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/jspodcast

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

www.begintocodewithjavascript.com

Feel free to send constructive comments to 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 is 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 you 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 go 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.

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

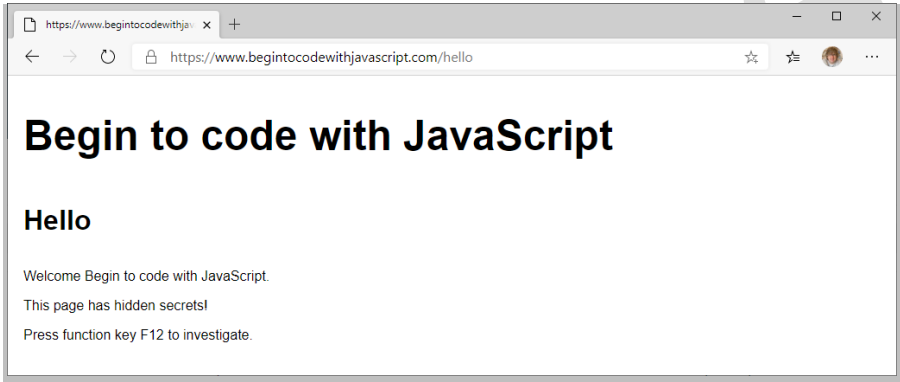
site on the World Wide Web. But then, with a single press of a key, you're going to explore a world behind the web page and get a glimpse of the role that JavaScript plays in making it work.

Make Something Happen

A web page with secrets

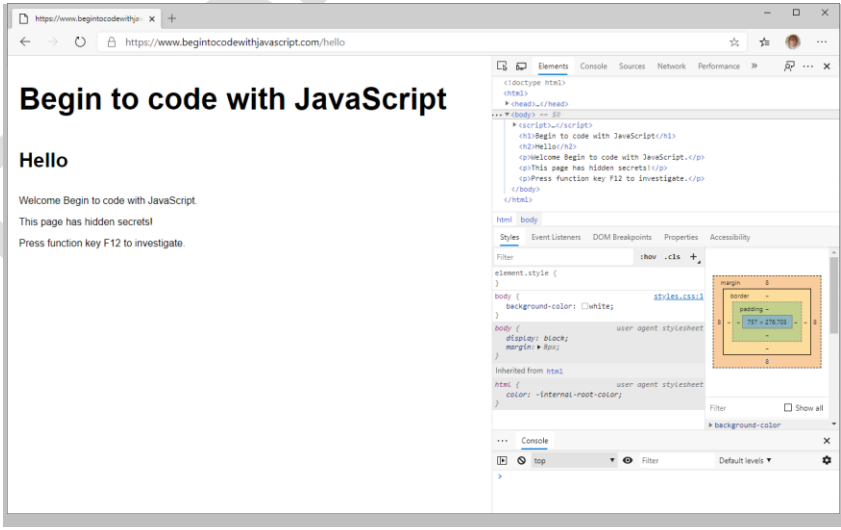
First you need to open your browser. Then visit the web page:

http://www.begintocodewithjavascript.com/hello



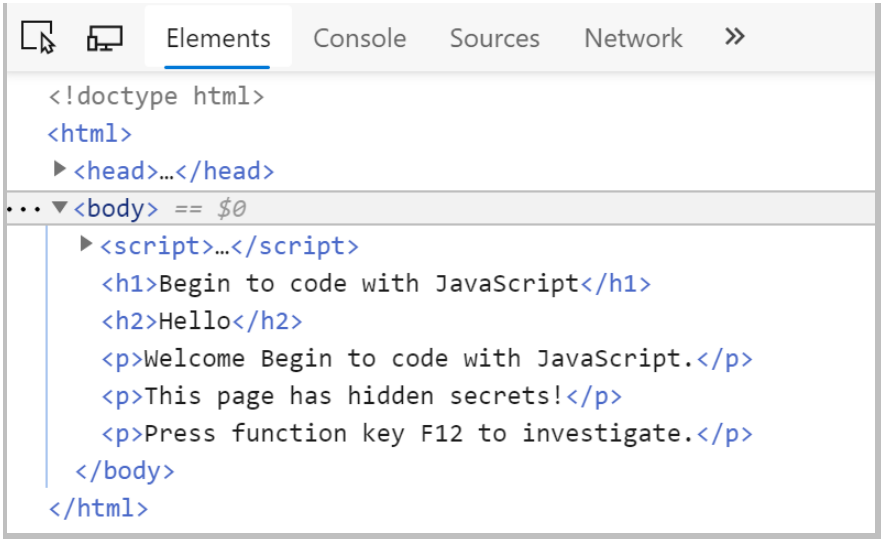
1.1 Ch01_inset01_01 A web page with secrets

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.



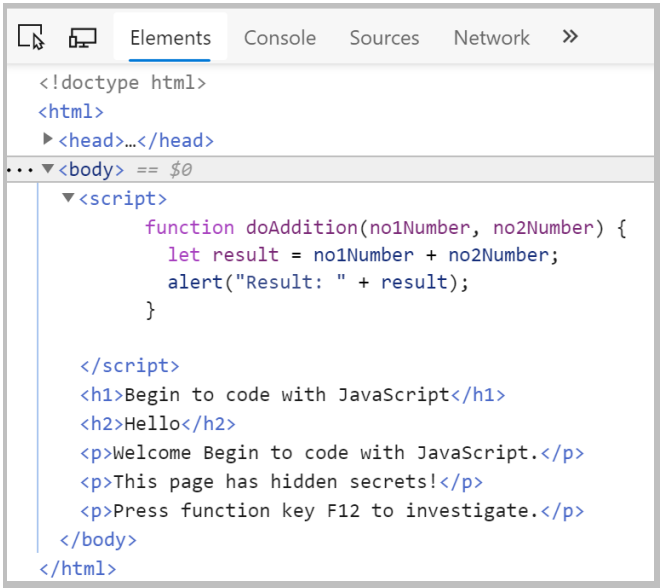
1.2 Ch01_inset01_02 Developer View

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.



1.3 Ch01_inset01_03 Page elements

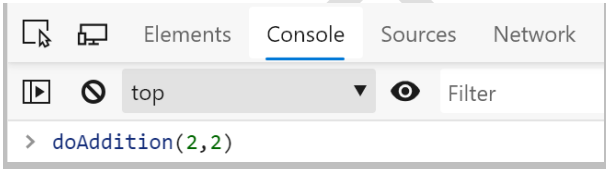
The Figure above shows the elements on this page. You can see that the text that appears on the page is here. Parts of the text are enclosed in what look like formatting instructions, for example some is marked as <h1> and some as <p>. If you look back at the web page as displayed you will notice that 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.



1.4 Ch01_inset01_04 JavaScript function

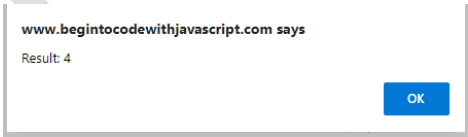
Clicking the arrow opens that part of the listing. The hidden feature is a function called `doAddition`. This takes two numbers, adds them together and displays the result using an alert. Later in the text we will go into detail of how this JavaScript works, but even at this stage it is quite clear what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



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 to 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
```

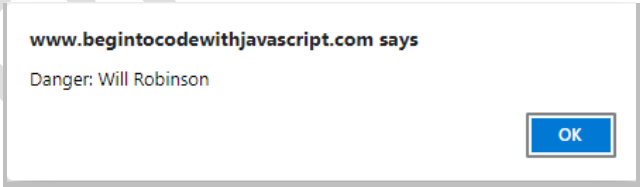
This calculation involves real numbers (i.e. ones with a fractional part). The calculation is adding 0.2 to 0.1 (a fifth to a tenth). This should produce the result 0.3 but what we get is interesting:

```
<-  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 `doAddtion` function displays the result it has calculated. Finally, lets 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.

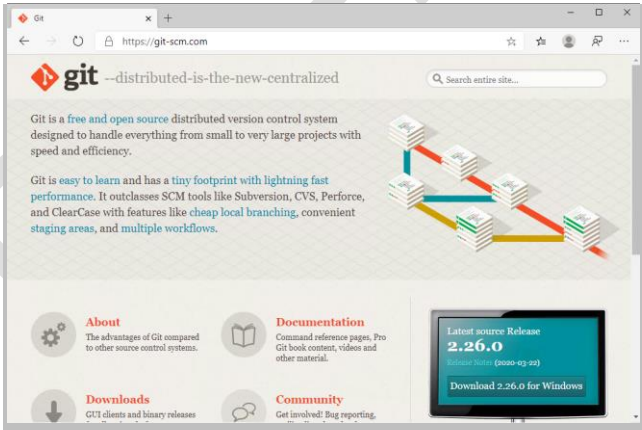
Git was created in 2005 by Linus Torvalds who was writing the Linux operating system at the time. He needed at tool that could track what he was doing and make it easy for him to work with other people. So he created his own. Git is a professional tool and very powerful. It lets large numbers of developers work together on a single project. Different teams can work on their own versions of the code which can them 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

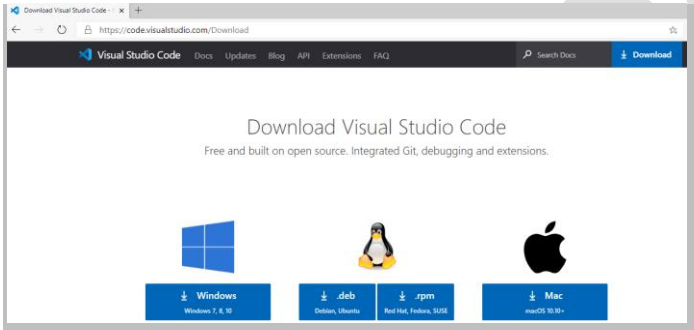
If you want to write a letter you would use a Word processor. To perform calculations, you might use a spreadsheet. Visual Studio Code is a tool that you can use to edit your program files. It can do a lot more than this, as we shall see later. But for now, we are going to use it as a super powerful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

Make Something Happen

Install Visual Studio Code

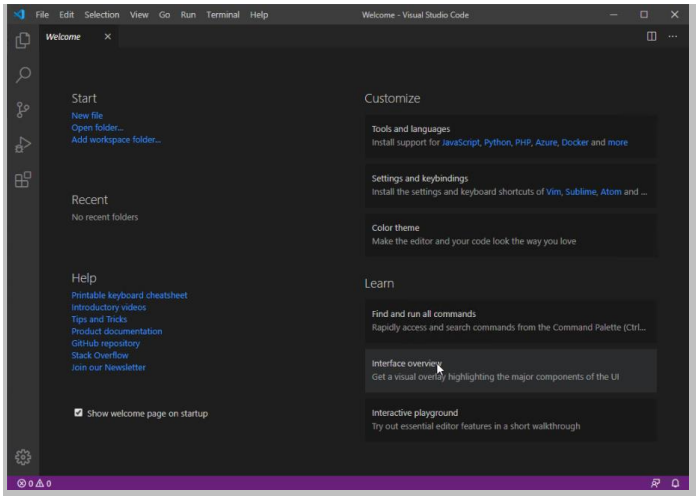
I'm going to give you instructions for Windows 10. The instructions for macOS are very similar. First you need to open your browser and visit the web page:

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.

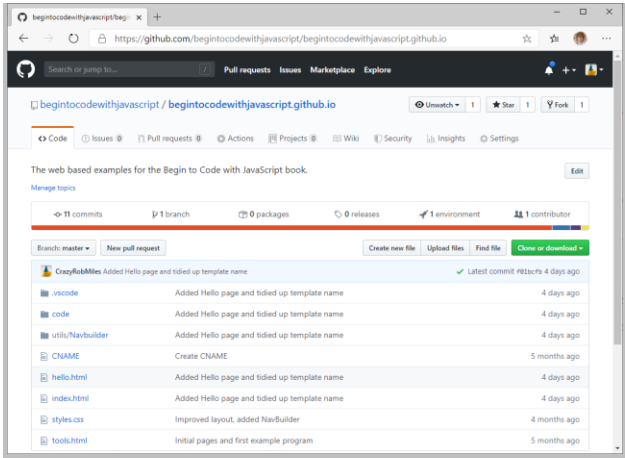
Make Something Happen

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):

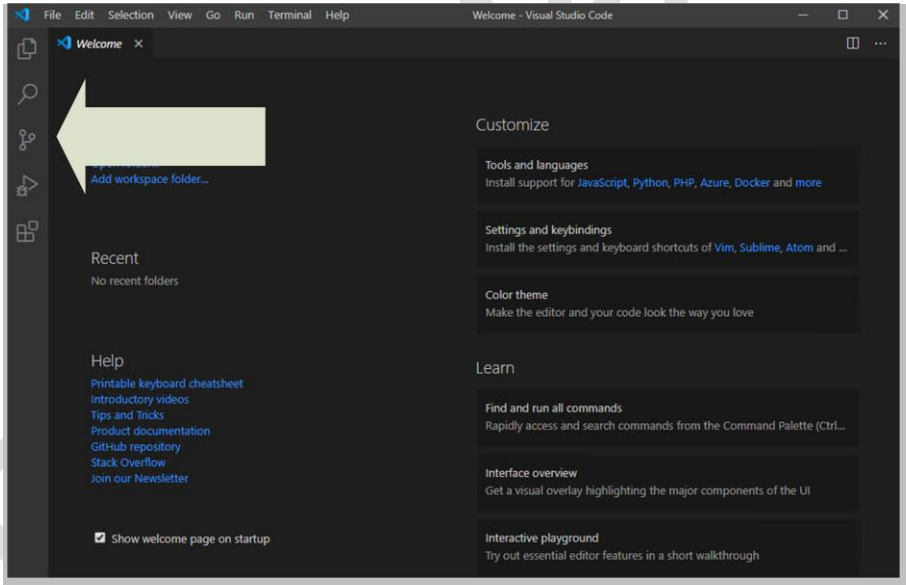
https://github.com/begintocodewithjavascript/begintocodewithjavascript.github.io

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.



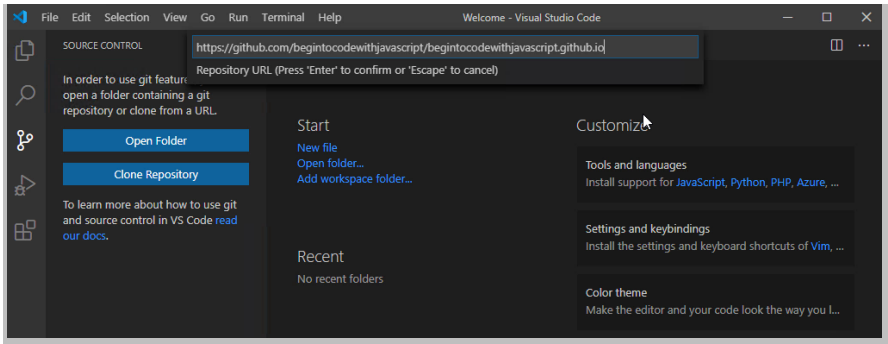
4.1 Ch01_inset04_01 Repository home page

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.

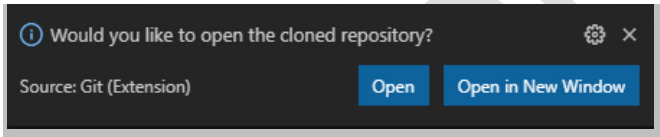


4.3 Ch01_inset04_03 Visual Studio Code Source Clone Repository

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/begintocodewithjavascript/begintocodewithjavascript.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 you're 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.

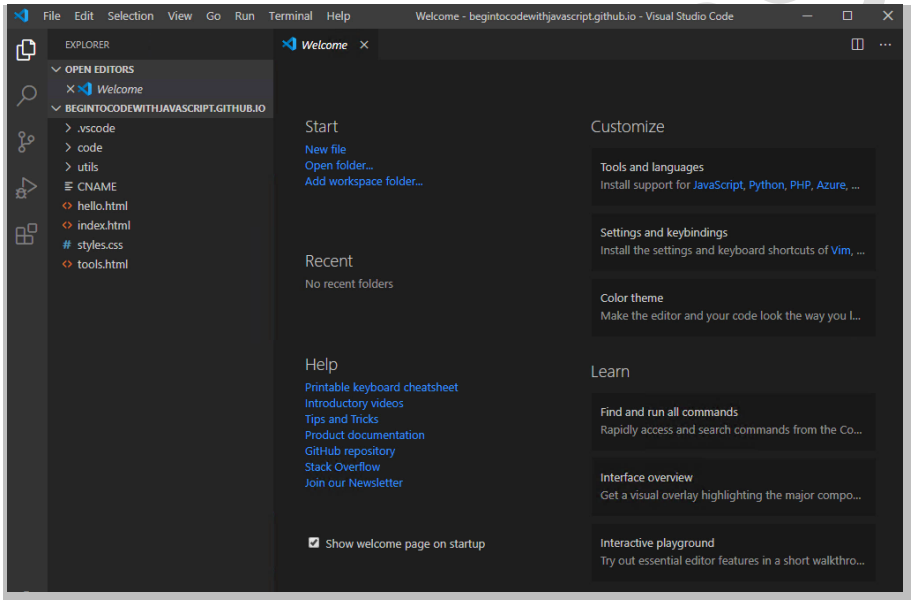
3. Use a web browser to view the HTML file and see what it does.

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

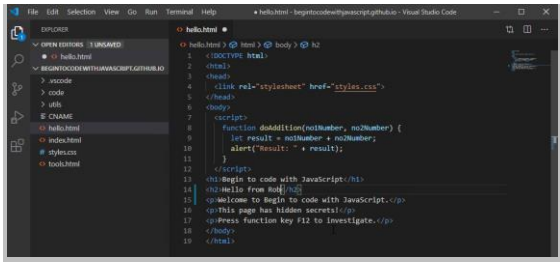
Edit the secret program

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

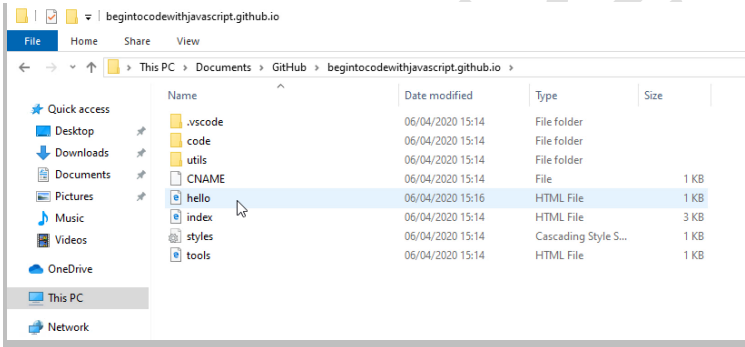
The Explorer window at the left hand side of the Visual Studio Code window provides a view of all the files and folders in the repository. You can click the ">" in front of folders in the Explorer view to open them and view their contents. For now, you are just going to look in the hello.html file, so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



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 opens 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 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 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> 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 emphsized **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: &amp;
```

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 and 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 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 “I” 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>
|  _ \ ___ | |__   |  \/  (_) | ___  ___
| |_) / _ \| '_ \  | |\/| | | |/ _ \/ __|
|  _ &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 (<) 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:

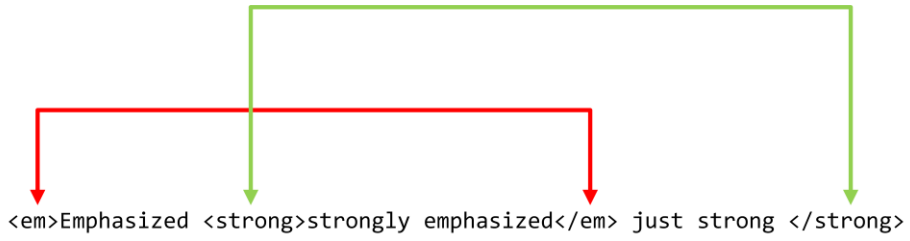
```
<em>Emphasized <strong>strongly emphazised</em> just strong </strong>
```

The browser will display what you would expect:

Emphasized ***strongly emphasized*** **just strong**

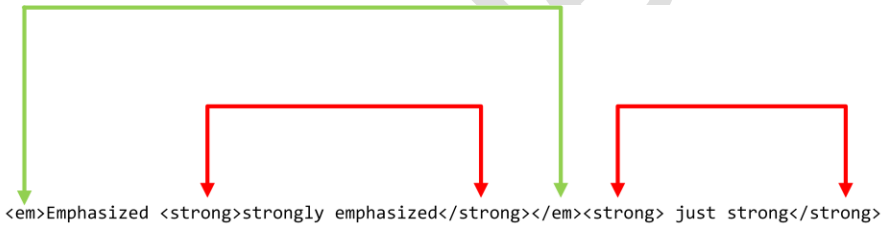
Ch02_Readeraid_01_Fig_01 emphasized text

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 tag "ends" inside the 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 Andreeson, 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:

```
<img src="seaside.JPG">
```

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.

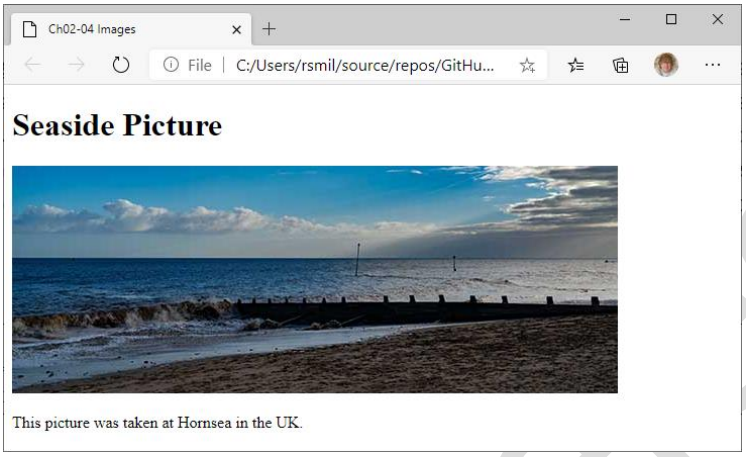
```
<img src="seaside1.JPG" alt="Maybe Rob got the filename wrong">
```

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><img src="seaside.JPG"></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><img src="seaside.JPG" width=400>
```

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 a 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, 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.

---

[1] Indicates that this is an HTML document

[2] HTML tab with a language attribute

[3] Start of the heading of the web page

[4] End of the heading

[5] Start of the body text of the web page

[6] End of the body text

[7] End of the HTML text

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 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.

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");
}
```

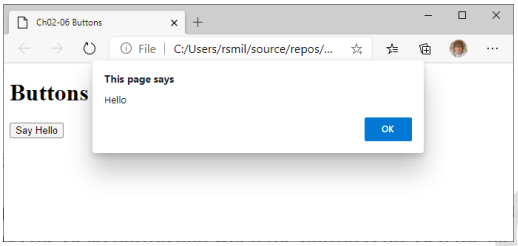
This function only performs a single action, it displays an alert that says "Hello" to the user when it is called. The line of JavaScript that displays the alert is called a *statement*. The end of a statement is marked by a semi-colon character. A function can contain many statements, each of which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

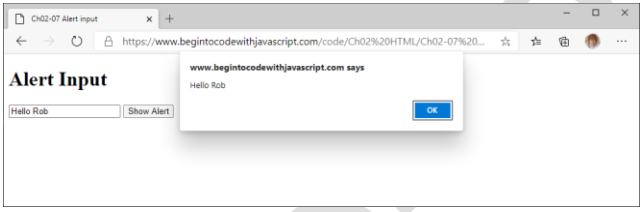
The `type` attribute tells the browser the type of input that is being read. In the code above we are reading text, so the attribute type is set to `text`. If you set the type attribute to `password` the contents of the input are hidden as they are typed. This is how JavaScript programs read passwords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

[8] Input text element

```
    <button onclick="doUpdateParagraph()">Update the Paragraph</button>[9]
    <p id="outputParagraph"></p>[10]
  </p>

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

---

[9] Button that calls doUpdateParagraph

[10] Paragraph for the output

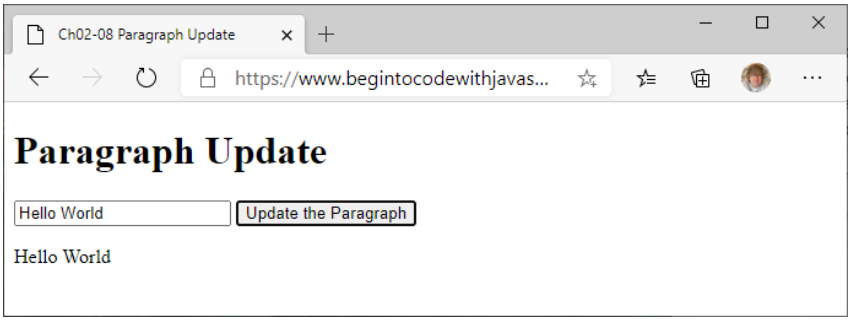
[11] Function that updates the paragraph

[12] Gets a reference to the input

[13] Gets a reference to the output

[14] Reads the text from the input

[15] Writes the text into the output



11. Ch02_fig11 Paragraph Update

MAKE SOMETHING HAPPEN

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 `setTimout` 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

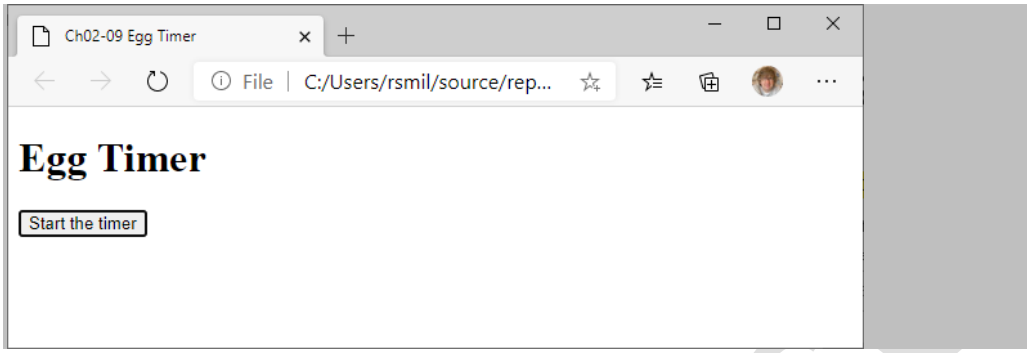
The `doStartTimer` function is connected to a button so that the user can start the timer. The `doEndTimer` will be called when the timer completes. I've added a calculation that works out the delay value. I want a five minute delay. There are 60 seconds in a minute and a value of 1000 would give me 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

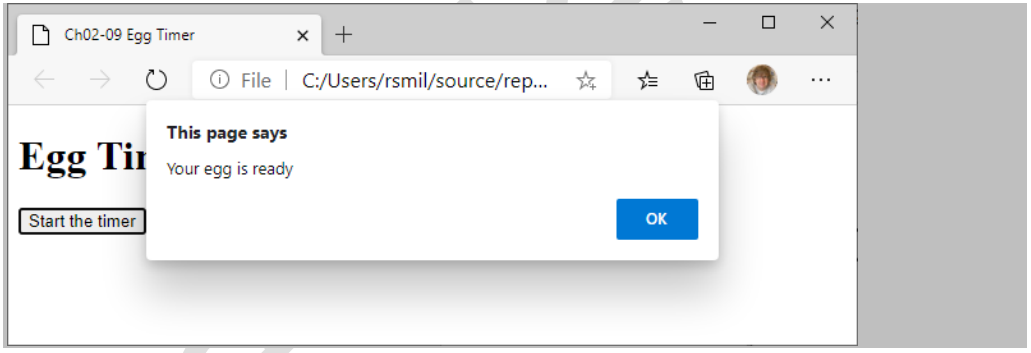
```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Ch02-09 Egg Timer</title>
</head>

<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_03_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 run 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:

```
    <img src="seaside1.JPG" alt="The image could not be found" id="pageImage"></p>
```

This `img` element displays the picture in the file seaside.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 them 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) 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 HMTL 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 rather than <i>?

The <i> (italic) tag means “use italic text”. The 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 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 and an end element?

No. Lots of tags do, for example marks the start of a paragraph and marks the end. But some, for example
 (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 <img src="seaside1.JPG"> 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 abut 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

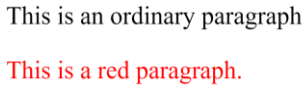
on a web page covers such things as the foreground and background color it has, the type of character design (font), size of the text and things like margins. You can think of a style as a lump of data that you want to apply to something. We are going to start by styling some text. Once we've worked 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.

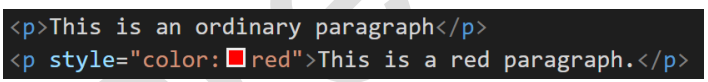
```
<p>This is an ordinary paragraph</p>
<p style="color:red">This is a red paragraph.</p>
```

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.



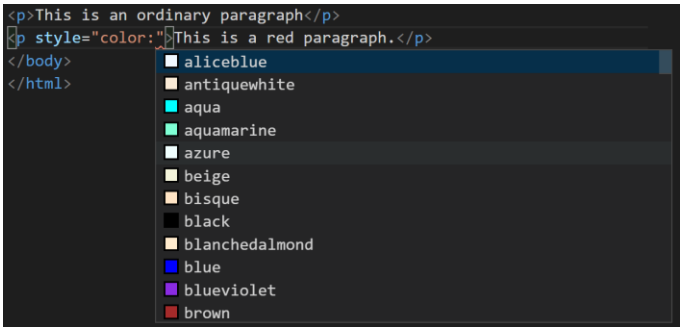
13. Figure 3.1 Styling text

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.



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

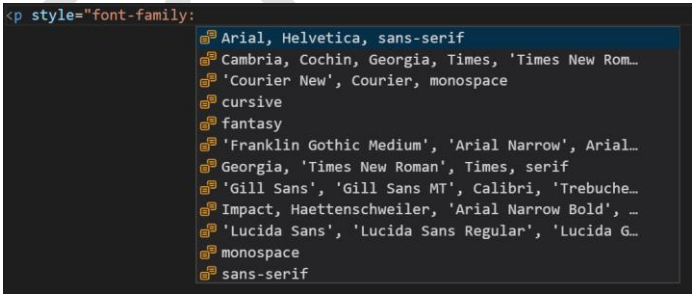
Fonts are one of many things about computers where computer manufacturers have "agreed to differ". They all agree that they need a font that looks like this (this is called a *serif* font because it has rounded corners) and a font that looks like this (this is called a *sans-serif* font because it doesn't 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.**

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.

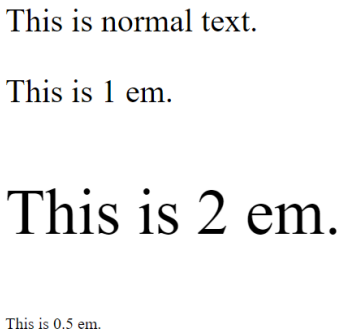
If you want to specify the size of text in a web pages there are a number of units you can use. You can express the size in inches, cm, pixels, points or as a percentage of the size of the display. But I would advise you to use the unit

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**



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="ΔdoStartClock()"[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);[19]
        }


    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

---

Δ

[16] *Calls doStartClock when the page is loaded.*

[17] *This is the paragraph that contains the clock display*

[18] *Function that starts the clock*

[19] *Call doClockTick every second*

[20] *Update the clock with the current time*

something, but you may be confused about exactly what is going on. In the case of `var`, a program is creating new variable:

```
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 `getElementByID` 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?

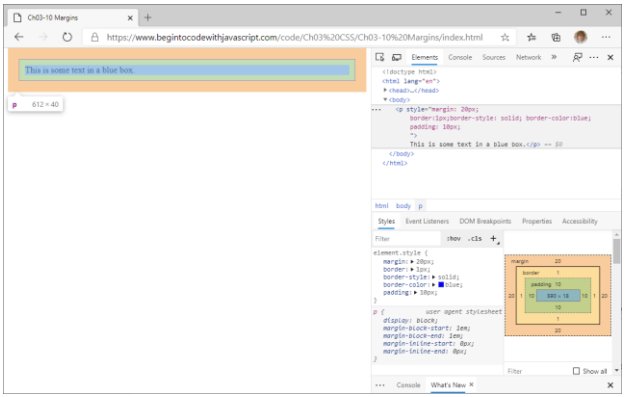
The solution is simple. The program must call `doClockTick` from the `doStartClock` function. Since the `doStartClock` method is called when the page is loaded, this will cause the display to be updated at the start. You can find my version of this in the folder **Ch03 HTML/Ch03-09 Clock Quick Start**.

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*

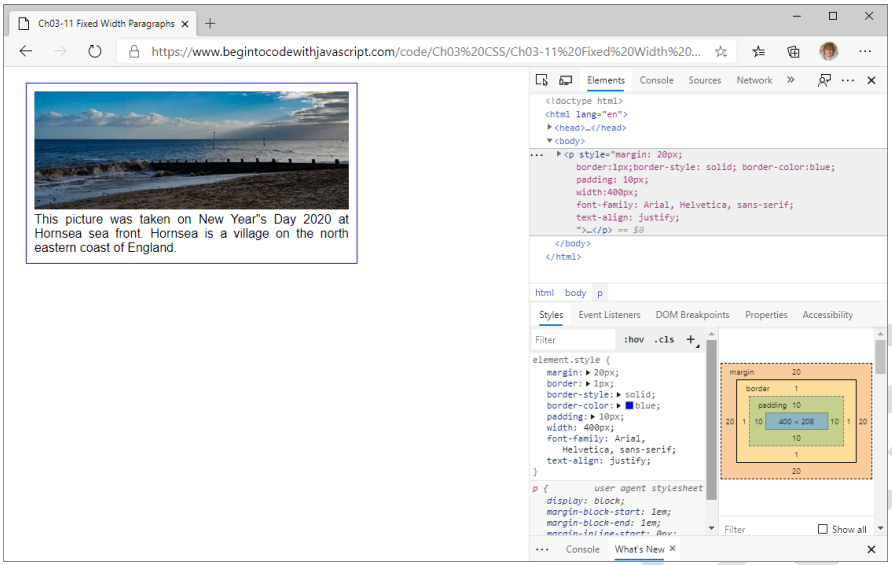


21. Figure 3.9 Margin Display

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;
    ">
    <img src "seaside.JPG" alt="The image could not be found" width="400">
    This picture was taken on New Year&amp;quot;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;
```

[21] *Style element*

[22] *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:

[23] 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:

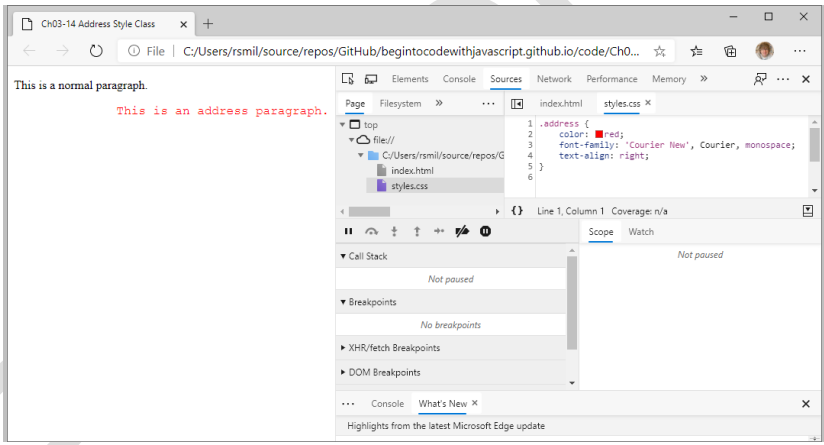
```
<p class="address"> This is an address paragraph.</p>
```

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 and 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 1OS</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

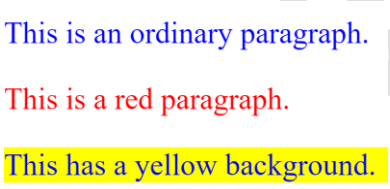
sure that any “nesting” is properly formatted. Take a look at the Programmer’s point “Don’t abuse the browser” in Chapter 2 for more details on this issue.

“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.



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**

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 JavaScrpt 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.

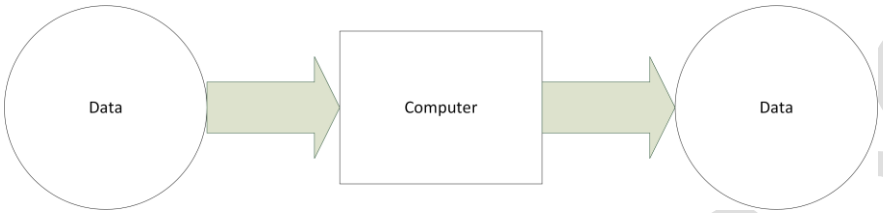


Figure 4.2 A computer as a data processor

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

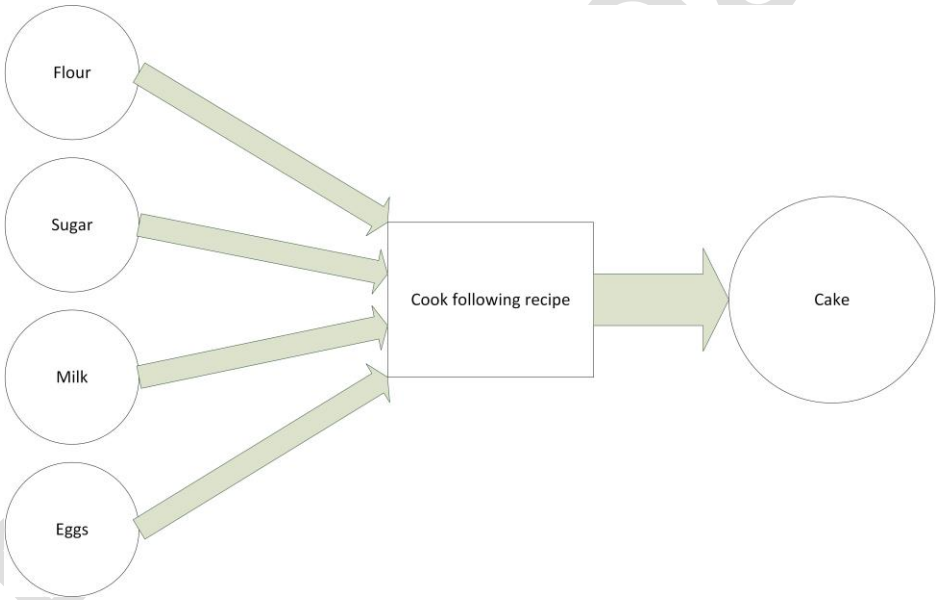


Figure 4.3 Recipes and programs

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

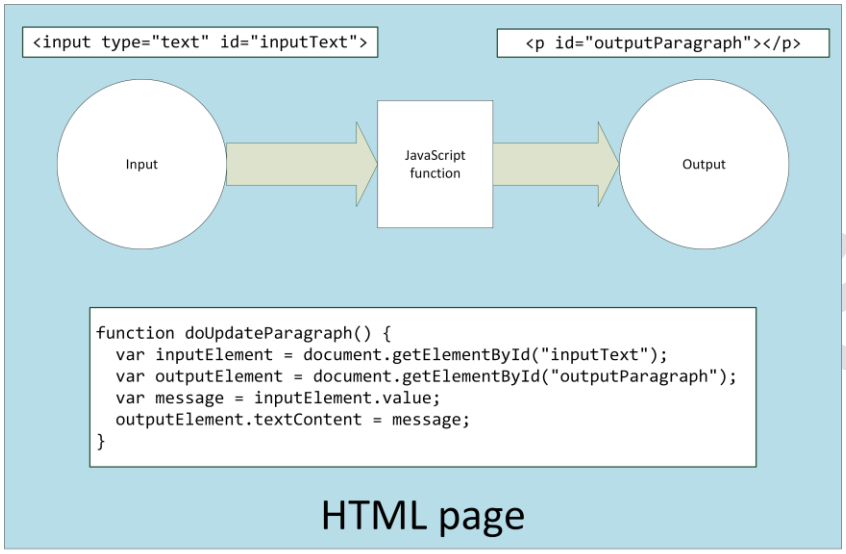


Figure 4.4 JavaScript as a data processor

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

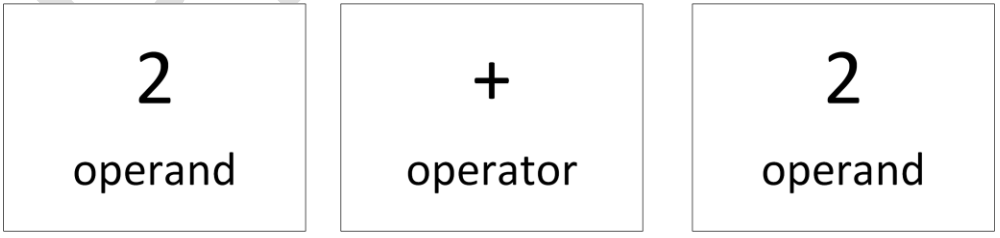


Figure 4.5 Ch04_Fig_05 Expression

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.

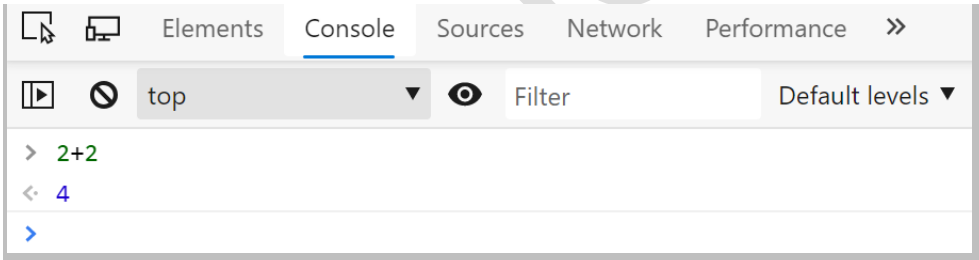
Code Analysis

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.



Ch04_inset_01_01

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 × (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.

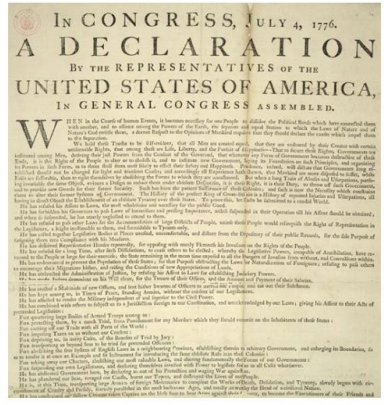
Data and information

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 2.6, which shows the difference.

What the Computer Sees

87 104 101 110 32 105 110 32 116 104 101
32 67 111 117 114 115 101 32 111 102 32
104 117 109 97 110 32 101 118 101 110 116
115 44 32 105 116 32 98 101 99 111 109 101
115 32 110 101 99 101 115 115 97 114 121
32 102 111 114 32 111 110 101 32 112 101
111 112 108 101 32 116 111 32 100 105 115
115 111 108 118 101 32 116 104 101 32 112
111 108 105 116 105 99 97 108 32 98 97 110
100 115 32 119 104 105 99 104 32 104 97
118 101 32 99 111 110 110 101 99 116 101
100 32 116 104 101 109 32 119 105 116 104
32 97 110 111 116 104 101 114 44 32 97 110
100 32 116 111 32 97 115 115 117 109 101
32 97 109 111 110 103 32 116 104 101 32
112 111 119 101 114 115 32 111 102 32 116
104 101 32 115 101 112 97 114 97 116 101
32 97 110 100 32 101 113 117 97 108 32 115
116 97 116 105 111 110 32 116 111 32 119
104 105 99 104 32 116 104 101 32 76 97 119
115 32 111 102 32 78 97 116 117 114 101 32
97 110 100 32 111 102 32 78 97 116 117 114
101 39 115 32 71 111 100 32 101 110 116
105 116 108 101 32 116 104 101 109 44 32
97 32 100 101 99 101 110 116 32 114 101
115 112 101 99 116 32 116 111 32 116 104
101 32 111 112 105 110 105 111 110 115 32
111 102 32 109 97 110 107 105 110 100 32
114 101 113 117 105 114 101 115 32 116 104
97 116 32 116 104 101 121 32 115 104 111
117 108 100 32 100 101 99 108 97 114 101
32 116 104 101 32 99 97 117 115 101 115 32
119 104 105 99 104 32 105 109 112 101 108
32 116 104 101 109 32 116 111 32 116 104
101 32 115 101 112 97 114 97 116 105 111
110 46

What We See



IN CONGRESS, JULY 4, 1776.

A DECLARATION

BY THE REPRESENTATIVES OF THE

UNITED STATES OF AMERICA,

IN GENERAL CONGRESS ASSEMBLED.

WHEN in the Course of human Events, it becomes necessary for one People to dissolve the Political Bands which have connected them with another, and to assume among the Powers of the Earth, the separate and equal Station to which the Laws of Nature and of Nature's God entitle them, a decent Respect to the Opinions of Mankind requires that they should declare the causes which impel them to the Separation.

We hold these Truths to be self-evident, that all Men are created equal, that they are endowed by their Creator with certain unalienable Rights, that among these are Life, Liberty and the Pursuit of Happiness.—That to secure these Rights, Governments are instituted among Men, deriving their just Powers from the Consent of the Governed, —That whenever any Form of Government becomes destructive of these Ends, it is the Right of the People to alter or to abolish it, and to institute new Government, laying its Foundation on such Principles and organizing its Powers in such Form, as to them shall seem most likely to effect their Safety and Happiness. Prudence, indeed, will dictate that Governments long established should not be changed for light and transient Causes; and accordingly all Experience hath shewn, that Mankind are more disposed to suffer, while Evils are sufferable, than to right themselves by abolishing the Forms to which they are accustomed. But when a long Train of Abuses and Usurpations, pursuing invariably the same Object, evinces a Design to reduce them under absolute Despotism, it is their Right, it is their Duty, to throw off such Government, and to provide new Guards for their future Security.—Such has been the patient Sufferance of these Colonies; and such is now the Necessity which constrains them to alter their former Systems of Government. The History of the present King of Great-Britain is a History of repeated Injuries and Usurpations, all having in direct Object the Establishment of an absolute Tyranny over these States. To prove this, let Facts be submitted to a candid World.

He has refused his Assent to Laws, the most wholesome and necessary for the public Good.

He has forbidden his Governors to pass Laws of immediate and pressing Importance, unless suspended in their Operation till his Assent should be obtained; and when so suspended, he has utterly neglected to attend to them.

He has refused to pass other Laws for the accommodation of large Districts of People, unless those People would relinquish the Right of Representation in the Legislature, a Right inestimable to them, and formidable to Tyrants only.

He has called together Legislative Bodies at Places unusual, uncomfortable, and distant from the Depository of their public Records, for the sole Purpose of fatiguing them into Compliance with his Measures.

He has dissolved Representative Houses repeatedly, for opposing with manly Firmness his Invasions on the Rights of the People.

He has refused for a long Time, after such Dissolutions, to cause others to be elected; whereby the Legislative Powers, incapable of Annihilation, have returned to the People at large for their exercise; the State remaining in the mean time exposed to all the Dangers of Invasion from without, and Convulsions within.

He has endeavored to prevent the Population of these States; for that Purpose obstructing the Laws for Naturalization of Foreigners; refusing to pass others to encourage their Migrations hither, and raising the Conditions of new Appropriations of Lands.

He has obstructed the Administration of Justice, by refusing his Assent to Laws for establishing Judiciary Powers.

He has made Judges dependent on his Will alone, for the Tenure of their Offices, and the Amount and Payment of their Salaries.

He has erected a Multitude of new Offices, and sent hither Swarms of Officers to harass our People, and eat out their Substance.

He has kept among us, in Times of Peace, Standing Armies, without the Consent of our Legislatures.

He has affected to render the Military independent of and superior to the Civil Power.

He has combined with others to subject us to a Jurisdiction foreign to our Constitution, and unacknowledged by our Laws; giving his Assent to their Acts of pretended Legislation:

For quartering large Bodies of Armed Troops among us:

For protecting them, by a mock Trial, from Punishment for any Murders which they should commit on the Inhabitants of these States:

For cutting off our Trade with all Parts of the World:

For imposing Taxes on us without our Consent:

For depriving us, in many Cases, of the Benefits of Trial by Jury:

For transporting us beyond Seas to be tried for pretended Offences:

For abolishing the free System of English Laws in a neighbouring Province, establishing therein an arbitrary Government, and enlarging its Boundaries, so as to render it at once an Example and fit Instrument for introducing the same absolute Rule into these Colonies:

For taking away our Charters, abolishing our most valuable Laws, and altering fundamentally the Forms of our Governments:

For suspending our own Legislatures, and declaring themselves invested with Power to legislate for us in all Cases whatsoever.

He has abdicated Government here, by declaring us out of his Protection and waging War against us.

He has plundered our Seas, ravaged our Coasts, burnt our Towns, and destroyed the Lives of our People.

He is at this Time transporting large Armies of foreign Mercenaries to compleat the Works of Death, Desolation, and Tyranny, already begun with circumstances of Cruelty and Perfidy, scarcely paralleled in the most barbarous Ages, and totally unworthy the Head of a civilized Nation.

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. V*ariables* 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:

```
var[24] total[25];
```

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

[24] Tells JavaScript that the program is creating a variable.

[25] The name of the variable.

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

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, `windowLengthInInches`. 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 `0.10000000000000000551115` 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.

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)
```

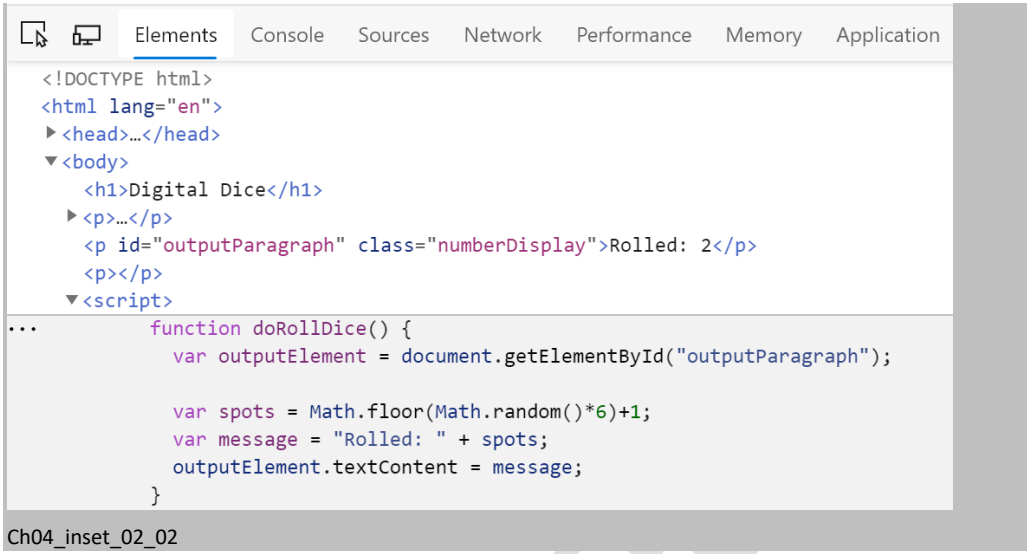
The value 1.9999 is very close to 2, but the floor function throws away the entire fractional part. We can apply the floor function to our random value. Try this.

```
> 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 the 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.



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 produced 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
<- "3hello34"
```

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)
<- "3hello7"
```

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.

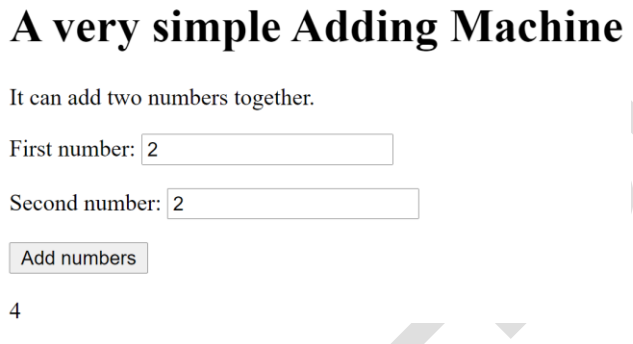


Figure 4.7 Adding machine

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 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>
  <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 `no2text`. 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.

[26] First number input

[27] Second number input

[28] Button to trigger the calculation

[29] Paragraph to display the answer.

```
var no1Element = document.getElementById("no1Text");[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 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:

---

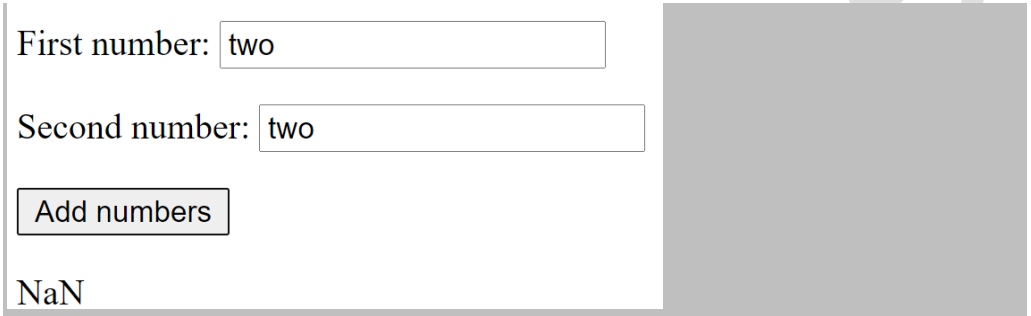
[30] *Get a reference to the element holding the user input*

[31] *Get the text from the user input element*

[32] *Convert the text into a 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:

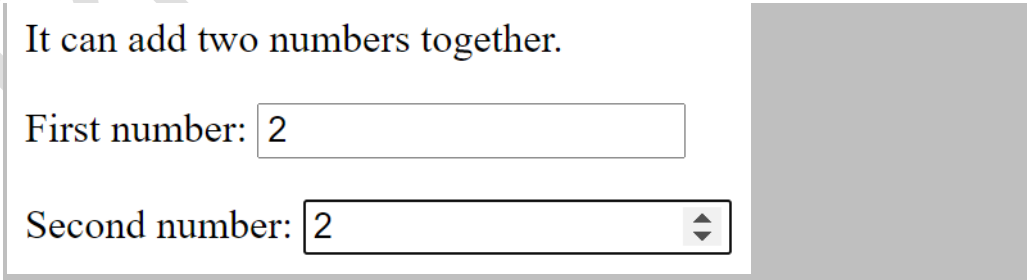


Ch04_inset_03_02

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.



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>&#x1f355; Pizza Calculater</h1>
  <p>Calculates the number of pizzas you&rsquo;ll need.</p>
  <p>
```

[33] *Using a stylesheet to add some style*

```
    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]

      var noOfPizzas = noOfStudents / 1.5;[39]

      var resultElement = document.getElementById("resultParagraph");[40]
      resultElement.innerText = "You need " + noOfPizzas + " pizzas.";[41]
    }
  </script>
</body>

</html>
```

---

[34] Input element for the number of students

[35] Button to press to display result

[36] Paragraph to display the result

[37] Function that calculates the result

[38] Get the number of students

[39] Perform the calculation

[40] Get the paragraph that displays the result

[41] Display the result

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:

Figure 4.9 Pizza Fractions

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 create 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 15.55555555 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 15.5555555 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.

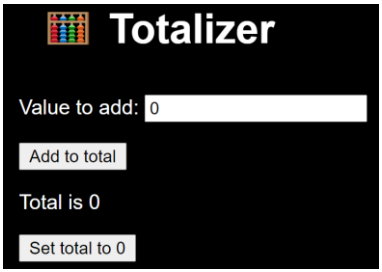


Figure 4.10 Totalizer

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 (🧮).

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(1) + " 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.

```
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
```

---

[42] Get a reference to the input element

[43] Get the text from the input element

[44] Convert the text into a numeric value

[45] Convert the value into Centigrade

[46] Get a reference to the output element

[47] Build the result string

[48] Put the result string on the output element

```
  // 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 in 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.innerText = "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.

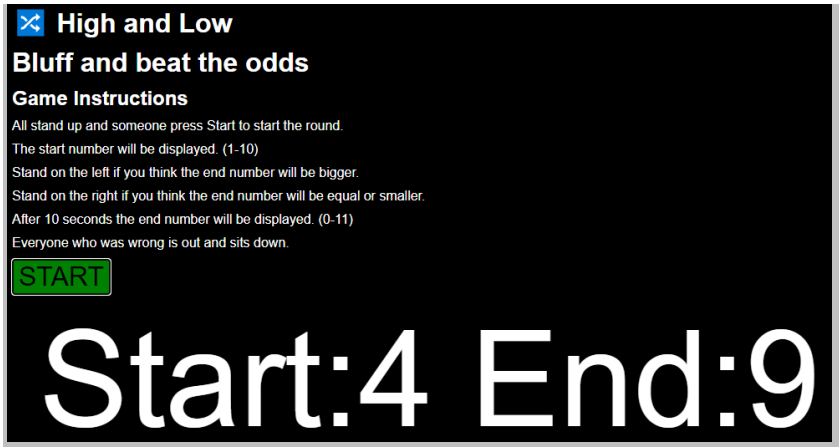
5. The program displays “Players stand.”

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 sanding 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

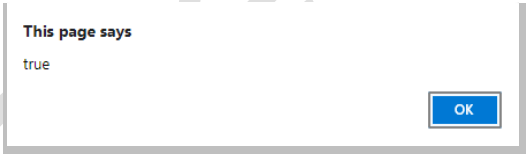
Boolean values

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 Bo olean 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".



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.

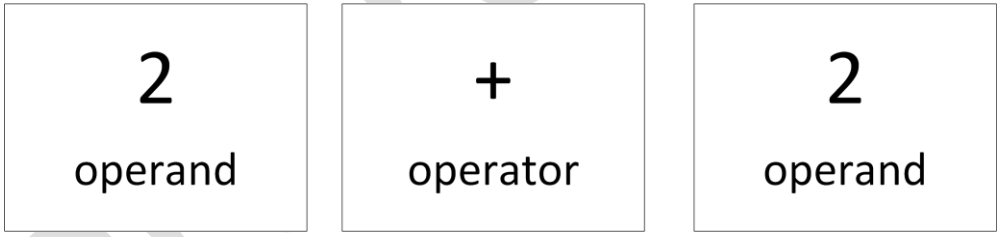


Figure 5.11 An arithmetic operator

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
>	Greater than	True if the value on the left is greater than the value on the right
<	Less than	True if the value on the left is less than the value on the right
>=	Greater than or equals	True if the value on the left is greater than or equal to the value on the right
<=	Less than or equals	True if the value on the left is less than or equal to the value on the right
==	Equals	True if the value on the left is equal to the value on the right
!=	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 devleoper 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 itIsTimeToGetUp= 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 itIsTimeToGetUp 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 itIsTimeToGetUp 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 itIsTimeToGetUp= (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.

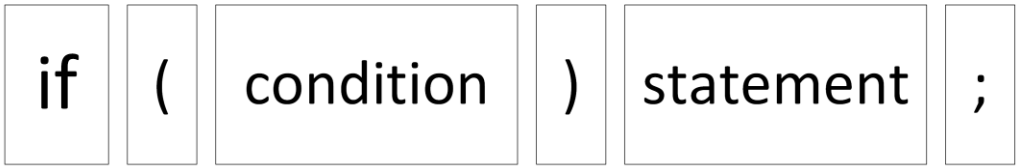


Figure 5.13 If construction

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>29);
```

```
            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. If 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.

[49] *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 what 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	Mininum 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:

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: 1

Enter your age: 18

Check 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>
```


[50] Include the CSS file

[51] Include the JavaScript file

[52] Start of numbered list of rides

```
</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 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, .menuNo[54]
{
    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-ceded 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

[53] This style adds shadows to the text

[54] These settings will be applied to all the classes.

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:

```
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 adding machine that we created earlier. It fetches text from the input elements in the HTML and uses the `Number` function 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 result. If you look at the HTML for the user interface you'll see that this paragraph has the ID `menuAnswerPar`:

---

[55] *Get the ride number that was entered*

[56] *Get the age of the person*

```
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 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==5) {
    // 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==5) {
    // 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 “Inifinity” 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.