribute to a paragraph to select red.

```
<p style="color:red">This is a red paragraph.</p>
```

An attribute is something that modifies an element. Each HTML element uses a particular set of attributes, for example a `p` element has a `style` attribute that sets the text style (as we saw above) and an `img` element has a `src` attribute that specifies the source file containing an image (as we saw when we displayed images). An element can also contain data attributes.

```
<ul class="menuRideList" id="rideList">
  <li data-MinAge="0" data-MaxAge="120" id="scenicRiver">Scenic River Cruise</li>
  <li data-MinAge="3" data-MaxAge="120" id="carnivalCarousel">Carnival Carousel</li>
  <li data-MinAge="6" data-MaxAge="120" id="jungleAdventure">Jungle Adventure Water Splash</li>
  <li data-MinAge="12" data-MaxAge="120" id="downhillMountain">Downhill Mountain Run</li>
  <li data-MinAge="12" data-MaxAge="70" id="regurgitator">The Regurgitator</li>
</ul>
```

The HTML above is like the earlier list of theme park rides, but each of the list items now has a `data-MinAge` and a `data-MaxAge` attribute. These give the age limits for each of the rides. This information about rides and ages is now all held in one place. The JavaScript program can use the `getAttribute` method provided by an element to read the contents of a data attribute.

```
// Get the carnival element
var carnivalCarouselElement = document.getElementById("carnivalCarousel");

// Get the min age from the carnival element data attribute
var carnivalMinAgeText = carnivalCarouselElement.getAttribute("data-MinAge");62
var carnivalMinAgeNo = Number(carnivalMinAgeText)

// Get the max age from the carnival element data attribute
var carnivalMaxAgeText = carnivalCarouselElement.getAttribute("data-MaxAge");63
var carnivalMaxAgeNo = Number(carnivalMaxAgeText);
```

The code above shows how this would work for the Carnival Carousel ride. Note that the value of an attribute is a string of text, so the `Number` function is used to convert the text into a numeric value. Once the program has the maximum and minimum ages it can then update the display to indicate whether the rider can go on the ride.

---

<sup>62</sup> Read the min age attribute

<sup>63</sup> Read the max age attribute

```
if(carnivalAgeNo < carnivalMinAgeNo){  
    carnivalCarouselElement.className = "menuNo";  
}  
else{  
    if(carnivalAgeNo > carnivalMaxAgeNo){  
        carnivalCarouselElement.className = "menuNo";  
    }  
    else{  
        carnivalCarouselElement.className = "menuYes";  
    }  
}
```

## CODE ANALYSIS

### Data attributes

You may have some questions about data attributes. Let's see if we can answer them by looking at some code. Start with the sample application in the example folder **Ch06 Repeating actions\Ch06-03 Data Ride Selector**. Open this in your browser.

Question: Does this code work?

# CRAZYADVENTUREWONDERFUNLAND

These are the rides that are available

- Scenic River Cruise
- Carnival Carousel

Enter your age:

Yes. It turns out that the code does work.

Question: What is the upper limit for the age of the Carnival Carousel ride?

We can answer this question by looking at how the data attributes on the Carnival Carousel list element in the HTML document. Press F12 to open the Developer View. Now type in the following JavaScript statement:

```
> var carnivalCarouselElement = document.getElementById("carnivalCarousel")
```

This statement creates a variable called `carnivalCruiseElement` that refers to the Carnival Carousel list element in the web page. Press enter to run it.

```
> var carnivalCarouselElement = document.getElementById("carnivalCarousel")
<- undefined
```

When you press Enter the variable is created and the statement that created the variable returns a value of `undefined`. Now we can use the `getAttribute` method to get data out of this element. Enter the following statement:

```
> carnivalCarouselElement.getAttribute("data-MaxAge")
<- "120"
```

This returns the value 120, which is the contents of the `data-MaxAge` attribute in the Carnival Carousel element. This means that the maximum age for that ride is set as 120 years. You can use the same method to read the minimum age value.

```
> carnivalCarouselElement.getAttribute("data-MinAge")
<- "3"
```

Question: What happens if we ask `getAttribute` to find a data attribute that doesn't exist?

Let's try it. Enter the statement below, that looks for a data attribute called `data-SillyAge`:

```
> carnivalCarouselElement.getAttribute("data-SillyAge")
<- null
```

This call of `getAttribute` returns of `null`, which is how JavaScript indicates that a reference value doesn't refer to anything. In other words, the `getAttribute` method couldn't find anything, and so it returned a value that indicates that nothing was found. We could use an if construction to test for a value of `null`, so our program could do something sensible if a data attribute is not found.

Question: Can I set the value of a data attribute in an element?

Yes, you can. There is a method called `setAttribute` which is provided with the name of a data attribute and the value to be set in the attribute. It sets the data attribute to that value. Enter the following statement to set the maximum age for the ride to 99.

```
> carnivalCarouselElement.setAttribute("data-maxAge", "99")
<- undefined
```

We can test this by reading back the value of the maximum age:

```
> carnivalCarouselElement.getAttribute("data-MaxAge")
<- "99"
```

Question: Can a program create new data attributes on an element?

Yes. This can be used to "bind" data to elements. The `setAttribute` method will create a new attribute if the one being set does not exist. Try the following statements which create a test data attribute called "data-test" on the Carnival Carousel element:

```
> carnivalCarouselElement.setAttribute("data-test", "test string")
<- null
> carnivalCarouselElement.getAttribute("data-test")
<- "test string"
```

Question: Does the name of a data attribute have to start with the characters "data-?"

JavaScript does not enforce this rule, but it is a very sensible convention so that someone reading the code does not mistake a data attribute for a "normal" attribute of that element.

Question: Can a program add data attributes to any HTML element?

Yes, you can. This is good way to store data values inside your web page.

## Using an unnumbered list as a container

The code in the example folder **Ch06 Repeating actions\Ch06-03 Data Ride Selector** only handles two rides. The reason for this is that I just didn't feel like copying the Carnival Carousel code for the other rides. It seemed like too much hard work and I was concerned I might make a mistake. It turns out that there is a much easier way of handling all the other rides which uses a loop to work through the list of rides.

```
<ul class="menuRideList" id="rideList">
  <li data-MinAge="0" data-MaxAge="120">Scenic River Cruise</li>
  <li data-MinAge="3" data-MaxAge="120">Carnival Carousel</li>
  <li data-MinAge="6" data-MaxAge="120">Jungle Adventure Water Splash</li>
  <li data-MinAge="12" data-MaxAge="120">Downhill Mountain Run</li>
  <li data-MinAge="12" data-MaxAge="70">The Regurgitator</li>
</ul>
```

The HTML above shows the list of rides in the Theme Park as they are displayed. Each of the rides is described by a list item (`<li>`) element and all the lists are enclosed inside an un-numbered list (`<ul>`) element. The `<ul>` element is a *container* element that holds other elements. In this case the list contains five list items. These list items are called the *children* of their container element. HTML elements provide methods that can be used to access the children of an element. Let's investigate.

MAKE SOMETHING HAPPEN

Investigate list elements

Start by opening the application in the **Ch06 Repeating actions\Ch06-04 Theme Park Ride For Loop** folder in the example code. This is a fully working ride selector that uses a for loop. We'll discover how the for loop works later in this section. For now, we are going to take a look at the list container. Open the Developer View by pressing the function key F12. Now we can start typing commands into the JavaScript command prompt. The first thing we are going to do is get a reference to the list that holds the ride names:

```
> var rideListElement= document.getElementById("rideList")
<- undefined
```

When you press enter the JavaScript console returns the result undefined because, as we know, the process of creating a variable does not return a value. However, the variable `rideListElement` now refers to the list of rides. To prove this, just type the name `rideListElement` and the Developer View will show you the HTML in the element it refers to:

```
> rideListElement
<- ><ul class="menuRideList" id="rideList">...</ul>
```

When you press enter this time the JavaScript console displays the element. Click the little right pointing arrow to expand the display.

```
> rideListElement
<- ><ul class="menuRideList" id="rideList">
  <li data-minage="0" data-maxage="120" class="menuYes">Scenic River Cruise</li>
  <li data-minage="3" data-maxage="120" class="menuYes">Carnival Carousel</li>
  <li data-minage="6" data-maxage="120" class="menuYes">Jungle Adventure Water Splash</li>
  <li data-minage="12" data-maxage="120" class="menuYes">Downhill Mountain Run</li>
  <li data-minage="12" data-maxage="70" class="menuYes">The Regurgitator</li>
</ul>
```

The list has five children which are the list items. These children are held in a property of the list which is called **children**. We can use an **index** to specify which of the children we want to look at. The index is expressed as a number enclosed in square brackets (`[]`): Let's have a look at the element at the start of the list. Type in the statement below.

```
> rideListElement.children[0]
```

Note that this is the element with an index of 0. When counting items in a container, JavaScript starts counting at zero, not one. When you press enter the list item for the Scenic River Cruise is displayed.

```
<- <li data-MinAge="0" data-MaxAge="120">Scenic River Cruise</li>
```

You can use this technique to obtain any of the items in the list. The statement below will view the list item for the Regurgitator. I tried it with a different number:

```
> rideListElement.children[4]
<- <li data-MinAge="12" data-MaxAge="70">The Regurgitator</li>
```

Remember that because JavaScript counts elements from 0 final element in a list containing four elements will have an index of 4. You might be wondering what will happen if you try to access an element that isn't there. Try this:

```
> rideListElement.children[5]
```

There is no Fifth Element in the list (although the film with that name is a classic). The result of trying to find a non-existent element is the `undefined` value:

```
<- undefined
```

We can find out how many children there are by using the `length` property of `children`. Try this statement to see how it works:

```
> rideListElement.children.length
```

There are five children, and so the `length` property will be 5:

```
<- 5
```

Leave your browser open for now, we will be using it to investigate loops a little later.

## The JavaScript for loop

Now that we know how to get hold of the list items in the ride list, the next thing we need to be able to do is use the items in our application. What we need is language construction that will loop round a block of code and count each time round the loop. It turns out that JavaScript has just the thing. It is called the *for loop*.

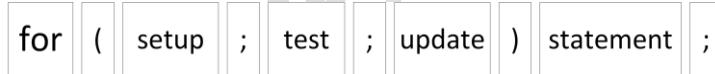


Figure 6.18 For loop

Figure 6.3 shows the elements of a JavaScript for loop.

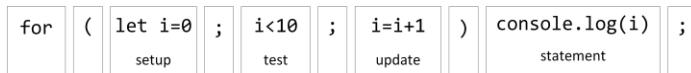
The **setup** element is performed once when the for loop is started

The **test** element is performed before every loop. If the test has a JavaScript “truthy” value of `true` the loop will continue. If the test has a “truthy” value of `false` the loop will end and the program will continue at the statement after the for loop.

The **update** element is performed after each execution of the loop.

The **statement** is a JavaScript statement that is to be repeated by the for loop. This can be a block of statements enclosed in curly braces if you want the loop to perform more than one statement.

A for loop is usually used with a *control variable*. This variable counts the number of times the loop has been performed. The **setup** element puts a starting value in the control variable. The **test** element tests the value of the control variable will stop the loop when the control variable reaches a particular value and the **update** element increases the value of the control variable.



**Figure 6.19** For loop counter

Figure 6.4 shows a complete for loop. This causes the loop statement to be performed 10 times. Let's take a proper look at it.

## CODE ANALYSIS

### The for loop

We can use the Developer View in the browser to investigate what a for loop does. Open the application in the **Ch06 Repeating actions\Ch06-04 Theme Park Ride For Loop** folder if you haven't already, and then open the Developer View by pressing F12.

Question: What does the for loop we have created actually do?

We can start by typing in the JavaScript that you have just seen in Figure 6.4

```
> for(let i=0;i<10;i=i+1)console.log(i)
```

You know (because I've told you) that this should repeat the statement `console.log(i)` 10 times. Whatever that does. Press enter and see what happens.

```
> for(let i=0;i<10;i=i+1)console.log(i)
0
1
2
3
4
5
6
7
8
9
<- undefined
```

The `console.log` statement is a useful tool for debugging a program. It logs values on the Developer View console. In this case it is logging the value of `i` each time it is called. The value of `i` starts at 0. Each time around the loop the value in `i` is increased by 1. The loop also tests the value in `i` using the condition `i < 10`. When `i` reaches the value 10 this condition is no longer true (the value 10 is not less than 10) and so the loop stops.

Question: Why does the for loop print out "undefined" at the end.

The for loop does not print out this message. This message is printed because the console always prints the value of a statement. A for loop construction does not generate a result in the same way that creating a variable using

```
var does not generate a result. In both cases the value undefined is displayed by the console.
```

Question: Can I use a for loop to control lots of statements?

Yes. A for loop can control a block of statements enclosed in curly braces (`{}`) in the same way as an if construction.

Question: What does the word `let` mean in front of the declaration of `i`?

Up until now we have created our variables using the JavaScript keyword `var`. However, we can also create a variable by using the keyword `let`. Variables created using `let` can be used anywhere in the program after they have been declared. Variables created using `let` are only useable in the block where they were declared. The value in `i` is only used inside the statement controlled by the for loop and so it should only inside that for loop. To prove that this has happened, enter the following statement to try and look at the value in `i`:

```
> i
```

This variable no longer exists because it was deleted when the for loop was completed.

```
<-Uncaught ReferenceError: i is not defined at <anonymous>:1:1
```

Question: Why is it a good idea for variables to disappear like this?

This is another piece of good design. I tend to use a variable called `i` for counting. Other programmers working on this code might decide to use a variable called `i` for something else. I don't want these two variables to clash so I make sure that my version of the variable only exists when I use it.

Question: What happens if JavaScript is given a loop that will never run?

```
> for(let i=0;i>10;i=i+1)console.log(i)
```

This loop is silly. The loop sets `i` to `0` and then continues while the value of `i` is greater than `10`. Since the value of `i` is not greater than `10` the loop will never be performed. If you run this code you will discover that it does not produce any output.

```
> undefined
```

Question: What happens if I put in a loop that will never end?

```
> for(let i=0;i<10;i=i-1)console.log(i)
```

This loop is also silly. Each time round the loop the value of `i` reduced by 1 rather than increased by 1. This means that the value of `i` will never reach `10` and so the loop will run for ever. If you run this statement (and I don't advise it) you will notice that the console will be filled with ever decreasing numbers, the fan on your computer will come on and your browser will become strangely unresponsive as it tries to run this loop as quickly as possible. The only way to stop this that always works is to close the browser tab holding the program. Sometimes when you visit a web page you find that your browser gets "stuck". Now you know one way that it can happen.

Now that we know about for loops, we can write the code for the Theme Park ride selector.

```
function doCheckAge() {  
  
    // get the age that was input  
    var ageElement = document.getElementById("ageText");  
    var ageText = ageElement.value;  
    var ageNo = Number(ageText);  
  
    // get the list of rides  
    var rideListElement= document.getElementById("rideList");  
  
    // get the number of child list items  
    var noOfRides = rideListElement.children.length;  
  
    // make a loop to count round the rides  
    for(i=0; i < noOfRides; i=i+1){64  
        // get the ride element out of the list  
        let rideElement = rideListElement.children[i];  
  
        // get the minimum age  
        let minAgeText = rideElement.getAttribute("data-MinAge");  
        let minAgeNo = Number(minAgeText)  
  
        // get the maximum age  
        let maxAgeText = rideElement.getAttribute("data-MaxAge");  
        let maxAgeNo = Number(maxAgeText);  
  
        // test the age and update the component  
        if(ageNo<minAgeNo){  
            rideElement.className="menuNo";  
        }  
        else{  
            if(ageNo>maxAgeNo){  
                rideElement.className="menuNo";  
            }  
            else{  
                rideElement.className="menuYes";  
            }  
        }  
    }  
}
```

---

<sup>64</sup> This is the for loop that updates the list

```
}
```

I love this code. I really like how it works. You should spend some time looking through it to make sure that you understand how it fits together. To appreciate how well designed it is, consider what changes you would have to make to the program if the theme park added another ride. The wonderful answer is that you would not have to change this code at all. You would just have to add another item to the list in the HTML file. You also don't have to change the program code if the age rating on one of the rides is changed. You just change the data in the HTML file and the program keeps going. The example solution in the folder **\Ch06 Repeating actions\Ch06-05 Lots of Theme Park Rides** has lots of additional rides but the JavaScript program code is the same.

#### PROGRAMMER'S POINT

##### Great programmers are “Constructively Lazy”

You can spot a great programmer by the way that they use their skills to avoid hard work. I call this “constructive laziness”. If I find myself having to write lots of code or worrying about synchronizing different parts of a solution I will try to find a way of using a loop and bringing things together into one place.

## Work through collections using for-of

We've seen how we can use a `for` loop to count through a range of values. We might find a use for this a bit later in the chapter. But we can also use a different form of the `for` loop to directly work through the items in a container. This new type of loop was not added to make something possible; we already know how to solve the problem. It was added to make something easier. This construction is called the `for - of` loop:

```
for (let rideElement^ of rideListElement.children^) {  
    // get the minimum age  
    let minAgeText = rideElement.getAttribute("data-MinAge");  
    let minAgeNo = Number(minAgeText)  
  
    // get the maximum age  
    let maxAgeText = rideElement.getAttribute("data-MaxAge");  
    let maxAgeNo = Number(maxAgeText);  
  
    // test the age and update the component
```

<sup>^</sup> This is set to the value of each element in the collection

<sup>^</sup> This is the collection that is being worked through

```
if (ageNo < minAgeNo) {
    rideElement.className = "menuNo";
}
else {
    if (ageNo > maxAgeNo) {
        rideElement.className = "menuNo";
    }
    else {
        rideElement.className = "menuYes";
    }
}
```

The loop above shows how it would be used. The loop will be performed for each of the children of the ride list. Each time round the loop the value of `rideElement` is set to the next child. There is no need to create a control variable and the code is much simpler. You can see a for of loop in action in the example in [Ch06 Repeating actions\Ch06-06 For of loop](#).

## Building web pages from code

The theme park ride selector makes good use of the way that a JavaScript program can interact with the elements in a web page. Now let's take this technique even further and find out how a JavaScript program can create new elements on the page when it runs. Then we can make web pages that construct themselves when they are loaded. For example, perhaps we might be asked to create a program to help people learn their times tables. Users enter a number and get a display of the times table for that number:

## Times Tables

1 times 2 is 2  
2 times 2 is 4  
3 times 2 is 6  
4 times 2 is 8  
5 times 2 is 10  
6 times 2 is 12  
7 times 2 is 14  
8 times 2 is 16  
9 times 2 is 18  
10 times 2 is 20  
11 times 2 is 22  
12 times 2 is 24

Which times table to you want :

**Make the times table**

Figure 6.20 Times tables

Figure 6.5 above shows how the program is used. The user enters the number of the times table that they want, and the program displays it. This program looks a lot like the Theme Park Ride Selector that we have already created. We could use a 12 element list to hold the results and then update the text in the list items with the times table that the user has requested in the same way that we update the color of each ride entry. However, that would mean we would have to type in a 12 element list, and that sounds like a bit too much work to me. So, let's start with an empty HTML document and then write some JavaScript to fill in the times table results by creating list items.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Ch06-07 Times Tables HTML generator</title>
  <link rel="stylesheet" href="styles.css">
  <script src="timestables.js"></script>
</head>

<body>
  <p class="menuHeading">Times Tables</p>
  <!-- Times table list will be built by code-->
```

```

<ul class="menuTimesTableList" id="timesTableList">65  

</ul>  

<p class="menuText">Which times table do you want :  

<input class="menuInput" type="number" id="timesTableText" value="2" min="2" max="12"> </p>66  

<button class="menuButton" onclick="doTimesTables()">Make the times table</button>67  

</body>  

</html>

```

This is the HTML page for the times table application. Note that in the middle there is an empty list with the id `timesTableList`. This is going to hold the times table that the program will generate. The user will press the "Make the times table" button which will call the JavaScript function `doTimesTable` to create the table. Let's have a look at that.

```

function doTimesTables() {  

    // get the times table number from the web page  

    var tableTextElement = document.getElementById("timesTableText");  

    var timesTableText = tableTextElement.value;  

    var timesTableNumber = Number(timesTableText);  

    // get the times table list from the web page  

    var timesTableListElement = document.getElementById("timesTableList");  

    // count through the times table producing results  

    for (let i=1; i<=12;i++) {  

        // calculate the result  

        let resultNumber = timesTableNumber * i;  

        // create a result string  

        let resultString = i + " times " + timesTableNumber + " is " + resultNumber;  

        // make new list item  

        let listItem = document.createElement("li");  

        // set the text of the new list item to the result string  

        listItem.innerText=resultString;
}

```

<sup>65</sup> Empty list to hold the times table results.

<sup>66</sup> Input for the required times table.

<sup>67</sup> Button that is pressed to produce the times table

```
// add it to the times table list
timesTableListElement.appendChild(listItem);
}
}
```

The most interesting part of this function is the last three statements which create a new element and add it to the list. Let's take a look at how they work.

## CODE ANALYSIS

### Building HTML from JavaScript

Take as your starting point the example application in the **Ch06 Repeating actions\Ch06-07 Times Tables HTML generator** folder. Open this application and then press F12 to open the Developer View.

Question: Does the program work?

You can test the program by entering a times table value into the web page and pressing the button "Make the times table". You will see a display like the one in Figure 6.5

Question: How do we create new web page elements?

The method `createElement` is provided by the document object. The method will make a new HTML element. Let's make another list item to add to the times table list. We give `createElement` a string that specifies the type of element we want, in this case we want a list item so we use the string "`li`". Type in the following and press enter:

```
> var newElement=document.createElement("li")
```

Now that we have our new element, we can set the `innerText` that it contains. Type in the following and press enter:

```
> newElement.innerText="Hello world"
```

We now have an html element that is a list item containing the text "Hello world".

Question: How do we add a web page element to an element on the page?

We can add a new child to any HTML element using the `appendChild` method to add a new child to the element. First we need to find the element we want to add to. Type the following statement and press enter:

```
> var timesTableListElement = document.getElementById("timesTableList")
```

We now have a variable called `timesTableListElement` that refers to the list holding the times table. Let's add our

new element to this. Type in the following statement and press enter.

```
> timesTableListElement.appendChild(newElement);
```

If you look back at the web page you will discover a new line has appeared underneath the last line of the times table. This line was added by us just now:

12 times 2 is 24  
Hello world

Which times table do you want :   
**Make the times table**

This new element is not part of the HTML file, it is part of the document object that is being maintained by the browser.

Question: Can we just keep adding lines to the list?

Yes you can. In fact there is a bug in our solution which means that it just keeps adding times tables to the list. If you press the "Make the times table" button again you will discover that another times table will be added to the page:

11 times 2 is 22  
12 times 2 is 24  
Hello world  
1 times 2 is 2  
2 times 2 is 4

Question: Can a program delete items from an element?

Yes it can. We can delete the element at the start of the list by entering the following:

```
> timesTableListElement.removeChild(timesTableListElement.children[0])
```

The `removeChild` method is given a reference an element that is then removed from the list. In the statement above the reference that is supplied is a reference to the element at the start of the list. If you look at the web page display you will find that the first result has been removed.

## Deleting elements from a document

In the preceding Code Analysis, we discovered a bug in the Times Table application. When you select a new times

table the results are added to the end of the existing one. Fortunately, we also discovered a method we can use to remove children from an element. We can use this method to “clean up” the list prior to adding new elements. To do this we can use another loop construction called a `while` loop. A while loop repeats a statement as long as a condition is true.

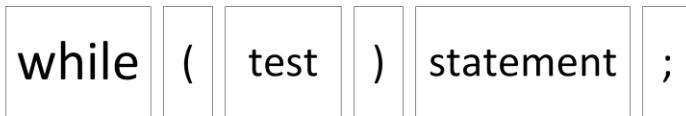


Figure 6.21 While loop

The `while` loop will perform the test and then, if the test is “truthy”, it will perform the statement. It will then loop back and repeat this process until the test is “falsey”. We can use this to create some JavaScript that will delete child elements until there are none left:

```
while(timesTableListElement.children.length > 0)
  timesTableListElement.removeChild(timesTableListElement.children[0]);
```

If this looks a little hard to understand, try converting it into English: “While the number of children is greater than 0, delete a child”. There is a working example of the Times Table program in the folder **Ch06 Repeating actions\Ch06-08 Times Tables HTML generator cleanup** in the sample programs.

## MAKE SOMETHING HAPPEN

### More times table fun

Here are some ideas that you might like to investigate.

### Twenty times table

Some people like to show off by knowing their twenty times table. Can you make a change to the program above to make a version that displays the 20 times table? You should be able to do this by changing just one number in the program. If you can't work out how to do this you can find my version in the folder **Ch06 Repeating actions\Ch06-09 Twenty times table**

### Times Table Tester

## Times Table Tester

1 times 2 is 2  
2 times 2 is 5  
3 times 2 is 6  
4 times 2 is 8  
5 times 2 is 11  
6 times 2 is 12  
7 times 2 is 14  
8 times 2 is 16  
9 times 2 is 18  
10 times 2 is 21  
11 times 2 is 22  
12 times 2 is 24

Which times table do you want :

The Times Table Tester produces a times table with some wrong numbers and then challenges the user to find them all. Then can press the "Check the times table" button to check which ones are correct. The JavaScript random number generator can be used to decide if a given value is to be displayed incorrectly. The program could work by using a data attribute in each list item that holds the value being displayed by that item. The marking process would be rather like the Theme Park Ride program. It would change the color of the list item depending on whether the displayed value is correct. You can find my version in the folder **Ch06 Repeating actions\Ch06-10 Times Table Tester**

## What you have learned

This chapter has shown you to the use loops in JavaScript programs. You've also picked up some useful programming design tips and discovered how JavaScript code can change the structure of the document being displayed by the browser.

- You only really learn how a program should work by making something and trying to use it. If you make a solution for someone you must be prepared to change the way it works when they think of a better way of using it.
- Bugs are a natural consequence of software development. When a fault is reported your loyalty should be to the process of fixing it, not assigning blame or responsibility.
- Some software writing processes, for example repeating behaviors by copying blocks of code, make bugs more likely.

- HTML elements can be given data attributes. This allows a program to bind data values directly to items in the document and for data to be embedded in the web page text.
- HTML elements can act as containers for other elements. Elements contained in an element are the child elements of the parent. Individual child elements can be accessed using an index value. The index values start at 0.
- A JavaScript for loop construction is used to repeatedly perform a statement. The for loop has a setup, test and update behaviors. The setup behavior is performed at the start of the loop. The test is performed after each loop and also before the first loop. If the test evaluates to false the loop stops. The update behavior is performed after each loop.
- A JavaScript for loop construction can be used to manage a control variable counter that can count between two values, allowing a program to work through the children of a container element.
- JavaScript provides a “for of” construction to work through the elements in a container.
- There are JavaScript functions that can be used to create new page elements ([document.createElement](#)) and add them as children to existing elements ([appendChild](#)). These elements are then rendered by the browser. This makes it possible for JavaScript code to create the contents of a web page programmatically.
- There is a JavaScript function ([removeChild](#)) that can be used to remove child elements from a container element.
- The while loop **construction allows** a program to repeat a block of statements while a
- .

Here are some questions that you might like to ponder about making decisions in programs:

**Question:** What is the difference between a bug and a fault?

**Answer:** A bug is something inside a program that makes it do the wrong thing. A bug that someone has noticed is a fault.

**Question:** Does every program contain bugs?

**Answer:** It is almost impossible to prove that a given program does not contain any bugs. Testing only ever proves that bugs exist, not that they don't.

**Question:** Why are lists indexed starting at zero?

**Answer:** This is just the way that JavaScript works. Some languages index their collections starting at 1. But JavaScript starts at 0. You can think of the index as the “distance” down the storage that you need to go to get to the item. So

**Question:** Can any HTML element be a container for other elements?

**Answer:** Yes. The HTML element in a document contains the HEAD and BODY elements. It is also possible for the child of an element to have children of its own, allowing for a hierarchy to be created.

**Question:** Does adding an attribute to an HTML element change the contents of the web page file?

**Answer:** No. You can think of the HTML file as defining the starting point of the document object that is displayed by the browser. Changes to this object will affect the appearance of the page but they will not change the contents of the HTML file.

**Question:** What is the difference between let and var?

**Answer:** Both keywords are used to create variables. Variables created with `var` exist from that point in the program. Variables created with `let` are destroyed when program execution leaves the block where they were declared. Declaring variables using `let` is a good idea as it reduces the chances of variables being reused by mistake.

**Question:** Could I create an entire web site using JavaScript code?

**Answer:** Yes. Lots of web pages you visit work this way. Rather than having HTML text stored on the server a site instead contains a small HTML file and a JavaScript program that loads data from the web and then uses it to build the website. We will be doing more of this in future chapters.

# 7

# Creating functions

## What you will learn

Functions are an essential part of program design. You can use functions to break up a large solution into individual components and to create libraries of behaviors that can be used by your programs. We have used some built in functions (for example `alert`) and created event handler functions (for example `doCheckAge`). In this chapter you'll learn how to create and use functions of your own. You'll see how to give functions data to work on and how a program can receive results that a function returns. Along the way you will pick up some tips about error handling.

## What makes a function?

A function is a chunk of JavaScript that has been given an identifier. A function should be *defined* before it is used. When JavaScript encounters a function definition it takes the statements that provide the behavior of the function and stores them ready for use by the program. A program can be *call* the function at which point the statements in it will be performed. We've created and called functions already, but now is the time to take a detailed look at how they work. Let's start with a look at a simple function that just says hello.

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
<title>Ch07-01 Greeter Function</title>
</head>

<body>
  <h1>Greeter</h1>
  <p></p>
  <p id="outputParagraph"></p>
</p>

<script>
  function greeter() {68
    var outputElement = document.getElementById("outputParagraph");69
    outputElement.textContent = "Hello";70
  }
</script>
</body>

</html>
```

This web page contains a function called `greeter`. The function finds an element on the screen with the ID “`outputParagraph`” and displays “Hello” in that element. Once the function has been defined, a program can call it. A call of a function runs the statements given when the function was defined. The `greeter` function doesn’t do much, but you can create functions that contain many statements. Remember that your program must define the function before it can be called.

## CODE ANALYSIS

### Investigating Functions

Start by opening the application in the **Ch07 Functions\Ch07-01 Greeter Function** folder in the example code. This application contains an HTML page with the output paragraph and a script section containing the `greeter` function. Open the Developer View by pressing F12.

Question: How do I call a function in my program?

<sup>68</sup> *greeter function*

<sup>69</sup> *Find the display element*

<sup>70</sup> *Set the display text to “Hello”*

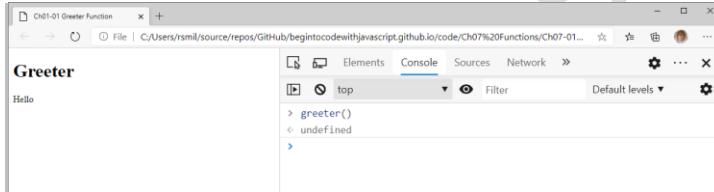
Up until now our programs have used functions that have been provided for us by JavaScript (for example the `alert` function). But the `greeter` function is one that we have written. However, we can call it in the same way, just by entering the name of the function, followed by a pair of braces.

```
> greeter();
```

When you press enter the `greeter` function is called. This function does not return a result (we will look at this later) and so the console displays “undefined”.

```
<- undefined
```

However, you will notice that the word “Hello” has now appeared on the browser display. This was displayed by the `greeter` function when it ran.



Question: Can we define functions in the console?

Yes, we can define our own functions in the console. Type in the statement below, which creates a function called `alerter`.

```
> function alerter() { alert("hello"); };
```

Press enter when you've typed in **exactly** what is given above. Make sure to use the right kinds of brackets for each part.

```
<- undefined
```

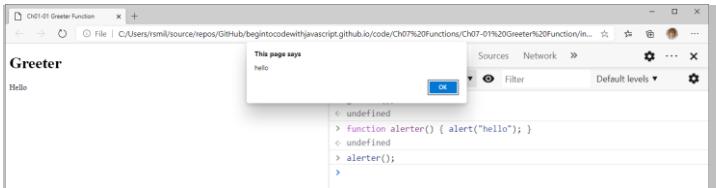
The application now contains a new function called `alerter`. The process of defining a function does not return a value, so the console shows the result “undefined”.

The JavaScript that you have just typed in contains a statement that calls the `alert` function, but this has not run because this statement is in the body of the `alerter` function. Now enter a statement that calls the newly created function:

```
> alerter();
```

When you press enter the statement will call `alerter` which will display an alert:

```
<- undefined
```



Question: Can one function call another function?

Yes it can. Consider the following functions:

```
function m2(){
  console.log("the");
}

function m3(){
  console.log("sat on");
  m2();
}

function m1(){
  m2();
  console.log("cat");
  m3();
  console.log("mat");
}
```

If `m1` is called what will be logged in the console? The best way to figure this out is to work through the functions one statement at a time, just like the computer does when it runs the program. Remember that when a function is complete, the program's execution continues at the statement following the function call. These functions are in the JavaScript for the Greeter page. You can find out what happens by calling function `m1`:

```
> m1();
```

When you press enter the `m1` function is called. This then calls `m2` and so on. The output is what you would expect:

```
the
cat
sat on
the
mat
```

Question: What happens if a function calls itself? For example, what if the `m1` function called `m1`?

Making a function calling itself is a bit like arranging two mirrors so that they face each other. In the mirrors, you see reflections going off into infinity. Does JavaScript go off into infinity when a function calls itself? Let's create a function and find out. Enter the following function definition:

```
> function mirror() { mirror(); };
```

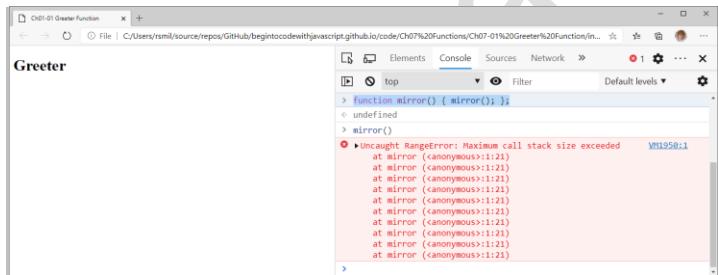
The `mirror` function just contains one statement which calls the `mirror` function. What happens when we create the function. Press enter to find out.

```
<- undefined
```

Nothing happens. Creating a function is not the same as running it. However, we now have a function called `mirror` that calls itself. Let's call `mirror` and see what happens:

```
> mirror();
```

When you press enter JavaScript will find the `mirror` function and start obeying the statements in it. The first statement in `mirror` is a call of the `mirror` function which JavaScript will start to execute. The program is stuck in a loop, but it does not get stuck forever. After a while the console stops with an error:



A screenshot of a browser developer tools console window titled "Greeter". The "Console" tab is selected. The console output shows the following sequence of events:

```
> function mirror() { mirror(); }
<- undefined
> mirror()
> 
    > Uncaught RangeError: Maximum call stack size exceeded  VM1958:1
        at mirror (<anonymous>:1:21)
        at mirror (<anonymous>:1:21)
```

Each time a function is called, JavaScript stores the return address (the place it must go back to) in a special piece of memory called the "stack." The idea is that when a running program reaches the end of a function, it grabs the most recently stored return address from the top of the stack and returns to where that address points. This means that as functions are being called and returned, the stack grows and shrinks.

However, when a function calls itself, JavaScript repeatedly adds return addresses on the stack. Each time the function calls itself, another return address is added to the top of the stack. The stack is finite in size and when this size is exceeded JavaScript stops the program.

Programmers have a name for a function that works by calling itself. They call it *recursion*. Recursion is occasionally useful in programs, particularly when the program is searching for values in large data structures. However, I've been programming for many years and have used recursion only a handful of times. I advise you to regard recursion as strong magic that you don't need to use now (or hardly ever). Loops are usually your best bet for repeating blocks of code.

function identifier ( arguments ) { statements }

Figure 7.22 Function definition

Figure 7-1 shows the form of a JavaScript function definition. We can work through each of these items in turn. The keyword `function` tells JavaScript that a function is being defined. JavaScript will allocate space for the function and get ready to start storing function statements. The word `function` is followed by the identifier of the function. We must create an identifier for each function we define. Because a function is associated with an action, it's a very good idea to make the name reflect this. I give functions names in the form `verbNoun`. The verb specifies the action the function will perform, and the noun specifies the item it will work on. An example would be `doTimesTables`.

After the function identifier, we have the arguments that are fed into the function. The arguments are separated by commas and enclosed within parentheses. Arguments provide a function with something to work on. So far, the functions we've created haven't had any arguments, so there has been nothing between the two parentheses. Finally, the definition contains a block JavaScript statements that form the body of the function. These are the statements that will be obeyed when the function is called.

## Give information to functions

The `greeter` function shows how functions can be used, but it isn't really that useful because it does the same thing each time it's called. To make a function truly useful, we need to give the function some data to work on. You've already seen many functions that are used in this way. The `alert` function accepts a string to display in the alert box. The `Number` function accepts an input to be turned into a numeric value. We could make a version of the `greeter` function that accepts a message to be displayed:

```
function greeter(message) {  
    var outputElement = document.getElementById("outputParagraph");  
    outputElement.textContent = message;  
}
```

### CODE ANALYSIS

#### Investigating Arguments

The new version of `greeter` is in the example application Ch07 Functions\Ch07-02 Greeter Arguments. Open this application and then open the Developer View by pressing F12.

Question: How do I give a function call an argument?

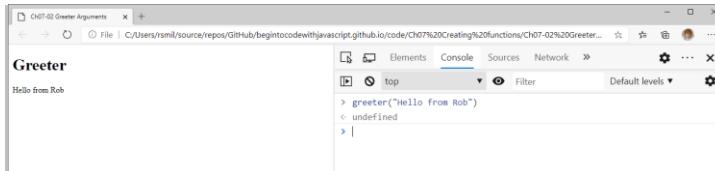
We can give a function an argument by putting something between the braces in the call of the function:

```
> greeter("Hello from Rob");
```

When you press enter the `greeter` function is called and the string "Hello from Rob" is passed as an argument to the function call.

```
<- undefined
```

The greeter function does not return a value, so the console displays “undefined”. However, you will notice that the phrase “Hello from Rob” has now appeared on the browser display. This was displayed by the `greeter` function when it ran.



A screenshot of a browser developer tools console window titled "Ch07-02 Greeter Arguments". The "Console" tab is selected. The output shows:

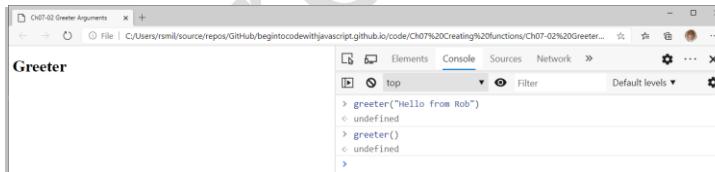
```
> greeter("Hello from Rob")
< undefined
```

Question: What happens if I miss out the argument from the function call?

This is bad programming practice. We've told JavaScript that the `greeter` function accepts an argument, and then we missed it off when we called it. Let's try it. Enter a call of `greeter` with no arguments.

```
> greeter();
```

You might think that this would cause an error. The `greeter` function is expecting to receive something which is missing from the call. If you've used other programming languages, for example C++ or C#, you will be used to getting errors if you do this kind of thing. However, it turns out that JavaScript is much more relaxed about this. There is no error, but no greeting either:



A screenshot of a browser developer tools console window titled "Ch07-02 Greeter Arguments". The "Console" tab is selected. The output shows:

```
> greeter("Hello from Rob")
< undefined
> greeter()
< undefined
```

This is what has happened:

5. The JavaScript system notices that there is an argument missing from the call.
6. It supplies the function with the value “undefined” for the missing argument.
7. The function sets the `textContent` property of the output paragraph to “undefined”. This is not displayed by the browser, so the text on the screen disappears.

If we are concerned about calls with missing arguments, we can make a version of the `greeter` function that checks to see if it been given an argument:

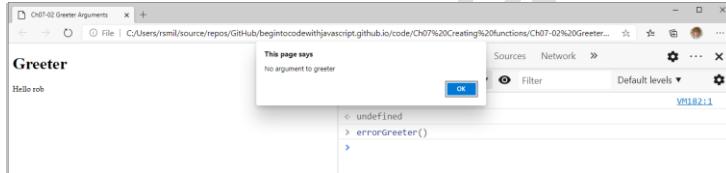
```
function errorGreeter(message) {
  if(message == undefined) {
```

```
    alert("No argument to greeter");
}
else
{
  let outputElement = document.getElementById("outputParagraph");
  outputElement.textContent = message;
}
```

The function above is called `errorGreeter`. It checks the message it has been supplied with and if the message is undefined it displays a warning alert. Otherwise it displays the message. This function is in the scripts of the example program. Try calling `errorGreeter` in the browser console with a missing argument:

```
> errorGreeter();
```

When you press enter you will see an alert because the function was not supplied with an argument.



If you call `errorGreeter` with an argument you will see that it displays the message as it should.

Question: What happens if I call a function with too many arguments?

We could try this:

```
> greeter("Hello world", 1,2,3, "something else");
```

If you call `greeter` as above JavaScript will not produce any errors, but the greeter function will ignore the extra arguments.

## Arguments and parameters

From the title of this section, you might expect that we will have a difference of opinion, but in JavaScript, the word *argument* has a particular meaning. In JavaScript, the word *argument* means "that thing you give to the call of a function."

```
greeter("Hello world");
```

In the above statement, the argument is the string "Hello world". So, when you hear the word argument you should think of the code that is making a call of the function. In JavaScript, the word *parameter* means "the name within the function that represents the argument." The parameters in a function are specified in the function definition.

```
function greeter(message)
```

This is the definition of a `greeter` function that has a single parameter. The parameter has the identifier `message`. When the function is called, the value of the `message` parameter is set to whatever has been given as an argument to the function call. Statements within the function can use the parameter in the same way as they could use a variable with that name.

## MAKE SOMETHING HAPPEN

### Parameters as values

When a function is called, the value of the argument is passed into the function parameter. What exactly does this mean? The following program contains a function (with the interesting name `whatWouldDo`) that accepts a single parameter. The function doesn't do much; it just sets the value of the parameter to 99.

```
function whatWouldDo(inputValue) {  
    inputValue = 99;  
}
```

This function is in the application **Ch07 Creating functions\Ch07-03 Parameters as values** folder in the example code. Open this application and then open the Developer View by pressing F12. First we are going to create a variable and put a value into it:

```
> var test=0;
```

When you press enter the console will create a variable called `test` that contains the value 100. We can use the `test` variable as an argument to the `whatWouldDo` function:

```
> whatWouldDo(test);
```

When the function has been called the question we must consider is "What value does `test` now contain?". Does it contain 0 (the value set when it was created) or does it contain 99 (the value set in the function `whatWouldDo`? We can answer the question by asking the console to display the value of `test`:

```
> test  
<- 0
```

When a function is called the value in the argument is *copied* into the parameter. So any changes to the parameter will not affect the argument at all.

## Multiple parameters in a function

A function can have multiple parameters. We might want a `greeter` function that can display text in different colors. We could add a second parameter that contains a color name:

```
function colorGreeter(message, colorName) {  
    var outputElement = document.getElementById("outputParagraph");  
    var elementColorStyle = "color:" + colorName;  
    outputElement.textContent = message;  
    outputElement.style = elementColorStyle;  
}
```

This function creates a style element using a supplied color name. The style is then applied to the output element along with the greeting text. We then provide arguments giving the string to be displayed and the color of the text:

```
colorGreeter("Hello in Red", "red");
```

### WHAT COULD GO WRONG

#### Muddled arguments

The `colorGreeter` function has two arguments which are mapped onto the two parameters that are used inside the function. The first argument has the greeting text and the second argument has the name of the color to be used for the text. It is important that these arguments are supplied in the correct order.

```
colorGreeter("red", "Hello in Red");
```

This call of `colorGreeter` has supplied the color name before the greeting text. In other words, they are the wrong way round. This would result in the `colorGreeter` trying to display the message "red" in the color "Hello in Red". This would not cause an error, but it would result in the program not doing what you want. So, when we make calls to functions, we need to make sure that the arguments that are supplied are given in the same order as the parameters. You can find the `colorGreeter` function in the sample **Ch07 Creating functions\Ch07-04 Color Greeter**.

## Using references as function arguments

In a JavaScript program there are essentially two kinds of variable. There are variables that contain a value and

variables that contain a reference to an object.

```
var age=12;
```

This statement creates a variable called `age` which contains a number with the value 12.

```
var outputElement = document.getElementById("outputParagraph");
```

This statement creates a variable called `outputElement` which contains a reference to an HTML element in the page document. This is how our JavaScript programs have located the page elements to be used to display information for the user. We can pass references into function calls as arguments:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Ch07-05 Reference arguments</title>
</head>

<body>
  <h1>Reference Arguments</h1>
  <p>
    <p id="outputParagraph">This is output text</p>71
  </p>

  <script>
    function makeGreen(element) {72
      element.style = "color:green";73
    }
  </script>
</body>
```

---

<sup>71</sup> Paragraph element for output

<sup>72</sup> makeGreen function

<sup>73</sup> Set the color of the element parameter to green

```
</html>
```

The HTML page above contains a function called `makeGreen`. The function `makeGreen` has a parameter called `element`. The function sets the style of the `element` parameter to the color “green”. A program can use this function to make any given HTML element green.

```
makeGreen(outputElement);
```

## MAKE SOMETHING HAPPEN

### Reference arguments

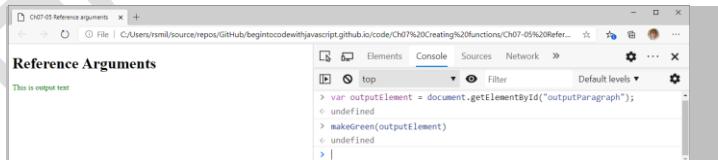
The function `makeGreen` is declared in the example application **Ch07 Creating functions\Ch07-05 Reference arguments** folder in the example code. Open this application and then open the Developer View by pressing F12. First we are going to get a reference to the display paragraph.

```
> var outputElement = document.getElementById("outputParagraph");
```

When you press enter a variable called `outputElement` is created that contains a reference to a paragraph on the screen. Now use this variable as an argument to a call of `makeGreen`:

```
> makeGreen(outputElement);
```

This will cause the text of the paragraph on the web page to turn green because the `makeGreen` function has set the style property of the element it has received a reference to:



The `makeGreen` function is designed to work on HTML display elements. What happens if we give it the wrong kind of argument. Try this:

```
> makeGreen(21);
```

The result of this statement is that nothing happens. No error is produced even though it is not meaningful to try and set the style of a numeric value to 21. Any mistakes that you make with function arguments will not produce errors, instead the program will just keep running.

## Arrays of arguments

There is another way that we can pass arguments into a JavaScript function, and that is as an *array* of items. We will discuss arrays in detail in the next chapter. An array is a form of *collection*. We have already seen one form of collection. The un-numbered list that we used to display ride information in the Theme Park in Chapter 06 held elements in the form of an array of items. We used the `for - of` loop construction to work through the array. If you're not sure about how this works, take a moment to return to Chapter 6 and read the section **Work through collections using `for-of`**. Within a function the keyword `arguments` means the array of arguments that were given when the function was called.

```
function calculateSum(){  
    let total = 0;74  
    for (value of arguments){75  
        total = total + value;76  
    }  
    var outputElement = document.getElementById("outputParagraph");77  
    outputElement.textContent=total;78  
}
```

The function `calculateSum` shows how this works. It uses a `for - of` loop to calculate the total of all the arguments given to the function call. This total value is then displayed on the page. The function can be called with any number of numeric arguments:

```
calculateSum(1,2,3,4,5,6,7,8,9,10);
```

---

<sup>74</sup> Set the total to 0

<sup>75</sup> Work through the arguments

<sup>76</sup> Add each value in the arguments to total

<sup>77</sup> Get a reference to the output element

<sup>78</sup> Display the output element

This would display the value 55. Of course, if you do something silly you might not get what you expect.

```
calculateSum(1,2,3,"Fred","Jim","Banana");
```

This call of the function would not cause an error. It would display the result "6FredJimBanana". This is because of the way that JavaScript combines strings and numbers. You can find out more on Chapter 4 in the section [Working with strings and numbers](#)

## Returning values from function calls

A function can return a value. You have seen this in many of the programs we've written. Here's an example:

```
var ageNo = Number(ageText);
```

This statement uses the [Number](#) function. The function accepts an argument (text expressing a number) and returns a value (the text as a numeric value). We can write our own functions that return a value.

### Dice spots method

In chapter 4 we created an application that displays a random dice. Each time we wanted a random dice value we had to perform some calculations. It would be useful to have these calculations in a function that we could use to get a random dice value.

```
function getDiceSpots() {  
    var spots = Math.floor(Math.random() * 6) + 1; 79  
    return spots;80  
}
```

This is the function [getDiceSpots](#). It calculates a random number of spots (check out the section [Creating a random dice](#) in Chapter 4 for details of how this works). The last statement in the function uses the JavaScript [return](#) keyword to return the calculated number to the caller. This function can be used whenever an application needs a dice value.

---

<sup>79</sup> Calculate the random result

<sup>80</sup> Return it to the caller

```
function doRollDice() {  
    var outputElement = document.getElementById("outputParagraph");81  
    var spots = getDiceSpots();82  
    var message = "Rolled: " + spots;83  
    outputElement.textContent = message;84  
}
```

The function `doRollDice` is called to display a dice value. It calls `getDiceSpots` to get the value and then display it. You can see this in action in the example application **Ch07 Creating functions\Ch07-07 Returning values** which works in exactly the same way as the dice example in **Ch04 Working with data\Ch04-02 Computer Dice** but uses the `doRollDice` function.

## Creating a customizable dice

Sometimes a program needs a random number in a different range from the 1 to 6 provided by a dice. We could use our newfound skills to make a function that accepts min and max values and then returns a random integer in that range.

```
function getCustomDiceSpots(min, max) {  
    var range = max - min + 1;85  
    var spots = Math.floor(Math.random() * (range)) + min;86  
    return spots;87  
}
```

The function `getCustomDiceSpots` is given two parameters that specify the minimum and maximum values of the required random number. It works out the range of numbers that are required (the difference between the maximum and minimum values), calculates a random number in this range and then adds that number to the minimum value to produce the number of spots. This number is then returned. One of the nice things about functions is that you

---

<sup>81</sup> Get a reference to the output paragraph

<sup>82</sup> Get the number of spots to display

<sup>83</sup> Build the message to display

<sup>84</sup> Display the message

<sup>85</sup> Calculate the range of values

<sup>86</sup> Calculate the random value

<sup>87</sup> Return result

don't have to understand how they work to use them.

```
var health = getCustomDiceSpots(70,90);
```

The above statement creates a variable called `health` that contains a random number in the range 70 to 90. It could be used in a video game to set the initial health value of a monster that fights the player. Sometimes the monster would be slightly harder to kill, which would make the game more interesting to play.

## The customizable dice app

I've used the `getCustomDiceSpots` function to make dice application where the user can select the min and max values for random numbers that they want to use to play a particular game:

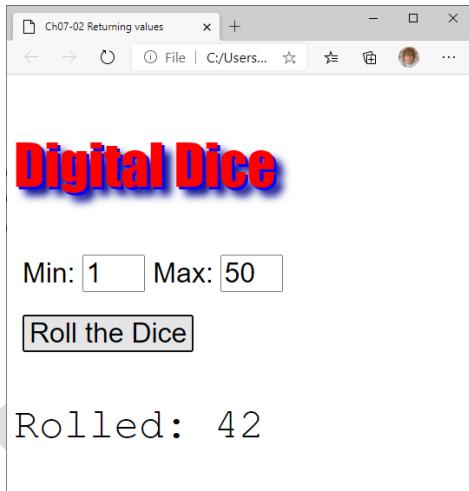


Figure 7.23 Customizable dice

Figure 7.2 shows how the application is used. The user fills in the text boxes for the the Min and Max values for the random number that they want and presses the "Roll the Dice" button to display a random number in that range. The settings above will produce a value in the range 1 to 50 that could be used to play a bingo game. Let's take a look at the code.

```

<!DOCTYPE html>
<html lang="en">

<head>
  <title>Ch07-02 Returning values</title>
  <link rel="stylesheet" href="styles.css">88
  <script src="customdice.js"></script>89
</head>

<body>
  <p class="menuHeading">Digital Dice</p>

  <p class="menuText">
    Min:
    <input class="menuInput" type="number" id="minNoText" value="1" min="1" max="100">90
    Max:
    <input class="menuInput" type="number" id="maxNoText" value="6" min="2" max="100">91
  </p>
  <button class="menuText" onclick="doRollDice('minNoText', 'maxNoText', 'outputParagraph')">
    Roll the Dice</button>92
  </p>
</p>

<p id="outputParagraph" class="numberDisplay">Press the roll button</p>93

<script>
</script>
</body>

</html>

```

---

<sup>88</sup> The stylesheet for the application.

<sup>89</sup> The JavaScript source file for the application.

<sup>90</sup> Min input

<sup>91</sup> Max input

<sup>92</sup> Button that rolls the dice

<sup>93</sup> Output paragraph

We have seen this kind of page before. This is some text to tell the user about the application. There are two input fields which are used to get the minimum and maximum values for the random number to be produced. There is a button to trigger the generation of the value and an output paragraph that displays the result. This page is very similar to the one used by the theme park ride selector application. The function `doRollDice` is called when the user clicks the button to roll the dice.

```
onclick="doRollDice('minNoText', 'maxNoText', 'outputParagraph')"
```

This is the `onclick` attribute for the dice roll button. This attribute contains a string of JavaScript code which will be obeyed when the button is clicked. If you're not clear on how this works, take a look in the section **Using a button** in chapter 2 for a refresh. The JavaScript that runs when the button is clicked makes a call of the function `doRollDice`.

```
doRollDice('minNoText', 'maxNoText', 'outputParagraph')
```

The `doRollDice` function has three arguments which are the id strings of the `minNoText`, `maxNoText` and `outputParagraph` elements in the web page. The function gets the minimum and maximum values from elements identified by the first two arguments to the function call. It then calculates the random result and then displays the result using the output element identified in the third element. We can take a look at this function to see how it works:

```
function doRollDice(minElementName, maxElementName, outputElementName) {  
  
    var min = getNumberFromElement(minElementName);94  
    var max = getNumberFromElement(maxElementName);95  
  
    var spots = getCustomDiceSpots(min, max);96  
  
    var message = "Rolled: " + spots;97  
  
    var outputElement = document.getElementById(outputElementName);98
```

---

<sup>94</sup> Get the minimum value

<sup>95</sup> Get the maximum value

<sup>96</sup> Get the number of spots

<sup>97</sup> Build the output message

<sup>98</sup> Get the output element

```
        outputElement.textContent = message;99  
    }
```

The function looks very small. It gets in the minimum and maximum values, calls `getCustomDiceSpots` to get the random value and then display the result. The function is small because it uses another function called `getNumberFromElement` to get the maximum and minimum values:

```
function getNumberFromElement(name)  
{  
    var element = document.getElementById(name);100  
    var text = element.value;101  
    var result = Number(text);102  
    return result;103  
}
```

This function is given the identifier of an input element on the web page as a parameter. It fetches a number from the specified input element. We can use this function in any program that needs to read numbers from the user. It gets a reference to the element with the identifier which has been supplied as a parameter, gets the text from this element, converts the text into a number and then returns the number to the caller. The function is called twice in the dice application; once to read the maximum value and once to read the minimum value. If I had an application that needed 10 inputs to be read from the screen I could call this function 10 times in the application.

#### PROGRAMMER'S POINT

#### Designing with functions

Functions are a very useful part of the programmer's toolkit and form an important part of the development process. Once you've worked out what a customer wants the application to do, you can start thinking about how you'll break down the program into functions. Once you've specified the behavior of each function in the application, you can write the function headers (in other words, pick the function name, the parameters, and any return

---

<sup>99</sup> Display the number

<sup>100</sup> Get the input element

<sup>101</sup> Get the text from the input element

<sup>102</sup> Convert the text into a result

<sup>103</sup> Return the result

value) and then you could even get someone else to write that function for you.

Functions save you from writing too much code. Often, you find that as you write a program, you write code that repeats a particular action. If you do this, you should consider taking that action and turning it into a function. There are two reasons why this is a good idea:

First, it saves you writing the same code twice; and secondly, if a fault is found in the code, you only need to fix it in one place.

Functions also make a program easy to test. You can regard each function as a “data processor.” Data goes into the function via the arguments, and output is produced via the return value. We can write what is called a “test harness” to call a function with test data and then check to ensure the output is sensible. In other words, we can make a program that tests itself. We will look at this later in the book.

## Add error handling to an application

The Digital Dice program works well as long as we use it correctly. However, it is not without its problems. It is quite easy to enter invalid values for the minimum and maximum values. You can experiment with the application in the examples folder Ch07 Creating functions\Ch07-08 Custom dice to discover what happens when it fails.



Min:  Max:

Rolled: 26

Figure 7.24 Dice errors

Figure 7.3 shows how the application can be used incorrectly. A minimum value of 50 and a maximum value of 2 is not correct. To make matters worse, the application has displayed a result which might be incorrect. We have no way of knowing if it has worked correctly because we are not sure how the `getCustomDiceSpots` function behaves if it is given incorrect maximum and minimum values. It turns out that there are also other ways to cause the program problems. A user could press “Roll the Dice” when the Min and Max input areas are both empty. We need to fix all these things. Fortunately, we have the JavaScript skills that we need to sort this out.

## Finding all the errors in advance

One thing that separates a good programmer from a great programmer is the way that they handle errors. A great programmer will start a project by thinking of all the things that could go wrong when the user starts to use their program. It's unlikely that they will think of every possible error (users can be very inventive when it comes to breaking things) but they will write down all the errors they can think of and then set out to deal with each error. Then, as the project continues, a great programmer will keep looking for possible errors and keep making sure that the errors that they know about are being handled correctly. In the case of the Digital Dice application I can think of three things that might go wrong:

The user might not enter anything into the max or min inputs.

The user might enter a number outside the correct range for one of the max or min inputs.

The user might enter a min value which is greater than or equal to the max value.

Now that we have identified the errors, the next thing we need to decide is what the program should do when they are detected. If you are writing a program for a customer, you **must** discuss with them what is supposed to happen. Do they want the program to assume default values (perhaps a min of 1 and a max of 6) or do they want error messages to be displayed? Do they want the error to be a message on the screen or a pop-up alert? These are all important questions that the programmer alone can't answer. The worst thing that can happen is that you make an assumption concerning what the customer wants. In my experience making assumptions is a neat way of doubling your workload. You show the customer what you've made, they tell you it is not what they want, and you have to do it again. In the case of our customer we have been told that the program should display a message and turn any invalid input areas on the screen red so that the user knows what to fix. So let's see how we can use our JavaScript skills to solve this problem.

Let's start with the error display. These are two style classes that I'm going to use to on the input elements on the web page. The `menuInput` style is the "normal" style. The program will set the `menuInputError` style to an input element that contains an invalid entry. Note that this has the background color set to red.

```
.menuInput {  
    background-color: white;  
    font-size: 1em;  
    width: 2em;  
}  
  
.menuInputError {  
    background-color: red;  
    font-size: 1em;  
    width: 2em;  
}
```

Our application uses a function called `getNumberFromElement` to get a number from an input element. Let's have a look at an improved function to deal with input errors:

```
function getNumberFromElement(elementID){  
    var element = document.getElementById(elementID);104  
    var text = element.value;105  
  
    var result = Number(text);106  
  
    if(isNaN(result)){107  
        // fail with bad number input  
        element.className="menuInputError";108  
        return NaN;  
    }  
  
    // get the max and min values from the input field  
    var max = Number(element.getAttribute("max"));  
    var min = Number(element.getAttribute("min"));  
  
    if(result>max || result<min){109  
        // fail because outside range  
        element.className="menuInputError";  
        return NaN;  
    }  
  
    // if we get here the number is valid  
    // set to normal background  
    element.className="menuInput";  
  
    return result;  
}
```

---

<sup>104</sup> Get the input element

<sup>105</sup> Get the text from the input element

<sup>106</sup> Convert the text into a number

<sup>107</sup> Make sure that we have a number

<sup>108</sup> Indicate an error

<sup>109</sup> Make sure that the number is in range

## CODE ANALYSIS

### The getNumberFromElement function

This function bears closer inspection. Here are answers to some questions about it:

Question: What does `NaN` mean? And what is the `isNaN` function doing?

We first saw `NaN` in chapter 4 when we wrote our first JavaScript to read numbers from the user. It is how JavaScript represents a value which is “not a number”. In this case the function is using the `Number` function to convert the text entered by the user into a number. If the `Number` function fails, either because the text is not numeric or the text is empty, the `Number` function returns the value `NaN` because it could not create a number result. The `isNaN` function takes an argument and returns `true` if the argument is not a number. The `getNumberFromElement` function uses `isNaN` to test the result value produced by the call to `Number`. If the result is not a number the function sets the style of the element to indicate that the input was in error and then returns the value `NaN` to the caller to indicate that a number could not be read.

Question: What are the `min` and `max` attributes? Where do they come from?

We can add `min` and `max` attributes to an HTML input element to help the browser validate numbers entered by the user. However, the browser does not enforce min and max completely, our program must also play a part in making sure that numbers are entered in the correct range.

```
<input class="menuInput" type="number" id="minNoText" value="1" min="1" max="99">
```

The HTML above is the input element for the minimum value. The `min` and `max` attributes are set to 1 and 99 respectively. The `getNumberFromElement` function can get the values of these attributes and then use them to perform input validation:

```
var max = Number(element.getAttribute("max"));
var min = Number(element.getAttribute("min"));
```

The function can then use these values to validate the number entered by the user.

```
if(result>max || result<min){
    // fail because outside range
    element.className="menuInputError";
    return NaN;
}
```

This is a really good way of handing minimum and maximum values since the values are obtained from the elements on the HTML Page and are not held in the JavaScript code.

Question: Why does the function sometimes return the value `NaN`?

We've discovered that it is possible for the `getNumberFromElement` to be unable to deliver a result. This happens when the user doesn't type in a numeric value or when the value they enter is out of range. The function needs a

way of informing the caller that it does not have a valid result, and returning `NaN` is how you do this in JavaScript. This makes using a function that returns a value a kind of “buyer beware” proposition, as the code that called the function must test to make sure that the result that was delivered was valid. We will see this in action when we look at the error handing version of `doRollDice`.

These changes to the `getNumberFromElement` function have addressed the first two errors that we identified. The user must now enter a valid number or the `getNumberFromElement` function returns the value `NaN` (not a number). Now the `doRollDice` function needs to be updated. This version checks that the min and max values are both numbers. It also checks that the minimum value is less than the maximum value. If either of these conditions is not met the function displays an alert.

```
function doRollDice (minElementName, maxElementName, outputElementName) {  
  
    var outputElement = document.getElementById(outputElementName);110  
  
    var minRand = getNumberFromElement(minElementName);111  
    var maxRand = getNumberFromElement(maxElementName);112  
  
    if (isNaN(minRand) || isNaN(maxRand)) {113  
        outputElement.textContent="Invalid range values";114  
        return;  
    }  
  
    if (minRand >= maxRand) {115  
        outputElement.textContent="Minimum above maximum";116  
        return;  
    }  
  
    var spots = getCustomDiceSpots(minRand, maxRand);117  
    var message = "Rolled: " + spots;118
```

<sup>110</sup> Get the output element

<sup>111</sup> Get the minimum value

<sup>112</sup> Get the maximum value

<sup>113</sup> Check for non-numbers

<sup>114</sup> Display error message

<sup>115</sup> Check maximum and minimum

<sup>116</sup> Display error message

<sup>117</sup> Get the spots value

<sup>118</sup> Build the reply

```
    outputElement.textContent = message;119  
}
```

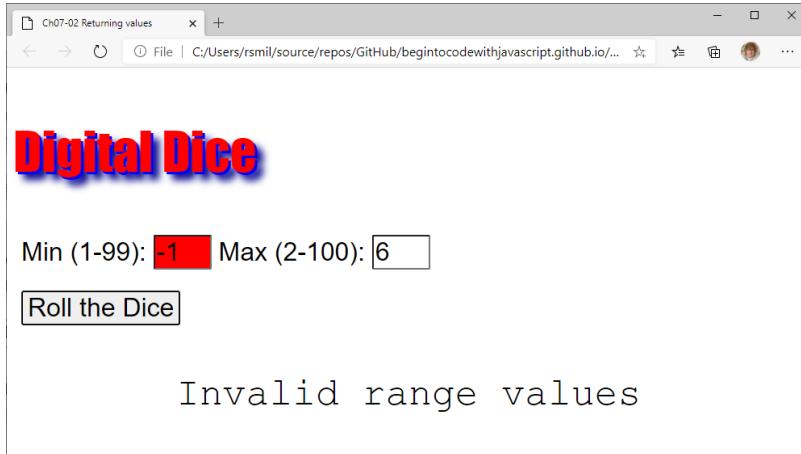


Figure 7.25 Error handling

Figure 7.4 above shows the error handling in action. Note that I've also made another change to the application so that it now shows the user the minimum and maximum values that are allowed for each input. This is a great way to reduce the chances of any errors in the first place.

#### PROGRAMMER'S POINT

##### You need to attack errors

I can be a very depressing person to have at a project meeting, particularly at the start of the project. I'll keep searching for things that could go wrong, and things we need to worry about. I'll also try and involve the customer in this process, and make sure we find out what should happen when things fail. I'm not doing this because I have a tendency to worry about things (although I probably do). I'm doing this because I don't want to be caught out by unexpected problems. If you are making something you care about you should take the same

<sup>119</sup> Display the reply

approach.

## Local variables in JavaScript functions

Imagine several cooks working together in a kitchen. Each cook is working on a different recipe. The kitchen contains a limited number of pots and pans for the cooks to share. The cooks would need to coordinate so that two of them didn't try to use the same pot. Otherwise, we might get sugar added to our soup and custard instead of gravy on our roast beef.

The designers of JavaScript faced a similar problem when creating functions. They didn't want functions to fight over variables in the same way that two cooks might fight over a particular frying pan. You might think that it would be unlikely that two functions would try to use variables with the same name, but this is actually very likely.

Many programmers (including me) have an affection for the variable name `i`, which they use for counting. If two functions use a variable called `i` and one function calls the other function, this could lead to programs that don't work properly because the second function might change `i` to a value that the first function didn't expect.

JavaScript solves this problem by providing a way that each function its own local variable space. This is the programming equivalent of giving each cook their own personal set of pots and pans. Any function can declare a local variable called `i` that is specific to that function call. We declare a variable as local by using the keyword `var`.

```
function fvar2() {
  var i=99;
}

function fvar1(){
  var i=0;

  fvar2();

  console.log("The value of i is:" + i);
}
```

When a function returns, all local variables are destroyed. The code above shows two functions that contain local variables. Both `fvar1` and `fvar2` use a variable called `i`. If we call `fvar1` JavaScript follows this sequence:

5. The function `fvar1` is called.
6. The first statement of `fvar1` creates a variable called `i` and sets it to `0`.
7. The second statement of `fvar1` makes a call to `fvar2`.
8. The first, and only, statement of `fvar2` creates a variable called `i` and sets it to `99`

9. The function `fvar2` finishes and control returns to the third statement of `fvar1`.

10. The third statement of `fvar1` prints out the value of `i`.

The question we must consider is, “What value is displayed in the console?” Is it the value 0 (which is set inside `fvar1`) or is it 99 (which is set inside `fvar2`)? If you’ve read the first part of this section, you know the value that will be printed is 0. The variables both have the same name (they are both called `i`), but they each “live” in different functions.

JavaScript uses the word *scope* when talking about variable lifetimes. The scope of a variable is that part of a program in which a variable can be used. Each version of `i` has a scope which is limited to the function body in which it is declared. This form of isolation is called encapsulation. Encapsulation means that the operation of one function is isolated from the operation of other functions. Different programmers can work on different functions with no danger of problems being caused by variable names clashing with each other. Now take look at this pair of functions:

```
function f2() {  
    i=99;  
}  
  
function f1(){  
    i=0;  
  
    f2();  
  
    console.log("The value of i is:" + i);  
}
```

Can you spot the difference? These two functions don’t use the keyword `var` to declare their variables. Is this important? Yes. It turns out that variables declared without the `var` keyword are made global to the whole application, so `f1` and `f2` are now sharing a single global variable called `i`. If we call the function `f1` it will print the value 99 because the call of `f2` will change the value of the global variable `i`. This means that you should always use `var` to declare variables in your functions unless you specifically want to share them. In chapter 4, in the section **Global and local variables** we discovered a situation where a program needs to have a variable which has *global scope*. Now we are starting to understand how this works.

## Local variables using `let`

In chapter 4, in the section **The JavaScript for loop**, we discovered that we could declare local variables using the keyword `let`. Variables declared using `let` are discarded when a program leaves the block of statements where they were declared, whereas variables declared using `var` are discarded when the program leaves the function in which they are declared.

```
function letDemo(){
  var i=0;120
  {
    let i=1;121
    var j=2;122
    console.log("The value of let i is:"+i);
  }

  console.log("The value of var i is:"+i);
  console.log("The value of var j is:"+j);
}
```

The function `letDemo` shows how this all works. It contains two versions of `i`. The first version is declared as `var` and exists for the entire function. The second version is declared inside a block and is discarded when the program leaves that block. Note that within the block it is not possible to use the outer version of `i` because any references to `i` will use the local version. In this case we say that the outer variables is *scoped out*. The function also contains a variable called `j` which is declared inside the inner block. However, because `j` is declared as `var` it can be used anywhere inside the function. These functions are declared in the example program at [Ch07 Creating functions\Ch07-10 Variable scope](#).

If you find this confusing, then I apologize. The best way to understand it is to consider the problem that they are trying to solve (stopping lots of cooks fighting over the same pots and pans) and then work from there. The most important point is to always use `var` or `let` when you create a variable. Otherwise you may find yourself a victim of very strange bugs in your programs.

## MAKE SOMETHING HAPPEN

### Double Dice

Quite a few games use two dice rather than one. Make a version of the dice program that displays the results from two dice. You could even make this super-customizable so that the user can select the range of values to be produced by each dice. If you are bit stuck with this; remember that there is nothing wrong with the `onClick` behavior of a button calling two methods. And that two dice are just one dice times two. Take a look at my example solution in the examples at [Ch07 Creating functions\Ch07-11 Two Dice](#) to discover the sneaky way I made the program work.

<sup>120</sup> This version of `i` exists for the whole function

<sup>121</sup> This version of `i` only exists in this block

<sup>122</sup> `j` exists for the whole function

# What you have learned

In this chapter, you learned how to take a block of code and turn it into a function that can be used from other parts of the program.

- You've seen that a function contains a header, which describes the function, and a block of code that is the body of the function. The function header supplies the name of the function and any parameters that are accepted by the function. When a function is called, the programmer supplies an argument that matches each parameter.
- Parameters are items that the function can work on. They are passed by value, in that a copy is made of the argument given in the function call. If the function body contains statements that change the value of the parameter, this change is local to the function body. A function can be given arguments which are references to objects in which case the function can interact with the object to which the reference refers.
- A function can regard parameters as an array and can work through the arguments supplied to the function by using the arguments keyword.
- A function returns a single value. This is achieved by using the return statement, which can be followed by a value to be returned. If no value is returned, or the function does not perform a return statement, the function will return a special JavaScript value called undefined, which is used to denote a missing value.
- The scope of a variable in a program is that part of the program in which the variable can be used. Variables created inside the body of a function using the keyword var have scope which is local to the function and cannot be used by statements outside that function.
- When creating software it is important to consider potential errors at the start of the project rather than finding out about them later.

Here are some questions that you might like to ponder about the use of functions in programs:

**Question:** Does using functions in programs slow down the program?

**Answer:** Not normally. There is a certain amount of work required to create the call of a function and then return from it, but this is not normally an issue. The benefits of functions far outweigh the performance issues.

**Question:** Can I use functions to spread work around a group of programmers?

**Answer:** Indeed, you can. This is a very good reason to use functions. There are several ways that you can use functions to spread work around. One popular way is to write placeholder functions and build the application from them. A function will have the correct parameters and return value, but the body will do very little. As the program develops, programmers fill in and test each function in turn.

**Question:** How do I come up with names for my functions?

**Answer:** The best functions have names given in a verb-noun form. `getNumberFromElement` is a good name for a

function. The first part indicates what it does, and the second part indicates where the value comes from. I find that thinking of function names (and variable names, for that matter) can be quite hard at times.

**Question:** What do I do if I want to return more than one value from a function?

**Answer:** JavaScript functions can only return a single value. However, in the next chapter we will discover how to create data structures that can contain multiple values that can be returned by functions.

Pre-release

# 8

## Storing data

### What you will learn

You might find this surprising, but you've already learned most of what you need to know to tell a computer what to do. You can write a program that gets data from the user, stores it, makes decisions based on data values, and repeats behaviors using loop constructions. You also know how to use functions to break a solution down into components. These are the fundamentals of programming, and all programs are built on these core capabilities.

However, there is one more thing you need to know before you can write most any kind of program. You need to be able to write programs that can manage large amounts of data. In this chapter, you'll learn just that, along with some extremely powerful JavaScript techniques for working with the HTML Document Object Model (DOM) that underpins the display of a page.

### Collections of data

Your fame as a programmer is beginning to spread far and wide. Now the owner of an ice-cream parlor comes to you and asks that you write a program to help her track sales results. She currently has 6 ice-cream stands around the city, and each day they sell ice-cream treats. What she wants is quite simple—she wants a program where she can enter the sales value from each stand and then get the total sales from all of them and the best and worst sales. She wants to use this analysis to help her plan the location of her stands and reward the best sellers. If you get this right, you might be getting some free ice cream, so you agree to help.

## Ice Cream Sales

### Ice Cream Sales

Stand 1 sales:

Stand 2 sales:

Stand 3 sales:

Stand 4 sales:

Stand 5 sales:

Stand 6 sales:

Total: 1270 Highest: 300 Lowest: 120

Figure 8.26 Ice cream calculator

As usual, the starting point for your program is a design that shows how the application should look. Figure 8.1 shows what the customer has drawn up. She wants to enter the sales values and then press a “Calculate” button to display the analysis. Because you have read chapter 7 and you know about error handling you ask her about the upper and lower limits on the data values and what the program should do if any of the values are out of range. Your customer hasn’t thought of this, but you discuss the application and agree on some additions to the design.

 Ice Cream Sales

Stand 1 sales (0-10000):

Stand 2 sales (0-10000):

Stand 3 sales (0-10000):

Stand 4 sales (0-10000):

Stand 5 sales (0-10000):

Stand 6 sales (0-10000):

Please enter numbers in the correct range

Figure 8.27 Ice cream calculator errors

Figure 8-2 shows the revised design. If a sales value is missing, too large or too small the input area will be highlighted in red and the program will display an error message. This is the same behavior as the number input for the **Custom Dice** program you created in Chapter 7, so you should be able to use some of the code from that for this program.

Now that you have a specification, all you must do now is write the actual program itself. The program will need variables to hold the sales values entered by the user and it can use logical expressions to compare sales values and choose the largest (so that it can find the biggest and smallest sales). You also know from earlier chapters how to display results to the user by setting the `innerText` of a paragraph in the page. We could start with the HTML for the application:

```

<!DOCTYPE html>
<html lang="en">

<head>
<title>Ice Cream Sales</title>
<link rel="stylesheet" href="styles.css">
<script src="icecreamsales.js"></script>
</head>

<body>
<p class="menuHeading"> &#127846; Ice Cream Sales</p>

<p>
<label class="menuLabel" for="s1SalesText">Stand 1 sales (0-10000):</label>
<input class="menuInput" type="number" id="s1SalesText" value="0" min="0" max="10000">
</p>

<p>
<label class="menuLabel" for="s2SalesText">Stand 2 sales (0-10000):</label>
<input class="menuInput" type="number" id="s3SalesText" value="0" min="0" max="10000">
</p>

<p class="menuText" id="outputParagraph"></p>

<p>
<button class="menuText" onclick="doCalc()>Calculate</button>
</p>

</body>

</html>

```

This HTML contains an input field for each of the 6 sales that are to be entered by the user (the listing above only shows the first two to save space). Each input field has an `id` attribute so that the program can find it and load the value stored in it. The input field uses a new feature of HTML that we have not seen before, the label:

## Labeling HTML input elements

Sales 1(0-10000):

**Figure 8.28** Single data entry element

Each of the elements on the application has a label next to it. In Figure 8.3 the label shows that the input is for the sales of the ice cream stand number 1. In earlier programs we have just displayed text next to the input element to label it, but it turns out that there is a better way to label inputs on an HTML page. We can use the `label` element to explicitly link a label with an input.

```
<p>
  <label class="menuLabel" for="s1SalesText">Stand 1 sales (0-10000):</label>
  <input class="menuInput" type="number" id="s1SalesText" value="0" min="0" max="10000">
</p>
```

The label and the input are held inside a paragraph. The label contains the label text for the input. It also contains a `for` attribute that matches the id of the target of the label. Both the label and the input are assigned stylesheet classes to manage their appearance on the page.

## Calculate total ice cream sales

The HTML contains a button which the users presses to perform the calculation. When this is pressed the JavaScript function `doCalc` is called. This function must get the values from the input elements on the HTML page and then calculate the results that the user wants to see.

```
Function doCalc() {
  var sales1, sales2, sales3, sales4, sales5, sales6;
  sales1 = getNumberFromElement("s1SalesText");
  sales2 = getNumberFromElement("s2SalesText");
  sales3 = getNumberFromElement("s3SalesText");
  sales4 = getNumberFromElement("s4SalesText");
  sales5 = getNumberFromElement("s5SalesText");
  sales6 = getNumberFromElement("s6SalesText");123
  var total = sales1 + sales2 + sales3 + sales4 + sales5 + sales6;
```

This is the first part of the function `doCalc`. The function runs when the user presses the `Calculate` button. The function runs it uses another function with the name `getNumberFromElement` that we created in the section **The customizable dice app** in Chapter 7. This function is supplied a string containing the id of the element in the HTML and it returns the value that the element contains. Once each sales value has been obtained the program then works out the

---

<sup>123</sup> Get the sales values from the inputs

total sales value by adding all the sales together.

## Find the highest ice cream sales

Finding the total sales was easy, so now we need to add the code to find the highest and the lowest sales. Fortunately we know about the use of relational operators to compare values and logical expressions to combine the comparisons. You can refresh your understanding of these in Chapter 5 in the section Boolean Expressions. With that knowledge we can write a JavaScript statement that can determine if the sales from ice cream stand 1 are the largest:

```
var highestSales;

if(sales1>sales2 && sales1>sales3 && sales1>sales4 && sales1>sales5 && sales1>sales6) {
    highestSales = sales1;124
}
```

The code above uses an `if` construction to decide whether the sales from stand 1 are the largest. The sales from stand 1 are largest if `sales1` is greater than the sales values from the other five stands. The logical expression checks to see if the sales from stand 1 are greater than the sales from stand 2, stand 3, stand 4 and stand 5. The expression sets the variable `highestSales` to the value in `sales1` if `sales1` is the largest value.

One problem with this design is that our program needs a test like this for all 6 sales values. So we need to write five more tests. Then we need another 6 tests to determine the lowest sales value. And if the ice cream parlor owner sets up more ice cream stands this would make our program even more complex.

We have hit this problem because we have started from the wrong place. Sometimes it is a good idea to use an existing program as the basis of a new one, but we have discovered that a design created to work with two values (the maximum and minimum values for a random number) does not scale up very well. We must do a lot of work to extend this structure to read in and manipulate six values.

I've watched a lot of people learn to program and I've seen quite a few work much harder than they needed to because they took something that they know how to do and tried to extend it to do a different task. This is a bit like trying to dig the foundations of a house with a spoon just because you know how use a spoon but don't fancy learning how to drive the mechanical digger. We have already decided that the best programmers are "Creatively Lazy". This might be a good point to try for some creative laziness. If you find yourself having to repeat chunks of code, you might want to stop and think about a better way of doing this. If you really want to do this the hard way, I've put my partially completed version of the program in the examples folder **Ch08 Storing data\Ch08-01 Unworkable Ice Cream Sales**.

---

<sup>124</sup> Test to see if `sales1` is the highest sales

## Creating an array

JavaScript also provides an *array* component that can be used to create indexed storage of a collection of data values. Each item in an array is called an *element*. A program addresses a particular element in the array by using an *indexer*, which is a number that identifies the element in the array. Some programmers refer to an indexer as a *script*. Let's investigate how arrays work.

### MAKE SOMETHING HAPPEN

#### Investigating arrays

Start by opening the application in the **Ch08 Storing data\Ch08-02 Array Ice Cream Sales** folder in the example code. This application works, you can enter some data and check the results. We are just going to investigate arrays from the developer console. Open the Developer View by pressing the function key F12. Now we can start typing commands into the JavaScript command prompt. The first thing we are going to do is create an empty array. Type in the statement below:

```
> var sales = []
```

When you press enter the JavaScript console creates an empty array with the identifier `sales`. This action does not return a result, so the console displays the message `undefined`:

```
> var sales = []
<- undefined
```

We can ask the JavaScript console to show the contents of any variable by entering the name of that variable. This works with an array too. Type in the identifier `sales` and press Enter.

```
> sales
<- []
```

The console shows us that the `sales` array is empty by displaying two brackets with nothing between them. Now let's store a sales value in the array. This statement will add an element at the start of the array:

```
> sales[0] = 100
```

When you press the Enter key JavaScript must store the value 100 in the array element with the index 0.

```
> sales[0] = 100;
<- 100
```

This element does not exist, so JavaScript adds it to the array automatically. Because an assignment statement returns the value assigned, the console displays the value 100, which is what was assigned to the array element. We can view the contents of the array again to see what has changed. Type in the identifier `sales` and press Enter.

```
> sales  
<- [100]
```

The console shows us that the sales array now contains a single value. We can use an array element as we would use any other variable. What do you think the following statement would do? Type it in and find out.

```
> sales[0] = sales[0] + 1  
<- 101
```

This statement adds 1 to the element at the start of the array. Let's add a second element to the array. We create the new element just by storing something in it.

```
> sales[1] = 150;  
<- 150
```

This element does not exist, so JavaScript adds it to the array automatically. Because an assignment statement returns the value assigned, the console displays the value 150. Let's see what has changed in the array. Type in the identifier `sales` and press Enter.

```
> sales  
<- [101, 150]
```

JavaScript shows us the element at the start of the array (which has 101 in it) and the next element (which has 150 in it). JavaScript will add new elements to the array each time whenever it needs to. You can think of the array "stretching" to hold whatever items it needs to.

The JavaScript in the example program **Ch08 Storing data\Ch08-02 Array Ice Cream Sales** uses arrays to store the sales values. Below you can see the statements in the program that get the data into the array for analysis.

```
var sales = [];125  
sales[0] = getNumberFromElement("s0SalesText");126  
sales[1] = getNumberFromElement("s1SalesText");  
sales[2] = getNumberFromElement("s2SalesText");  
sales[3] = getNumberFromElement("s3SalesText");  
sales[4] = getNumberFromElement("s4SalesText");  
sales[5] = getNumberFromElement("s5SalesText");
```

The statements above create an array called `sales` and then set elements in the array with the sales values from the elements on the HTML page. Note that because the array elements are indexed from 0 I've changed the ids

---

<sup>125</sup> Create the sales array

<sup>126</sup> Store sales values in each element of the array

for the input elements to match. In other words, the element at the start of the array has an index value of 0 and is assigned a value from an element with an id that matches.

```
sales[0] = getNumberFromElement("s0SalesText");
```

#### PROGRAMMER'S POINT

#### Counting from zero is just something you will have to get used to

If you are used to counting from 1 you will find that the way that arrays count from zero is a bit irritating. However, you will just have to get use to it, as it is how JavaScript (and lots of other programming languages) work. Just remember that an array containing 6 elements (like the sales array above) will index these in the range 0 to 5. Note that this means that sometimes you will have count in one way for the program (array elements 0 to 5) and another for the user (ice cream stands 1 to 6).

## Processing data in an array

Storing the sales in an array makes it easy for a program to work through the elements in it. We can access individual elements by using index values, or we can use the `for - of` loop to work through the elements.

```
total = 0;127  
  
for(let saleValue of sales){128  
    total = total + saleValue;129  
}
```

The statements above work through the `sales` array and calculate the total number of sales. The great thing about this code is that it would work for any size of array.

## Finding the highest and lowest sales values

Another request the customer made was for the program to find the highest and lowest sales in the set of results.

---

<sup>127</sup> Set the total to zero

<sup>128</sup> Work through the sales array

<sup>129</sup> Add each sales value to the total

Before you write the code to do this, it's worth thinking about the algorithm to use. In this case, the program can implement an approach very similar to one that a human would use. If you gave me some numbers and asked me to find the highest value, I would compare each number with the highest value I had seen so far and update the current highest value each time I found a larger one. In programming terms, this algorithm would look a bit like the following. (This is not JavaScript as such; a description like this is sometimes called *pseudocode*. It looks something like a program, but it is there to express an algorithm, not to run inside a computer.)

```
if(new value > highest I've seen)
    highest I've seen = new value
```

At the start we set the "highest I've seen" value to the value of the element at the start of the array since this is the highest we've seen at the start of the process. I can put this behavior into a function which will calculate the highest value in any array that is passed into it.

```
function getHighest(inputArray){

    var max = inputArray[0];130

    for(let value of inputArray) {131
        if(value>max){132
            max = value;133
        }
    }
    return max;134
}
```

A program can use the `getHighest` function to get the highest value from any array of values. We can create helper functions called `getHighest`, `getLowest` and `getTotal` to use in our application.

---

<sup>130</sup> Set the maximum to the start element

<sup>131</sup> Work through the input array

<sup>132</sup> Test for a new maximum

<sup>133</sup> Set the maximum to the new value

<sup>134</sup> Return the maximum value

```
function doCalc() {  
  
    var sales = [];  
  
    sales[0] = getNumberFromElement("s0SalesText");  
    sales[1] = getNumberFromElement("s1SalesText");  
    sales[2] = getNumberFromElement("s2SalesText");  
    sales[3] = getNumberFromElement("s3SalesText");  
    sales[4] = getNumberFromElement("s4SalesText");  
    sales[5] = getNumberFromElement("s5SalesText");  
  
    var totalSales = getTotal(sales);  
    var highestSales = getHighest(sales);  
    var lowestSales = getLowest(sales);  
  
    var result = "Total:" + totalSales + " Highest:" + highestSales + " Lowest:" + lowestSales ;  
  
    var outputElement = document.getElementById('outputParagraph');  
  
    outputElement.textContent = result;  
  
}
```

The completed version of `doCalc` is shown above. It creates a `sales` array, uses the analysis functions to create the results and then displays them. You can find this version of the program in the examples folder **Ch08 Storing data\Ch08-02 Array Ice Cream Sales**.

## WHAT COULD GO WRONG

### Detecting invalid sales values

The ice cream sales application configures the input elements in the HTML document to have an input type of number.

```
<input class="menuInput" type="number" id="s0SalesText" value="0" min="0" max="10000">
```

However, using this input type would not stop the user from directly entering invalid values:

Stand 1 sales (0-10000):

In this case the user has accidentally caught the minus key when typing in the number and entered a negative

value. The good news is that the `getNumberFromElement` function that the program is using to read the sales value will return a result of `NaN` (not a number) when it reads a value outside the min and max settings for an element. (find out more about how this works in Chapter 7 in the section **Add error handling to an application**).

The bad news is that our program will not handle this correctly. We can fix the problem by using the way that JavaScript works with numbers. Remember that any mathematical calculation involving the value “not a number” will return a result of “not a number”. So, if the total of all the sales values is not a number this means that at least one of the sales values is not a number. This is something our program can test for.

```
var totalSales= getTotal(sales);
var result;

if (isNaN(totalSales)) {
    result = "Please enter numbers in the correct range"
}
else {

    var highestSales = getHighest(sales);
    var lowestSales = getLowest(sales);

    result = "Total:" + totalSales + "Highest:" + highestSales + "Lowest:" + lowestSales;
}
```

The code above shows how this works. If the `totalSales` value is not a number, the `result` variable is set to an error message. Otherwise `result` is set to the calculated values. You can find this version of the program in the example folder **Ch08 Storing data\Ch08-03 Error handling Ice Cream Sales**.

## Build a user interface

The program that we have created meets the specification set by the customer. Which makes the next phone call from our customer rather unwelcome. She says has some good news. Her company has just got another two ice cream stands. This means that we will have to add two extra elements to the HTML for the application and then make sure that the data analysis program loads these new values in correctly.

A way to make our lives easier, both in terms of creating the HTML data entry page and processing the data, is to make more use of the Document Object Model (DOM) that underpins the application. In chapter 6, in the section **Building web pages from code** we made some use of this when we created the **Times Table Generator** application. The times table application generates HTML output elements using a `for` loop and adds the elements to the document for display to the user. If you are not sure how this works, take a look at the Code Analysis section **Building HTML from JavaScript** in chapter 6.

```
<!DOCTYPE html>
<html lang="en">
<head>
```

```
<title>Ice Cream Sales</title>135  
<link rel="stylesheet" href="styles.css">136  
<script src="icecreamsales.js"></script>  
</head>  
  
<body onload="doBuildSalesInputItems('salesItems',0, 10000, 6);">137  
  <p class="menuHeading"> &#127846; Ice Cream Sales</p>138  
  
  <div id="salesItems">139  
  </div>  
  
  <p class="menuText" id="outputParagraph">140  
  </p>  
  
  <p>  
    <button class="menuText" onclick="doCalc('salesItems','outputParagraph')">Calculate</button>  
  </p>  
  
</body>  
  
</html>
```

This is the HTML page for a version of the ice cream sales program that automatically generates the input paragraphs. The paragraphs will be generated by the method `doBuildSalesInputItems` which is connected to the `onload` event. We first saw the `onload` event in Chapter 3 in the section **Create a ticking clock** when we used `onload` to start the clock ticking when the clock page was loaded.

For the ice cream sales application, the `onload` event will trigger the creation of the input items when the page is loaded. The input items will be added as children of the `div` element with the id `salesItems`. A `div` element is an HTML element that is used to group things together. We first saw `div` in chapter 3 in the section **Formatting parts of a document using div and span**.

---

<sup>135</sup> Title for the page

<sup>136</sup> Stylesheet for the application

<sup>137</sup> Function called to build the page

<sup>138</sup> Heading for the page

<sup>139</sup> Container for the sales items

<sup>140</sup> Output paragraph

```
doBuildSalesInputItems ('salesItems', 0, 10000, 6);
```

The `doBuildSalesInputItems` function is called with four arguments:

11. The ID string of the element that will contain the input items to be created. The input items will be added to the children of that element.
12. The minimum value that can be input to this element. This value is used to create the input element. In the case of our application the smallest number of sales is 0.
13. The maximum value that can be input to this element. In the case of our application the customer has said that sales of more than 10000 are impossible.
14. The number of input paragraphs that are to be created. The statement above will create 6 input elements.

Now that we've seen how the `doBuildSalesInputItems` function is called; let's look at the function code itself.

```
function doBuildSalesInputItems(containerElementID, min, max, noOfItems)  
{  
    var containerElement = document.getElementById(containerElementID);141  
  
    for(let itemCount=1; itemCount<=noOfItems; itemCount=itemCount+1){142  
        let labelText="Sales "+itemCount;143  
        let itemPar=makeInputPar(labelText, min, max);144  
        containerElement.appendChild(itemPar);145  
    }  
}
```

The function uses a for loop to generate each input paragraph in turn by calling the function `makeInputPar`. The end point of the loop is determined by the parameter `noOfItems`. The values of the `min` and `max` parameters are passed into the function `makeInputPar`. Let's have a look at how this works.

---

<sup>141</sup> Find the container element

<sup>142</sup> Loop round for each item to be added

<sup>143</sup> Build the label for the item

<sup>144</sup> Get the paragraph to be added

<sup>145</sup> Add the paragraph to the container

## Create an input paragraph

The program must build the JavaScript code that for the HTML elements into which the user will enter the data. I've created this as a function so that we can use it anywhere we want to read a value from a user.

Figure 8.3 above shows the display that the input paragraph should produce. This reads in the sales for ice cream stand number 1. The HTML that describes this input paragraph is shown below:

```
<p>
  <label class="menuLabel" for="s1SalesText">Stand 1 sales (0-10000):</label>
  <input class="menuInput" type="number" id="s1SalesText" value="0" min="0" max="10000">
</p>
```

The input paragraph is comprised of a label element that contains the label "Stand 1 sales (0-10000)" and an input element to receive the data. The label and the input elements are assigned style classes to make it easy to manage their appearance. The input element has `min` and `max` attributes that are used by the browser to limit the values that the user can enter.

In our first version of this application the input paragraphs were defined in the HTML file for the application. There were six such paragraphs, one for each of the ice cream stands. However, we can simplify the application by using JavaScript to make the input paragraphs.

```
function makeInputPar(labelText, min, max) {
  var inputPar = document.createElement("p");146
  var labelElement = document.createElement("label");147
  labelElement.innerText = labelText + " (" + min + "-" + max + ")";
  labelElement.className = "menuLabel";149
  labelElement.setAttribute("for", labelText);150
  inputPar.appendChild(labelElement);151   let inputElement = document.createElement("input");152
```

<sup>146</sup> Create the enclosing paragraph

<sup>147</sup> Create the input label

<sup>148</sup> Add the max and min to the label

<sup>149</sup> Set the style class for the label

<sup>150</sup> Connect the label to the input element

<sup>151</sup> Add the input to the children of the enclosing paragraph

<sup>152</sup> Create the input element

```
inputElement.setAttribute("max", max);153
inputElement.setAttribute("min", min);154
inputElement.setAttribute("value", 0);155
inputElement.setAttribute("type", "number");156
inputElement.className = "menuInput";157
inputElement.setAttribute("id", labelText);158
inputPar.appendChild(inputElement);159

return inputPar;
}
```

The function `makeInputPar` uses the `createElement` function provided by the `document` object to create a paragraph element (`inputPar`) and give the paragraph two child elements. The children are a label element (`labelElement`) and an input element (`inputElement`). The `makeInputPar` function is supplied with three parameters:

15. The label to be used for the input
16. The maximum input value
17. The minimum input value

A program could make the input element for the first ice cream stand by making the following call:

```
var standInput = makeInputPar("Sales 1", 0, 10000);
```

However, we are not going to do this because we are using a loop to create the inputs and add them to a container.

## CODE ANALYSIS

---

<sup>153</sup> Set the maximum value

<sup>154</sup> Set the minimum value

<sup>155</sup> Set the initial value to 0

<sup>156</sup> Set the input type to number

<sup>157</sup> Set the style class for the input

<sup>158</sup> Set the ID to connect to the label

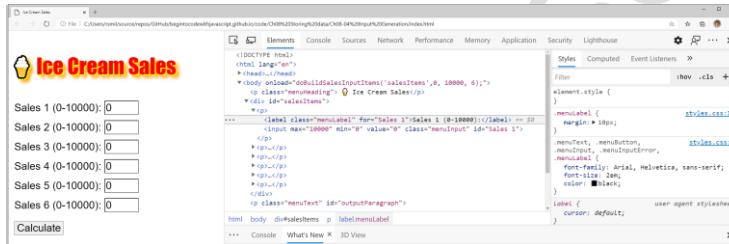
<sup>159</sup> Add the input to the enclosing paragraph

## Input Generation

We can see this input generation in action by using the JavaScript console. We might also have some questions about it. Start by opening the application in the **Ch08 Storing data\Ch08-04 Input Generation** folder in the example code. This application contains an HTML page which has been generated by the `doBuildSalesInputItems` function. Open the Developer View by pressing F12.

Question: Can we look at the HTML that was generated by the `doBuildSalesInputItems` function?

Yes we can. If we click the **Elements** tab in the Developer View it will display a view of the document object. We can then open up the view to see the HTML for each element.



Question: Can we add another input element to the document from the console?

Yes, we can. Click the **Console** tab to open the console. First we need a reference to the container element that holds the input paragraphs. Type in the following and press Enter:

```
> var containerElement = document.getElementById('salesItems');
```

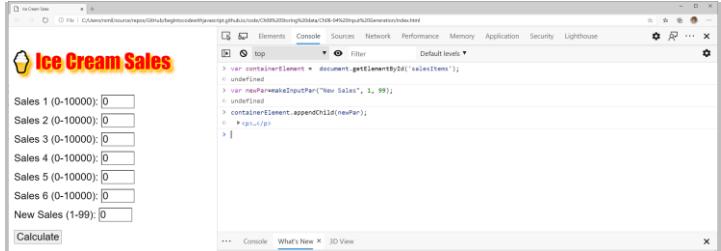
Next, we need to create a new input item. The function `makeInputPar` will do this for us. We will look at how this function works in the next section. Call the function to make a new input item called "New Sales" with a min of 1 and a max of 99:

```
> var newPar=makeInputPar("New Sales", 1, 99);
```

Now that we have our new paragraph, the next thing we do is add it to the children of the container element:

```
> containerElement.appendChild(newPar);
```

This adds our newly created paragraph to the page. You should see it appear. It will have the label "New Sales" and an input range of 1 to 99.



Above you can see the new element (with the caption "New Sales") that has been added at the bottom of the list.

Question: How would I increase the number of ice cream stands?

Using a loop to generate the makes it very easy to change the number of ice cream stands. If you take a look at the HTML code you will find the call of the function `doBuildSalesInputItems`. The final argument of this call is the number of stands to be created, which is presently 6. If we want to handle 12 stands we just have to change the HTML to reflect this. We don't need to make any changes to the JavaScript program.

```
doBuildSalesInputItems('salesItems', 0, 10000, 0);
```

Now we need to move on to how we can make a JavaScript program that can process the data that has been entered. To do that we need to write some code to create a `sales` array from the contents of the input elements in the document.

## Read back the values

In the previous version of the ice cream program we had a separate statement that create an array element for each sales value. Below is the statement that got the sales for the initial two ice cream stands.

```
sales[0] = getNumberFromElement("s0SalesText");
sales[1] = getNumberFromElement("s1SalesText");
```

If the customer added more stands we would then have to modify the program and add more statements. For 12 ice cream stands we would need 12 of these statements. There is a better way to do this. We can use a loop to read the values from the HMTL elements. The loop will work through all the input paragraphs on the page. The input paragraphs are children of the sales item element

```

▼<div id="salesItems">
  ▼<p>
    <label class="menuLabel" for="Sales 1">Sales 1 (0-10000):</label>
    <input max="10000" min="0" value="0" class="menuInput" id="Sales 1">
  </p>
  ▷<p>...</p>
  ▷<p>...</p>
  ▷<p>...</p>
  ▷<p>...</p>
  ▷<p>...</p>
</div>

```

**Figure 8.29** Input paragraphs

Figure 8.4 shows how these elements are structured. On the left we have the `div` element with the ID `salesElements` that contains all the elements. The `div` element contains 6 paragraphs, one for each ice cream stand. The first paragraph gets the input for ice cream stand 1. It contains a `label` and an `input` item. To get the user input the program must work through the sales items and extract the data from the `input` element.

The program can use a `for - of` loop to work through the children and extract the value of each item. The code above shows how this works.

```

var sales = []160;
var salesPos = 0161;

var salesElement = document.getElementById('salesItems')162;

for (const item of salesElement.children) {163

  let salesValue = getNumberFromElement(item.children[0])164

  sales[salesPos] = salesValue;165

```

<sup>160</sup> Create an empty array

<sup>161</sup> Start at the beginning of the array

<sup>162</sup> Get a reference to the element containing the sales items

<sup>163</sup> Work through the children of this element

<sup>164</sup> Get the number from the input element

<sup>165</sup> Store the number in the array

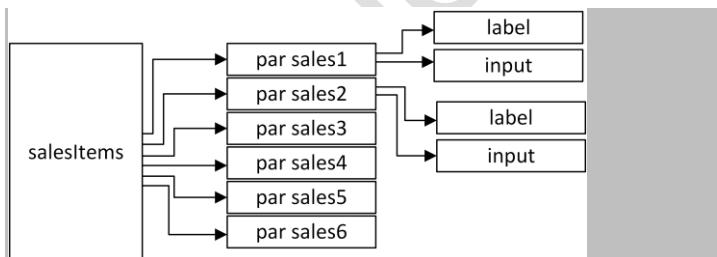
```
    salesPos = salesPos + 1;166  
}
```

## CODE ANALYSIS

### Reading numbers

Question: What is this code doing?

Good question. When the user presses the Calculate button they have just typed in the sales values and want to see the total, highest and lowest values displayed. The first thing the program needs to do is get the values out the input elements on the page and into the sales array. The input elements are children of the paragraphs that are themselves children of the `salesItems` element. Think of the `salesItem` element as a grandfather element. The grandfather has six children. These are the paragraphs that were generated when the page was built. Each of the six children has two more children which are the label and the input elements.



The `for - of` loop will work through each of the child elements of the `salesItems` container. Each time round the loop the variable `item` refers to the next paragraph element in the `salesItems`. Each paragraph element contains two children, the label for the input and the input itself. The input element that we need is the second child of the sales paragraph, which will have index number 1 (because indexes always start at 0).

```
let salesValue = getNumberFromElement(item.children[1]);
```

This version of the `getNumberFromElement` function is supplied with a reference to the element it is reading. So it fetches the value `input` and returns the result as a number.

Question: What happens if the input element contains an invalid value?

The input element contains `max` and `min` attributes that are used by `getNumberFromElement` to validate the

<sup>166</sup> Move on to the next element in the array

number that the user types in. If the number the user has entered is outside this range, or the user has not entered a number, the `getNumberFromElement` function returns a result of Not a Number (NaN).

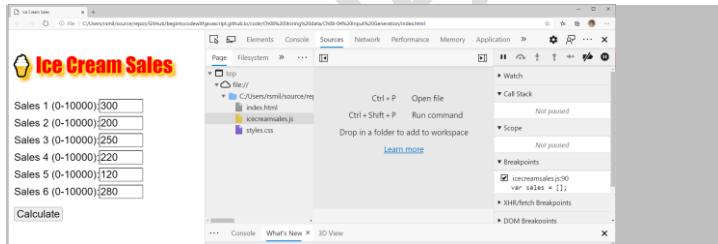
Question: Would I have to change this code if we added more ice cream stands?

No. That's the great thing about this loop. It will work through any number of input paragraphs.

## MAKE SOMETHING HAPPEN

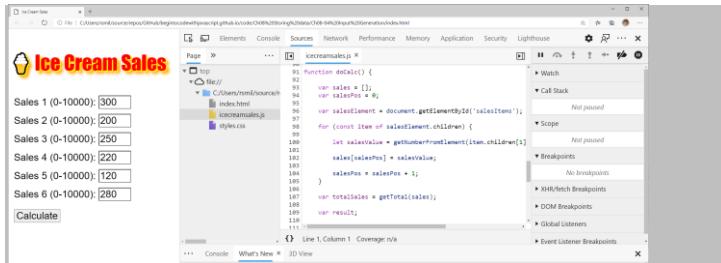
### Use the debugger to see code run

You might be finding the code above a little hard to follow. You might be thinking that it would be wonderful to be able to watch it execute and see what each statement does. It turns out that you can do just that by using the *debugger* which is built into the Developer View in the browser. In this case we are not debugging the program because there's something wrong with it, we just want to see how it works. Start by opening the application in the Ch08 Storing data\Ch08-04 Input Generation folder in the example code. This application contains an HTML page which has been generated by the `doBuildSalesInputs` function. Enter some sales figures for the ice cream stands and then open the Developer View by pressing F12. Select the **Sources** tab in the developer view.



The **Sources** tab will show you all the files that make up the application. Note that the browser display that you see might not match the one above. I've changed the arrangement of the windows in the browser and zoomed in on the developer view to make it clearer for the book. However, you can drag the window borders around to get a similar view.

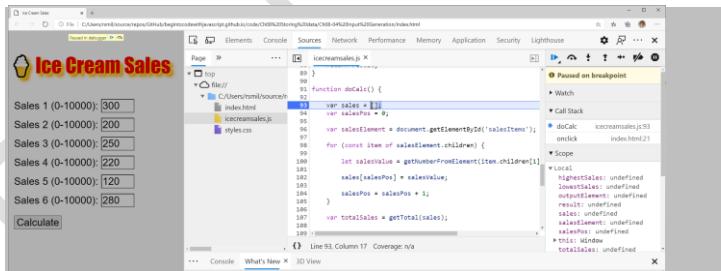
We want to take a look at the JavaScript code which is in the file `icecreamsales.js`. Click this filename (it is highlighted in darker gray in the figure above).



The Sources view now shows us the JavaScript code in a window. Use the scroll bars to move down the code to find the `doCalc` function. This function is called from the HTML when the Calculate button is pressed. We are going to put a *breakpoint* at the first statement of this function. When the program reaches the breakpoint, it will "takes a break". It will pause and we will be able to investigate what the function is doing and even run through the statements one at a time. We set a breakpoint by clicking just to the left of the line number in the listing.



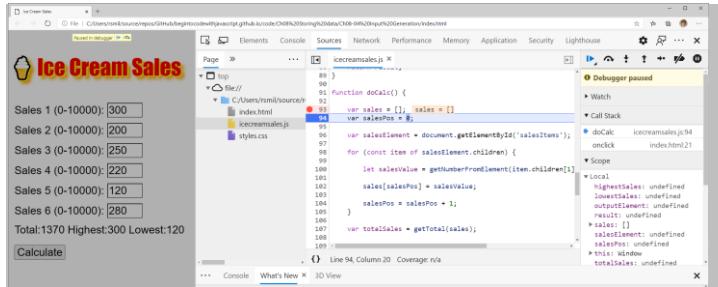
The display will show a red dot at that position as shown above. Click to add a breakpoint. If you click in the wrong place you can remove a breakpoint by clicking on the red dot. Once you have set the breakpoint; click the Calculate button in the application to run the function. The browser will call the `doCalc` function and then pause when it hits the breakpoint.



The program is paused at the statement where we set the breakpoint. You can see it highlighted in blue. We can use the transport controls to make the program run one statement at a time.



The transport controls look a bit like Egyptian Hieroglyphics. They are in the top right-hand area of the screen in my browser, but they might be somewhere else on yours. We will find out more about each control as we go. For now we want to use the **Step** control which is indicated above. Click this and the program will execute the indicated statement and then move on to the next one.



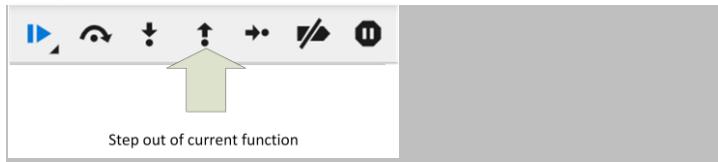
Note that the content of the `sales` array is shown for you. If you press the Step button again you are shown the contents of the `salePos` variable. You can keep pressing the button (or use the function key F9) to step through the program. You can watch the program get a reference to the `sales` element and then enter the `for` loop which will work through the elements. When the program calls the `getTotal` function you will find that the view moves into that function and you step through the statements in that.

```
62 function getNumberFromElement(element) { element = input#Sales 1.menuInput
63
64     var text = element.value; text = "300", element = input#Sales 1.menuInput
65
66     var result = Number(text); result = 300, text = "300"
67
68     if (isNaN(result)) {
69         // fail with bad number input
70         element.className = "menuInputError";
71         return NaN;
72     }

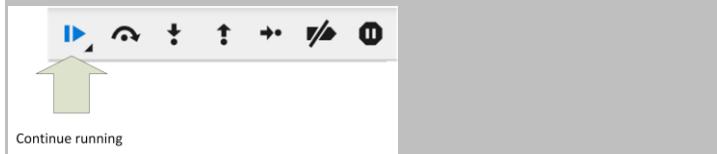
```

The `getNumberFromElement` function reads the text from an input element and converts it into a number. Here you can see that the text is "300" as a string and the result is 300 as a value. If you look back at the application you will find that the first sale value is 300.

You can use another transport button to step out of the `getNumberFromElement` function and return to where it was called.



When you have seen enough you can press the transport control that resumes normal program execution. If you leave the breakpoint set you will find that if you press the Calculate button in the application it will stop at the breakpoint next time around.



You can spend as much time as you like working through code. You can also use this technique to put breakpoints in any of the other sample programs and watch what they do too.

Now that you know how the program works you can make the change that adds two new ice cream stands to the system. The good news is that you only have to change the value 6 to the value 8 somewhere in the application. You can find my version in the examples in the folder **Ch08 Storing data\Ch08-05 Eight Stand Version**.

## Arrays as lookup tables

The application has a title bar "Ice Cream Sales". Below it is a list of locations with their corresponding sales values in a table:

Location	Sales
Riverside Walk (0-10000):	0
City Plaza (0-10000):	0
Central Park (0-10000):	0
Zoo Entrance (0-10000):	0
Main Library (0-10000):	0
North Station (0-10000):	0
New Theatre (0-10000):	0
Movie House (0-10000):	0

At the bottom left is a "Calculate" button.

Figure 8.30 Named Stands

Your customer has had another idea. Rather than the application asking for the sales of Stand 1, she would like the application to show named locations as shown in Figure 8.5 above. This looks like it might be quite hard to achieve, but it turns out that we can use arrays to help. In this case we are going to use an array to hold a list of place names. When we first created an array we used the following statement:

```
var newArray=[];
```

The characters "[" and "]" mark the start and end of the array that is being created. This statement creates an empty array, because there is nothing between the "[" and the "]". We set values values into an array when we create it by putting values here:

```
var newArray=[1,2];
```

This statement creates an array called `newArray` that contains two numbers. The element at the start of the array contains the value 1 and the next element contains the value 2. This statement is exactly equivalent to these three:

```
var newArray=[];
newArray[0]=1;
newArray[1]=2;
```

We can also create arrays of strings. Consider the following statement:

```
var standNames = ['Riverside Walk','City Plaza','Central Park','Zoo Entrance',
'Main Library','North Station','New Theatre','Movie House'];
```

This statement creates a variable called `standNames` that contains the names of all the stands. The item on the right of the assignment is the actual array. We can also use arrays like this in function calls:

```
doBuildSalesInputItems('salesItems',0, 10000,
['Riverside Walk', 'City Plaza', 'Central Park', 'Zoo Entrance',
'Main Library','North Station','New Theatre','Movie House']);
```

This statement calls the function `doBuildSalesItems`. The final parameter to this call is an array that contains the names of all the ice cream stand locations. This version of the function can then use these names to build the HTML page:

```
function doBuildSalesInputItems(containerElementID, min, max, placeNames) {
    var containerElement = document.getElementById(containerElementID);167

    for (placeName of placeNames) {168
        let itemPar = makeInputPar(placeName, min, max);169
        containerElement.appendChild(itemPar);
    }
}
```

Rather than creating the name "Sales x", where x is the number of the stand, this version of the function just works through the place names and feeds each name in turn into the `makeInputPar` function. The function `doBuildSalesInputItems` is called from the HTML when the page is loaded by the browser. The function call is triggered by the `onload` event:

```
<body onload="var standNames = doBuildSalesInputItems('salesItems', 0, 10000,
    ['Riverside Walk', 'City Plaza', 'Central Park','Zoo Entrance',
    'Main Library','North Station','New Theatre','Movie House']);">
```

The text above comes from the HTML file for the application. The `body` element can contain an attribute called `onload` which contains a string of JavaScript to be obeyed when the page is loaded by the browser. For this application the string of JavaScript calls the `doBuildInputSalesItems` to create the input paragraphs for the sales values. You can find this function used in the example program in the folder **Ch08 Storing data\Ch08-06 Named Stands**.

## CODE ANALYSIS

### Creating named Ice Cream Stands

This is all rather clever stuff. But you might have some questions:

Question: What is passed into a function when an array is used as an argument to a function?

The parameter in the function is a reference to the array. This means that if the function changes the contents of

---

<sup>167</sup> Get the container that will hold the items

<sup>168</sup> Work through the place names

<sup>169</sup> Create an input par

<sup>170</sup> Add the input par to the children in the container

the array (by assigning something to a value in the array or by adding a new element on the end of the array) this will change the array because there is only ever one copy of the array itself. Consider this function:

```
function changeArrayElement(inputArray)
{
    inputArray[0]=99;
}
```

The function `changeArrayElement` has a single parameter which is an array. It changes the value in the start of the array to 99.

```
var testArray = [0];
```

This statement creates an array called `testArray` which contains a single element which is set to the value 0.

```
changeArrayElement(testArray);
```

This statement calls the `changeArray` method and gives it `testArray` as an argument. The question that we need to consider is: "What is the value in element 0 of `testArray` after the call of `changeArrayElement`?". The answer is that it contains 99 because the function is passed a reference to the array. When the function runs the reference is used to find the array and change the element in it. If you think about it, passing arrays by reference makes very good sense. Otherwise a program would have to make a copy of an array to pass into a function call.

Question: How would I add another ice cream stand location to the application?

This application design means that to add another stand to the application we just have to add the name of the new stand to the array value that is passed into the call of `doBuildInputSalesItems`. The JavaScript code does not have to be changed at all.

Question: Can an array hold numbers and strings in different elements?

Yes it can. You can regard each element of an array as a totally separate variable that can hold any value. It can be confusing if you use arrays in this way, in the next chapter we will discover how to design variables that collections of items of different types.

## Creating fixed width layouts

If you run the program in the folder **Ch08 Storing data\Ch08-06 Named Stands** you will notice that it works perfectly well, but the layout of the input elements is not very consistent:

## Ice Cream Sales

Riverside Walk (0-10000):

City Plaza (0-10000):

Central Park (0-10000):

Zoo Entrance (0-10000):

Main Library (0-10000):

North Station (0-10000):

New Theatre (0-10000):

Movie House (0-10000):

**Figure 8.31** Poor layout

The input elements are not lined up vertically because the names of the different locations are of different lengths. This is not a programming problem, so we don't need to make changes to the JavaScript, but it would be useful to be able to modify the style of the labels for each item so that they are the same size.

```
.menuLabel {  
    display:inline-block;171  
    width:12em;172  
    margin: 10px;  
}
```

The CSS text above is the definition of the `menuLabel` style class which is used to style the menu label (hence the name). I've added two new attributes to the style definition. The first tells the browser that this element is a block that should be displayed inline with enclosing elements. The second item sets the width of the element to 12 characters. If we make these changes to the stylesheet the input display matches what the customer requested in Figure 8.6. You can find this version of the application in the example folder `Ch08 Storing data\Ch08-07 Named Stands fixed width`.

---

<sup>171</sup> Display this element as a block inline with the text

<sup>172</sup> Set the width of the block to 12 characters

## MAKE SOMETHING HAPPEN

### Highlight the best and worst sales locations

Your customer (who by now has given you a lot of ice cream) has one final request. She would like the program to highlight the best and worst sales locations in the display. She would like the highest sales to be highlighted yellow and the lowest to be highlighted in blue. Hint: to do this the program must make a second pass through the sales elements after it has determined the highest and lowest values. Any sales elements with a sales value that matches the highest can be assigned a yellow style class. Any sales elements with a sales value that matches the lowest can be assigned a blue style class. This is the best way to do it because it is possible to have several sales values that are the highest or the lowest. You can find my version in the examples in the folder Ch08 Storing data\Ch08-08 Highlight High and Low

## Interactive Times Table Tester

### Times Table Tester

1 times 13 is:

2 times 13 is:

3 times 13 is:

4 times 13 is:

5 times 13 is:

6 times 13 is:

Which times table do you want :

How many lines do you want :

We can use our new skills to improve the times table tester application that we created in Chapter 6. The version

above generates a times table for any value and then checks what the user enters. You can find the working application in the example folder **Ch08 Storing data\Ch08-08 Times Table Tester**.

It has been suggested that the program could be improved if it displayed the scores at the end:

You got 2 correct out of 6

To make this work you have to add a new paragraph to the HTML to display the result and then work through the program to find out where the table is scored. Have a go yourself and then take a look at my version in the example folder **Ch08 Storing data\Ch08-09 Times Table Tester with scores**.

## What you have learned

In this chapter, you discovered how a program can use arrays to store large amounts of data. You've also built on your knowledge of how a JavaScript program can generate document elements and display them.

- A variable can be declared as an array and acts as a container for multiple values. Each value in an array is stored in an element and particular element can be identified by an index value. An index is sometimes called a *subscript*.
- Elements in an array with a particular index value are created when a value is assigned to that element. There is no need to specify the size of an array when it is created.
- An array value exposes a length property that gives the number of elements in the array.
- Element index values start at 0 and extend up to the value of (length-1). This range of index values is called the *bounds* of the array. There is no element with an index value of the length of the array.
- If a program accesses uses an index value outside the bounds of the array a value of undefined is returned.
- A program can use the for – of loop construction to work through the elements in an array. A program can also work through elements of an array by creating a variable that counts through the index values in the array.
- An array with a number of elements can be created using a single statement.
- Inputs in an HTML document can be assigned label elements that display a prompt for that input.
- The Developer Console in the browser can be used to step through JavaScript programs and view the contents of variables held in them.

Here are some questions that you might like to ponder about the use of functions in programs:

**Question:** How do I find out the size of an array?

**Answer:** An array variable provides a property called `length` which contains the number of elements in the array.

**Question:** Does every element in an array have to be the same kind of data?

**Answer:** No. An array can contain numbers, strings and references to other objects.

**Question:** Do I have to put a value into every element?

**Answer:** No. For example, you could create an element with index 0 and another element with index 5. All the elements in the middle (i.e. those with index values 1,2,3 and 4) would be set to `undefined`.

**Question:** Can I store a table of data as an array?

**Answer:** A JavaScript array has only one dimension – the length of the array. You can think of it as a row. Some programming languages allow you to create “two dimensional arrays”. You can think of these as a grid with width and height. JavaScript does not allow you to create two dimensional arrays. However, you could create a table data structure by creating an array that has an array as each element.

**Question:** What happens to my program if I used an index value outside the range of the array?

**Answer:** If the program is storing data in this location the array will be extended to hold the value at the new index. If the program is reading data from this location the array access will return the value `undefined`.

**Question:** Can I use indexes on other data items?

**Answer:** Yes. You can use an index to obtain the individual characters in a string.

**Question:** What happens if I don't initialize the elements in a new array?

**Answer:** Any elements that you don't initialize are set to the value `undefined`.

**Question:** What happens if I add two arrays together?

**Answer:** It would be nice if adding two arrays together created a long array with one set of elements appended to the other. Unfortunately, this does not happen. Instead Java JavaScript creates the string version of each array and then appends one string to the other.

**Question:** Can I use an image as a label for an input item?

**Answer:** Yes you can. The content of a label can be an image, or some text and an image.

**Question:** What is the difference between the children of an HTML element and a JavaScript array?

**Answer:** They are implemented in slightly different ways, but you can use an index to access values in each and they both expose a `length` property. You can also use the `for` – `of` loop construction to work through the elements in either.

**Question:** Can I use an array to allow a function to return more than one value?

**Answer:** What a good question. Yes. A program can return an array which can contain multiple values. However,

the function and the caller would have to agree on what was in each of the array elements. In the next chapter we will discover a much neater way of creating objects that contain named data items.

**Question:** Can I use the JavaScript debugger to debug the JavaScript in any web page?

**Answer:** Yes. If you press F12 when viewing your favorite web page you can then open up the JavaScript files and take a look at them.

Pre-release

# 9

# Objects

## What you will learn

Programs can work with many different types of data, including integers, floating-point numbers, and strings of text. They can also create arrays of a particular data type. However, the data that programs need to work with is often more complex than single values. In this chapter, you'll learn how to use objects to store related data items. You discover how software can create custom objects and have your first go at solving a real computing problem.

## Make a tiny contacts app

Suppose one of your friends is a lawyer who wants someone to create a personal, confidential contacts app. The client wants a tiny “lightweight” application to provide a quick way of storing contact details—names, addresses, and telephone numbers—for her important clients. You sit down with her and decide how the application will work:

## Tiny Contacts

Name:

Address:

Phone:

Figure 9.32 Tiny Contacts Prototype

Figure 9.1 shows how the application will look. It has three text boxes for data entry and two buttons. When the **Save Contact** button is pressed the data entered into the text boxes is stored. To find a contact the user enters the name and then presses the **Find Contact** button. The program then searches the contact store for a contact with that name. If the contact is found the program displays the address and phone number information. If the contact is not found the program displays an alert as shown in Figure 9.2 below.



This page says  
Not found

Name:

Address:

Phone:

Figure 9.33 Contact not found

## Prototype html

The best way to show the lawyer what her program will look like is to create a prototype that behaves in the same way as the finished product. We can do this by creating an HTML page and then adding just enough JavaScript to allow us to demonstrate how the program will be used.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Tiny Contacts</title>
  <link rel="stylesheet" href="styles.css">173
  <script src="tinycontacts.js"></script>174
</head>

<body>
  <p class="menuHeading"> &#128199; Tiny Contacts</p>

  <p>
    <label class="InputLabel" for="name">Name:</label>
    <input class="inputText" id="name">
  </p>

  <p>
    <label class="InputLabel" for="address">Address:</label>
    <textarea class="inputTextarea" rows="5" cols="40" id="address"></textarea>
  </p>

  <p>
    <label class="InputLabel" for="phone">Phone:</label>
    <input class="inputText" id="phone">
  </p>

  <p>
    <button class="menuButton" onclick="doFind()>Find Contact</button>175
    <button class="menuButton" onclick="doSave()>Save Contact</button>176
  </p>
</body>
```

---

<sup>173</sup> The stylesheet for the application

<sup>174</sup> The JavaScript program for the application

<sup>175</sup> Call doFind to find a contact

<sup>176</sup> Call doSave to save a contact

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

This is the HTML for the prototype application. It defines the three input areas and the two buttons. There is one new element in this page; the `textarea` element.

## The `textarea` element

If you look at Figure 9.1 you will see that the address of a contact is can be entered as multiple lines of text. We can't use an `input` element to read multi-line text because an `input` element only supports a single line. A `textarea` is configured by two attributes called `rows` and `cols`. These define the size of the data entry area on the screen. The Tiny Contacts application uses an area which has 5 rows and 40 columns.

```
<textarea class="inputTextarea" rows="5" cols="40" id="address"></textarea>
```

The `textarea` will display a scroll bar if the user enters more than 5 lines of text and a `textarea` will automatically wrap text if the user types off the end of the line. A JavaScript program can use the `value` property of a `textarea` to interact with the content of the text area.

## Prototype stylesheet

The HTML file contains the elements that make up the application display. The CSS (Cascading Style Sheet) file controls how the display will look. Your customer would like a clean black and white display and so you propose this stylesheet:

```
.menuHeading {  
    font-size: .4em;  
    font-family: Impact, Haettenschweiler, 'Arial Narrow Bold', sans-serif;  
    color: black;  
    text-shadow: 3px 3px blue, 10px 10px 10px grey;177  
}  
  
.InputLabel, .inputText, .inputTextarea, .menuButton
```

**Commented [RM1]:** Not sure if I need to mention that a `textarea` does not have a `value` attribute in HTML. I think at the moment that it is a bit of a digression that the reader doesn't need to know.

<sup>177</sup> Style class for the application heading

```
{178
  font-family:Arial, Helvetica, sans-serif;
  font-size: 2em;
  color:black;
}

.inputLabel
{179
  display:inline-block;180
  vertical-align: top;181
  width:6em;182
}
```

The css file defines a style for the heading. It then contains settings shared by all the other text styles in the document, followed by settings which are only applied to the labels. This makes it easy to change the font, size or color of all of the text. It also makes it possible for a designer to change the style of any element on the page without changing the HTML file at all.

## Prototype JavaScript

The third component of the tiny contacts application is the JavaScript that provides the behaviors. The prototype application doesn't do much. It just displays contact information if the user searches for "Rob Miles" and displays "Contact not found" if the user searches for any other name. The **Save Contact** button just displays an alert when it is pressed.

```
function displayElementValue(id, text){183
  var element = document.getElementById(id);
  element.value = text;
}
```

---

<sup>178</sup> These settings are applied to all the text style classes

<sup>179</sup> These settings are only applied to the label style classes

<sup>180</sup> Make the label into an inline block

<sup>181</sup> Put the text at the top of the block

<sup>182</sup> Fix the width of the label

<sup>183</sup> Display the text in the element with the specified id

```

function getElementValue(id){184
    var element = document.getElementById(id);
    return element.value;
}

function doSave() {185
    alert("Saves a contact in the store");
}

function doFind() {186
    var name = getElementValue("name");187
    if(name=="Rob Miles")188
    {
        displayElementValue ("addressText", "18 Pussycat Mews\nLondon\nNE1 4IOS");
        displayElementValue ("phoneText", "+44(1234) 56789");
    }
    else {189
        displayContactNotFound ();
    }
}

```

You can find the prototype application in the sample folder **Ch09 Objects\Ch09-01 Tiny Contacts Prototype**. You can use it to enter and search for contact information. The application will only display output if you search for a contact named Rob Miles, and it doesn't store any entered data. However, for demonstrating how the application will work it's very useful. Your client agrees that the application can work like this and you can start building it.

## Helper functions

If you look closely at the prototype JavaScript code you'll notice a pair of tiny functions. I've written these to help move data between the HTML page and the application:

---

<sup>184</sup> Get the data value from the element with the specified id

<sup>185</sup> Save the contact details

<sup>186</sup> Find a contact

<sup>187</sup> Get the name value

<sup>188</sup> Is the name “Rob Miles”

<sup>189</sup> Clear the details if the name is not “Rob Miles”

```
function displayElementValue(id, text){  
    var element = document.getElementById(id);190  
    element.value = text;191  
}  
  
function getElementValue(id){  
    var element = document.getElementById(id);192  
    return element.value;193  
}
```

The function `displayElementValue` accepts two arguments: the id of the element and the text to be displayed. It is used to display values on the HTML page. The call of the function below would display the phone number text on the page. Note that although the phone number is described as a number the data itself is a string of text.

```
displayElementValue("phoneText", "+44(1234) 56789"
```

The function `getElementValue` accepts a single argument: the id of the input element to be read. The call of the function below would get the name that had been entered onto the HTML page and store it in a variable called `name`.

```
var name = getElementValue("name");
```

You might wonder why I wrote such small functions. They don't make the program much smaller. If I wrote the application without these functions it would only add a few statements. However, I think they do make the program much clearer because the functions express exactly what is being done without the distraction of how they work. As an example, consider the function that is called when contact information is not found. At the time the function is called we can tell from the name of the function exactly the program is doing.

```
function displayContactNotFound()  
{
```

---

<sup>190</sup> Get the element being changed

<sup>191</sup> Set the value in the element

<sup>192</sup> Get the element being read

<sup>193</sup> Return the value in the element

```
    alert("Not found");
}
```

Another advantage of using a function like this is that it makes it easy to change what the program does when a contact is not found. If the customer says that she would like a sound to be produced when the item is not found we can just add this to the `displayContactNotFound` function without having to work through the entire program to find the code that deals with finding contacts.

## Storing contact details

Now that the customer has agreed how the program should work and we have got the HTML page and style sheet files, we can start to create the code. However, before we can write the code to save a contact, we need to decide how the contact information will be stored. There will be lots of contacts. When we wrote the ice cream sales application we used an array to hold the sales information. One way to store the contact details would be to use three arrays, one for the name, one for the address and third for the phone number:

```
var contactNames = [];
var contactAddresses = [];
var contactPhones = [];
```

The data for a contact would be held in array elements with a particular index value. In other words `contactNames[0]` would hold the name of the contact at the start of the storage, `contactAddresses[0]` would hold the address and `contactPhones[0]` the phone number.

Array index values

	0	1	2	3
name	Rob Miles	Fred Bloggs	Joe Smith	Imogen Bloggs
address	18 Pussycat Mews	1 Forthe Road	House of Joe	Immy Villas
phone	(1234) 567 567	(1234) 556 767	(1234) 367 547	(1234) 723 523

Data arrays

**Figure 9.34** Array storage

Figure 9.3 shows how this works. The elements at index 0 in each array hold the details for Rob Miles. We can use the elements at a given index to build up a set of contact information for a specific person. For example, Imogen Bloggs lives at Immy Villas and has a phone number of (01234) 723523. We can write a function that stores a contact at a particular index position in all the arrays:

```
function storeContact(pos){  
    contactNames[pos] = getElementValue("name");194  
    contactAddresses[pos] = getElementValue("address");195  
    contactPhones[pos] = getElementValue("phone");196  
}
```

The function `storeContact` accepts an argument that gives the position in the array. This is where the contact will be stored. In other words, if the value of the argument is 0 the contact will be stored at the start of the storage.

Each statement in the function fetches data from an element on the HTML page and stores it at the given index in the array that holds the data for that type of data. Remember that a JavaScript program automatically creates an array element at the specified index when the program stores a value in that element. The first contact will be stored at the element with index 0 (remember that arrays are indexed from 0), the next at index 1 and so on. The statement below saves contact information on the HTML page into the elements at the start of the array.

```
storeContact(0);
```

## Finding contacts

Now that we know how to store contact information the next thing we can work on is finding it. The customer wants to be able to search for contacts by name. We need something that works through the stored contacts and finds position of the one we want to display. In other words, asked to find the entry for “Joe Smith” in the contact store shown in Figure 9.3 it would return the value 2.

---

<sup>194</sup> Store the name

<sup>195</sup> Store the address

<sup>196</sup> Store the phone number

```
function findContactPos(name) {  
  
    for(let pos=0; pos<contactNames.length; pos+=1){197  
        if(contactNames[pos]==name) {198  
            return pos;199  
        }  
    }  
    return NaN;200  
}
```

The `findContactPos` returns the position in the array of the contact with the name supplied as a parameter. It uses a for loop to work through the `contactNames` array comparing each element in the array with the name it is searching for.

## CODE ANALYSIS

### The `findContacts` function

The `findContacts` function is a very important part of the application. You may have some questions about it.

Question: What does the for loop do?

The function must look at each element in the `contactNames` array to see if it matches the name the function is searching for. If I asked you to search through a row of pigeonholes to find the one with my name on you'd start at the first pigeonhole, check the name of that and then move on to the next one if there was no match. The for loop controls the contents of a variable called `pos` (an abbreviation of position). The value of `pos` starts at 0 and the loop ends when the value in `pos` reaches the length of the array.

Question: What would happen if the `contactNames` array is empty when the `findContacts` function is called?

The loop only runs as long as the value of `pos` is less than the length of the array. An empty array has a length of 0. The initial value of `pos` is set to 0 when the loop is set up. This means that the loop is never performed if the array is empty because 0 is not less than 0.

<sup>197</sup> Count through the array elements

<sup>198</sup> See if the stored name matches the parameter

<sup>199</sup> If the name matches return the index value

<sup>200</sup> Return Not a Number (NaN) if the name was not found

Question: What happens when a match is found for the name being searched?

When the program finds a matching name the value of the index variable is returned to the caller. This is the position in the array of the name.

```
var pos = findContactPos("Fred Bloggs");
```

As an example, using the data store in Figure 9.3 the above statement would create a variable called `pos` which would be set to 1 by the call of `findContactPos`.

Question: What does the `findContactPos` function do if the name is not found in the `contactNames` array?

If the name is not found in the array the for loop will complete without finding a match. The program would then move on to the statement after the for loop which returns the value Not a Number (NaN) to indicate that the name was not found. It is perfectly sensible for a function to contain more than one return statement. Note that it is up to the program calling `findContactPos` to check to see if the name was found.

This type of search behavior is common in systems that you use every day. Whenever you use a credit card to pay for something the computer in the bank will use the number of the credit card to search for your bank account information so that the transaction can be authorized and added to your statement.

## Displaying contacts

The final function that we need is one that displays the contact on the HTML page. This function is given a position in contacts store and displays the elements in each array at that position. It uses the `displayElementValue` function that we saw earlier to display each item:

```
function displayContact(pos){  
    displayElementValue("name", contactNames[pos]);  
    displayElementValue("address", contactAddresses[pos]);  
    displayElementValue("phone", contactPhones[pos]);  
}
```

## Saving a contact

The final pieces of code that we need are the functions that respond to the button presses in the application. When the user clicks the Save Contact button the function `doSave` is called by the browser. This function must work in two different ways:

- It must create a new contact store for a new contact
- It must update the contact information for an existing contact

The `doSave` function gets the contact name from the web page and then uses the `findContactPos` function to see if the contact store already contains someone with that name. The `findContactPos` returns a value which is stored in a variable called `pos`. If the value of `pos` that is returned is not a number (NaN) the function sets the value of `pos` to the length of the array, so that the contact information will be stored at the end of the array. Otherwise the newly saved data will overwrite the existing data. I've added some comments to this code to make it clear what each section is doing.

```
function doSave() {  
  
    // get the name of the contact being saved  
    var name = getElementValue("name");  
  
    // find the position of the name to save  
    var pos = findContactPos(name);  
  
    if(isNaN(pos)){  
        // if we didn't find an existing contact name  
        // we store the contact on the end of the array  
        pos = contactNames.length;  
    }  
  
    storeContact(pos);  
}
```

## CODE ANALYSIS

### The `doSave` function

The `doSave` function implements the editing behavior of the application. You may have some questions about it.

Question: Why does the `findContactPos` function sometimes return NaN?

We know that the JavaScript language provides a variable value called "not a number" or NaN. This is used by JavaScript itself. If we ask the JavaScript `Number` function to convert the string "Fred" into a numeric value it will return NaN. In the Tiny Contacts application I'm using the value NaN in the function `findContactPos` to indicate that a position could not be found. If I asked you to search through a row of pigeonholes to find the one with my name on, and you couldn't find it, you'd say "Sorry Rob, can't find it". The value NaN is how `findContactPos` indicates that it couldn't find what it had been asked to look for. I could have used a different flag value if I wanted, perhaps -1 (which would not be meaningful as an index into an array). This form of notification relies on the user of a function checking the reply of the function. Later in the book we will look at ways that a program can explicitly stop running by raising an exception when things go wrong.

Question: What does the function use the length of the `contactNames` array for?

If the `findContactPos` function indicates that it can't find a contact with that name (which it does by returning `NaN`) this means that the store does not contain a contact with this name. The contact information must be stored in a new location. When the program is first run the length of the array is 0, so the first contact will be stored at element 0, the start of the array. The next contact will be stored at element 1 (because the length of the array is now 1). And so on.

Question: What happens if the user changes the name of a contact and then saves it?

This is a very good question. It points to a potential bug in the application. We can answer the question by working through the code with an example. If the user opens a contact called "Rob", changes the name to "Rob Miles" and then presses the Save Contact button, what will the `doSave` function do? The function will fail to find a contact with the name "Rob Miles" and then create a new contact with that name. This could be dangerous because there would now be two contact entries for the same contact. If the user searches for the old name by mistake she might see data which is out of date.

This kind of issue crops up frequently when software is being created. As you write the program code you discover questions that you didn't consider when you discussed the design of the application. The only way to resolve this issue is to ask the customer what she wants to happen. She says that she will never need to change the name of a contact, so she is OK for the program to be left as it is.

## Finding a contact

The `doFind` function runs when the user clicks the **Find Contact** button. It can use the `findContactPos` function to search for a contact with a name that matches the one entered. If a contact is found the contact contents are displayed. If a contact is not found the function must display a message:

```
function doFind() {  
  
    var name = getElementValue("name");  
  
    var pos = findContactPos(name);  
  
    if(isNaN(pos)){  
        displayContactNotFound();  
    }  
    else{  
        displayContact(pos);  
    }  
}
```

You can find the complete application in the folder **Ch09 Objects\Ch09-02 Array Tiny Contacts**. If you try it, you will discover that it works fine. You can enter contact details, save them and search for them.

## Use an object to store contact details

As you've seen, we can create a perfectly workable Tiny Contacts application by using a list for each piece of information we want to store for our contact. However, working with data stored in this way is not as easy as we might like. If we ever add a new data item for a contact (perhaps we want to add their email address), we would need to add a new list and then make sure that items in it were managed correctly.

We want a way of holding all the information about a contact in one place. We need some kind of "container" could hold the name, address, phone number, and any other items we want to store. One possible solution might be to use an array to store information about each customer, but this wouldn't make it very easy to access specific detail items. Instead, we'll use a JavaScript object.

You'll hear a lot about objects over the next few chapters, as they are one of the fundamental building blocks that underpin the language. You might have heard the term "object-oriented programming." Now you are going to find out what an object is and how to use it in a JavaScript program.

### MAKE SOMETHING HAPPEN

#### Create an object

We can start our description of objects by making some using the JavaScript console in the browser. Open the example application at **Ch09 Objects\Ch09-03 Object Tiny Contacts** in the examples folder browser and press F12 to open the Developer View. Select the Console tab in the view.

In JavaScript we can create an empty object by using a pair of open and close braces. Type in the statement below and press Enter. Do not use a pair of brackets (`[` and `]`) – they are for creating arrays.

```
> var contact = {}
```

The process of creating an object does not return a value, so when you press Enter you will see a familiar undefined message:

```
> var contact = {}
<- undefined
```

As usual with the JavaScript console, you can view the contents of a variable simply by typing the name of the variable. Type in `contact` and press Enter:

```
> contact
```

JavaScript shows you that the `contact` object does not contain any properties.

```
> contact  
<- {}
```

Let's add a name property to the `contact` object. Type in the statement below, which sets the `name` property of the `contact` object to the string "Rob Miles". Press Enter.

```
> contact.name = "Rob Miles"
```

This is an assignment operation (that's what the `=` means) and an assignment expression always returns the value being assigned so JavaScript replies with the string "Rob Miles".

```
> contact.name = "Rob Miles"  
<- "Rob Miles"
```

Let's take a look at what has changed in the object. Type in `contact` and press Enter to view the contents of the `contact` object:

```
> contact
```

The `contact` object contains one property which is the name of the contact.

```
> contact  
<- {name: "Rob Miles"}
```

We can add a second property to the contact by setting an address property in exactly the same way as we added the name. Type in the statement below and press Enter.

```
> contact.address = "18 Pussycat Mews"
```

The address is added to the contact and the console shows us the value assigned as before.

```
> contact.address = "18 Pussycat Mews"  
<- "18 Pussycat Mews"
```

Type in the statement below to view the contact and press enter to see what has been added to the contact.

```
> contact
```

The contact object now contains a name and address. We could keep adding properties to match the requirements of our application.

```
> contact  
<- {name: "Rob Miles", address = "18 Pussycat Mews"}
```

Objects are managed by *reference*. In other words, the `contact` variable we have created refers to a location in computer memory where the `contact` object is stored. We can create a new variable and set it to refer to the same location as the `contact` variable. Type the statement below, which creates a variable called `refDemo` which refers to the same object as `contact`. Press enter when you've typed in the statement.

```
> var refDemo = contact
```

Creating a new variable always returns a result of undefined, which the console will now show us.

```
> var refDemo = contact  
<- undefined
```

Now let's take a look at the object that `refDemo` refers to. Type in `refDemo` and press enter to get the console to display the object that `refDemo` refers to.

```
> refDemo
```

Because `refDemo` refers to the same object as `contact` you will see the same properties displayed.

```
> refDemo  
<- {name: "Rob Miles", address = "18 Pussycat Mews"}
```

The `refDemo` variable refers to the same object as `contact`. We can prove this. Enter the following statement and press Enter:

```
> refDemo.name = "Rob Bloggs"
```

This statement changes the name property of the object referred to by `refDemo`. As with other assignments it displays the value being assigned.

```
> refDemo.name = "Rob Bloggs"  
<- "Rob Bloggs"
```

Now view the contents of the `contact` variable by typing its name and pressing Enter:

```
> contact
```

Both `contact` and `refDemo` refer to the same object, so changes made via `refDemo` will also change the object that `contact` refers to.

```
> contact  
<- {name: "Rob Bloggs", address = "18 Pussycat Mews"}
```

The name property is now "Rob Bloggs". This change was not made via the `contact` reference but via the `refDemo` reference.

## Use an object in the Tiny Contacts program

We can use an object to simplify the Tiny Contacts program. The program now only needs a single array to hold all the contact information.

```
var contactStore = [];
```

Each of the objects in the `contactStore` array will have a set of properties that are the stored data for that contact. To make this work we will have to modify some of the functions in our program. Rather than storing the different parts of contact data in individual arrays the `storeContact` function now adds the elements as properties which are added to a contact object:

```
function storeContact(pos){  
    var contact = {};201  
    contact.name = getElementValue("name");202  
    contact.address = getElementValue("address");203  
    contact.phone = getElementValue("phone");204  
    contactStore[pos]=contact;205  
}
```

The `displayContact` function now displays the properties in a contact object rather than elements from the three arrays that store contact data.

```
function displayContact(pos){  
    var contact = contactStore[pos];  
    displayElementValue("name", contact.name);  
    displayElementValue("address", contact.address);  
    displayElementValue("phone", contact.phone);  
}
```

The example program in the folder **Ch09 Objects\Ch09-03 Object Tiny Contacts** contains a version of the tiny contacts program that uses an object rather than separate arrays. If you take a careful look at the JavaScript code you will discover that some of the functions are completely unchanged.

---

<sup>201</sup> Create an empty object

<sup>202</sup> Add the name

<sup>203</sup> Add the address

<sup>204</sup> Add the phone number

<sup>205</sup> Store the contact in the array

## Store data in JavaScript local storage

The Tiny Contacts program works perfectly well except for one thing. There is no way of persisting the contacts data. Each time the user opens the page it starts with an empty set of contacts. For the Tiny Contacts program to be properly useful it needs a way of storing data. Some computer languages can save and load data in the file storage on the system they are running on. A JavaScript program running inside a browser is not usually allowed to interact with files stored on the host computer. JavaScript programs are often part of pages downloaded from anywhere on the internet and it would be very dangerous to give these programs access to files on your computer. However, JavaScript does provide *local storage*. This is managed by the browser and our JavaScript applications can use it to store reasonable amounts of data.

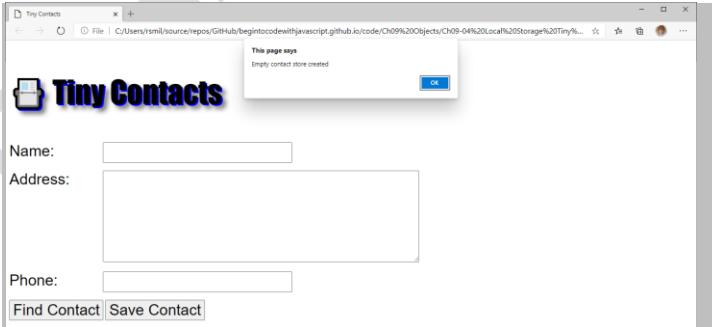
You can think of local storage as a data object that is persisted by the browser on the host computer. This means that every machine has its own copy of local storage. Fortunately, our customer only has one computer, a laptop that she uses for everything, so this is not a problem. Later in this book we'll discover how we can use network storage to store data.

Each browser program has its own implementation of local storage which means that items stored when using the Edge browser would not be accessible when using the Chrome browser. The browser maintains an object called `localStorage` which has methods that are used to save items in local storage and get them back.

### MAKE SOMETHING HAPPEN

### Investigate local storage

The example program in the folder **Ch09 Objects\Ch09-04 Local Storage Tiny Contacts** uses local storage to store contact information. Let's find out how it works. Open the application in your browser.



The first time the contacts page is opened there will be no contacts store. The page displays an alert to indicate this. Click OK in the alert to clear it and press the F12 key to open the Developer View. Now move to the console

tab so that you can enter some JavaScript. Programs use local storage by placing strings of text in named locations. The method to store an item in local storage is called `setItem`. Type the statement below and press Enter.

```
> localStorage.setItem("test", "This is some test data we are storing")
```

When you press Enter the method `setItem` will run and store the string "This is some test data we are storing" at the location "test". The `setItem` method returns the value undefined.

```
> localStorage.setItem("test", "This is some test data we are storing")
<- undefined
```

We can use the method `getItem` to view the contents of an item stored in local storage. Type in the statement below.

```
> localStorage.getItem("test")
```

When you press Enter the `getItem` method will run and return the contents of that location.

```
> localStorage.getItem("test")
<- "This is some test data we are storing"
```

The content of location test will still be there if we return to this web page after having shut down the browser. It will even be present after we have rebooted our computer. We can see how the data storage for our Tiny Contacts application works. Type some contact details into the application and press the Save Contact button. The contact information is held in a location called "TinyContactsStore". Type in the following statement to read the contents of this location.

```
> localStorage.getItem("TinyContactsStore")
```

When you press Enter you will see the contents of the contacts store now contains the data that was entered.

The screenshot shows a web browser window titled 'Tiny Contacts'. The application interface has fields for 'Name' (Rob Miles), 'Address' (18 Pussycat Mews, London, NE14 10S), and 'Phone' (+44(1234) 56789). Below these are 'Find Contact' and 'Save Contact' buttons. To the right of the browser is the browser's developer tools, specifically the 'Console' tab. The console output shows the following JavaScript code and its execution:

```
> localStorage.setItem("test", "This is some test data we are storing")
<- undefined
> localStorage.getItem("test")
<- "This is some test data we are storing"
> localStorage.getItem("TinyContactsStore")
<- "[{"name":"Rob Miles","address":"18 Pussycat Mews\\nLondon\\nNE14 10S","phone":"+44(1234) 56789"}]"
```

If you store some more contacts you will find that they are added to the stored item. If you restart your PC and

then load the same application you will find that the contact information has been retained.

The functions `setItem` and `getItem` can be used to store and retrieve data in local storage. If you want to remove the contents of a local storage the function `removeItem` can be used to remove an item. The statement below would remove the item `TinyContactsStore` from the browser storage.

```
localStorage.removeItem("TinyContactsStore")
```

## Use JSON to encode object data

Now it is time to discover how JavaScript programs can store data as formatted text. This is one of the most powerful features of the language and it underpins many JavaScript applications that store and transfer data around the internet. The technology is called JSON or, to give the full name *JavaScript Object Notation*. In the previous “Make Something Happen” we saw that the contacts store for our application is retained as a string of text. This text is encoded using JSON.

```
[{"name": "Rob Miles", "address": "18 Pussycat Mews\nLondon\nNE14 10S", "phone": "+44(1234) 56789"}]
```

The JSON string above represents a contact store that contains a single contact. If you look at it carefully you can see some strings which are the names of properties (for example “phone”) and some strings which are values (for example “+44(1234) 56789”). The string above actually creates an array (which is why the outer characters are square brackets ([ and ])) and the array contains a single object (which is the element enclosed in braces ( { and } ) inside the array).

### MAKE SOMETHING HAPPEN

#### Working with JSON

You should still have your browser open at the example program in the folder **Ch09 Objects\Ch09-04 Local Storage Tiny Contacts**. If you haven’t, open up the application, press F12 to open Developer View and select the Console tab. We can start by making an object and adding some properties to it. Type the following statement and press Enter:

```
> var contact = {name: "Rob", address: "Rob's House", phone: "1234"}
```

This is a quick way of creating JavaScript objects that we have not seen before. Rather than creating an empty object we have populated a new object with three properties. Each property is given as the name of the property,

followed by a colon (: ) and the value for that property. Successive properties are separated by the comma ( , ) character. We now have a variable called `contact` which contains my details. If we just type the name of the variable the console will show us the contents:

```
> contact  
<- {name: "Rob", address: "Rob's House", phone: "1234"}
```

The console is showing us the JavaScript representation of the `contact` object. We can convert the object into a JSON string by using the `stringify` method provided by the `JSON` object. Type the following statement and press Enter:

```
> var jsonContact = JSON.stringify(contact)  
<- undefined
```

The `stringify` method returns a string of text which has been displayed by the console. The variable `jsonContact` now contains a JSON string that represents the contact. Type in `jsonContact` and press Enter to view it:

```
> jsonContact  
<- "{"name":"Rob", "address":"Robs House", "phone":"1234"}"
```

The JSON string version of an object looks very like the JavaScript representation, with the difference that the names of properties are also enclosed in double quotes (""). Another method, called `parse`, can be used to convert this string back into an object:

```
> var decodedContact = JSON.parse(jsonContact)  
<- undefined
```

We now have a new variable called `decodedContact` which contains the contact information that was stored in the string `jsonContact`. We can view this by just entering the name of the variable:

```
> decodedContact  
<- {"name":"Rob", "address":"Robs House", "phone":"1234"}"
```

So, in this make something happen we have "round tripped" a contact object. We have converted it into a string of text by using the wonderfully named `stringify` method and we have converted the string back into an object by using the `parse` method.

We can combine local storage and JSON to make two methods that will save and load the contacts in the Tiny Contacts application.

```
var contactStore = []206
```

---

<sup>206</sup> Array that holds the contacts

```
var storeName;207

function saveContactStore(){208

    var storeJson = JSON.stringify(contactStore);209

    localStorage.setItem(storeName, storeJson);210
}

function loadContactStore(){211

    var dataString = localStorage.getItem(storeName);212

    if(dataString==null){
        contactStore = [];
        return false;
    }

    contactStore = JSON.parse(dataString);

    return true;
}
```

The `loadContactStore` function is called when the page is first loaded. The `saveContactStore` function is called each time a contact is saved. The name of the local storage item to be used to store the contact data is held in a variable called `storeName`. The value in this variable is set when the application is started.

```
function doStartTinyContacts(storeNameToUse){
    storeName = storeNameToUse;

    if(!loadContactStore()){
        alert("Empty contact store created");
    }
}
```

---

<sup>207</sup> Constant variable that holds the name in local storage

<sup>208</sup> Saves the contacts

<sup>209</sup> Convert the contacts to a string

<sup>210</sup> Store the string in local storage

<sup>211</sup> Loads the contacts

<sup>212</sup> Get the string from local storage

```
}
```

The function `doStartTinyContacts` is called to start the contacts store and load the contact values. If the load fails the function displays an alert to indicate that the program has made an empty contact store. The HTML page for the application calls the function `doStartTinyContacts` when the page is loaded.

```
<body onLoad="doStartTinyContacts("TinyContactsStore");">
```

The string "TinyContactsStore" is given as an argument to the call of `doStartTinyContacts` from within the web page. This makes it easy for us to change the location of the stored data. You can find this code in the example application in the samples in the folder **Ch09 Objects\Ch09-04 Local Storage Tiny Contacts**.

## CODE ANALYSIS

### Using JSON

JSON makes it much easier to move data between JavaScript applications and also to store data in the form of strings. But you might have some questions about it:

Question: Where is the JSON standard defined?

One of the great things about JSON is that it is human readable. You can see how property names and values go together. If you want to see the standard that defines what a JSON document can contain you can find it at the web page <https://www.json.org/>

Question: What types of data can a JSON document store?

A JSON document can hold strings of text (which are enclosed in quote characters), numeric values and the logical values true and false.

Question: How long can a JSON string be?

Strings in JavaScript can extend to many thousands of characters, so you can use JSON to store large objects if required.

Question: What happens if I store a reference in the JSON string?

The reference is followed and the contents of the object inserted. We've seen this in action already. The `contactStore` in the Tiny Contacts application is an array of references to objects. When it is converted into objects

each reference is followed and the object contents are inserted into the string.

Question: What happens if a program tries to store an invalid value like Not a Number in a JSON string?

We've seen that JavaScript variables can hold values such as Not an Number (NaN) or infinity. These values are stored in a JSON string as the special value `null` which is used in JSON to indicate that a value is missing.

Question: What happens if the program tries to parse a string that does not contain valid JSON text?

The `JSON.parse` method expects to be given a string of JSON to work on. If the `parse` method can't decode the string it does something we haven't seen before. It throws an *exception* and ends the execution of the program. Exceptions are a way of forcing a program to respond to an error. We will discuss them in the next chapter.

## Use property accessors

This section has perhaps the most confusing heading in the whole book. JavaScript programs can use *property accessors* to work with the properties in an object. Before we talk about property accessors, let's refresh our understanding of what a property is.

```
var contact = {};
contact.phone = "+44(1234) 56789";
```

A JavaScript program can create an empty object and then add properties to it. The two statements above create a contact object and add a `phone` property to the object which is the string "+44(1234) 56789". We have seen that programs can add properties to an object which acts as a container for the property values that have been added to it. We can identify these properties in the program by name as you can see above, where we create a property called `phone`. This method is called *dot notation* because the name of the property is given after a dot (.) in the program text.

A property accessor looks a lot like an array indexer. It is called *bracket notation* because the property identifier is enclosed in square brackets. It works a bit like an array too, except that the item used as an index can be any value. This makes a JavaScript object behave like a dictionary. One of my favorite jokes is to tell someone that the word "gullible" is not in the dictionary and then watch as they look up the word and proudly read it out to me. A dictionary lets you look up definitions starting from a given word. If you look up the word "gullible" you will get a result along the lines of "easily persuaded to believe something". In JavaScript, if you look up a property accessor you get the value of the property with the specified name:

```
var contact = {};213  
contact["phone"] = "+44(1234) 56789";214
```

The statements above show this works. The first statement creates an empty object called `contact`. The second statement adds a property “phone” to the empty object by using a property accessor. Note that the property accessor is the string “phone”. This is a very powerful feature of JSON but it can also be a confusing one, particularly when you try to compare JavaScript objects with JavaScript arrays. Let’s if we can’t make things clearer.

## MAKE SOMETHING HAPPEN

### Investigating property accessors

Open the application in the example folder **Ch09 Objects\Ch09-05 Universal Tiny Contacts**. It is a contact manager that works in the same way as previous ones. Press F12 to open Developer View and select the Console tab. First, we want to investigate property accessors. Let’s start by making an empty `contact`. Type the following statement and press Enter:

```
> var contact = {}
```

Creating a new variable does not return a value, so the console displays the familiar message “undefined”.

```
> var contact = {}  
<- undefined
```

We now have an object called `contact` which has no properties. We can view the contents of `contact` by entering its name and pressing Enter :

```
> contact
```

JavaScript will tell us that the `contact` object is empty by printing a pair of curly brackets with nothing between them.

```
> contact  
<- {}
```

Now, let’s add a name property to the `contact` object. Enter the following and press Enter:

```
> contact.name = "Rob"
```

<sup>213</sup> Create an empty object

<sup>214</sup> Add a phone property to the object using a property accessor

This statement returns the value being assigned, so the console prints the value "Rob".

```
> contact.name = "Rob"  
<- "Rob"
```

The `contact` object now contains a property called `name` which contains the value "Rob". Now let's add another property, but this time we will refer to the property using a *property accessor*. Type in the statement below and press Enter.

```
> contact["address"] = "House of Rob"
```

This statement adds a property called `address` which is set to the value "House of Rob". It also displays the value being assigned, as usual.

```
> contact["address"] = "House of Rob"  
<- "House of Rob"
```

If we look at the contents of the `contact` object we will see both these properties. Type in the statement below and press Enter to take another look at the contents of the `contact` object:

```
> contact
```

The `contact` object now contains `name` and `address` properties.

```
> contact  
<- {name: "Rob", address: "House of Rob"}
```

At this point you might start to think that a property accessor makes every object look like an array. Let's see what happens if we try to use an object in the same way we would an array. The following statement tries to store the string "Hello world" at element 0 of an array. Let's try to do that with the `contact` object. Type in the statement below and press Enter.

```
> contact[0] = "Hello world"
```

JavaScript does not produce an error message, so the statement must have done something. The value "Hello world" has now been added as a property of the `contact` object.

```
> contact[0] = "Hello world"  
<- "Hello world"
```

Let's take a look at the contents of the `contact` object. Type in the object name and press Enter:

```
> contact
```

This is very confusing. The `contact` object now has a property called "0" which contains the string "Hello world".

```
> contact  
<- 0: "Hello world", name: "Rob", address: "House of Rob"}
```

The next thing to do is to try and use "0" as a property identifier in our program. The statement below is trying to access a property called "0". Type it in and press Enter to see what it does:

```
> contact.0
```

JavaScript will not let you use a number as an object property identifier so this will generate an error:

```
> contact.0  
VM142:1 Uncaught SyntaxError: Unexpected number
```

Property identifiers in a program must obey the same rules that are applied to all JavaScript identifiers. However, we can still access this property by using 0 as a property accessor. Type this statement and press Enter to prove this:

```
> contact[0]
```

You have correctly identified a property and so JavaScript shows you the value it holds:

```
> contact[0]  
<- "Hello world"
```

Please remember that I am in no way recommending that you should use anything other than strings of text as property accessors. As with lots of things in life, just because you can do something does not mean that you should try to do it.

We have discovered that we can use strings of text as property accessors rather than putting property names in the program code. This is interesting, but at the moment we are not sure why it is so useful. It turns out that property accessors give us a lot of power. Our programs can be made to create their own object designs. Consider these two statements:

```
var propertyName = "phone";  
contact[propertyName] = "+44(1234) 56789";
```

The first statement creates a variable called `propertyName` that contains the string "phone". The second statement uses the value of `propertyName` as property accessor. The effect of these statements would be to change the `phone` property of the `contact` variable. If I want to create an object with a particular set of properties, I can now do this from within the program, rather than having to create the code by hand. Why is this useful? Perhaps we want to make other kinds of data storage applications. We can consider why this is in the next section.

## Use a data schema

Your Tiny Contact program is becoming quite popular. A friend has seen you working on it and would like you to

create a program to keep track of his video game high scores. Another friend wants a program to store her recipes. In fact there are lots of people who seem to be looking for simple data stores. You could create all these different programs by editing the TinyContacts application, modifying the HTML elements and then changing the JavaScript code to match. However, this looks like it might be hard work. Instead we can step back from the original problem using a software development technique called *abstraction* and see if we can't create a single application that will work in all these scenarios.

#### PROGRAMMER'S POINT

##### Objects are useful for abstraction

Abstraction is a way of “stepping back” from the problem you are trying to solve and trying to think about the “bigger picture” surrounding the code that you are writing. It is something that experienced programmers do a lot.

If you asked an experienced programmer to solve the Tiny Contacts problem they would not think about storing names, addresses and phone numbers. They would think about storing “stuff”. Each item of stuff would have a particular name (like name, address, phone number etc) and some characteristics (for example the address stuff would be edited over several lines of text but the name stuff would be edited as one line of text). They would then design a solution which allows named items with different characteristics to be stored.

This would make it easy to modify their solution. If the Tiny Contacts customer wanted to add an email address to the contacts store this would be easy to do, and if someone wanted an application to store a different type of data, this would be easy to do as well. The programmer would just have to design a different kind of container for each application.

Objects are a great tool for abstraction because you can start with the idea of an object as a generalized thing which can hold “stuff” and then add detail by setting out the properties that you want for the particular stuff that you are storing. For example, a contact will have a name, address and phone number properties whereas a recipe will have properties that describe the name, ingredients and how to cook the food.

Learning when to step back, and how far to step is something that comes with experience. It's best not to get too abstract too early. However, it is also important to think about “the bigger picture” when you start writing programs.

JavaScript property accessors let us create data objects that describe the data items that are to be stored. These can then be used by the application to create objects that will hold the data. We've already done this kind of thing in the times table and Ice Cream sales applications, where an application creates the HTML page that will be used to enter data. Now we are extending this process to allow an application to create bespoke objects to hold data. If you look in the HTML page of the [Ch09 Objects\Ch09-05 Universal Tiny Contacts](#) application, you will find a call of a function `doBuildPage`.

## Build HTML from a schema

```
doBuildPage( "contactItems",215
  [ { id:"name", prompt:"Name", type:"input"},216
    { id:"address", prompt:"Address", type:"textarea", rows:"3", cols:"40"},217
    { id:"phone", prompt:"Phone", type:"input"} ] );218
```

The `doBuildPage` function takes two arguments. The first argument is the name of the display element on the HTML page which is to contain all the HTML that will be generated to provide the user interface. We've used this technique before in the Ice Cream sales application, where the program generates the input fields for all the ice cream stands.

The second argument is an array of objects, each of which describes one of the items to be stored. A description like this is called a *schema*. A schema is a lump of data that describes a data structure. The schema for the Tiny Contacts application defines three data items, the name, address and phone number items. All the descriptions contain at least three property values:

- The id to be used to identify the property in the object that is created to store a contact.
- The prompt gives the text that is to be displayed in the HTML when this property is being edited.
- The type gives the type of the item. There are two types of property. An `input` type uses the `input` HTML element to read a single line of text. The `textarea` type uses a text area. For a `textarea` the description also contains row and col values to specify the size of the text area to be used.

The `doBuildPage` function works through each of the items in the schema and creates an HTML element for each item. This is then added to the container element to build the display. This function is called when the page is loaded.

```
function doBuildPage(containerElementID, schema){  
  // store the schema for use later by the application  
  dataSchema = schema;
```

<sup>215</sup> Container object on the web page for the HTML

<sup>216</sup> Description of the name entry

<sup>217</sup> Description of the address entry

<sup>218</sup> Description of the phone entry

```
// get a reference to the element containing the edit items
var containerElement = document.getElementById(containerElementID);

// work through each of the items in the schema
for (item of dataSchema) {
    // make an element for that item
    let itemElement = makeElement(item);
    // add the element to the container
    containerElement.appendChild(itemElement);
}
}
```

The `doBuildPage` function makes uses a function called `makeElement` to make each HTML element for display on the page. This function uses the description from the schema to decide which what kind of element to make.

```
function makeElement(description) {
    // Create the enclosing paragraph
    var inputPar = document.createElement("p");

    // Create the label for the element
    var labelElement = document.createElement("label");
    labelElement.innerText = description.prompt + ":";
    labelElement.className = "InputLabel";
    labelElement.setAttribute("for", description.id);
    inputPar.appendChild(labelElement);

    // decide what kind of element to make
    switch (description.type) {
        case "input":
            inputElement = document.createElement("input");
            inputElement.className="inputText";
            break;

        case "textarea":
            inputElement = document.createElement("textarea");
            inputElement.className = "inputTextarea";
            inputElement.setAttribute("rows", description.rows);
            inputElement.setAttribute("cols", description.cols);
            break;
    }
    // add new kinds of element here
}
```

```

// set the id for the element
inputElement.setAttribute("id", description.id);
// give the element an initial value
inputElement.setAttribute("value", "");
// add the element to the paragraph
inputPar.appendChild(inputElement);
// return the whole paragraph
return inputPar;
}

```

The code in `makeElement` is very similar to the code in the function `makeInputPar` that we created in the section **Create an input paragraph** in chapter 8. It creates an HTML paragraph that contains a label and an input element. The function is supplied with an object that describes the kind of input that is required. The object contains the three properties of `id`, `type` and `prompt` that we saw earlier. These are used to build the HTML display.

## Build a data object from a schema

When the user presses the Save Contact button the `storeData` function is used to read data from the HTML elements on the page and create properties in a new data object that is then stored.

```

function storeData(pos){
    // Create an empty data item
    var newData = {};

    // Work through the data schema
    for(property of dataSchema){
        // Get the data out of the HTML element
        let itemData = getElementValue(property.id);
        // Create a property to store that data
        newData[property.id] = itemData;
    }
    // put the new data in the storage array
    dataStore[pos]=newData;

    // save the data store
    saveDataStore();
}

```

The schema array holds the application together. It describes each of the data items to be displayed by the editor

and it is used to build the HTML page and also to create each data object for storage. We can extend the schema design by adding new types of data input, perhaps a number input type.

## MAKE SOMETHING HAPPEN

### Create your own data storage

In this Make Something Happen I want you to investigate some working programs and try to produce your own schema for a data store. The example application in the folder **Ch09 Objects\Ch09-05 Universal Tiny Contacts** holds a tiny contacts data manager that uses the schema above. The example application in the folder **Ch09 Objects\Ch09-06 Recipe Store** holds a recipe storage program. Both programs use the same JavaScript and stylesheet files. The only file that is changed for each application is the HTML file that contains the schema for the application. Take a look at both these applications and use what you find out to create a third application that can store video game scores. The items to be stored are:

- The name of the video game
- The high score reached
- The time in minutes that the game lasted
- A details panel that can accept 5 lines of text which are 40 columns wide

You should be able to achieve this by only changing the contents of the HTML file. If you get stuck you can take a look at my version which you can find in **Ch09 Objects\Ch09-07 Game Scores**.

## Improving the user interface

We now have a working data store, but it is not very easy to use. When searching for an item the user must enter an exact match for the item name for it to be found. If they mistype one letter of the name, perhaps by typing “rob” rather than “Rob” the name will not be matched. These problems are caused by the simple nature of the function that finds items in the data store:

```
function findDataPos(name) {  
    for(let pos=0;pos<dataStore.length;pos=pos+1){  
        let storedName=dataStore[pos].name;  
        if(storedName==name) {219  
            return pos;  
        }  
    }  
}
```

---

<sup>219</sup> Very simple test for a name match

```
        }
        return NaN;
    }
```

We can make the find process much easier by making the test ignore whether the text was typed in UPPER CASE or lower case. This can be achieved by converting the search string and all the items it is searching for to lower case before performing the test:

```
function findDataPos(name) {
    name=name.toLowerCase();
    for(let pos=0;pos<dataStore.length;pos=pos+1){
        let storedName=dataStore[pos].name;
        storeName = storedName.toLowerCase();
        if(storeName==name) {
            return pos;
        }
    }
    return NaN;
}
```

The version of Tiny Contacts in the example folder **Ch09 Objects\Ch09-08 Tiny Contacts Improved Search** uses this version of the function.

## Add “Super Search” to Tiny Contacts

You show the improved search to your lawyer customer and she is very pleased. However, after a while she comes to see you, saying that she has had an awesome idea to make the application even better. She doesn't like having to type in the entire name to search for a contact. Instead she wants to type in just the first part of the name and then use the Find Contact button to step through all the contacts that start with that part. For example, she might type just the letter “R” and then repeated presses of the Find Contact button would find the entries for “Rob”, “Ronald” and “Rita”. She calls this her “Super Search” feature and she would really like you to add it to the Tiny Contacts program.

### PROGRAMMER'S POINT

#### User ideas are often the best kinds of ideas

Making a great application is only the first step in making a successful one. You also have to work hard to

engage with your users and make them into your sales force. We have seen that one way to do this is to always respond constructively to error reports. Another good technique is to engage with users and encourage them to suggest improvements. This is a win-win proposition. They get a solution tailored to their needs and you get another feature that you can use to sell your solution.

## Solve a problem

Making “Super Search” work is an interesting problem. I suppose you could call it our first “proper” programming problem. By that I mean that it is not obvious how we can solve it. With all the other applications we have created we have been automating a behavior that we know how to do. We knew how to calculate times tables and add up ice cream sales before we went anywhere near a computer. But for this problem we don’t know how to do what is being requested. So, the best possible answer we can give our lawyer friend is that we will think about it and get back to her later.

The first step towards solving a problem is to determine whether the problem can be solved. Trying to do the impossible never ends well. In this case I can tell you that I’ve used systems that work in a similar way to that suggested by the lawyer, so it is possible. So, with that useful knowledge, let’s work out how we can solve the problem. The first step to solving the problem is to write down the sequence of actions that need to be performed to use the action in the application:

18. Type part of the name into the name input. (for example “R”).
19. Press the **Find Contact** button. The first contact (i.e. the one nearest the start of the `contactStore` array) with a name that starts with R will be displayed.
20. If the user presses Find Contact again the next contact with a name that starts with R will be displayed.
21. If the user enters a new name, the Find behavior resets and the **Find Contact** button now searches for the new name.

I had a look at this and thought about it and I found three things that the application needs to be able to do to make these behaviors work:

22. The application needs to be able to test whether one string starts with the search string. This is so it can match the contact with the name “Rob” to the search string “R”.
23. The application needs to store the search string and the search position it has reached in the `contactStore` array. This is so when the user presses **Find Contact** the application knows what to search for and where to start the search from.
24. The application needs to know when the user types in a new name. This is so when the name changes the application can reload the search string and restart the search at the start of the data array.

## Find out if one string starts with another

The JavaScript string object provides a method called `startsWith` that returns true if the string starts with another string.

```
var name="Rob";
if(name.startsWith("R")){
    alert("Rob starts with R!")
}
```

The JavaScript code above shows how this works. So now we know that we can search the contacts for names that start with a string, we can move on to the next problem. The program needs to store the current search string and the position it has reached in the array of contacts.

## Retain the find string and find position

```
var findString = "";
var findPos = 0;
```

These two variables hold the string we are finding and the current find position. They are global variables that are declared outside any function so that they are visible to all the functions in the application. We start by setting the find string to an empty string and the find position to 0. This means that if the user presses the Find Contact button as soon as the application has been started the search will match all the contacts (all names start with an empty string) and search from the start of the contact store (remember that arrays are indexed from 0).

When the user presses the **Find Contact** button the **doFind** function is called to search. We can make this function start searching with the find string at the current find position:

```
function doFind() {
    var pos = findStartsWithDataPos(findString,findPos);220
    if(isNaN(pos)){221
        displayDataNotFound();222
    }
    else{
        displayData(pos);223
    }
}
```

<sup>220</sup> Search for a contact

<sup>221</sup> Have we found a match?

<sup>222</sup> Display a message if no contacts match the search

<sup>223</sup> Display the contact that was found

```
    findPos = pos+1;224  
}  
}
```

This `doFind` function uses a function called `findStartsWithDataPos` which is given the current `findString` (perhaps the string "R") and the current value of `findPos` (perhaps the position 0). If `findStartsWithDataPos` doesn't find a match it returns Not a Number (NaN) and the `displayDataNotFound` function is called to tell the user about this.

The `findStartsWithDataPos` function is provided with a name to search for and a start position. It searches for a contact which has a name that starts with the search string. It starts the search at the supplied start position.

```
function findStartsWithDataPos(name, startPos) {  
    name=name.toLowerCase();225  
    for(let pos=startPos;pos<dataStore.length;pos=pos+1){226  
        let storedName=dataStore[pos].name;227  
        var lowerCaseStoredName = storedName.toLowerCase();228  
        if(lowerCaseStoredName.startsWith(name)) {229  
            return pos;230  
        }  
    }  
    return NaN;231  
}
```

Our application can use these two functions to search through the datastore array for contacts that start with a particular string. Each time `doFind` is called it will use `findStartsWithDataPos` to find the next contact to display. The final piece of the solution is code that restarts the search each time the user changes the contents of the name input field.

---

<sup>224</sup> Store the find position that we have reached

<sup>225</sup> Convert the name to lower case

<sup>226</sup> Start the search at the search position

<sup>227</sup> Get the name out of the search contact

<sup>228</sup> Convert the name to lower case

<sup>229</sup> Check for a match

<sup>230</sup> Return the search result

<sup>231</sup> If the search fails return this

## Detect changes to the name input

This is perhaps the easiest part of the solution. We have seen that HTML elements can be made to call JavaScript functions when something happens in the document. We've used the `onload` attribute to specify a JavaScript function to be called when an HTML document is loaded into the browser. HTML provides an `oninput` function that will call a JavaScript function when the user inputs text into an input field. We can use this to specify a method to run when the user changes the name of a contact when using the application:

```
<input class="inputText" oninput="resetFind()" id="name" value="">
```

This is a very powerful feature. If you've ever used a web page which automatically updates or checks your text as you type it in, you now know how it works. The `resetFind` function sets up a new search:

```
function resetFind(){
    nameString=getElementValue("name");232
    findString=nameString.trim();233
    findPos=0;234
}
```

The `resetFind` function gets the name element from the HTML document and then uses a JavaScript string feature that we've not seen before. The `trim` function returns a version of the string which has had all its leading and trailing spaces removed. It would convert " R " to "R". This is very useful. The user will become confused if a search for "Rob" doesn't find "Rob" and this prevents that problem from happening. The value of `findString` is then set to this trimmed value and `findPos` is set to 0 to make the next search start at the beginning of the stored contacts.

## CODE ANALYSIS

### Investigating Super Search

The "super search" version of Tiny Contacts is in the examples folder in **Ch09 Objects\Ch09-09 Tiny Contacts Super Search**. Let's open it up and take a look inside. Open the application in your browser and press F12 to open

<sup>232</sup> Get the name from the HTML input element

<sup>233</sup> Remove leading and trailing spaces

<sup>234</sup> Set the find position to the start of the array

the Developer View of the application.

Question: Does it work?

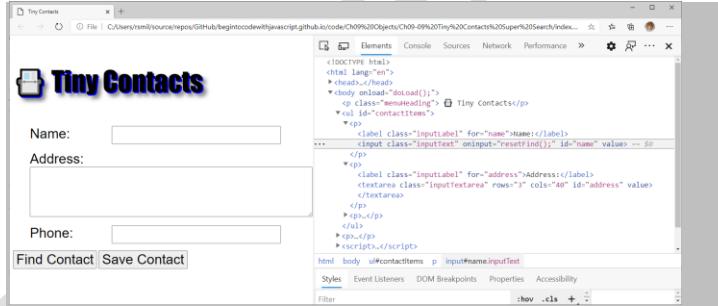
Good question. Try it. Enter some data and save it, and then enter the first letter of one of the names into the name field and press Find Contact. The application will display the first match and continue to display matches up to the end of the data.

Question: What happens if I enter an empty name to search for?

Try it. The behavior of the `statsWith` function means that all strings start with an empty string. So the `findStartsWithDataPos` function will match every contact. The user of the Tiny Contacts program will really like this, as it provides her with a way of stepping through all the saved contacts.

Question: How does the application add the `oninput` behaviour to the name input field.

Changes to the name input field must be detected and used to update the search string. If you use the Elements tab in the Developer View to take a look at the web page HTML you will see that the `oninput` attribute has been added to the name input field. However, it has not been added to the address underneath. How does this work?



I've made a tiny change to the function that generates the HTML for each data item that needs to be stored. The `makeElement` function regards the element with the id "name" as special. When it generates that element it adds the attribute to it. Open the Sources tab in the Developer View and take a look in the `tinyData.js` file.

```

    tinydata.js X
    Page >> ... tinydata.js
    top // add new kinds of element here
    file://
    C:/Users/rsmil/source/repos/GitHub/beginnertocodeswithjavascript.github.io/code/Ch09-09%20Objects/Ch09-09%20Tiny%20Contacts%20Super%20Search/index...
    Sources Network Performance Memory ...
    tinydata.js
    index.html
    styles.css
    121 // add new kinds of element here
    122 }
    123 // special handling for the name input which
    124 // must reset the find when it is changed
    125 if(description.id === "name") {
    126     ((inputElement) =>
    127         inputElement.setAttribute("oninput", "resetFind()");
    128     )
    129 }
    130 // set the id for the element
    131 inputElement.setAttribute("id", description.id);
    132 if(description.value) {
    133     inputElement.setAttribute("value", description.value);
    134     inputElement.setAttribute("value", "");
    135     // add the element to the paragraph
    136     inputElement.parentElement.appendChild(inputElement);
    137     // return the whole paragraph
    138     return inputElement;
    139 }
    140 ...
    141

```

3 characters selected Coverage: n/a Scope Watch

The statement that I have highlighted tests the id of the element being created and adds the `oninput` attribute if it is the name element.

Question: How hard would it be to add this feature to the Recipe and Video Game Score applications?

The wonderful thing about the way that our solution works is that such a change would require no programming effort at all. We simply have to update the versions of `tinydata.js` with the super search one.

## MAKE SOMETHING HAPPEN

### Improving data storage

At this point your lawyer friend is very keen to progress the application and is having lots of ideas about how it could be made better. She is even suggesting some kind of partnership where she takes a stake in the business and gets a share of any income that you get from selling the application. This sounds reasonable to you, it is always useful to have a good lawyer on your side when you are writing software. Here are a few of the ideas that she would like you to add.

- She would like to add an “email” input so that the program can keep track of the email address of each contact.
- She would like a button that clears the HTML form to make it easy to enter new contacts.
- She would like the search to “wrap round” so that when she searches off the end of the list she then returns to searching at the start.

See if you can make these features work. If you need ideas, you can take a look at my solution which you can find in the examples in the folder Ch09 Objects\Ch09-10 Super Tiny Contacts.

As a further exercise you might like to make a datastore application that can hold data that you fancy storing.

## What you have learned

In this chapter, you learned how to JavaScript objects allow programs to store related items as properties of a single object and how to make programs that work with objects.

- An object can contain any number of named property values. Each property value can be regarded as a variable holding a particular kind of value. An object can contain numbers, text and references to other objects.
- A program can create an empty object using a pair of braces ({ and }) with nothing between them.
- A populated object (one with properties) is created if the braces enclose a comma separated list of property values. Each property value is expressed as the name of the property followed by a colon and the property value, for example you could create the following implausible object: `{name:"Rob Miles",age:21}` which contains two properties called `name` and `age`.
- Objects are managed by reference. A variable can contain a reference to an object. A single object can have multiple reference variables referring to it. An object can contain a property which is a reference to another object. This allows a program to build data structures such as lists.
- An object reference can be given as an argument to a function call and passed into the function as a parameter of that function. This allows the function to interact with the properties in the object and add additional properties to the object.
- A program can store a large amount of structured data by using an array of object references.
- The browser maintains a *local storage* area for each different web site the user visits. Values placed in local storage are maintained when the browser is not running and the host machine is switched off. The values that are stored are strings of text. Each string is accessed by using a string that gives the item a name.
- JavaScript Object Notation is a standard for encoding JavaScript variables into strings of text. The JSON object provides a `stringify` method to convert a variable into a JSON string and `parse` method to convert JSON strings into variables.
- There are two ways that a program can access a property. Dot notation uses a dot to separate the object variable name and the property name (`contact.name`). Bracket notation uses a property accessor (usually a string) enclosed in square brackets (`contact["name"]`).
- A data schema is an object that contains a design for other objects. Schemas can be used in association with bracket notation property accessors (see above) to allow a program to dynamically create custom software objects.

- JavaScript provides functions to convert text in a string to upper and lower case (`toLowerCase` and `toUpperCase`) and also to remove leading and trailing spaces (`trim`).
- JavaScript provides a `startsWith` function that can be used to detect whether one string starts with another.
- A JavaScript function can be made to run when the user of an application changes the text inside an input element in the document. This is achieved using the `onchanged` attribute of the `input` or `textarea` element.

Here are some questions that you might like to ponder about what we have learned in this chapter:

**Question:** What is the difference between a `textarea` element and an `input` element in HTML?

Both elements have a `value` attribute that can be used by a JavaScript to read and write their contents. A `textarea` can be made to span an area of the HTML document to allow multiple lines of text to be entered. The `textarea` element can be given rows and cols attributes that specify the size of the area on the page.

**Question:** How does a `textarea` represent multiple lines of text?

When the `value` attribute of a `textarea` is read by a JavaScript program the browser separates each individual line using the linefeed character which is used in strings to mark the end of each line. The `textarea` also uses linefeed characters to arrange the display of any text loaded into it. The `textarea` will also break text and display scroll bars as appropriate to make the text fit on the screen.

**Question:** Can I add a property to any JavaScript variable?

No. JavaScript will not complain if your program tries to do this, but properties added to variables containing boolean values, strings or numbers are not stored with the variable. If you want to add properties to an object you need to tell JavaScript that your program is creating an object by using the {} notation.

**Question:** Can I use an object as an argument to a function call?

Yes. We have done this many times in our programs already. The argument is passed as a reference to the object.

**Question:** What happens if an object in memory has no variable referring to it?

Good question. As a program runs variables that refer to an object could be reassigned to refer to different objects. In addition, variables defined using `let` and `var` may be discarded as program execution moves in and out of blocks of code. This can lead to objects in memory that no longer have any variables that refer to them. The browser that is running the JavaScript program also runs a special process called the “garbage collector” which searches for objects that have no references and reclaims the memory used by them. This is an automatic process as far as our programs are concerned, although we should be careful not to write code that creates and discards large numbers of objects so that our applications are not slowed down by the garbage collector having to come in and clean up all the time.

**Question:** Does a browser only keep one local storage area on a given machine?

A browser keeps one local storage area per web connection *origin*. Two web sites on the same host will be on the same origin so the sites <https://robmiles.com/blog> and <https://robmiles.com/javascript> would share the same

local storage. JavaScript programs running from pages stored on a particular PC will all share the same local storage on that PC.

**Question:** How do property accessors really work?

A property accessor uses an accessor value to get the name of a property on an object. For example `contact["name"]` would access the `name` property of an object referred to by the variable `contact`. A program can also use other value types to access a property of an object, for example `contact[99]` would also access a property (using a number as an accessor) as would `contact[true]` (using a boolean value as an accessor). JavaScript managers this by converting the property to a string and then using this to search the properties of the object. **You should not however use anything other than a string as a property accessor. Just because something is possible does not mean that you should do it.**

**Question:** Can I use a reference as a property accessor?

This would be legal JavaScript but it would be a very dangerous thing to do because when JavaScript converts the reference to a string (see above) it gets the same string for all object references.

**Question:** What happens if a program uses an accessor value that does not exist?

If JavaScript can't find a property with a particular accessor name it returns the value "undefined".

**Question:** What is the difference between `[]` and `{}`?

A pair of square brackets creates an empty array object. A pair of braces (curly brackets) creates an empty object. Arrays can use indexers to access array elements and JavaScript will automatically create array elements as a program populates the array. Objects use property accessors to access properties.

**Question:** Can I use a variable to identify the property to be accessed?

Yes you can. If the program creates a string variable that contains the name of the property a program can use the string contents as a property accessor.

**Question:** Can I use a single object to store all the tiny contacts by using the name of a contact as a property accessor?

Good question. We could write something like `contacts["Rob Miles"] = robMilesContact`. This would make a "Rob Miles" property of the `contacts` object which contains a reference to `robMilesContact`. This would remove the need to have an array of contacts but it would also make for strange behavior because the name of the contact would be used to locate it in the contact store. If the name property of the contact was changed we would have to make sure that the property name was also changed otherwise the user would find it difficult to locate the contact. I try to make sure that I store a single piece of data in one place only, rather than having two pieces of linked data elements that have to be kept synchronized.

# 10

## Advanced JavaScript

### What you will learn

In this chapter we are going to take our JavaScript knowledge to the next level. We are going to discover how applications manage errors with exceptions and how to design data storage with classes. We'll discover how class inheritance can save us time when creating applications and how we can use object-oriented techniques to make data components that can look after themselves. We'll be doing this by creating a fully featured data storage application you could use to run a business.

### Manage errors with exceptions

In Chapter 9, when we were looking at JSON, we discovered that JSON uses exceptions to signal when things go wrong. An exception is an object that describes something bad that has just happened. A piece of JavaScript can `raise` or `throw` an exception to interrupt a running program. Now is the time to find out what this means and discover the

part that exceptions play in creating reliable applications.

If you remember, JSON (JavaScript Object Notation) is a standard for encoding the contents of JavaScript variables into text so that they can be stored or transferred to another machine. We used JSON to convert the Tiny Contacts store into a string so that it could be stored using Local Storage in the browser. The JSON object provides methods called `stringify` and `parse` that can move JavaScript objects to and from text strings.

```
var test = {};
test.name = "Rob Miles";
test.age = 21;
var JSONstring = JSON.stringify(test);
```

The statements above create a JavaScript object called `test` which contains `name` and `age` properties. The contents of this object are then converted into a string called `JSONstring` by the `stringify` method.

```
{"name": "Rob", "age": 21}
```

This is the string that would be stored in `JSONstring`. This string can be converted back into a JavaScript object by using the JSON `parse` method:

```
var test = JSON.parse(jsonString);
```

## MAKE SOMETHING HAPPEN

### Breaking JSON

The `stringify` and `parse` methods will fail if they are given invalid inputs. They fail by *raising* or *throwing* an exception. Let's investigate what this means and how programs can be made to handle this failure. Use your browser to open the example application at [Ch10 Advanced JavaScript\Ch10-01 JSON Validator](#) in the examples folder.



This application tests strings of text to find out if they contain valid JSON. It works by handing exceptions thrown by the JSON `parse` method. When the **Parse** button is pressed the application reads the string from the input and displays whether or not the string contains valid JSON. Look back through Chapter 9 to find some valid JSON to use to test it. Try it with a few strings to prove that it works.

Now press F12 to open the Developer View and select the Console tab so that we can investigate the application. The JSON `parse` method fails by throwing an exception. Let's see what that means. Type in the following statement. This statement attempts to parse a string with the dangerous sounding contents "kaboom". Press Enter to see what happens.

```
> JSON.parse("kaboom");
```

The JSON `parse` method is not able to make sense of "kaboom" and it indicates this by *throwing an exception*. We have not included code to catch the exception and so the JavaScript console displays an error message in red:

```
> JSON.parse("kaboom");
Uncaught SyntaxError: Unexpected token k in JSON at position 0
at JSON.parse (<anonymous>
at <anonymous>:1:6
```

We are not used to JavaScript programs complaining in the event of an error in the program code. JavaScript will tend to use the values `Not a Number` (`NaN`), `undefined` or `overflow` to indicate that something has gone wrong if our program combines values incorrectly. Our program must test for these values to decide whether an action has worked properly.

The error created by `parse` is called an *exception*. It denotes the fact that the requested action can't be performed. If the exception is not caught the program will stop running. Statements that might throw an exception can be placed inside a `try` block of code. The `try` block is followed by a `catch` block which contains statements to be performed if an exception is thrown. This is called a `try – catch` construction. Let's type one in. Enter the code below, remembering to press return at the end of every line.

```
> try {
  JSON.parse("kaboom");
} catch {
  console.log("bad json");
```

```
}
```

When you press return after the closing curly bracket of the `catch` block the JavaScript code runs. The `parse` method will fail and throw an exception, but this time the exception is thrown inside a `try` block and the associated `catch` clause will run and log a message on the console:

```
> try {
    JSON.parse("kaboom");
} catch {
    console.log("bad json");
}
bad json
```

Enter the same construction replacing the word “`kaboom`” with some valid JSON “`{}`”. Note that this time the message “`bad json`” is not displayed because `parse` doesn’t throw an exception if the JSON string is valid.

## Catching exceptions

The JSON validator above uses the try-catch construction to display an appropriate message.

```
function doValidate(){
    var inputElement = document.getElementById("inputString");
    var outputElement = document.getElementById("outputResult");

    var inputText = inputElement.value;
    try {235
        var result = JSON.parse(inputText);236
        outputElement.innerText = "Valid JSON";237
    }
    catch {238
        outputElement.innerText = "Invalid JSON";239
    }
}
```

---

<sup>235</sup> Start of the try block

<sup>236</sup> Code that might cause an exception

<sup>237</sup> Statement that is only performed if no exception is caused

<sup>238</sup> Start of the catch block

<sup>239</sup> Code that is obeyed if the exception is caused

The `doValidate` function is called when the `Parse` button is pressed in the JSON validator web page. If the user types in valid JSON, the `JSON.parse` method doesn't throw an exception and the program does not obey any of the statements in the `catch` block. Instead, it goes straight to the end of the method. However, if the `JSON.parse` method can't parse the string that was typed in, the exception is generated and execution transfers immediately to the block of code under the `catch` keyword.

The `try` block can contain many statements. However, this might make it hard for you to work out which statement caused the exception.

## CODE ANALYSIS

### Exception handling

You might have some questions about how exceptions are handled.

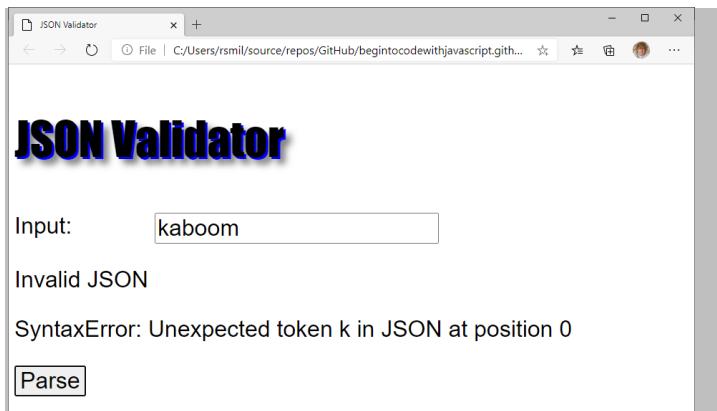
Question: How does the exception stop the display of the "Valid JSON" message?

```
var result = JSON.parse(inputText);
outputElement.innerText = "Valid JSON";
```

The statements in `doValidate` that parse the input JSON and display the message are shown above. When the `parse` method throws an exception the normal execution of this sequence of code is interrupted and program execution moves to the `catch` part of the `try - catch` construction. The reason that "Valid JSON" is not displayed when `parse` throws an exception is because the program does not get that far.

If no exception is thrown the program completes the code in the `try` block and then skips the statements in the `catch` block.

Question: How can a program get hold of the exception object?



The first version of the JSON Validator program ignored the error object produced by the `JSON.parse` function. However, we can add the display of the error message by picking up the error object as a parameter to the `catch` construction:

```
catch (error) {240
    var errorElement = document.getElementById("outputError");
    errorElement.innerText = error;241
    outputElement.innerText = "Invalid JSON";
}
```

This is the code that runs in the catch part of the try – catch construction. It displays the error value (which is highlighted) on an element in the HTML to give the display above. You can find this version of the validator application in the example files at [Ch10 Advanced JavaScript\Ch10-02 JSON Validator Error](#)

Question: Can I throw my own exceptions?

Yes you can. The `throw` statement is followed by the value that is being thrown to describe the error.

```
throw "something bad happened";
```

Question: What happens when an exception is thrown?

<sup>240</sup> Catch with error parameter

<sup>241</sup> Display the error

Throwing an exception stops the program. Any statements following the `throw` statement are not performed.

```
> throw "something bad happened";
> Console.log("This message is never printed");
```

In the code sample above the second statement which logs a message on the console is never performed because when the `throw` is performed the running of this sequence of instructions is ended.

Question: Is it possible to return to a program after an exception has been thrown?

No. Throwing an exception ends the running of a sequence of statements.

Question: Is it possible for the JSON `stringify` method to throw an exception?

Yes. It turns out that there are some JavaScript objects that can't be saved into a string of text. Let's see if we can make one. Go back to the Developer Console in the browser and start by making an empty object called `infiniteLoop`. Type the statement below and press enter.

```
> var infiniteLoop = {}
```

Creating variables always returns an undefined result, so you should see the undefined message.

```
> var infiniteLoop = {}
<- undefined
```

Now we are going to add a property to the object that contains a reference that refers to the object itself. Type in the following statement and press enter.

```
> infiniteLoop.loopRef = infiniteLoop
```

When you press enter the JavaScript console adds a new property to the `infiniteLoop` variable that contains a reference to the `infiniteLoop` variable. In other words, this variable now contains a reference to itself.

```
> infiniteLoop.loopRef = infiniteLoop
<- {loopRef: {...}}
```

Now let's try and `stringify` the value in `infiniteLoop`. Type the following and press Enter:

```
> JSON.stringify(infiniteLoop)
```

When you press enter the `stringify` method tries to save the contents of `infiniteLoop`. It does this by working through each of the properties inside the value and saving each one in turn. It finds the `loopRef` property and so it follows that reference to save that value. The reference leads to the `infiniteLoop` variable so `stringify` has to save that.... This save process would go on forever, rather like a reflection between two parallel mirrors. Fortunately the people who created the `stringify` method are aware of the problem and have added a test for what is called a "circular" reference. If you try to `stringify` an object that contains a reference to itself you will get an error.

```
> JSON.stringify(infiniteLoop)
```

```
Uncaught TypeError: Converting circular structure to JSON
--> starting at object with constructor 'Object'
--- property 'loopRef' closes the circle
at JSON.stringify (<anonymous>)
at <anonymous>:1:6
```

## Exceptions and errors

Now that we know how to throw and catch exceptions, we can consider how we can use them in our applications. However, before we do that I want to talk about the are two types of fault that can occur in a program:

- Things that shouldn't happen
- Things that **really** shouldn't happen.

Things that shouldn't happen include users typing in numbers that are out of range (perhaps an age value of -99), and network connections failing. These are bad things that we expect. Things that **really** shouldn't happen include faults in functions and methods that are used by our applications.

The starting point for the discussion of exceptions was the way that the `JSON.parse` method throws an exception if it is used to parse a string that does not contain valid JSON (for example "kaboom"). This should never happen in the Tiny Contacts program. The only way that this could happen is if the browser storage is corrupted in some way. Is this really a problem? I would say yes, and I've added code to my Tiny Contacts application to deal with this.

```
const STORE_LOAD_OK = 0;
const STORE_EMPTY = 1;
const STORE_INVALID = 2;242

function loadDataStore() {

    // get the data from local storage
    var dataString = localStorage.getItem(storeName);

    // if there is no data make an empty store
    if (dataString == null) {
        dataStore = [];
        return STORE_EMPTY;243
    }

    // read the stored contacts
    try {
```

---

<sup>242</sup> Constant values for the status codes

<sup>243</sup> Return a status code

```
        dataStore = JSON.parse(dataString);
    }
    catch {
        // if the parse fails make an empty store
        dataStore = [];
        return STORE_INVALID;244
    }
    return STORE_LOAD_OK;245
}
```

The code above is a modified version of the `loadDataStore` function from the Tiny Contacts application in the example programs in the folder **Ch10 Advanced JavaScript\Ch10-03 Tiny Contacts Secure**. The call of the `JSON.parse` method is now enclosed in a `try` block and the `catch` block creates an empty contact store if the `parse` method fails with an exception. The first version of `loadDataStore` returned the Boolean value `true` if it worked and `false` if it failed. There are three possible ways that this version of `loadDataStore` can fail:

- The data is not present in local storage is not present because this is the first time the application has been used.
- The data loaded from local storage is not valid JSON (this will cause an exception in parse)
- The data was found and loaded successfully.

The function returns one of three status values to indicate which of these possible outcomes happened when `loadDataStore` was called. Programmers can then test this value to determine what happened when the contacts store was loaded.

```
const STORE_LOAD_OK = 0;
const STORE_EMPTY = 1;
const STORE_INVALID = 2;246
```

These values have been declared using the keyword `const`. This means that their value cannot be changed by the program when it runs. This is sensible because they are being used to indicate status values. The return values from `loadDataStore` are used by the function `doStartTinyData`. If the store is empty or invalid an alert is displayed for the

---

<sup>244</sup> Return a status code

<sup>245</sup> Return a success code

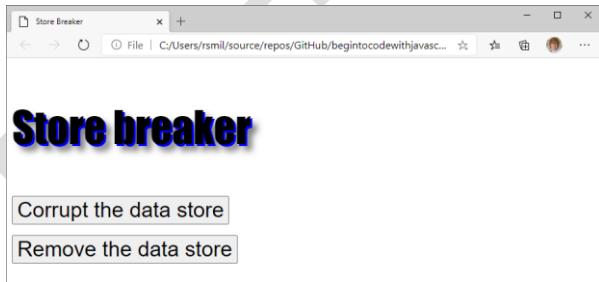
<sup>246</sup> Constant values for the status codes

user.

```
function doStartTinyData(storeNameToUse) {
  storeName = storeNameToUse;

  var loadResult = loadDataStore();247

  switch(loadResult){
    case STORE_LOAD_OK:248
      break;
    case STORE_EMPTY:249
      alert("Empty store created");
      saveDataStore();
      break;
    case STORE_INVALID:250
      alert("Store invalid. Empty store created");
      saveDataStore();
      break;
  }
}
```



<sup>247</sup> Try to load the contact store

<sup>248</sup> If the store loaded OK we don't do anything

<sup>249</sup> If the store was empty we make a new one

<sup>250</sup> If the store was corrupted, we make a new one

**Figure 10.35** Store breaker

It is very important that once we have created some error handling code we also create a way of testing it. In this case I created a new application which breaks the storage. You can see it in Figure 10.1 above and find the application itself in the folder **Ch10 Advanced JavaScript\Ch10-04 Store Breaker** in the examples. You can use this application to “break” the data storage for the Tiny Contacts application.

```
<!DOCTYPE html>
<html lang="en">

<head>
  <title>Store Breaker</title>
  <link rel="stylesheet" href="styles.css">
  <script src="breakstore.js"></script>
</head>

<p class="menuHeading"> Store breaker</p>

<p>
  <button class="menuButton" onclick="doBreakStore()">Corrupt the data store</button>
</p>
<p>
  <button class="menuButton" onclick="doEmptyStore()">Remove the data store</button>
</p>
</body>
</html>
```

This is the HTML for the store breaker application. It contains two buttons. One button is pressed to corrupt the data store. It stores the string “kaboom” in the data store which will cause `JSON.parse()` to fail. The second button is pressed to remove the data store completely. These allow us to test the two possible errors.

```
const storeName = "TinyContactsStore";251
function doBreakStore() {252
  var reply = confirm("Click OK to corrupt the data store");253
```

<sup>251</sup> Name of the store in local storage

<sup>252</sup> Called to break the JSON storage

<sup>253</sup> Show a confirm dialog

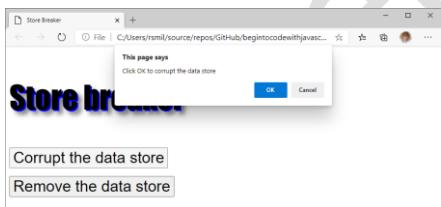
```

if (reply) {254
  localStorage.setItem(storeName, "kaboom");255
}
}

function doEmptyStore() {256
  var reply = confirm("Click OK to remove the data store");
  if (reply) {
    localStorage.removeItem(storeName);257
  }
}

```

These functions use a feature of JavaScript that we haven't seen before. The `confirm` function allows a user to confirm an action. It pops up a message box containing the prompt string and offers the user a chance to either confirm the action or cancel. If the user confirms the action by clicking the OK button the `confirm` function returns the value `true`.



**Figure 10.36** The `confirm` function display

#### PROGRAMMER'S POINT

#### Error management is crucial

Dealing with errors is time consuming and difficult. It can be hard to test an error handler. We had to write a special “breaking” program to test the error handling in the Tiny Contacts application. However, error management is a crucial part of software. Viruses and malware often attack systems by causing errors and then

<sup>254</sup> If the user confirms perform the action

<sup>255</sup> Save an invalid store name

<sup>256</sup> Called to empty local storage

<sup>257</sup> Remove the local storage item

exploiting badly written or non-existent error handlers. When you are deciding how long it will take to create an application make sure that you include the time it will take you to decide how errors are to be handled and write and test the components to deal with them.

## Class design

From what we've learned up to now, we can regard a Java object as a container. We've seen that a program can add properties to an object so that it can assemble related items. We explored this by creating an object-based application that stores contact details. Object properties were added by the program one at a time, building up a complete set of contact information by adding name, address and phone number properties to an "empty" object. This works well for small applications, but sometimes you want to map out your class design rather than building it up as the program runs. And, as we shall see, using classes to design objects also brings benefits by reducing the amount of code that we write.

### Fashion Shop application

Your lawyer client is very happy with her Time Tracker application. She's been showing it to her friends, and they've been very impressed—particularly a friend who runs a fashion shop and has been looking for an application to help her manage her stock. She sells a range of clothing items and needs help tracking inventory. She's keen to get your help, and she's offering discounted prices, or even free clothing, in exchange. Free fashion sounds like an interesting idea, so you sit down with your new client and talk about what she needs.

She tells you that stock arrives from suppliers, and she enters the details in her stock book. For each different item that she sells she stores a page of data in the book. She updates the stock level in the book when stock arrives from her suppliers and when she sells something. She shows you two of the pages from her book.

#### Imogen's Fashions

##### DRESS

STOCK REFERENCE:	221
STOCK LEVEL:	541098
PRICE:	60
Description:	Strapless evening dress
COLOR:	Red
PATTERN:	Swirly
SIZE:	10

#### Imogen's Fashions

##### PANTS

STOCK REFERENCE:	222
STOCK LEVEL:	321
PRICE:	45
Description:	Good for the workplace
COLOR:	Black
PATTERN:	Plain
LENGTH:	30
WAIST:	30

Figure 10.37 Fashion shop stock items

Figure 10.3 shows us what Imogen does when she works with the stock data. We can use this as the basis of our program specification. As usual we draw some designs showing how the program will be used. There will be more than one design because this program will be spread across several “pages”. For each page we will need to also write a “story” which describes how the page will be used. This will also allow us to work through the feature provided by that page to make sure that we know exactly what the program should do.



Figure 10.38 Fashion shop main menu

Figure 10.4 shows the main menu of the program. Imogen can click the buttons to select pages to add dresses, pants, skirts and tops to her stock. She can select an update page to edit any existing stock items to change their description or stock levels. She can also obtain a list of all her stock items by pressing the List button.

Figure 10.39 Fashion shop add item

Figure 10.5 shows how an item would be added to stock. When she clicks **Save** the program will assign a stock number to the item and save it.



Figure 10.40 Add complete

When a new item is added to the store the program displays an alert as shown in Figure 10.6. This gives the stock number which was assigned to the new item. The **Update** stock item button is used to search for a given stock number and display it for editing.



Figure 10.41 Update Stock

Figure 10.7 shows how Imogen will search for stock to be updated. When the **Find** button is clicked the program will search for the given stock item and display it for editing. The final button on the main menu generates a list of stock items that Imogen can look through. Each item has an **Update** button that she can click to open the update page for that item.



Figure 10.42 Stock List

Imogen reckons that this will be a good start for the application, you agree a price in stylish clothing and you start work on the program. The first thing that you need to do is decide how the different stock items are going to be stored.

MAKE SOMETHING HAPPEN

## Manage Imogen's Fashions

The application in the sample folder **Ch10 Advanced JavaScript\Ch10-05 Fashion Shop** implements a working fashion shop store. It creates a set of test data that you can view. You can also create your own fashion items, store them and then search for them by their stock reference. You should spend some time adding and editing stock items to get a feel for how it is used. Then we can start working out how each part works

### Store stock data

We could store information for a particular stock item in a JavaScript object by creating an empty object and then adding properties to it:

```
myDress = {}258
myDress.stockRef=21259;
myDress.stockLevel=8;
myDress.price=60;
myDress.description="Strapless evening dress";
myDress.color="Red";
myDress.pattern="Swirly";
myDress.size=10;
```

The statements above create an object called `myDress` which contains all the data for the dress in Figure 10.3. However, there is a much easier way of creating objects that contain properties. We can create a `class` that tells JavaScript how to make a `Dress` object and what the object contains:

```
class Dress{
    constructor(stockRef, stockLevel, price, description, color, pattern, size){
        this.stockRef = stockRef;
        this.stockLevel = stockLevel;
        this.price = price;
        this.description = description;
        this.color = color;
        this.pattern = pattern;
        this.size = size;
    }
}
```

<sup>258</sup> Create an empty object

<sup>259</sup> Add a stockRef property

```
}
```

The JavaScript above doesn't store any data. Instead it tells JavaScript the properties that are stored inside a `Dress` object and how to construct one. The `constructor` method is called to create an *instance* of the `Dress` class. Now we can create a new `Dress` much more easily:

```
myDress=new Dress(221,8,60,"Strapless evening dress","red","swirly",10);
```

The `new` keyword tells JavaScript to find the specified class and run the `constructor` method in that class to create a new instance of the class. When the `constructor` method runs it copies the parameter values into properties in the newly created object. The keyword `this` which you can see in the `constructor` method above means "a reference to the object that this method is running inside".

```
this.pri^ce=price;
```

The confusing looking statement above takes the `price` value that was supplied as an argument to the `constructor` and assigns it into a newly created `price` property on the object that is being created.

```
yourDress=new Dress(221,8,60,"Elegant party dress","blue","plain",12);
herDress=new Dress(222,5,65,"Floaty summer dress","green","floral",10);
```

If the word `this` is confusing, consider the two statements above. They create two `Dress` instances. Each time the `constructor` in the `Dress` class runs it must set up a different object. In the first call of the `constructor` to set up `yourDress` the keyword `this` represents a reference to the `yourDress` object. In the second call of the `constructor` to set up `herDress` the keyword `this` represents "a reference to the `herDress` object".

## CODE ANALYSIS

---

<sup>^</sup> property of the object

<sup>^</sup> parameter to the function

## Objects and constructors

Question: What happens if I create a class that doesn't contain a constructor?

If you miss out the constructor method JavaScript will create one for you which is empty.

Question: What happens if I miss out some arguments from the call of a constructor?

If you miss out the arguments to a call of a JavaScript function or method the values of those parameters are set to "undefined".

```
shortDress = new Dress(221,0,50);
```

This statement would create a dress with the stock reference (221), stock level (0) and price (50) but the description, color, pattern and size properties would be set to the value undefined.

Question: What is the difference between a function and a method?

A method is exactly like a function, but it is declared within a class. The constructor for a class is the first method that we have created. Later in this section we will discover how we can add methods to classes to make objects that can provide services for our programs. A function is declared outside any class.

Question: I still don't understand `this`. What does it mean?

To understand what `this` does, it is a good idea to remember the problem it is solving. When a constructor method runs it must write property values in the object that it is setting up. The constructor for the `Dress` object needs to set the values of stock reference, stock level and price etc. JavaScript provides the keyword `this` to represent that reference.

## Object-oriented design

It would make sense to create a class to hold each kind of data we wish to store. Programmers call this object-oriented programming. The idea is that elements in a solution are represented by software "objects." The first step in creating an application is to identify these objects.

In the English language, words that identify things are called nouns. When trying to work out what classes a system should contain, it's a good idea to look through the description of a system and find all the nouns. As an example, consider the following description of a fast-food delivery application.

\*\*\*PRODUCTION: Please highlight (yellow) the following words below: customer, dish, menu, order. \*\*\*

"The **customer** will select a **dish** from the **menu** and add it to his **order**."

I've identified four nouns in the description, each of which will map to a specific class in the application. If I were working for the fast-food delivery company, I would next ask them what data they stored about customers, dishes,

menus, and orders.

#### PROGRAMMER'S POINT

##### Don't write any code before you have completed your data design

For a commercial project, you would spend a lot of time on the design of the classes in your system before you wrote a single line of code. This is because design mistakes are much easier to fix at the beginning of the project, rather than after code has been written.

In the case of our fast-food management example above, we would want to make sure that the customer class holds all the information required to make the business work. We would do this by creating "paper" versions of the classes and then working through all the usage scenarios (creating an order, cooking an order, delivering an order) to make sure that all the data the application needs is being captured.

If the application must store a customer telephone number so that the delivery driver can call for directions if needed, it is best to discover this at the beginning of the project, rather than after the entire user interface has been created.

We will write code and discuss it as we go along because we are learning about data design and JavaScript programming. However, if I were creating a professional solution, I'd spend a lot of time away from JavaScript working out the design before I created any classes.

When we talk to our fashion shop customer, she'll talk about the dresses, pants, hats, tops, and other items that she wants the application to manage. Each of these could be objects in the application and can be represented by a class. Each class will contain the properties that describe that item of clothing. Let's start by considering just the information for dresses and pants and create some classes for these objects. We already have a class for [Dress](#), so let's make one for [Pants](#).

```
class Pants{  
    constructor(stockRef, stockLevel, price, description, color, pattern, length, waist){  
        this.stockRef = stockRef;  
        this.stockLevel = stockLevel;  
        this.price=price;  
        this.description=description;  
        this.color = color;  
        this.pattern = pattern;  
        this.length = length;  
        this.waist = waist;  
    }  
}
```

The code above defines a `Pants` class. It contains a constructor method to set up the contents of that class. Our program can now create instances of these classes:

```
myDress=new Dress(221,8,60,"Strapless evening dress","red","swirly",10);
myPants=new Pants(222,1,45,"Good for the workplace","black","plain",30,30);
```

When I wrote this sample code, I found myself using a lot of block-copy commands in the editor. This is not a good thing.

#### PROGRAMMER'S POINT

##### Block copy is not your friend

Visual Studio Code will let you select a block of statements and copy them into another point in your program. I call this action "block copy." And it is not your friend.

When writing the code for the `Dress` and `Pants` classes, you might think it is efficient programming to just block copy the repeated elements from one class to another. However, this is not a good idea. If you are copying the same code from one part of your program to another, you are not programming most efficiently. A good programmer will try to write a piece of code exactly once. If the code is used more than once in an application, a good programmer will convert the code into a method or function and then call the method each time it's needed.

This is not about making sure that our programs are as small as we can make them. It's about self-preservation. If you block copy a piece of code into lots of different places in your application, you'll have a real problem if you find a bug in the copied code. You'll need to go through your entire application and fix all the broken copies of that code. On the other hand, if you find a bug in a method, you can fix it just once, and it is fixed for every situation in which that method is used. Fortunately, there is a way we can remove the need for numerous copies of the same code, which we will discuss now.

If I find myself copying program text from one place to another, I take this as a trigger to step back from the problem and think about different ways of structuring my solution.

## Creating superclasses and subclasses

JavaScript classes support a mechanism called inheritance. This is another aspect of object-oriented design. Inheritance lets us base one class on an existing superclass. This is called extending the superclass. We can greatly simplify the design of our classes for the Fashion Shop program by creating a superclass, which we can call `StockItem`.

The `StockItem` class will store all the attributes common to all the data items in the shop. These are the stock reference, price, color, and stock level. The `Dress` and `Pants` classes will extend the `StockItem` class and add the properties particular to dresses and pants. Figure 10-9 shows the arrangement of the classes we're creating. In software design

terms, this is called a class diagram.

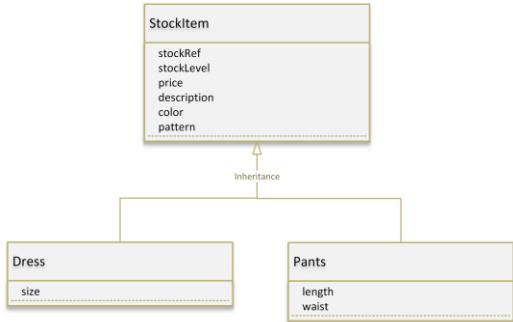


Figure 10.43 Fashion shop class diagram

The class diagram shows the relationship between classes in a system. Figure 10.9 shows that both `Pants` and `Dress` are subclasses of the `StockItem` class (meaning they are based on that class). We could also say that the `StockItem` class is the superclass of `Dress` and `Pants`.

In real life, inheritance means stuff that you get from people who are older than you. In JavaScript terms, inheritance means the attributes a subclass gets from its superclass. Some programmers call the superclass the *parent* class and the subclass the *child* class.

The key to understanding inheritance is to focus on the problem it is solving. We're working with a collection of related data items. The related items have some properties in common. We want to implement the shared properties in a superclass and then use this superclass as the basis of subclasses that will hold data specific to their item type. That way, we only need to implement the common properties once, and any faults in the implementation of those properties need only be fixed once.

Working in this way has another advantage. If the fashion shop owner decides that she would find it useful to be able to store the manufacturer of the items she's selling, we can add a manufacturer attribute to the `StockItem` class, and all the subclasses will inherit that attribute too. This will be easier than adding the attribute to each class.

## Abstraction using objects

Another way to think of this is to consider what we are doing as *abstraction*. We first encountered abstraction in chapter 9 in the section "Use a schema" where we created a design for all data stores rather than building individual ones. We have seen abstraction means "stepping back" from the problem and taking a more general view. In our conversations with the fashion shop owner, we would like to talk in general terms about the things she would like to do with the stock in her shop. She will want to add stock items, sell stock items, find out what stock items she has, and so on. We can talk to her about her stock in abstract terms and then later go back and fill in the specific details about each type of stock and give them appropriate behaviors.

Programmers use abstraction a lot. They talk about things like stock items, customers, and orders without considering specific details. Later, they can go back and "fill in the details" and decide what particular kinds of stock items, customers, and orders with which the application will work. We'll create different kinds of stock items in our Fashion Shop program. The `StockItem` class will contain the fundamental properties for all the stock, and the subclasses will represent more specific items.

The diagram in Figure 10-9 is called a *class hierarchy*. It shows the superclass at the top and subclasses below. When you travel down a class hierarchy, you should find that you move from the abstract toward the more concrete. The least abstract classes are `Pants` and `Dress` because these represent actual physical objects in our application.

## CODE ANALYSIS

### Understanding inheritance

Here are some questions about object-oriented design and inheritance. Try to come up with your own answers before reading the answers I've provided.

Question: Why is the superclass called super?

This is a good question and one that has confused me for a long time. The word *super* usually implies something better or more powerful. A "superhero" has special powers that ordinary people do not. However, in the case of a superclass, this doesn't seem to be the case. The superclass has fewer powers (fewer properties) than the subclass that extends it.

I think the word *super* makes sense if you consider it as something that classes descend from. The *super* object is above the *sub* object, just like superscript text is above subscript text. The object is the superclass because it is above everything else.

Question: Which is most abstract, a superclass or a subclass?

If you can work out the answer to this question, you can start to consider yourself an "object-oriented ninja." Remember that we use abstraction as a way of "stepping back" from the elements in a system. We'll say "receipt" rather than "cash receipt" or "StockItem" rather than "Pants."

Question: Can you extend a subclass?

Yes, you can extend a subclass. We could create a `Jeans` class that extended the `Pants` class and contained a `style` property that could be "skinny", "high waist", "bootcut" or "flared". In JavaScript, there is no limit to how many times you can extend classes, although I try to keep my class diagrams fairly shallow, with no more than two or three subclasses.

Question: Why is the `pattern` property not in the `StockItem` class?

Most impressive. Well spotted. The `pattern` property is in both the `Dress` and `Pants` classes. It might seem sensible to move the property into the `StockItem` class with the `color`, `stockLevel`, and `price` attributes.

The reason I haven't done this is that I think that the fashion shop might sell some stock items that have no pattern—for example, items of jewelry. I want to avoid a class having data properties that aren't relevant to that item type, so I've put [pattern](#) values into the [Dress](#) and the [Pants](#) class instead.

I'm not particularly happy with this, in that ideally a property should appear only in one class, but in real-world design you come across these issues quite often. One possible way to resolve the issue would be to create a subclass called [PatternedStock](#) that is the superclass for [Dress](#) and [Pants](#), but I think that would be too confusing.

Question: Will our system ever create a [StockItem](#) object?

JavaScript will allow the creation of a [StockItem](#) object (an instance of the [StockItem](#) class), but it's unlikely that we would ever actually create a [StockItem](#) on its own.

Some programming languages, for example, C++, Java, and C# allow you to specify that a class definition is abstract, which stops a program from making instances of that class. In these languages, an abstract class exists solely as the superclass for subclasses. However, JavaScript does not provide this feature.

Question: The owner of the fashion shop thinks that one day she might like to keep track of which customer has bought which item of stock. That way she can look at their past purchases and make recommendations for future purchases. Here are three ways to do this. Which would make the most sense?

8. Extend the [StockItem](#) class to make a [Customer](#) subclass that contains the customer details because customers buy stock.
9. Add [Customer](#) details to each [StockItem](#).
10. Create a new [Customer](#) class that contains a list of the [StockItems](#) that the [Customer](#) has bought.

Option 1 is a bad idea because a class hierarchy should hold items that are in the same "family." In other words, they should all be different versions of the same fundamental type. We can see that there is some association between a [Customer](#) and a [StockItem](#), but making a [Customer](#) a subclass of [StockItem](#) is a bad idea because they're different kinds of objects. The [StockItem](#) holds attributes such as [price](#) and [stockLevel](#), which are meaningless when applied to a [Customer](#).

Option 2 is a bad idea because several customers might buy the same [StockItem](#). The customer details cannot be stored inside the [StockItem](#).

Option 3, adding a new [Customer](#) class, is the best way to do this. Remember that because objects in JavaScript are managed by references, the list of items in the [Customer](#) class (the items the customer has bought) will just be a list of references, not copies of [StockItem](#) information.

## Store data in a class hierarchy

Now that we've decided using inheritance is a good idea, we need to consider how to make it work with our classes.

```
class StockItem{
    constructor(stockRef, stockLevel, price, description, color){
        this.stockRef = stockRef;
        this.price=price;
        this.description=description;
        this.stockLevel = stockLevel;
        this.color = color;
    }
}
```

This is the `StockItem` class file. It contains a constructor method to set up a `StockItem` instance. The `StockItem` class will be the superclass of all the objects that the fashion shop will be selling. We can create a `Dress` class that is a subclass of the `StockItem` class to hold information about dresses that the fashion shop will be selling.

```
class BrokenDress extends StockItem{
    constructor(stockRef, stockLevel, price, description, color, pattern, size){
        this.pattern = pattern;
        this.size = size;
    }
}
```

The `BrokenDress` class extends the `StockItem` class. It only contains the properties that are specific dresses. However, we have a problem if we try to use the `BrokenDress` class:

```
myDress=new BrokenDress(21,4,50,"Strapless evening dress","red","swirly",10);
```

The statement above tries to create an instance of the `BrokenDress` class. This statement will fail with an error:

```
Uncaught ReferenceError: Must call super constructor in derived class before accessing 'this' or re-
turning from derived constructor at new BrokenDress
```

JavaScript is telling us that to create a `BrokenDress` our constructor must first create a `StockItem`. The constructor in the `BrokenDress` class that we have created doesn't do this. So this class is broken. Hence the name. To fix it we need to make a `Dress` class that contains a constructor that first constructs the super object. JavaScript provides the `super` keyword which can be used in a constructor to call the constructor in the super class. The constructor method for the `Dress` class calls the constructor method in the `StockItem` class by means of the `super` keyword.

```
class Dress extends StockItem{
    constructor(stockRef, stockLevel, price, color, pattern, size){
        super(stockRef, stockLevel, price, color);260
        this.pattern = pattern;
        this.size = size;
    }
}
```

## MAKE SOMETHING HAPPEN

### Investigate using super to create instances

The example application in the folder Ch10 Advanced JavaScript\Ch10-06 Fashion Shop Classes contains the `StockItem`, `BrokenDress` and `Dress` classes that you can experiment with. You can use the Developer View to debug the process of creating a `Dress` by placing a breakpoint at the first statement of the `Dress` constructor and then stepping through the JavaScript as the `super` keyword is used to call the constructor in the `StockItem`.

## Add a method to give an object a behavior

Things that are part of a class are called the *members* of the class. We know how to create properties which are members of a class. Now we are going to find out how to add method members. Adding a method member to a class allows it to do things for our program. At the moment the classes we have created don't contain any methods. A useful method might be one that allows a `Dress` to provide us with a string describing its contents. We can use this to produce the text to be displayed for the Stock List menu item.

```
class Dress extends StockItem{

    constructor(stockRef, stockLevel, price, description, color, pattern, size){
        super(stockRef, stockLevel, price, description, color);
        this.pattern = pattern;
        this.size = size;
    }

    getDescription(){261
        var result = "Ref:" + this.stockRef +
    }
}
```

<sup>260</sup> Call the constructor in the super class

<sup>261</sup> getDescription method

```
    " Price:" + this.price +
    " Stock:" + this.stockLevel +
    " Description:" + this.description +
    " Color:" + this.color +
    " Pattern:" + this.pattern +
    " Size:" + this.size;262
    return result;263
}
}
```

The `Dress` class above contains a method called `getDescription` that can be called to get a description of the contents of the object. Note that the method uses the `this` reference to access the properties of the object that is being described. A program can use this method to get a string that describes a particular dress.

```
myDress=new Dress(221,8,60,"Strapless evening dress","red","swirly",10);
console.log(myDress.getDescription());
```

The first statement above creates a dress object called `myDress`. The second statement uses the `getDescription` method on the `myDress` object to display a description of the dress. It would display the following:

```
Ref:221 Price:60 Stock:8 Description:Strapless evening dress Color:red Pattern:swirly Size:10
```

The Fashion Shop application will use the `getDescription` method to build an HTML element to be displayed in a stock list.

## MAKE SOMETHING HAPPEN

### Investigate use the `getDescription` method

The example application in the folder **Ch10 Advanced JavaScript \Ch10-07 Fashion Shop Methods** contains the `StockItem`, `Dress` and `Pants` classes that you can experiment with. You can use the Developer View to create `Dress`

---

<sup>262</sup> Assemble a string containing a description

<sup>263</sup> Return the string

and `Pants` instances and call the `getDescription` method to view their contents.

## Objects and polymorphism

The next thing I want to talk about has the most impressive name in the entire book. The word polymorphism comes from the Greek language and means "the condition of occurring in multiple forms." In software engineering it means regarding an object in terms of what it can do rather than what it is.

A great thing about the `getDescription` method is that other parts of the Fashion Shop application don't need to know how the `Dress` and `Pants` classes store their data or even what data is stored inside them. A part of the program that needs to produce a description of a `Dress` doesn't have to pull out the various properties from a `Dress` instance, it just has to call the `getDescription` method to get a string that describes that particular dress. What's more, this part of the program doesn't need to care whether it is dealing with dresses or pants, it can just view these items as "things I can use `getDescription` to get a description of".

Polymorphism means thinking about objects in terms of what they can do, rather than what they are, and allowing each object to perform a particular action in a way specific to that object. A given object can be viewed in many ways, depending on what you want to do with it. Different parts of the Fashion Shop will view objects in terms of abilities such as "get a description string", "set a discount", "save" and "Load". Each of these abilities can be provided by methods inside the object with a characteristic name which can then be used by the rest of the system to perform that action. Part of an object-based design process involves identifying the behaviors required of objects and specifying them as methods.

## Overriding methods in sub classes

A fundamental principle of object-oriented design is that a given object contains all the behaviors for that particular object. In this respect the `getDescription` method in the `Dress` class is not good. The first four items that are used to build the description string are not held in the `Dress` class they are held in the `StockItem` super class.

```
getDescription(){
    var result = "Ref:" + this.stockRef +264
        " Price:" + this.price +265
        " Stock:" + this.stockLevel +266
        " Description:" + this.description +267
        " Color:" + this.color +268
```

<sup>264</sup> value is from the `StockItem` class

<sup>265</sup> value is from the `StockItem` class

<sup>266</sup> value is from the `StockItem` class

<sup>267</sup> value is from the `StockItem` class

<sup>268</sup> value is from the `StockItem` class

```

        " Pattern:" + this.pattern +
        " Size:" + this.size ;
    return result;
}
}

```

If we added a new property to the `StockItem` class we would have to also change the `getDescription` method in the `Dress` class to display the new property. If `StockItem` had many sub-classes we would have to change every one. What we would like to do is make the `StockItem` class responsible for delivering the description of what it contains, and then use that description in the `Dress` class. It turns out that we can do this by creating a `getDescription` method in the `StockItem` class and then *overriding* the `getDescription` method in sub classes of `StockItem`.

```

class StockItem{
    getDescription(){
        var result = "Ref:" + this.stockRef +
        " Price:" + this.price +
        " Stock:" + this.stockLevel +
        " Description:" + this.description +
        " Color:" + this.color;
        return result;
    }
}

class Dress extends StockItem{269
    getDescription(){
        var result = super.getDescription() +270
        " Pattern:" + this.pattern +
        " Size:" + this.size ;
        return result;
    }
}

```

Both `StockItem` and `Dress` contain a `getDescription` method. We say that the `getDescription` method in the `Dress` class *overrides* the `getDescription` method in the `StockItem` class. A description of a `Dress` must include a description of the contents of the super object of `Dress`, and so the JavaScript `super` keyword is used to invoke the `getDescription` method in the `StockItem` object. Within a method in a class the word `super` is a reference to the `super` object (the one

---

<sup>269</sup> Override the `getDescription` method in the super class

<sup>270</sup> Call `getDescription` from the super class

above the object in the class hierarchy).

## CODE ANALYSIS

### Understanding super

The example application in the folder **Ch10 Advanced JavaScript \Ch10-08 Dress Shop Override Methods** contains the `StockItem`, `Dress` and `Parts` classes that work in exactly the same way as the previous example. But these versions use overridden versions of the `getDescription` method. You can use the debugger to explore how the overridden method is called. However, you might also have some questions.

Question: How does the `super` reference work?

When a program is running JavaScript keeps a lump of data about each class that is being used. One of the items of information that JavaScript stores is the super class of each class. When JavaScript sees the keyword `super` in a statement it looks in the definition of the class the method is part of to find the super class. It then finds the specified method in the super class and runs it.

Question: What would happen if you tried to use `super` in a class at the top of a hierarchy?

This would cause an error because JavaScript would not find a `super` method.

Question: Can you override a function in a JavaScript program?

No. A function is not part of an object. You can only override a method which is part of an object.

## Static class members

You will have noticed that the demonstration Fashion Shop app contains a lot of test data. This is generated by the data objects themselves. The data and methods to create test dresses are not part of any `Dress` instance. Instead they are part of the `Dress` class itself. JavaScript lets us do this by creating properties and methods which are *static*. The word *static* in this context means “always there” rather than unchanging. We don’t need to use `new` to create an instance of the `Dress` class to get hold of the static members of the class. These members exist as soon as the JavaScript class is loaded by the browser.

```
static colors = ["red", "blue", "green", "yellow"];271
static patterns = ['plain', 'striped', 'spotted', 'swirly'];272
```

<sup>271</sup> Static array of colors

<sup>272</sup> Static array of patterns

```

static sizes = [8, 10, 12, 14];273

static getTestItems(dest) {274
    var stockNo = StockItem.getLargestStockRef(dest) + 1;275
    for (let color of Dress.colors) {276
        for (let pattern of Dress.patterns) {277
            for (let size of Dress.sizes) {278
                let price = StockItem.getRandomInt(10, 200);279
                let stock = StockItem.getRandomInt(0, 15);280
                let description = color + " " + pattern + " dress";281
                dest[dest.length] = new Dress(stockNo, stock, price, description, color, pattern, size);
            };282
            stockNo = stockNo + 1;283
        }
    }
}

```

The method `getTestItems` above works through arrays of colors, patterns and sizes to create a large number of dress stock items which it adds to an array supplied as a parameter. The `getTestItems` method is `static` so that it can be called without the program needing to make an instance of the `Dress` class. The data arrays that are used by the `getTestItems` method are also defined as `static`. The `getTestItems` method also uses the static method `getRandomInt` which is declared in the `StockItem` class. The `getRandomInt` method is used to obtain random prices and stock levels for the dresses that are created. It is based on the function that we created to make a random dice. The `getLargestStockRef` method is used to search the fashion store to find the highest stock number in it. This ensures that the function

---

<sup>273</sup> Static array of sizes

<sup>274</sup> Static function to get test data

<sup>275</sup> Make sure we don't repeat stock numbers

<sup>276</sup> Work through the colors

<sup>277</sup> Work through the patterns

<sup>278</sup> Work through the sizes

<sup>279</sup> Get a random price

<sup>280</sup> Get a random stock level

<sup>281</sup> Build a description string

<sup>282</sup> Create a new Dress and store it

<sup>283</sup> Advance the stock number

doesn't create any stock items with the same stock number as existing ones.

The static members of a class are marked with the `static` keyword. Static class members are accessed via the class identifier so the above method can be used as follows to create an array full of `Dress` values:

```
demo = []
Dress.getTestItems(demo);
```

## CODE ANALYSIS

### Understanding static

The example application in the folder **Ch10 Advanced JavaScript \Ch10-09 Fashion Shop Static Members** contains `Dress` and `Pants` classes that contain static `getTestItems` method that create large amounts of data that can be used to test our system. However, you might also have some questions.

Question: Does `static` mean that a class member cannot be changed?

No. The `static` keyword affects where a class member is stored, not what you can do with it. Static marks members of a class that are part of a class, not an instance of the class. If you want to make a variable that cannot be changed you can declare it as `const`.

Question: Can you give me an example of a good use for static members of a class?

Static members are things that you want to store for the entire class. Suppose that you wanted to set a maximum price for all the dresses that the fashion shop sells. This could be used to help detect when someone miss-types a price value. This value should be stored in a static member of the class since it does not need to be stored with every dress. If we need to change the maximum price of a dress we can just update the static value and the maximum price for all the dresses will be changed.

You make a method member of a class static when you want to use the method without creating an instance of the class. The JavaScript Math class contains lots of static methods to perform mathematical functions. These are declared as static so that we don't have to create an instance of the Math class to be able to use them.

Question: Where does the stock ref number come from when we create a new stock item?

Each item of stock has a stock reference number. This number identifies a stock item in the same way that a given credit card number identifies a credit card. Imogen will enter the stock reference number to find an item she wants to edit. It is very important that each stock item has a unique stock reference number. The `getLargestStockRef` method searches through all the items and returns the largest stock reference that it has found. The program then adds one to this value to get the next stock number to be used.

```
static getLargestStockRef(items) {
    if (items.length == 0) {284
        return 0;
    }

    var largest = items[0].stockRef;285

    for (const item of items) {286
        if (item.stockRef > largest) {287
            largest = item.stockRef;
        }
    }

    return largest;288
}
```

## Data storage

We have one last problem to solve before we can build the finished solution. We need a way of saving the dress shop data. The Tiny Contacts application that we created in Chapter 9 used JSON to encode the contact objects into text that was then stored as strings in browser local storage. We can do something similar, but the use of a class hierarchy makes it a little bit trickier. To understand why, consider what happens when we convert a dress value to a string using the `JSON.stringify` method.

```
{"stockRef":1,"stockLevel":"11","price":"85",
"description":"red plain dress","color":"red","pattern":"plain","size":"8"}
```

The text above is a JSON string that describes a dress. It contains all the properties in a `Dress` instance, including those in the `StockItem` object superclass. When we load this object back into our program we want to be able to use it as a `Dress` object. Unfortunately there is nothing in the JSON string that tells a program reading it that this is a stored `Dress` value.

## Add type information to JSON

The way to fix this is to add an extra property to our data that gives the type of the data. We can do this in the

---

<sup>284</sup> If there are no items return 0

<sup>285</sup> Start with the first stock reference number

<sup>286</sup> Search through the stock items

<sup>287</sup> If the item has a larger stock reference, save it

<sup>288</sup> Return the largest stock number

constructor for the class:

```
constructor(stockRef, stockLevel, price, description, color, pattern, size) {  
    super(stockRef, stockLevel, price, description, color);  
    this.type = "dress";289  
    this.pattern = pattern;  
    this.size = size;  
}
```

This is the modified [Dress](#) constructor. We can modify the constructor for the other types of clothing so that they create an appropriate type property (i.e. the [Pants](#) constructor must set a type value of “pants” and so on).

## Use the stored type property

```
{"stockRef":1,"stockLevel":11,"price":85,"type":"dress",  
"description":"red plain dress","color":"red","pattern":"plain","size":8"}
```

The JSON above describes a dress that contains a type property. I've highlighted the type information in yellow. What we need now is a way of creating a [Dress](#) instance from this JSON source string. I created a member function in the [StockItem](#) class to do this. The function is called [JSONparse](#). It takes in a string of JSON and returns an object of the type specified by type value in the JSON. It uses a [switch](#) construction to decide what type of object to create.

```
static JSONparse(text) {290  
    var rawObject = JSON.parse(text);291  
    var result = null;292  
  
    switch (rawObject.type) {293  
        case "dress":
```

---

<sup>289</sup> Add a type property to the dress

<sup>290</sup> Static method to read a stock object

<sup>291</sup> Create a raw object from the JSON string

<sup>292</sup> Create an empty result

<sup>293</sup> Decide which type of object is being loaded

```

        result = new Dress();294
        break;
    case "pants":
        result = new Pants();295
        break;
    }
    Object.assign(result, rawObject);296
    return result;
}

```

The `JSONparse` method also uses a method that we have not seen before. It is called `Object.assign` and you can see it used in the last but one statement in the method. The `Object.assign` function copies all the data properties from one object to another. The program uses this to take all the data properties from the JSON object that we read and copy them into the empty object that we have just created. The first argument to the `assign` function is the destination object for the copy. This is the newly created object of the required type. The second argument to the `assign` function is the `rawObject` that was loaded from JSON. This contains all the data properties.

## CODE ANALYSIS

### Load and save

The example application in the folder **Ch10 Advanced JavaScript \ Ch10-10 Fashion Shop No Test Data** is a version of the Fashion Shop that doesn't generate test data when first started. You can create and store stock items and they will be persisted when the browser is closed. It uses local storage in exactly the same way as the Tiny Contacts program, but it creates an array of JSON strings to store the data.

Question: The object that we loaded from the JSON object contains all the data properties that the program needs.  
Why do we have to copy it into another object?

This is because a `Dress` instance contains function members as well as data members. The system needs to be able to ask a `Dress` instance to do things like `getDescription`. An object created from JSON will not have these methods. So we need to create an "empty" `Dress` instance that contains the required methods and then add the data from the object that we have read from JSON.

Question: Why has the `JSONparse` method been made `static`?

---

<sup>294</sup> Make an empty Dress object

<sup>295</sup> Make some empty Pant object

<sup>296</sup> Copy the properties from the object into the result

The `JSONparse` method is called to read stock items from storage. It has to be `static` because when the program is first started there are no stock items loaded.

Question: Do you need a copy of the `JSONparse` method in every stock class (for example `Dress` and `Pants`)?

No. The nice thing about this is that a single copy of the `JSONparse` method in the `StockItem` class will load data for any of the classes because it just copies what has been saved. However, you do need to make sure that the `switch` statement in the `JSONparse` method is kept up to date. If you add a new type of stock (perhaps hats) you would have to add a `case` for that type to the switch.

## Build a user interface

We now have all the behaviors that we need to make the Fashion Shop application. We can put all the different kinds of data in objects which can be saved and loaded. Our class based design means that data properties shared by all the different objects are only stored in one place. The only thing missing is the user interface element. We need a menu system for the application along with views of the different types of data.

### Make stock items display themselves

We have already built HTML documents that look very similar to parts of the Fashion Shop application. When we created the Tiny Contacts application, we created *schemas* objects to describe the display elements that were needed. We can use the same approach to design HTML elements for each of the data classes in the Fashion Shop. If you're not sure how we used schemas to design an HTML document, take a look in the section **Use a data schema** in chapter 9.

```
static StockItemSchema = [
  { id: "price", prompt: "Price", type: "input" },
  { id: "stockLevel", prompt: "Stock Level", type: "input" },
  { id: "description", prompt: "Description", type: "textarea", rows: 5, cols: 40 },
  { id: "color", prompt: "Color", type: "input" }];
```

The code above shows the display schema for the `StockItem` class. This is the same schema design as we used for the Tiny Contacts application. There is an entry for each item to be displayed. The item gives the id of the property, the prompt to be displayed and the type of the input. Three of the items are single line inputs and one is a textarea. If you take a look at Figure 10.5 from earlier in this chapter you will see how this schema defines the display of the top four values to be entered.

```
static buildElementsFromSchema(HTMLdisplay, dataSchema) {
  // work through each of the items in the schema
```

```
for (let item of dataSchema) {297  
    // make an element for that item  
    let itemElement = StockItem.createElement(item);298  
    // add the element to the container  
    HTMLdisplay.appendChild(itemElement);299  
}  
}
```

The `buildElementsFromSchema` method works through the elements in a schema and uses a function called `makeElement` to create each element from the schema information and add it to an HTML element. It is the same mechanism that was used to create the display of the Tiny Contacts application.

```
getHTML(containerElementId) {  
    StockItem.buildElementsFromSchema(containerElementId, StockItem.StockItemSchema);  
}
```

The `StockItem` class contains a method called `getHTML` that calls `buildElementsFromSchema` with the parameters required to build the part of a display needed to edit a `StockItem`. Now that we know how to build the display for a `StockItem` we can consider how to build the display for a `Dress`.

```
static DressSchema = [{ id: "pattern", prompt: "Pattern", type: "input" },  
                     { id: "size", prompt: "Size", type: "input" }];300  
  
getHTML(containerElementId) {301  
    super.getHTML(containerElementId);302
```

---

<sup>297</sup> Work through the schema

<sup>298</sup> Create the display element

<sup>299</sup> Add the element to the page

<sup>300</sup> Schema for the Dress display

<sup>301</sup> Function to add the property elements to an HTML document

<sup>302</sup> Add the elements from the parent object

```
    StockItem.buildElementsFromSchema(containerElementId, Dress.DressSchema);303  
}
```

The code above is the code that builds the display for the dress. The `getHTML` method uses a schema that defines just the extra elements that need to be added to the HTML document for the dress. It calls the `getHTML` function of its super object (which is the `StockItem`) to get the HTML for that object and then adds its own elements on the end.

## CODE ANALYSIS

### Creating HTML

A great programming language is one where you can create code that you are proud of. I'm quite proud of this HTML generating code. It is easy to use and easy to extend. We can add more stock types and easily express what each stock type contains. However, you might have some questions about it.

Question: What does this code do again?

Good question. Remember that our data design has given us some classes that hold all the different data properties in a stock item. We have used inheritance to create a super-class called `StockItem` that holds all the properties shared by all the stock items (for example the price and the number of items in stock). Then we've created sub-classes of `StockItem` that hold data specific to that type of item (for example the length and waist properties of `Pants`).

If we want the user to interact with these properties we will need to create HTML (labels and input elements) in the HTML document displayed by the browser. We could do this by hand but it would be tedious. When we made the Tiny Contacts application we created a schema object that defined the properties to be displayed and then wrote a method that worked through the schema making HTML elements to edit each item. Think of a schema as a "shopping list" of items to be displayed.

The process described above takes the same approach as we used for Tiny Contacts and adds a schema to each type of stock item. It uses the `super` keyword so that the display builder (the `getHTML` method) in `Dress` can call the `getHTML` method in the `StockItem` class. It is necessarily complicated, but if you stare at it hard enough it does make sense. And if you understand it you can call yourself a "class hierarchy ninja".

Question: What would I need to do if I wanted to add a new data property to the `StockItem` class?

This is a situation in which this approach would pay off. If Imogen decides that she wants the system to record some new data in the `StockItem` class we could add a new item to the `StockItem` schema. The display elements produced would be added to the edit displays of all the stock items because they are all sub classes of `StockItem`.

<sup>303</sup> Add the elements from the Dress

## Start the application

```
var mainPage; // HTML element that contains the user interface  
var dataStore; // Array of stock items  
var storeName; // name of save data in local storage  
var activeItem; // currently active stock item (for entry and edit)
```

The application uses four variables that are shared between all the functions. When the Fashion Shop starts these variables must be set up. The `body` element of the HTML document containing the application contains an `onload` attribute that specifies the function to be called when the page is loaded. This function starts the Fashion Shop running. The function is called `doStartFashionShop`.

```
<body onload="doStartFashionShop('mainPage','fashionShop')" class="mainPage">
```

The function `doStartFashionShop` is very similar to the function `doStartTinyContacts` that we created in chapter 9 to start the Tiny Contacts application. This function needs to load the stock data from the browser and set up the other shared variables.

```
function doStartFashionShop(mainPageId, storeNameToUse) {  
  
    mainPage = document.getElementById(mainPageId);304  
  
    storeName = storeNameToUse;305  
  
    loadDataStore();306
```

---

<sup>304</sup> Set `mainPage` to refer to the page container

<sup>305</sup> Set the name of the local storage string

<sup>306</sup> Load the stock data

```
    doShowMainMenu();307  
}
```

The last statement in `doStartFashionShop` displays the user menu. Lets take a look at how that works.

## Create user menus

The user interacts with our program by pressing buttons on the screen that select the various menu options. If you look back to Figure 10.4 you can see the main menu for the display. Each of the program options is selected by pressing a button, and when the button is pressed the application calls the function for that option.

```
function doShowMainMenu() {  
    openPage("Main Menu");  
  
    showMenu(  
        [{ desc: "Add Dress", label: "Dress", func: "doAddDress()" },  
         { desc: "Add Pants", label: "Pants", func: "doAddPants()" },  
         { desc: "Add Skirt", label: "Skirt", func: "doAddSkirt()" },  
         { desc: "Add Top", label: "Top", func: "doAddTop()" },  
         { desc: "Update stock item", label: "Update", func: "doUpdateStock()" },  
         { desc: "List stock items", label: "List", func: "doListFashionShop()" }]);  
}
```

I'm using yet another schema to describe each menu option. The function `showMenu` works through the schema and builds the display. If you look back to Figure 10.4 you can map the application main menu onto the items in the schema above. The function `openPage` removes all the elements on the page and displays a heading.

## Add a stock item

The menu calls an add function for each stock item. The `doAddDress` function looks like this:

```
function doAddDress() {  
    addStock(Dress);  
}
```

---

<sup>307</sup> Show the main menu

It calls the `addStock` function and does something we've not seen before. It uses the `Dress` class as an argument to the call of `addStock`. We do this so that we can have a single `addStock` function that can create any type of stock item.

```
function addStock(StockClass) {  
  
    activeItem = new StockClass();308  
  
    openPage("Add " + activeItem.type);309  
  
    activeItem.getHTML(mainPage);310  
  
    showMenu(  
        [{ desc: "Save item", label: "Save", func: "doSaveAdd()", },  
         { desc: "Cancel add", label: "Cancel", func: "doShowMainMenu()" }]);311  
}
```

The `addStock` function makes a new item of the required class (the class is supplied as a parameter). It then creates a new display page and fills it with the HTML generated by the new item. At the end of the page the function builds a menu containing **Save** and **Cancel** buttons. The function handler for the **Cancel** button just displays the main menu. The function for the **Save** button copies the inputs from the HTML elements into a new copy of the stock item. This function has the name `doSaveAdd`. The job of `doSaveAdd` is to copy the data from the HTML document into the currently active item. This item is then stored in the data store.

```
function doSaveAdd() {  
  
    activeItem.loadFromHTML();312  
    activeItem.stockRef = StockItem.getLargestStockRef(dataStore) + 1;313  
    dataStore[dataStore.length] = activeItem;314  
    alert(activeItem.type + " " + activeItem.stockRef + " added");  
}
```

---

<sup>308</sup> Create a new item

<sup>309</sup> Display a new page

<sup>310</sup> Get the HTML for the new item

<sup>311</sup> Show a menu for the add page

<sup>312</sup> Load the data from the HTML into the new item

<sup>313</sup> Assign a stock number

<sup>314</sup> Store the item

```
    saveDataStore();315  
    doShowMainMenu();316  
}
```

## Exploring the Fashion Shop application

There is a lot to explore in the Fashion Shop and you can learn a lot by exploring it. The application is heavily based on the Time Tracker application. I would strongly advise you to spend some time going through the code. You can use the Developer View debugger to work through the code as it runs. The great thing about the application is that it is very clear what the intent of each function is. For example the edit function, which we have not explored in this chapter, must find a stock item to be edited, make that item the active item and then, when the edit is completed, copy the edited properties from the HTML document into the data store.

Note that we have not explored the function that provides a list of stock items. You can use it, and you can look at the code that makes it work, but we will be investigating that function and adding some great features to it in the next chapter.

### MAKE SOMETHING HAPPEN

#### Expand the fashion shop

You can also learn a lot about programming by adding features to an existing application. Here are some things that you might like to do with the Fashion Shop application:

- Add a new type of clothing called `suit`. A suit has the properties, jacket size, pant size, color, pattern and style. You can do this by adding a new class which is a sub class of `StockItem`.
- Add a new property called `manufacturer` which is to be stored for all the items in stock. You can do this by adding a new attribute to the `StockItem` class.
- Add some data validation to the application. At the moment the user can save stock item records that have missing data fields. Write a function that tests for empty fields and only allows a record to be saved if all the fields have been filled in.
- Create a totally new data storage application which can store information. You should find this quite easy to do. You can use the Fashion Shop application as a great starting point.

---

<sup>315</sup> Save the data store

<sup>316</sup> Display the main menu

# What you have learned

In this chapter, you learned how JavaScript objects allow programs to store related items as properties of a single object and how to make programs that work with objects.

- JavaScript programs can create and *throw* exception objects that describe an error that has been detected by the code. JavaScript statements that may throw exception objects can be enclosed in a try block as part of a try – catch construction. The catch element of the construction contains JavaScript code that only runs in the event of an exception being thrown. This construction allows a program to detect and deal with errors in a managed way.
- A program should only raise an exception when something exceptional has occurred. Errors that are to be expected in the normal running of the program (for example invalid user input or network failures) should not be managed using exceptions.
- A JavaScript class lets a programmer design the contents of an object by specifying data to be used in the class constructor method to set initial values of properties in the class.
- A JavaScript class constructor method accepts parameters that can be used to initialize properties in a newly created class instance.
- The JavaScript `new` keyword is used to create a new instance of a class by calling a constructor method to that class. Because missing method arguments to a JavaScript method call are replaced by the value `undefined` you can use a constructor call with no parameters to create an "empty" class instance that contains undefined values for all the properties.
- A JavaScript class can contain methods which are members of the class. A member method can be called by code outside the class to allow an object to provide behaviors for that code.
- Within a class method the reference `this` refers to the object within which the method is running.
- JavaScript inheritance allows the creation of "super" or "parent" classes which can be extended to create "sub" or "child" classes. A sub class contains all the members of the parent. This allows attributes shared by a number of related classes to be stored in a single super class.
- A sub class can *override* methods in the super class by providing their own implementations of the method. The keyword `super` allows a method in a sub class to call the super class. The constructor of a sub class must contain a constructor method that makes use of the `super` mechanism to initialize the properties of the super class.
- A class can contain *static* data and method members which are stored as part of the class rather than being part of any instance of a class. A static method is a way that class can provide a behavior or data property that can be used without the need to create an instance of the enclosing class.

Here are some questions that you might like to ponder about what we have learned in this chapter:

**Question:** What happens if a JavaScript program doesn't catch an exception that has been thrown?

If an exception is thrown in some JavaScript that is not part of a try-catch construction the exception will be caught by the browser and the program will end. If you have the Developer View open you will see the exception displayed in the form of a red error message.

**Question:** When should a JavaScript program throw an exception?

Exceptions should only be used in exceptional circumstances. Some JavaScript functions, for example `JSON.parse`, use exceptions to signal error conditions. You can also make your code generate exceptions. When you start work on a project you need to decide on all the possible error conditions and then decide how each should be handled. Exceptions are useful because they provide a way that a low level failure can be quickly propagated to a higher level error handler.

**Question:** Must my JavaScript programs catch all exceptions?

No. For me the biggest concern when I write a program is not that the program might fail. It is that the user might think that it has worked when it has not. If an application fails with an obvious error the user will get upset. If an application "pretends" that it has worked and the user later finds out that they have lost all their data they will get very upset. You should ensure that exceptions are logged and reported in a way that makes their error reports useful.

**Question:** Do I have to use classes in my programs?

No. However, they can make some kinds of programs (particularly those that need to deal with different types of related data) easier to write.

**Question:** Can a JavaScript class have multiple constructors?

No. Some programming languages have a mechanism called "overloading" where a class can contain multiple versions of a method that all share the same name but have different parameters. JavaScript does not support overloading so a class can only contain a single constructor method. If you want to provide different ways to construct an object you must write code in the constructor method to decide what the parameters to the function mean.

**Question:** What is the difference between a method and a function?

A method is declared as part of a class, whereas a function is declared outside any class. Both can accept parameters and return a result.

**Question:** Can the `this` reference be used as an argument to a function call?

Yes. Within a method the `this` reference refers to the object which the method is running within. If an object wants to send another object a reference to itself it can pass the value of `this` as a function argument. This is like me calling you on the phone and telling you my phone number.

**Question:** Can I use the `this` reference inside a static method?

No. A static method is a member of the enclosing class and is not associated with an existing instance.