Computer Programming

A Guide for the Perplexed

Tukolor

Computer Programming

A Guide for the P e r p l e x e d

by

Tukolor

Cover by Tukolor

Contents

1. Introduction

This is a very short guide to programming. What is programming? it asks and tries to answer.

Who is the guide for? I suppose the answer is myself. Not myself now, but me when I first broached the subject of computing, long ago in the early 1990's.

My own background studiously avoided the technical, which was an intellectual black hole. For Christmas 1979, or maybe it was 1980, I got an Amstrad 464. That is factually a computer, but for me it was a tape machine that played games. How it worked was a mystery to me, as was any technology. Television was more or less magic, for example how the picture appeared on the screen. The Amstrad 464 was a sort of television that played games.

When I became curious about computers in the early 1990's — and I cannot remember why this was — I still had the 464. As I remember, I started to try and learn about computing as follows. I bought a few issues of 'Byte' magazine, which seemed to be an authoritative voice; I took out a book about BASIC programming from the local library; and I bought a cheap (but very good) book on the same subject. Byte taught me that computing was full of impenetrable jargon and incomprehensible acronyms, but nevertheless with a couple of books about BASIC

programming and an Amstrad 464 and its BASIC interpreter, I was at least on my way.

However, I remember the things that made understanding how to program more difficult, things not so difficult in themselves but hard to understand due to certain misconceptions, for example that 'computing is a sort of math'. My guide is aimed at the level of understanding underlying the writing of code and which precedes the writing of code. I assume the reader knows nothing at all about computers, at least at the level of programming them, but would like to know what programming *is* and what computers *are*.

This is absolutely not a guide on how to program. You learn how to program by writing code and actually writing programs. Writing code is knowing to how to write the line of code you are writing now and getting it to run. Writing code is always in the now. It is never theory. Planning code, yes, but not writing it. Write it, run it. That is the ideal, but often the code won't run, or you don't know something you need to know to make it run. A guide on how to program needs to give you the whole picture, and what you don't know you look up.

But before you can learn to program it is at least useful to know what a program is and what a computer is (aside from being a magic box that as you touch it brings to you bright and brave new worlds). To understand what a program is is not to know *how to* code. Therefore, the code examples in

this guide are written at first in simplified 'made up' languages and even when I discuss real programming languages I have left out most of the gory details. All code examples have been made as simple as possible and are intended to be read in the text, understood as text, and not run in a code editor.

In a nutshell, this is a programming book for someone who has never programmed but who is curious about what programming is.

2. Entwork

Two ways come immediately to mind as a basic definition of programming. We can say it is:

- programming a computer; or it is
- writing a program.

This is a useful starting point because, while both of these definitions appear to be more or less tautological, in fact neither of them is really true. Not false, but not really true.

2.1 Systems

Take a very simple program. The UI consists of a single button that, when pressed, prints a document called `foo`. Here is the entire code for the program, written in an imaginary language called `fop` ('foo-only-printer') that consists entirely of this one command.

```
fop::print!
```

When the `foo` document successfully prints, it cannot be denied that we have programmed the computer and have written a program. But. How did our tiny 'one-liner' actually do all that? How did the computer we just programmed know how to do what it just did — to print a document?

The answer, of course, is that we are standing on the shoulders of giants. We have written a program that sits on top of a host of other programs. In other words, we did write a program, but ours was a little program, as little as a program can possibly be. The giants wrote all the big bits, all the good stuff. The code that actually does stuff.

For our UI we used a designer to plunk a button on our main screen. We also wrote the `fop` code (`fop::print!`) that runs when the button is clicked. Our *click handler*.

That's what *we* did.

Here's (very very roughly) what the giants of the screen code did. Our screen is a rectangular region (a '**window**') within the main display. The main display is in itself more or less dumb. It paints pictures. It is filled with regions just like our own screen — with 'windows'. Our monitor **refreshes** the display maybe sixty times a second and at each refresh each window is **painted** at its current pixel position. As the user, say, moves windows around the screen, each refresh updates each window to its new position, so it appears to move. Of course, one window can cover another, so the giants have to make sure in their code that hidden regions are hidden and visible ones stay seen. Within each window, or rectangular region, there is a complex layout of **controls**, our button just for example. These need to be painted onto the display, each in its

current state. Buttons, for example, ought to look different when they are pressed.

Worse yet for the giants, windows have interrelationships with each other. A program (like our `foo` one) is a **process** and each window in it belongs to that process. Even this one-liner `foo` doc program pops up a window informing the user all is well after a successful print. Even `Foo`, then, has a main window and a secondary window. Other more complex programs will have many more windows. The giants must ensure that, when a process (that is, a program) exits (that is, the program is shut down) all its windows correctly disappear. Finally, user input has to be tracked (mouse or keyboard or, nowadays, fingers). The system needs to know — and right now — what window what user input is directed at. Each window must send a **message** as '*I'm clicked!*', says our `foo` print button. Why then, `fop::print!` replies our little program.

This is **kernel** stuff that lies deep within the heart of the computer operating system. The kernel though needs much more than a dumb windowing display. Windows are sparked off by processes and the giants need to manage these. Processes can launch lightweight versions of themselves called **threads**. Managing processes and their threads is a complex task.

Windows and processes are nothing without **memory**. Processes must be loaded into memory as soon as they are started, and the system

must allocate each program (that is, its root process) a set amount of memory (that is, within the available **RAM**). In turn, RAM is merely short-term memory. What use is a computer that forgets everything? A running program exists in RAM, but the source code needs to be hard-stored, usually on a **HDD**. The giants have to include that in their shuffle, *the shuffle between window, process, quick-memory and long-memory*.

Finally, let us mention **peripherals** that exist outside the computer box, and in particular draw attention to the printer which, after all, is what our little `foo` program is all about. Printers come in a whole array of shapes and sizes. To deal with each on an individual basis would require an enormous amount of unnecessary effort. So, the giants devised a generic blueprint called a **driver**. To the system all drivers have the same shape. It is the properties that differ from implementation to implementation. Each printer has a driver written for it and the system interfaces with the driver, not the printer itself.

In this foo-scenario, of course, the real action takes place beyond even the system printer driver. The driver sends the print instructions and data (that is, the foo document) and the printer carries out the instructions on the data. The actual code-to-print resides in the printer itself. It is *this* code, the sleeping giant beyond even the giants of the

computer system (the **operating system**), that makes the `foo` document print.

Our little one-liner, then, is a program within a program that makes a call to the peripheral program that actually does the printing. We therefore cannot really say to ourself that we have programmed a computer or that we have written a program, though in a sense these statements are true. If we do claim we've written the program that prints the doc, we might as well say we have built a house just because we rang its doorbell. For that is our `foo` program. It rings the bell that summons the giant's code that wakes the sleeping peripheral giant.

2.2 Frameworks

In the previous section we met with the *system* and the *peripheral*. Developers, generally, do not do close encounters of such a kind. In between the operating system and the developer there nowadays lies the **framework**. These days, you would not use a `fop` method but a `fop` framework to take care of printing stuff for you. If you used the new `ffs` framework (said to stand for 'foo framework services'), you would write something like this:

```
#foosing fooprint


// Print out the Foo doc.
FooDoc foodoc = few FooDoc()!
foodoc.prefoomat()!
foodoc.confooguresPrinter("Foo!")!
```

```
ifoo (foodoc.sendToPrinter() == fue) {
    infoom("The document was printed successfoolly.")!
    }
```

Now the path to the giants runs basically like this:

```
your-code > framework (ffs) > system > peripheral
```

Another layer of giants — the framework giants — has been interpolated! In fact, the days of monolithic do-it-all frameworks seem to be in retreat and are being replaced by patchworks of littler entwork (as the old Anglo-Saxons called the handiwork of giants). Here we have **DI** ('dependency injection'), **plugins**, **services** (including microservices), **containers**, **libraries**, **components**, **API**'s ('application programming interfaces') etc. What used to be a whole body is now a composite of artificial limbs and iron lungs and glass eyes. The giants have shrunk! In the end though, these little patches of entwork do add up to the equivalent of a framework.

We see then that the framework sits on the shoulders of the system and the modern developer sits on the shoulders of the framework. Programming is therefore not writing a program or programming a computer, it is *interpolating code into a system*, whether cloud or Android or PC.

Programming (in feng shui terms) is not to create the garden, it is to lay a flower bed, or plant a peony. For some — the support team — it more of a case of digging up weeds. The *computer* does not run your

program, *your program runs on the computer* placed on top of the entwork.

That is what a programmer of a computer does and what programming a computer is. In the next chapter, however, we will see that even the notion of 'programming a computer' is an increasingly outmoded one, for nowadays a single program runs on many computers. In fact, it does not so much *run* on your computer, rather it *arrives* at your computer over the Internet.

3. Computer to Computers

In the old days when there were just computers and programs and no 'devices' and apps you would code something like this,

```
form.Height = 300
form.Width = 300
```

and there you were. Your program had a nice square **form** (window) 300 by 300 **pixel**s. If this was a Windows program, you were programming for the Windows system on a Windows PC. A pixel was a dot on the video display and a line 300 pixels long was 300 dots long.

3.1 Pixels and Dots

Those were the days, not necessarily better, but simpler. Today, screen resolutions differ widely as do the devices they run on. The code above (bearing in mind '300' equates to '300 dots') is in itself quite inadequate. The thing is, 300×300 is clearly intended to mark out a region of the screen and, when a pixel was a pixel, 300 dots did the job. But nowadays there are dots and there are dots. More and more dots are being crammed into the modern screen. This is Hi-Res (or some such term as 'retina' in Applespeak). You want to say, 'this screen should take up this amount of space', but you can no longer specify the number of dots.

On Android, for example, the problem is solved via 'virtual' pixels. These allow you to indeed specify a 300×300 screen, but behind the scenes, the giants are calculating how many dots there will be in that 300 amount.

> I came across this pixel-issue recently when I used WinForms to write a quick utility program. (WinForms is a now-old Microsoft technology for building UI screens.) I hadn't used WinForms in a long long time, as all my home projects use WPF[1]. WinForms does things the old way, where screens and their text boxes and checkboxes and buttons have a fixed size specified in pixels. WPF is more modern and flexible and generally less pixel-centric. WinForms is designed for programming on a PC running the Windows system and how refreshing it was in terms of knocking out a quick app. Here you have — you really do — a WYSIWYG design system. You create a form (screen, window) and it looks just how you designed it when you run your program! You see what you design! Except I didn't, for all of a sudden, the screen would shrink when I clicked on it.
>
> Times and screens have changed since I last used WinForms. The assumption of a generic PC running on Windows no longer holds. It's the pixels, of course. Some screens are more blessed with pixels than others. The WinForms team have implemented a fix for it; rather hacky and inelegant I felt. The fix was also what was making my

[1] 'Windows Presentation Foundation', which is supposed to have been the new WinForms replacement for over a decade now, so don't make a note in your diary to pencil in when it *will* replace it

> screen shrink. It was 'compensating' for — well, what I am not exactly sure in my case, as I don't have either a new or a Hi-Res monitor. As is so often the case in modern computing, there was a one-line fix with a configuration setting (to fix the WinForms fix), but in searching for the fix for the fix it struck me how even old tech like WinForms must fit in with the new tech world.

The point is that in this NTW there are computers and computers. The different resolutions of modern devices is only the beginning. The tablet and the phone are essentially computers that are rarely called computers. Each of these has a tiltable interface that can be switched between portrait and landscape. A phone is also moreover very small. You nowadays have to code your program to make nice with all these sizes and tiltings.

3.2 Computer to Device

At least though you do still write apps for a PC or a tablet or a phone (even if you write a single program with a version for each, each version is an independent app). What about the Internet, though? A website is loaded into a browser and the popular browsers have a version for pretty much any popular device. Here we have ventured far beyond mere pixels. A website ought to look good on both a large Hi-Res screen for a PC and a phone on a phone in portrait. Even watches can be little computers these days. Does your site look good on a watch, be it Apple or Google? And what if you wrote the following code?

```
button.Text = "OK"
```

Kerrang, don't do that! A perhaps overlooked counterpart to globalisation is *localisation*. Even little programs are often localised these days. So, your button text should be able to be configured into the local language of the user.

```
// set text to 'OK' in the local language of the user
// in German fex, OK is simply
// 'Wiederholungsanpassungswertantwortverschliessung'
button text = localised.OK
```

Just these three things — screen resolution, device type, localisation — illustrate how far we have come from programming *for a computer*.

3.3 Device to Browser

There is more, though. The earliest web pages were written in **HTML**, a '**markup**' language that organises and formats text. With HTML all the basic formatting functionality is there, making it possible to create a simple 'word-processed' document. The early www was thus a matter of downloading documents. But the limitations of this were soon realised and so the giants invented the '*gateway*'. This was a gamechanger and the foundation of the modern www. For this gateway existed on the server — the central exchange that sends web pages to your browser. The gateway was a place pages could be sent to for processing before being returned.

Take a product page, for example. With HTML you could design a product page, but you would need a separate page for each product. With the gateway, you could now design a product page *template* and pass this to the gateway. The product details were retrieved from a database and the template filled in with the data. The gateway then sent the HTML page back to the server which forwarded it on to you. The significance of this new toy was huge, for it can safely be said 'no gateway, no Amazon'.

Meanwhile, in browserland, the giants invented the idea of **scripting** in the form of **JavaScript**. It was JavaScript that could read the DOM and alter it. A simple example of this can be seen in the following code:

```
// HTML
<p>This is a paragraph</p>


// DOM + JavaScript
p.innerHTML = "I have just modified the paragraph text!";
```

All of a sudden, annoying animations appeared in web pages. Web pages were now dynamic. They could be programmed on the client (that is, the browser).

Of course, the giants didn't leave things like that. Web browsers have become more or less like a 'Windows on top of Windows', unfeasibly complex systems with a host of programming possibilities.

Today, then, the web is more or less a case of '**client**, meet **server** and server, meet client'. A web page is still in the end an HTML document, but this end result is almost always produced by a myriad of technologies both on the client and at the server end. The document is, of course, sent to the device that requested it, any of the many types of devices in use today. It may be viewed on a gigantic monitor, or maybe a tiny watch-computer.

That, in a nutshell, is a million miles away from the old days of a generic PC running on a Windows system. The old WinForms days. The path of your web-programming activity might now look something like this:

```
[ Yaml, XML, SASS, React.js, TypeScript, Node.js,
SQL (or NOSQL), Apache/Kestrel, Razor,
REST API (or GraphQL), JSON ]
     > server > client >
          [ JavaScript, CSS, WebAssembly ]
               > HTML > browser > system (> peripheral)
```

Even the basic apparently tautological definitions we began with — that programming is 'programming a computer' or 'writing a program' — are now mere dust. The notion of both 'computer' and 'program' have vanished. This does not mean to say that a web site is not a specific thing all in all, or that it does not in the end run on a computer. But just because a particular car's destination is Penzance and all cars have a destination, that does not mean all cars have a destination in Penzance.

That is to say, you can't build your website on your computer and say, 'That looks perfect, all done', for the destination of your live website is not limited to your computer (your 'Penzance', if you like). These days you are, if you like, 'programming other people's computers'.

3.4 What is programming, then?

So, what then is programming nowadays? Before anything else can be done, it is *planning* and *designing* and *thinking*. Plan in place, then it is *writing code*, perhaps the core activity or at least the sine qua non of it all. Code written, then it is *assembling codes* into a running unit. Unit created, then it is *testing the runnable*. Tests done, then it is *fixing code*. This is a feedback loop — writing, assembling, running, fixing — until the glorious Day Of Beta. Beta built, then it is *reviewing the app* by all the interested parties. Fail? Back to the feedback loop. Pass? the hallelujah Day Of Release. Release done, then it is *maintaining the app*, more or less an as-and-when process (as and when an issue arises).

So,

- planning
- coding
- assembling
- running (testing)
- fixing

- reviewing
- releasing
- maintaining

Only the second stage is focused on *writing* code. Fixing and maintaining are based on code too, but not necessarily writing it. We could therefore define 'programming' as all this or offer up a narrow definition of 'coding'. Which is a how long is a piece of string question. Perhaps it is best to say that programming is both coding and a whole lot more.

Whichever and whatever, it is on that hallelujah Day Of Release — as your little app is at last sent out to become a part of Ymir, the World Giant, the giant of the giants who created the code your app runs in and on — that you become a programmer. After all, you have a program so ergo even M. Descartes in his deepest doubt would agree that *you have programmed*.

P R O G R A M M O E R G O
P R O G R A M M A T O R / T R I X
S U M

4. What is a Computer?

As all programs are run via a computer, it is essential to have a basic idea of how computers run in order to program one. Programs that work closely with hardware, such as drivers (programs that control hardware such as printers of graphics cards) or to a lesser extent audio and video processing software, require a detailed knowledge of how this or that hardware runs.

For most programs, however, a generalist's view is sufficient. This chapter examines in suitably generalist terms what a computer is, how it is architected and how code is related to the architecture at the machine level.

4.1 Math

Is programming a form of math? Take the following algebraic equation

```
x + y = z
```

and compare it to this line of code

```
z = x + y
```

As these look more or less identical, they support the view that computing ('hey, the clue is in the name') is math. This view is however a misconception and the algebra and code are fundamentally different.

The algebra represents numbers but the code is an instruction to the computer.

When a mathematician says 'x + y = z', they mean that the abstract values 'x' and 'y', when added, are equal to the abstract value 'z'.

However, the code `z = x + y` is not an algebraic equation. 'x', 'y' and 'z' here are pointers to data stored in the computer's memory. 'x', for example, might have a value of 'Hello,' and 'y' of ' world!'. So, the value assigned to 'z' would be 'Hello, world!'.

```
Hello, +  world! = z
```

This is not an algebraic equation!

Even integral numbers would not make it an algebraic equation, for 'z' = '1' + '2' simply means 'put the sum of 1 and 2 into the memory pointed to by z'. An algebraic 'x' is an abstraction of a number; a code 'x' is a reference to a value in the computer's memory.

However, to pick apart what a computer's memory actually is, we need do understand what a computer is. We will begin with a brief overview of the history of computing machines.

> It is interesting that the once-universal term 'computer' given to computing machines is in danger of vanishing. Tablets and phones are clearly computers but never, or hardly ever, called by that name. Nowadays computers are an essential part of cars and televisions, but cars and

> televisions are not computers. As more and more aspects of life and technology 'go digital' it is probably the case that the computer itself is becoming a historical device, like the video player or analog TV. The computer, perhaps we can begin to see, was only a computer in its heyday, when it was the only computing machine around (pace the pocket calculator).

4.2 Computing Machines

What is a computer? Before computers, the answer would invariably be 'someone who computes'. There were no machines that could compute (apart from outlier contraptions such as the genial Babbage 'engines'). There were devices such as the abacus that aided computation, but an abacus is not in itself a computer.

Perhaps the nearest commonplace thing to a machine-computer was the *clock*, which effectively computed the time. The most sophisticated clocks (and watches) were marvels of mechanical engineering.

It is in the late 1940's that electronic machines able to compute were first built. The groundwork for these machines was laid over a long reach of history via the work of many mathematicians, logicians, linguists and engineers, but the reality of 'what a computer is' begins only with the building of a fully-functional one. Functional electronic computers first appeared in the mid-late 1940's.

The new computers were at first gigantic devices that performed very specific and limited tasks. They were, however limited, proper commercial products bought by big businesses. The early computers did what was asked of them well enough.

The development of the computer in those days was characterised by evolution coupled to many dead-ends. Each new computer required a new design and solved a new problem. Some of these designs and solutions stuck, many did not. Gradually, however, certain basic principles emerged. The computer came to be a processor of **code** stamped onto a punched card. At this groundbreaking stage, there was now *code* and there was *machine* and the resulting computations could be *stored* externally onto *tape*.

In other words, *there was the computer and there was computing* and the computer had now reached the point of being a generalist computing machine.

```
code > [ computer] > data
```

Many well-known components of the modern computer, however, emerged gradually, including the most basic functional unit of any computer: the **byte**. Things in the 'real world' are collections of electrons and quarks. A computer is a collection of **bits**. Just as quarks are collected into protons, bits are collected into bytes. How many bits in a byte? It took a while, but eventually a canonical figure of eight was generally adopted.

4.3 The Personal Computer

It was the **PC** that gave us the 'classical' shape and form of the computer, which takes us up to the early-mid 1980's.

A generalist can understand a classical PC as working somewhat as follows. The PC's activity began with the **BIOS**, a chip that ran when the machine started. The BIOS checked the hardware and then fired up the operating system (popularly MS-DOS in the early days). A classical early PC had at its core a *hard disk*, memory (*RAM*), a *graphics card* and a *processor*. It was attached to more or less essential '*peripherals*' such as a *monitor*, *keyboard* and of course the *floppy disk drive* (in all its creaking glory).

The classical PC stored files on its HDD. Files contained data or code ('*programs*'). Programs were bought off the shelf — *word processors* and *spreadsheets* and *games* and the like.

A classical PC, then, was a collection of codes stored on a HDD ready to be converted into running programs by being executed as a process by the operating system (for example, MS-DOS).

```
FILE > OPERATING SYSTEM > PROCESS
WRITER.EXE > MS-DOS > [PROCID_14E77]
```

4.4 Architecture

The fundamental structure of a computer, from the first machines to the classical PC, is called its **architecture**. A programmer ought to possess a generalist's understanding of computer architecture.

As we have seen, the architecture of computers evolved between the 1940's and the 1980's. By the latter period, the fundamental architecture of the modern computer was a done deal. Even today's tablets and phones are simply reimagining's of the first classical PC's (so we have monitor > *touchscreen*, *on-screen keyboard*, mouse > *touch*, HDD > *SSD*, graphics card > *graphics chip*).

If the atomic core of a computer is the bit, the active core is the processor and the world of the processor is the world of the byte (eight bits), a world which is illustrated below:

```
10110101
11100111
00110001
10000011
```

Fascinating, eh? Well, fascinating or not, that is bytes and that is computing. But why bytes? To answer that is not to answer why computers exist at all, but why they exist in the form they do.

4.5 Counting

Mathematics is in the end a sophisticated form of counting. Counting, in turn, is a counting of things. Things is what pre-scientific people counted. They counted 5 or 67 or 123 sheep. No one was interested in no sheep.

The default European method of counting is based on the ten fingers of the human body. It therefore has ten names for the numbers. All bigger numbers are based on these ten names, though this is often hidden by the changes in the sound of language over time.

> While 'hundred' for example is unlike any other English number's name, it was probably once pronounced something like '*dkmtom*' and so clearly based on '*dekmt*' (as 'ten' was pronounced in them days).

With the adoption of writing, numbers as well as words were set down on the page (or tree bark or stone). An example to consider here is the ingenious Roman system, which worked within three numeric groupings: 10, 100 and 1,000. Each grouping had a beginning, a midpoint and an endpoint, written with a single symbol. The numbers in between were written using an additive or subtractive principle. So, V is the midpoint between 1 and 10, and 4 is '5 – 1' (IV) and 6 is '5 + 1' (VI). Similarly, 9 is '10 - 1' (IX). The additive principle is seen in the sequences I, II, III and VI, VII and VIII.

The three groupings are shown below:

```
I .. V .. X     1-10
X .. L .. C     10-100
C .. D .. M     100-1,000
```

With a mere six symbols, all the numbers from 1 to 1,000 can be easily written down. So just as we have VII (7), there is LXX (70) and DCC (700), or IV (4) and XL (40) and CD (400). It is very noticeable how this writing system is tied to the decimal system preserved in the names of Roman numbers.

The significant things to notice here is how this Roman writing counts only things and how much this makes sense in pragmatic terms. We might even ask, *Why would anyone want to count anything but things*?

4.6 Decimal

The notion of a zero symbol is a profoundly unobvious one. It is only because it is so familiar to us that we find it difficult to conceive of a counting system without it. However, if we compare it to the Roman system, things do not look good at a first glance, for set against the mere six Roman symbols (IVXLCDM) that can be used for every number up to 1,000, the decimal system requires nine counting symbols (123456789) plus the zero symbol (0) to get us to 10. Oh dear.

Things begin to get better though. When we have used up our counting symbols as we reach the

number ‘ten’, we can now bring our zero symbol into play. We shift our first counting symbol to the left of the zero symbol, which gives us ‘10’. The ‘1’, we now say, means ‘one lot of ten’, to which we add the zero. 10 + 0 of course remains 10. In that lies the zero’s profound little trick.

We can now use the same principle *for all the rest of the counting symbols*. ‘15’ is simply ‘one lot of ten plus 5’. Note here that there is no longer any conceptual shift between additive (VI) and subtractive (IV) symbols. We can also scale up our counting symbols on the left and say, for example, that ‘20’ means ‘two lots of ten plus zero’. Counting from 10 to 100 is the same as counting from 1 to 10 except for the extra zero on the end.

When we max out our symbols at ‘99’, we simply re-use our basic principle and shift the ‘1’ to the left again so that we have two zeros to the right. The ‘1’ now means ‘one hundred’ which of course remains as-is when we add our two zeros to it. From this point, we can go on shifting the ‘1’ indefinitely: 1000, 10000, 100000000000000, etc.

With our zero-based system in place, we can write numbers that would be absurdly complex to express using the Roman system and all because we introduced a symbol that is an anathema to counting — zero!

4.7 Binary

When the first computers were being designed, counting was a decimal thing. So how would you store numbers on the computing machine you are trying to build? Clearly, that is fundamentally asking, *How do I store digits on the machine?* That is to say, decimal digits.

Early computers used valves and an obvious way to indicate a numeric value was by using voltages. For a decimal digit, you would measure any of ten voltages and each voltage would represent a digit. If you had two valves, you could fetch back a two-digit number. Which is clumsy, unworkable.

This is where the genius of the zero-based writing system comes in, for it is eminently capable of further abstraction in the way the Roman system is not. The underlying principle of the zero-system is that it uses x counting symbols plus another for zero and it shifts the first symbol to the left when the symbols have been used up.

If we use seven counting symbols, '10' means 'one lot of eight plus zero'. If there are more than nine counting symbols, we can simply re-use the letters of the alphabet for numbers greater than 9. With fifteen counting symbols, we end up with: 123456789ABCDEF. Here, 'F' means fifteen and '10' means 'one lot of sixteen plus zero'. The minimum number of counting symbols you can have is one,

where '10' means 'one lot of two plus zero' and so here, meet binary.

Binary is terrible for humans, but for designing computers it is booyakasha because all we need now is two voltages to make a digit, one for the solitary counting symbol and one for the zero symbol. Let's say zero volts equals '0' and 1 volt equals '1'. So '16' is 10000000 in binary and that would be eight valves, the leftmost being fired up with 1 volt, the rest set to zero volts.

The *binary* counting system is a perfect fit for the fundamental atom of the computer, the bit. It is binary that makes it possible to build a working computer. Binary and decimal *are arithmetically identical* in the zero-based writing system. Better still, so are octal (seven counting digits) and hexadecimal (fifteen counting digits). What is a standard byte? Eight bits. Octal. Two bytes, sixteen bits: hexadecimal.

Binary, octal, hexadecimal: every programmer needs to have a basic understanding of these.

> Note that a counting system above hexadecimal would not be practical. The next step up from hexadecimal (base 32) would have thirty-one counting numbers:
>
> 123456789ABCDEFGHIJKLMNOPQRSTUV.
>
> Would you be able to work out what the number BOAW2T is in decimal? That is certainly not BOOJAKASHA!

4.8 Code

What is 1 + 1? How could you make a computer be able to answer this question? The answer is, if you think on the question, not at all obvious. A key to understanding the answer is to realise that there are four actors in the sum: the two *addends*, the *sum*, and the '+' symbol. Understand this and you start to see how a computer 'thinks'.

Below is a very simplified computer-like way of saying one plus one:

```
01. LOAD 1
02. ADD 1
03. STORE
```

The key to the answer is that there needs to be somewhere for data to both *come from* and *go to*. Even for something as simple as 1 + 1, the 1's have to exist somewhere. In this very simple sequence of instructions, the first *loads* a '1' value (the data). Into what and from where we do not care for the moment. The point is that we have made it the computer's focus of attention. This done, we can run our add command. This will *add 1 to the data we have just loaded*. The data now has a value of '2'. The last thing to do is *store our data*. Again, where we do not care for now.

We can use this super-simple system to observe how our initial 'algebra' sum (z = x + y) might work and

so see how un-algebra-like it really is. Here is the sum:

```
01. SET x 1
02. SET y 1
03. LOAD x
04. ADD y
05. STORE z
```

Note that we could just as well have written (as per our early example):

```
01. SET x "Hello,"
02. SET y " world!"
```

The core of the processor is not, then, monolithic, for we see **commands** (SET, LOAD, ADD, and STORE) along with *data*. These are handled by the processor using **registers**. *Registers run commands and store data.*

When we say 'SET x 1', that is where 'x' will go — to a data register — and the register will now be storing a value of '1'.

Our first and simplest example (LOAD 1, ADD 1, STORE) has three registers (A, B, C), one that executes commands, one that stores the data that commands act on and a third that stores data permanently (until it is overwritten). So, this is what our command sequence is saying:

```
load the value 1 onto register B
add the value of 1 to the contents of register B
store the contents of register B onto register C
```

4.9 Language

Computers do not in any way shape or form directly understand English. In computerese, our simple sequence of commands might look something like this:

```
00000001 00000001
00000002 00000001
00000003
```

This is **machine code**. It is this *and only this* that a computer understands. In the above example, we observe commands and data as the machine sees them. Look at the first line. The first and second parts seem identical, but course they are not. Each *command* has a numeric look-up value. Here, '00000001' stands for the 'LOAD' command. The second part of the statement is data and it has a value of 1. Note that, when the computer receives the LOAD command, it knows what to do with the data value: it will set registry B to '1'.

As humans do not do reading machine code at all well, a programmer would work with the englished equivalent of the sequence, which we have met with earlier:

```
01. LOAD 1
02. ADD 1
```

```
03. STORE
```

This represents what is called **assembly language**. Assembly languages map very closely to machine code. Without the binary, assembly language is easier to read, but not easy to read. Because it must follow the machine so closely, it is very precise and very verbose.

Enter the **programming language**. *C, C++, C#, Java, Kotlin, Haskell, Prolog, LISP, Lua, Ruby, Python, JavaScript, COBOL, BASIC, Fortran, Pascal, Ada, Go, Rust, Brainfuck* — you could almost utter a syllable and that there is or was the name of a programming language.

Programming languages — even C — are human-readable. They work at a much higher level than machine code or assembly language. It is important to understand, though, that they are the same thing. That is to say, a program written in any programming language *cannot be understood by the computer*. Your program code must be converted into machine code (perhaps via assembled code). That done, the high-level language and the machine code are one and the same.

In many languages, this conversion process is called **compiling**. Compilation is performed by — a compiler. Other languages are **interpreted**. A compiler takes your entire program and turns it into machine code; an interpreter does the same thing line by line as the program runs. C is an example of

a compiled language and BASIC of an interpreted one.

It is programming languages that a programmer learns, just as a French interpreter must learn French. A programming language is in a sense interpreting the computer. To use a programming language, then, does not require a detailed *knowledge* of the underlying machine, but a general *awareness* of it is I would say more or less essential. It is with this awareness that a programmer can distinguish between algebra and code: between x + y = z and `z = x + y`.

Summary

Computing is not algebra. It is not math. It can do algebra and all sorts of math. But that is not what it is. At a fundamental level, it is commands and data. In the simple sum of 1 + 1, the '+' is a command and the ones are data. The computer is not (not to its knowledge) doing a sum. As far as it can tell it is performing a command on data. The data, moreover, must always be stored somewhere. As the machine processes any sequence of command, data must be both fetched and sent to and from registers.

That is a computer program, a relentless sequence of fetching and sending and commanding, this sequence having been read from that alien thing — alien to the computer — we call 'code'.

5. Towards A Language

In the next chapter, we will at last breach the world of programming and look at a real programming language, C#. This is one of the most highly-regarded programming languages around today. However, before we get to C# itself, the aim of the current chapter is to place it in a wider context as to 1) what a programming language is and 2) in particular what a 'C-like' programming language is.

5.1 Algorithms

I finish this section on algorithms with a second metaphor, but let me begin with a first: an **algorithm** is a script and code that uses an algorithm is an actor. Code reads algorithms like scripts, for an algorithm is a canonical way of solving a problem. If you use an algorithm, you are wise to follow it to the letter.

Take as an example problem a computer screen. It is filled up with lines and each line needs to be *drawn* onto the screen and drawn *fast*. To solve the problem, *enter the algorithm*.

The problem here being? Lines can be described algebraically with a linear equation and that equation can be mapped to cartesian coordinates. These equations though are *decimal* things, not integers, and decimals use up more time and space

than integers when they are being processed. Linear equations are not ideal for computer graphics.

Enter Jack Bresenham in the year 1962, Jack being one of those 'little giants' who figured out one of the key pieces that became a secure part of Ymir the world giant. He invented the 'Line Algorithm' named after him, a method of drawing a line using only integers. His key insight was to see that pixels are effectively the cartesian coordinates of the computer display. The problem of line drawing is therefore getting to pixel Y from pixel X. The algorithm, in a nutshell, goes from pixel to pixel, nudging the line in the right direction as it goes. If we advance for example from coordinate { *300*: *300* } to { *301*: *300* }, we need only a simple integer addition to move forward.

The algorithm is an important one. So many lines are drawn onto the computer screen every second that shaving off even a tiny amount of processing time adds up (or in this case subtracts down) and this illustrates a basic principle in programming: *the more times an action is repeated the more need there is for speed, and after a certain threshold the need becomes a* ***requirement***.

Algorithms can be considered as the cartilage of programming, flexing its joints and helping it run. Fundamental algorithms form part of most programs because they are integrated into the frameworks used in every program. Behind the scenes, a simple line of code such as

```
DrawLine(300, 300, 400, 400)
```

will be using at least a variant of Jack Bresenham's algorithm.

5.2 Assembly

While algorithms can provide a *global optimisation*, assembly language can optimise individual tasks. It offers a *local optimisation*.

Although code written in high-level programming languages ends up as machine code via a compiler, compilers are generalists. Of course, they aim to create good clean assembly code but it is obviously quite a task to make that code the most optimised possible in every case. Enter the specialist assembly language programmer. The ALP has a huge advantage over the compiler because they can focus on the specific problem at hand. In the past a good ALP was a better bet than any compiler. For music or audio processing, say, an ALP was a must in order to optimise the code as much as possible.

However, as we know by now the giants of Ymir are always busy and among them are the giants of the compiler workshop. As a result, today's compilers are masterpieces of entwork. Only a supreme assembly language programmer could hope to improve on the assembly code produced by a modern compiler. There are still edge cases where a human can still outshine the giants, but generally speaking only a fool would chance their hand these days at assembly code in the hope that their code would

improve on the compiler's code. So a programmer ought to know that there is assembly language and what assembly language is but has no need to learn how to code in it.

In the far-off days of old though, low-level was an achievable aim. So, C.

5.3 C

C, it should not be surprising, is based on a language called 'B' which in turn was based on a long-forgotten 'A' language. C was developed in 1972 by one of the giants of computing, Dennis Ritchie[2]. Anyway, C is undoubtedly one of the great programming languages.

A basic C program looks like this:

```
int main() {
  // do stuff
}
```

As you familiarise yourself with programming, you will immediately recognise this as a 'C-like' language by those {} curly braces. C is a high-level language with a deliberately concise syntax. Before C, programming languages tended to aim for a chatty English-like appearance that made the code, it was thought, more readable.

[2] He even, I believe, got to be on first-name terms with Ymir who although he only has one name likes to be called Mr. Ymir by the likes of you and me.

Here is a simple *Pascal* example (Pascal being a language designed by Niklaus Wirth in 1970):

```
function Add(X, Y: Integer): Integer;
var
   Result: Integer;
begin
   Result:= X + Y;
end;
```

Here is the much more concise C equivalent:

```
int add(int x, int y) {
  return x + y;
}
```

C is of particular interest here as it is a high-level programming language that goes low-level with a feature called '*pointers*', which constitute a first-class part of the language. Pointers look like this:

```
int* ptr = &value;
```

A pointer is how C bridges the gap between a high and a low-level language. So, in our example, `&value` 'points' to a memory location on the computer. `*ptr` points in turn to `&value`. Pointers allow the programmer to do assembly-language type things in a high-level language and to communicate with the machine at the lowest level. (Pointers also have the uncanny ability to make C code look like Martian.) Anyway, C had all the high-level stuff other languages had, but it was pointers that made it C.

In the long run though, what with imitation being the sincerest etc, it is a testament to C that it became not just a language but a *type* of language. A lot of people it seems just *like* those curly braces.

5.4 C-type Languages

C-type languages include C++, Java (and its successor Kotlin), JavaScript and C#. That is an impressive collection of descendants.

Here is our `add` code in JavaScript:

```
function add(x, y) {
  return x + y;
}
```

Here is a Kotlin equivalent:

```
fun add(x: Int, y: Int) {
  return x + y;
}
```

Spot the difference?

C itself began all of a sudden to be old when a group of giants invented the **object**. We will discuss objects in detail in Chapters 7 and 8, but for now we can say that an object is a way of grouping code into the expression of a *thing*. A common example used to explain objects is an Animal. In the real-world things tend to be types of things and so a Mammal is a type of animal and a Cat is a type of mammal. So, animal <> mammal <> cat.

If we code for an object called 'Animal', we can then create another object called 'Mammal' that **inherits** from the 'Animal' object. A 'Cat' object can likewise inherit from the 'Mammal' object (and therefore indirectly from an Animal object).

C does not do objects, so Bjarne Stroustrup invented a C-like language called *C++* that did and does do objects. Here are our objects in C++:

```
class Animal{
};

class Mammal: public Animal {
};

class Cat: public Mammal {
};
```

> Class? An object is a *thing*. A class is what *defines* an object. In your code you would create a Cat object you had defined as a class. As an analogy, we can say that the FA Cup Final is played between 'two teams', a specific final between 'Liverpool' and 'Arsenal'. Here 'team' is a class, 'Liverpool' an object.

C++ is still today a go-to language because it does all of the low-level C things while offering the advantages of objects, but C++ programs tend towards the verbose, despite the concision of C. Common tasks need lines and lines of code. It is still popular for certain tasks, but many would say 'showing its age' (like C itself).

A big issue with C and C++ is memory. These languages — this is of course due to their facilitation of low-level coding, also their great strong point — require careful memory management. All computer programs, as we have seen, have commands and data and all data points to a memory location. What about when the program no longer needs the data? At that point, data should be *dereferenced*. In C and C++, programmers are responsible for *deallocating* memory. With this responsibility comes great power but does the average program need this? If not, why must the poor programmer needlessly endure a responsibility which is so obviously a recipe for disaster in the form of the dreaded **bug**?

Enter *Java* (designed by James 'the giant' Gosling) and meet the **garbage collector**. In a Java program, all memory references are logged by the Java ecosystem. The ecosystem regularly checks if data is being referenced. Every so often, dereferenced memory is declared as 'garbage' in a great sweep (a 'garbage collection') and is cleared away. The programmer, therefore, does not have to worry about 'leaking' memory. Java does it all for you.

Another innovation of Java is that it is not compiled into machine code (or assembly language). Java runs via the 'Java Virtual Machine' that exists in a world of *bytecode*. Bytecode is Java-specific and is essentially a generic form of assembly language. It is this bytecode that is compiled to machine code.

What this means is that, because bytecode is universal, Java can run on any operating system. The bytecode is compiled into the machine code of whatever system is running, be it Windows, Mac or Linux.

Java, then, is a C-like language that is nothing like C. It is not a hybrid low-high-level language or a replacement assembly language. It is a modern language. Or was. Even Java is now 'showing its age'.

Kotlin is the new Java, and is a language strongly influenced by C#. The interesting thing about Kotlin here is that it is compiled (the term is *transpiled*) into Java, indicating the profound changes in computing technology over the years:

```
THEN    C -> machine code
NOW     Kotlin -> Java -> bytecode -> machine code
```

Those are the C's, then. Next let us take a look at .NET, the natural habitus of C# that first entered the world at the beginning of 2002.

5.5 .NET and its Ancestors

The story of .NET begins, in this narrative at least, with the release of *Visual Basic* way back in 1991. This was a landmark release that revolutionised software development — **WYSIWYG** programming, who'd'a'thunkit? What you got with VB was a language (VB itself) and a designer. You created a screen (called a 'form') and then wrote code to

respond to button clicks and checks and mouse moves and so on. You would also, of course, write non-user-interface code for all the general functionality that your program required. The clue to the intent of VB lies in the language chosen — BASIC, the more or less toy language often used to teach children how to program. VB was not a toy and it was not BASIC, but it was a limited language. It was a brilliant piece of technology — revolutionary, as I said — but the limitation was all too obvious.

Two years later in 1993, Microsoft released the first version of *Visual C++*. This, the company seemed to say, was for *proper* programmers. As we have seen, C++ is a C-like language equally at home as a low- or a high-level of coding. VC++ also shipped with the *MFC* ('Microsoft Foundation Classes'), its helper-library. Still, helper-library or not, VC++ is verbose. As a low-level language, the argument went, that was a necessary burden for the programmer and VC++ did indeed offer a low-level of control inconceivable with VB.

By 1993, then, Microsoft had the over-simplified VB and the over-complicated VC++. So, meet *Delphi* from Borland, released two years after VC++ in 1995 together with its *VCL* ('Visual Component Library'). Delphi situated itself in the perfect middle space between VB and VC++. Rather than BASIC, Delphi used Pascal. No one would deny that Pascal is a better general-purpose language than BASIC. The Delphi dialect was Borland's own

Object Pascal. While VB was *object-based* its implementation of '**Object-Oriented Programming**' (to be discussed later) was limited. Object Pascal was fully-**OOP**. Moreover, Delphi had the beautifully-designed VCL, which aimed to cover about 80% of development needs, leaving the power of the Object Pascal (OP) language to cover the rest. Finally, there was a RAD ('*Rapid Application Development*') screen designer tied in to a well-thought-out hierarchy of screen ('form') objects. The major player behind all this was another of the all-time giants, Anders Hejlsberg. Poached by Microsoft, he went on to design C#, the language we will shortly be looking at.

Lastly, a brief mention of *COM* ('Component Object Model'), another Microsoft technology that first appeared in 1992. This technology tried to solve the problem of sharing data and functionality between Windows programs. Using COM, `me.dll` could as it were 'merge' with `you.dll`, and `me` code run *inside* `you` code. COM was clever, COM worked, but it was not pleasant to work with. Like VB, it was kind of good and kind of bad in equal measure.

A question worth asking, then, is 'Why didn't Delphi sweep the board?' Delphi was, at least it seems to me, superior to both VB and VC++ as a generalist **IDE** ('Integrated Development Environment') by some margin and ought to have done more than it did. Perhaps it didn't because Borland was a much smaller company than Microsoft, but I wonder if

another issue was with Pascal. This is a very clean language, but it does mean that a Delphi program is necessarily much more verbose than one written in a concise C-like language.

When the .NET Framework appeared, any Delphi developer would recognise the VCL in WinForms. In the broadest of senses .NET was a sort of supercharged Delphi. But there at its core was the clean and concise C#, not the verbose Pascal.

5.6 .NET, m'okay?

The .NET Framework could be described as a mix of Java, COM, VB and Delphi.

Like Java, .NET Framework languages didn't touch machine code. Instead of bytecode, .NET used the Intermediate Language (IL). However, .NET was not a 'universal' platform. It was Windows-only. The purpose of IL was to unify all the .NET languages. Whatever language you used to code in, the result was always IL. Any .NET language could be used in any project.

The .NET Framework was a world of assemblies each of which was all IL. The framework libraries were assemblies, your program was an assembly, *everything* was an assembly. Everything now being IL, COM was redundant. .NET did the job of COM, but did it far better. Now `me.dll` and `you.dll`, both rewritten in .NET, could run inside other assemblies. Both `me.dll` and

you.dll were, after all, assemblies and both therefore IL.

```
// this code is in me.exe
using yousoft.you; // this is the you.dll assembly

. . .

// me.exe is calling code in you.dll
me = you.foo();

. . .

// this code is in you.exe
using mesoft.me; // this is the me.dll assembly

. . .

// you.exe is calling code in me.dll
you = me.foo();
```

With the newer .NET Core, IL is much more like Java's Virtual Machine. With .NET Core, IL compiles into Windows, Mac and Linux machine code. Modern .NET programs (should we call them 'apps' nowadays?) can therefore run on Macs and Linux boxes[3].

.NET today is a mature (quarter of a century-old!) platform that (with .NET Core) is adapting to a

[3] This has certainly expanded .NET's reach, though I suspect the root cause of this is the ever-decreasing range of Microsoft's own reach.

rapidly-changing development environment, what with the Internet and the tablets and phones and watches all encroaching in on Microsoft's safety-zone, the Windows PC. The monolithic Framework is gone, replaced by a deliberately modular architecture. The general principle nowadays is that, if you need to *foo* something, you just choose the tool you think foos best. This might be *Gengal* or *Broobant* or *Woolp* or any of the other foo frameworks out there.

The one stable thing though seems to be C#. The .NET Core pretty much is C# these days, which sounds to me like a lead-in to the next chapter.

Summary

C-like languages begin with the eponymous 'C', a sort of low-high-level hybrid. Then 'Hello, objects!' and objects came to C via the good offices of C++. Next came Java, a language with a C-like syntax that removed the low-level stuff. Last not least is C#, the core language of .NET, a language that builds in one way or another on each of these earlier C's.

6. Low-level: Pillars

There are three fundamental pillars of coding: **sequence**, **iteration** and **branching**. This section therefore introduces coding by explaining these pillars and their significance.

6.1 Sequence

A computer program is in essence a sequence of instructions. This works at two levels. Firstly, the programmer writes code in a high-level programming language. Secondly, the program is compiled into machine code. It is a sequence of instructions but the machine code sequence is what is executed.

> A computer program is a sequence of instructions in machine code.

What are the instructions in the sequence are made out of? Well, let us first of all look at this sequence:

```
Print out "foo"
Output "foo"
Send the following to the screen: "foo"
the word is now "foo"
show us the word "foo"
```

This sequence is written in a human language and illustrates how human languages are too imprecise to write computer code in. Look for example at how

easy it is to say the same thing in so many different ways. This imprecision unacceptable if we want to deliver instructions to a computer.

We have already sketched out the early days of computers, of valves and machines big as castles, but back in the 1950's a more refined aspect of computing was also active courtesy of the theorists. These thinkers were studying the idea of computer intelligence and in doing so investigated the possibility communicating with the computer and the consequent relationship between computer and language.

The thinkers soon realised language is a very complex thing. Sure, you can make a computer understand a sentence like 'The cat sat on the mat'. It is relatively simple to write rules that say a sentence is not syntactically incorrect or a word is not in the dictionary (an asterisk here denotes an ungrammatical sentence):

```
*cat the on the sat mat [ syntax check fails ]
*sat mat cat the the on
*the clat sat on the mlat [ dictionary lookup fails ]
*dhe cat apped ob dhe mat
```

Semantics was the deal-breaker:

```
The cat ate on the mat
?*The cat ate the mat
?*The cat wandered through the mat
?*The cat expressed the mat
```

The simplest human conversation turns on an immense and ever-expanding database of context. Any human would know that mats don't sit on cats. That is context. A computer does not know this and has no sense of context. A poet, however, might use the phrase, knowing it is absurd. Cats do crawl under mats and that would be a context to use the phrase 'The mat sat on the cat'. This deliberate misuse of language is even further beyond computer's reach.

> The cat sat on the mat
> But the cat was very fat
> So *~~the mat sat on the cat~~ [*SEMERROR 2X43h!*]
> Said how do you like that?

This was also the era of Noam Chomsky and his revolutionary work 'Syntactic Structures' (1957). The basic and revolutionary idea was that language can be seen in mathematical terms. Here a 'dictionary' is a set of values that can be transformed by rules ('grammar') into phrases. One of his great insights was that there are 'phrase structure' grammars that, while inadequate to describe *human* language, are *nevertheless grammars that describe language*.

This is where the computer thinkers come in. We have seen how human language is too diffuse to be used to program a computer. But a simpler phrase-structure language is another thing. The key point now is that the new way of thinking about language was mathematical. If you defined a language in a

mathematical way, the correctness of its syntax is *provable* and *testable*. If you could design a *computer language* this way, you could read any code written in it and *prove if it was syntactically correct*. Note that there could be no semantics problems as per human language. Such a language would be perfectly unambiguous.

That, in a nutshell, is what the compiler we have often talked about does. It constructs the rules of the language, reads the code, checks the syntax.

A programming language defined so is built around a dictionary of **keywords** and **operators** and a grammar. A dictionary might include items like this:

```
keywords: if then while do for break
operators: +-*/%^&()
```

Even at the simplest level, it can be seen that a program written in that language would be incorrect if it used non-keywords such as 'boo' or 'yaka' or 'sha'.

The grammar tells the compiler what the rules are for putting keywords and operatives in the right order. Say this syntax is correct:

```
x = 3
```

The following instructions must therefore be wrong (unless the language is appallingly designed):

```
= 3 x
3 x =
3 = x
```

> The last line is worthy of note, for it provides a neat little insight into how code works. The intent of the language here is clearly to say: place the value on the right hand side (rhs) into the memory referenced by the left hand side (lhs). So `x = 3` means 'place the value 3' (rhs) into the memory referenced by `x` (lhs). If we reverse the `x` and `3` we can see that `3` is a literal and the code makes no sense. There is nowhere for the value in `x` to go!

Returning to our first example, we can replace the imprecise English with proper computer code:

```
print("foo");
```

Because the syntax can be proved correct, all ambiguity is removed:

```
sat cat mat [ GREAT! OK! ]
sat mat cat [ NO! 2X43h! SEMERROR! ]
```

and no poet could argue with that, not even Homer himself.

6.2 Iteration

Computer code would be perfectly rotten if there was just an endless sequence of code. Imagine a million lines of solid uninterrupted statements. The sequence is the first pillar of programming. The second pillar is iteration.

Look at this code from a world with no iteration:

```
print("foo");
print("foo");
print("foo");
print("foo");
print("foo");
```

With iteration:

```
// iterator plus control code in parentheses
for (i = 1; i <= 5; i = i + 1)
{ // code block start
     print("foo"); // code within block
} // code block end
```

This is a **`for`** loop (iteration). Note the familiar curly braces. These define a **block** of code. The block begins with the opening brace and ends with the closing one. All the code in between (all of one line here) belongs to the `for` loop. Between the `for` and the opening brace you can see how the loop is controlled.

The code is saying 'loop 5 times'. First it places the value of '1' into the memory referenced by `i`. Second it tests to see if the value in `i` is less than or equal to 5. If the value in `i` is 6, the iteration stops and we pass to the closing brace (which means 'end'). If the value in `i` is less than or equal to 5, the third control segment in executed, and 1 is added to the value in `i`. Having got past the control code, the code within the braces is executed. Here of course, the word 'foo' is printed.

In this code, the value in `i` is initially set to 1, so it will pass into the block and output 'foo'. At the second point of iteration, `i` will be 2, then 3, then 4, then 5. So 'foo' will be output 5 times.

The other common iterator found in programming languages is called a **`while`** loop:

```
i = 1;
while (i <= 5)
{
     print("foo");
     i++;
}

. . .

i = 1;
do
{
     print("foo");
} while (++i <= 5);
```

As you can see, there are two variations, each useful depending on the situation. The general meaning of the code should be clear. Both loops output 'foo' five times. In the first loop, the control code is at the start of the iteration, in the second (a '`do while`' iteration) it is placed at the end.

> This example introduces a neat bit of concision common to all C-like languages, the 'plusplus' notation. Instead of saying `i = i + 1`, we can express the same thing concisely

> with either **`i++`** or **`++i`**. Both add 1 to the value of **`i`**. However, **`i++`** means 'run the code in this line then increment **`i`**' and **`++i`** means 'increment i and then run the code in this line'. Generally, **`i++`** is better, being more readable. In the do-while loop, though, observe how the value of **`i`** must be incremented before the test in the control code. Here, **`++i`** is the one to use. Anyway, therein lies the witty joke behind the name C++. (Was the beta design called ++C?)

6.3 Branching

The third pillar of programming is branching. Consider the following code:

```
// get my winnings!
winnings = casino.GetBalance(me);
// invest them!
myaccount.Invest(winnings);
```

This code uses **objects**, which we encountered in the last chapter. Don't worry yet about what exactly an object is, the code here is easy enough to understand in itself. The casino object has code that sends back the user's balance and the account object has code that invests the specified amount. The code thus retrieves the user's casino balance and invests it into their bank account.

It should be obvious that there is something profoundly wrong with this code. Since when are casinos associated with winning? Even if people do

sometimes win at casinos, this code assumes they do without fail.

This is where *branching* comes into play. Without branching, the problem with code above would be unsolvable. With branching, the solution is obvious:

```
casinoBalance = casino.GetBalance(me);
if (casinoBalance > 0)
{ // "then"
     myaccount.Invest(casinoBalance );
} // "endif"
```

A branch can be thought of as an **`if-then`** statement. A lot of programming languages in fact use these as keywords, Pascal for example:

```
if (casinoBalance > 0) then
begin
     // code
end;
```

> This shows how elegantly concise C-like languages are, for they remove the need for a 'then' statement whilst retaining full readability.

The `if` statement is the most obvious example of the incorporation of **Boolean logic** into computing. In essence, this logic reduces to *statements* that are either *true* or *false*. So, from a Boolean point of view *The cat sat on the mat* is either true or false. If the cat is or was sitting of the mat, it is true; otherwise, it is false.

> Note here how a Boolean statement needs to be very precise. 'The cat sat on the mat' is very imprecise. For example, it doesn't specify which cat, nor when. If *any* cat *ever* sat on the mat, the statement could be considered true, but that is not what the English sentence means: *that* implies a specific cat is or was just now sitting on the mat. If a mat was found in the great palace of Ur, and a cat sat on it 5,000 years ago, no one would use *that* phrase to express this.

Because there are just two values in Boolean logic, the values '1' and '0' are often used [4]in programming (where the universal convention is the '1' means 'true' and '0' means 'false').

But with a `bool` type in a language such as C#, it is perhaps better to think in abstract terms simply of *true* and *false*.

Summary

These three pillars of programming — sequence, iteration and branching — sound so simple, but they are so fundamental that they can express practically anything. Of course, while learning to program merely begins here, you now understand how a program is structured at the most fundamental level.

[4] The single characters 'Y' and 'N' are also often used.

7. Mid-level: from object

The three pillars of programming have taken us a long way, so now it is time to examine general everyday-level coding concepts that are only slightly less fundamental.

7.1 object

C# is an **OOP** (Object-Oriented Programming) language in which *everything* is an object. The following code is illegal:

```
// start of program illegal.exe
int x = 1; // ILLEGAL!!!
// rest of code omitted
```

The x *variable* here is 'loose'. It does not belong to an object. To avoid such looseness, you must write this instead:

```
// start of program legal.exe
class Legal
{
     int x = 1; // LEGAL!!!
     // C-like Main() omitted
     // rest of code omitted
}
```

That's better. The x is now safely tucked away within a class.

The base on which C# is built is called **`object`**. Everything in a C# program is at root an object. An object is created with the `new` keyword like this:

```
object o1 = new Object();
object o2 = new object(); // equivalent
object o3 = new(); // newer shortcut syntax
```

`Object` is also a **type**, the root type. Tautologically, in the code above `o2` is an object of type `Object`. It is a variable, so it is also a variable of type `Object`. As `Object` is important rather than interesting, let us quickly move on to the variables and types that programmers actually use.

7.2 Variables and Types

At the lowest level, C# is **variables** which come in various *types*. You already know that data points to a memory location in the machine. So, if you read the C# code

```
var x = 1;
```

you should know it means that `x` references a memory location that contains a value of '1'. However, in a high-level language that is too simplistic, as the following code illustrates:

```
var x = 1;
x = "foo"; // ERROR!!!!
```

The `var` keyword in C# is a sometimes-useful way of declaring a variable. It essentially means 'any', so `var x` means that `x` can be of any type *at the point of*

declaration. Once declared, however, the type of `x` is fixed. If you write `var x = 1`, then x is an integer. You cannot later assign a string to it. If we change `var` to `int`, the code looks a lot clearer:

```
int x = 1;
x = "foo"; // OBVIOUS ERROR!!!!
```

It is important to understand that `int` is an object, so we can if we want declare it like this:

```
int x = new int();
int y = new Int32(); // equivalent
```

> The declaration `int x = 1` is what is known as 'syntactic sugar', referring to when a common programming task is provided with a shortcut baked into the language itself.

We should also mention the *constant*. Conceptually, a constant is a variable that can't be changed. So,

```
const x = 0;
var y = 0;
x = 1; // ERROR!
y = 1; // OK!
```

Constants are becoming increasingly fashionable in modern code-think, and with good reason. Constants don't change, bringing a welcome level of predictability to very large programs (we might prefer to call them applications). There is growing consensus these days that if a value *can* be a `const` it *should* be.

7.3 Arrays

You wouldn't get far with just variables, which can only store a single bit of data. Hence the very venerable **array**. Here is array of integers:

```
// verbose
int[] i = new int[3];
i[0] = 1;
i[1] = 2;
i[2] = 3;

// quick'n'easy
int[] j = { 1, 2, 3 };
```

Given what you already know, that code should be easy to follow. You know that declaring `i` points to a location in memory. An array is just a group of i's with a specified size (or **length** in C/C# terminology). So `int[3]` simply means a variable with a 'length' of three `int`'s. An array has an index that starts at zero, so compare:

```
int i = 1;
int[] a = int[1]; // length in the square brackets
a[0] = 1; // index in the square brackets
```

The variable and the array say the same thing. The variable `i` is set to 1, and the array index of 0 is set to 1. So, the variable `i` points to a value of 1, and the first and only element of a points to a value of 1.

The code below illustrates the basic usefulness of arrays:

```
// use a variable here
string single = "Honky Tonk Women";
// use an array here
string[] album = { "Gimme Shelter", "Love In Vain",
     "Country Honk", "Live With Me", "Let It Bleed",
     "Midnight Rambler", "You Got The Silver",
     "Monkey Man", "You Can't Always Get What You Want" }
```

7.4 Operators

Operators are fundamental to programming. Assigning values, arithmetic and testing for equality are all things that need to be done, and frequently, in any program.

Certain operators need no explanation:

```
x = y + 1; // addition operator
x = y - 1; // subtraction operator
```

Some are familiar in function but maybe not in appearance:

```
x = y / 1; // division operator
x = y * 1; // multiplication operator
```

The '=' operator that everyone knows and loves is deceptive though in C-like languages, for it does not mean 'equals'. The code below explains the initially-confusing family of equality operators, including those relevant to Boolean logic.

```
// this does *not* mean 'equals' but something
// like 'assign to' or 'place the value in', so
// here 'assign the value 1 to x'
x = 1;

// the equality sign uses two equals characters '=='
// 'if the value of x is equal to the value of y'
if (x == y) {}

// an exclamation mark means 'not',
// so != means 'not equal to'
// 'if the value of x is not equal to the value of yv
if (x != y) {}

// bool (Boolean) values can be true or false
bool t = true;
// set f to not t: here, as t is true, not t is false
bool f = !t; // 'not' operator

if (!(x == y)) {} // same as x != y
// note: '!' is less readable here but often makes complex
// evaluations more readable

// two ampersands mean 'and'
if ((x == y) && (a != b)) {} // both tests must be true

// two bars mean 'or'
if ((x == y) || (a != b)) {} // either tests can be true
```

There are other operators, but that should be sufficient to illustrate what an operator is and how it is used.

7.5 Threads

Threads are among the gnarliest and most difficult aspects of programming. But this is so because of what they do, not what they are. The idea of threads (threading) is actually very simple and even coding for threads (**multithreading**) has been made easier and easier.

A computer program executes in a **process** created by the operating system. It is this process that streams to the processor all the commands and data contained in the machine code.

Some tasks take longer than others in running code. Say a particular task takes five seconds. With just the one process, the program would freeze until the task ended.

This is the purpose of threads. A thread is a sort of lightweight process — it can stream commands and data to the processor too — spawned by the main process, the process. A process can spawn multiple threads. So, with a thread created, the lengthy task can be delegated to it. Now the thread takes five seconds but the main process is free and the user is happy because the program doesn't freeze.

The code below illustrates that a simple piece of threading code is almost as easy to read as non-threaded code:

```
// async signals that Add is threaded
// 'asynchronous programming' is another term
// for 'threading'
/*
     ( The <int> indicates 'Task' is a 'generic' type, a
     topic discussed in the next chapter, but it is enough
     to understand that 'Task' here is tied to the integer
     type. )
*/
public async Task<int> Add(int i)
{
    // wait for the ++i task to be completed
    return await Task.FromResult(++i);
}

. . .

// wait for a billionth of a nanosecond
int added = await Add(1);

. . .

// non-threaded code
public int Add(int i)
{
     return ++i;
}
```

```
. . .


int added = Add(1);
```

Threading can never be simplified however or made not gnarly. It is easier to create threads nowadays, but they are still threads. That is, if you had five threads running, you have six (including the main process) pieces of code active. If a certain `i` value is floating in and around this code, we need to ensure that its value does not get set by one thread and reset by another. This is one reason for the current enthusiasm for constants. No thread can change a value that cannot be reset. 'Threading issues' are the stuff of a programmer's nightmares. (A programmer dreaming of being chased by a monster is usually far more worried about that threading bug their dream can't solve.)

7.6 Exceptions

Look at this code (well, imagine this code):

```
{
     // imagine a gigantic mass of code
}
```

The more gigantic in mass code gets, the more likely it will generate an **error** at some point. The modern way of dealing with errors is via **exceptions**. An exception can be thought of as *an anticipated error*. Exceptions work with a **try-catch** block.

We can rewrite the above code as follows:

```
{
     try
     {
          // imagine a gigantic mass of code
     }
     catch (Exception e)
     {
          // handle error
     }
}
```

There are many types of exception, so we can handle exceptions in a very granular way. If our program was about processing Mallumps, we could create our own custom exception:

```
catch (MallumpOverloadedException e)
{
     // handle error
}
catch (MallumpNotFoundException e)
{
     // handle error
}
```

This example also illustrates how you can finesse your exception handling with multiple 'catch' blocks.

With exceptions, we conclude our (very selective) look at 'mid-level' coding features.

8. High-level: to Assemblies

This chapter provides an overview of the features of C# that lead to increasingly greater levels of abstraction until, at the assembly level, the program itself is capable of becoming a service for other .NET programs.

8.1 OOP Objects

We have met with objects — they are hard to avoid in modern computing. This section describes custom objects created by the programmer.

The pillars of objects are:

- **inheritance**,
- **encapsulation**, and
- **polymorphism**.

Which is a grand way of obfuscating the fact that objects actually fairly easy to understand at a basic level.

In an earlier chapter, we looked at objects in an abstract way and talked about *hierarchies*, giving as an illustration a hierarchy proceeding from Animal to Mammal to Cat. We said that in this hierarchy Mammal *inherits* from Animal, Cat from Mammal.

In C#, such a hierarchy can be expressed like this:

```
// note: a class has a code block (empty here)
class Animal {} // curly braces: code block!
class Mammal {}: Animal
class Cat {}: Mammal
```

This is quite useless to anyone, so let's add the ability to give an animal a name:

```
class Animal
{
     string Name;
}
```

The variable `Name` is called a **field**. It is a **member** of `Animal`. We can now create an `Animal` **object** (as defined in the `Animal` class) and assign a value to the name field, like so:

```
Animal a = new Animal();
a.Name = "Lion"; // class members are denoted by a dot
```

Here `a` is an object. This object instantiates the **class** `Animal`. We use the word `new` to create an object defined by a class. Here we set the `Name` field to 'Lion'. Members of a class are denoted with a dot: `{ class } dot { member }`.

With the line `new Animal()`, we meet with a class **constructor**. This is a special method called when an object is created:

```
// all classes have a constructor, so this class
// has one even though it has not been coded for
class Animal {}


// this declaration provides an empty constructor
class Animal
{
     Animal()
     {
          // initialisation code here!
     }
}
```

The code below illustrates that an object is very specific about how it can be used. An object is defined by its class and anything not defined there will lead to an error:

```
// ERROR!!! No field 'Car' defined in class
animal.Car = "Ferrari";
```

The following code provides an example of how **inheritance** works. It shows how classes can re-use what is defined in their parent. Here, the `Name` field defined in `Animal` is re-used:

```
mammal.Name = "Goat"; // Name << Animal
cat.Name = "Felix"; // Name << Mammal << Animal
```

A more difficult concept to understand is **polymorphism**. To help here, first consider that nature has produced no less than three sets of flying creatures: birds, insects and bats. Classes can express this fact via polymorphism:

```
class FlyingAnimal: Animal
{
     // to be overridden a method
     // must be marked as virtual
     virtual void Fly() {}
}

class Bird: FlyingAnimal
{
     override void Fly() {} // polymorphic 'Fly'
}

class Insect: FlyingAnimal
{
     override void Fly() {} // polymorphic 'Fly'
}

class Bat: FlyingAnimal
{
     override void Fly() {} // polymorphic 'Fly'
}
```

We create a `FlyingAnimal` class that defines code for flying. We then create classes for the three types of flying animal and we **override** the parent class flying code. Each child class implements its own 'flying code', just as nature intended.

8.2 OOP Methods

A programming language can be said to consist of *things* (variables, constants, objects) and actors.

In OOP, methods are the *actors* (not to be confused with method actors). Methods *belong to objects* and are *defined in classes*.

Compare C# to C, which does not have objects. In C, actors are called '*functions*'. A function can be thought of as a 'free' actor. In C# a method is an actor 'tied' to an object:

```
/*
     In C, all actors are 'free' because there are no objects
     to tie them to. Free actors are called FUNCTIONS.
*/
void login(char[] user, char[] pwd) {
     // implementation here
}

/*
     In C#, all actors are 'tied' to an OBJECT. Tied actors
     are called METHODS.
*/
// 'Login' is a METHOD of the 'user' OBJECT
user.Login("me", "password");
```

If we want the animals of our `Animal` object to be able to start performing actions, we must give the class some methods, which become members of the class.

```
class Animal
{
     string Name;
     void Sleep() {}
```

```
    void Watch() {}
}

class Predator: Animal
{
     void Eat(Prey prey) {}
}

class Prey: Animal
{
     bool Escape() { return true; }
}

// usage

// here we create two OBJECTS from the
// CLASS definitions above
Prey fly = new(); // shortcut for 'new Prey()'
Predator spider = new();

// call the METHODS (actors) of the 'Prey'
// and 'Predator' classes
// note: 'call' is the most commonly-used terminology
// for using methods, so we say we 'call'
// Escape(), we 'call' Eat()
if (!fly.Escape()) // nb 'dot notation'
{
     spider.Eat(fly); // nb 'dot notation'
}
```

Let us pick apart these three lines to understand more about how methods work:

```
void Eat() {}
bool Escape() { return true; }
if (!fly.Escape())
```

The first line contains the curious term `void`. This implies to the unenlightened that there is something up. In fact, the term is an old C one that simply means 'nothing' or more specifically 'has no value'.

The line `void Eat() {}` contains three elements.

- The curly braces we are familiar with, so we know that they form the **block** of the method's code (here empty).

- `Eat()` is the name of the method and the parentheses (here empty) show that it is a method. In between the parentheses is where we put any **parameters**.

- `void` is the **return value** of the method. That is the point of `void` ('has no value'). `void` allows a method to act without responding.

What all this means will become clearer by examining the line `bool Escape() { return true; }`.

- `Escape()` is as per `Eat()`. There are no parameters here either.

- `void` has been replaced by `bool`. This is a value. It means that `Escape()` is expected to respond.

In C (and C#) terminology this is called **returning a value**. This makes the method act like a sort of variable. So just as we have can have a `bool test` variable, we can also have a `bool Test()` method. The method though is what we might term an *active* value, whereas a variable is effectively a *passive* value. The `Escape()` method, far kinder than Nature, responds here with the line `return true`.

- In the line `if (!fly.Escape())`, the exclamation mark is the C# boolean symbol for 'not', as we saw in the previous chapter. If its response decides the fly can escape, it will return `true`, which after applying the `!` ('not') will evaluate to `false`. If the fly can't escape, the return value will be false and the `if` will evaluate to `true`. (Our code here is kind and the fly will always escape.)

The last line to examine is `void Eat(Prey prey) {}`, which treads mostly familiar territory. The method is `void` and not expected to respond, so it has no `return` statement. The new thing here though is the parameter, which is the poor prey. Fortunately, this code too is kind, for the empty code block means this predator can do nothing to its prey.

Methods, then, allow objects to act. The *things* of an object are termed **properties**, which we need to examine next.

8.3 OOP Properties

Our animals can keep a secret if we add the following field to the class definition:

```
string Secret = "secret!";
```

But say your `Animal` class has been released into the wild and coders are busy coding away with it. Given this design of the class, a mischievous coder might write:

```
animal.Secret = "Not any more!"; // goodbye, secret
```

That is not good. The class design has made the secret public. There is no **encapsulation**. The `Secret` property is simply thrown open to any code using the `Animal` object. So rather than a field, it is usually best practice to employ a **property**, for properties enable encapsulation.

In the latest versions of C#, properties can be defined in multiple ways:

```
// simplest type of declaration, more or less
// equivalent to a field . . .
public string Foo { get; set; }
// . . . except we can also have . . .
public string Foo { get; }
// . . . a read-only state field syntax cannot express.
```

```
// wordy property declaration (older syntax)
private string _foo;
public string Foo
{
     get { return _foo; }
     set { _foo= value; }
}

// concise property declaration (new syntax)
private string _foo;
public string Foo
{
     get => _foo;
     set => _foo= value;
}
```

The most important new thing here is the keywords `private` and `public` that have so far been avoided in the code examples. These are **access modifiers**. They describe what can and cannot 'see' the value. A `public` value is accessible outside an object but a `private` one is only accessible within the class code. The secret can now be kept:

```
// DEFINITION
private string _secret = "secret!";

. . .

// USAGE
animal._secret = "Not any more!"; // ERROR! private
```

Now note the keywords `get` and `set`. These belong to properties and are logically enough called a **getter** and a **setter**. At this point, it should be clear what the `_foo` variable is doing. It effectively holds the actual value of the public-facing `Foo` property. The getter simply returns the value of `_foo` while the setter assigns the special variable `value` to it.

This allows us to encapsulate our class because we can now define which properties can be seen by outside classes. If there is no setter, the property is read-only.

```
private float _pi = 3.14;
public float Pi
{
     get { return _pi; }
}

. . .

numbers.Pi = 1.43; // ERROR! readonly! no setter!
float pi = numbers.Pi; // OK! getter exists!
```

Methods can also easily be encapsulated using access modifiers:

```
private void GenerateSecret() {}

. . .
```

```
// secret stays secret here
myclass.GenerateSecret(); // ERROR! defined as private
```

Another important access modifier worth mentioning here is **`protected`**. Look at this class:

```
public class Tale
{
     private string[] _characters;
}
public class FairyTale: Tale
{
     public void EnumerateCharacters()
     {
          // reference to '_characters': ERROR! private!
          for (int i = 0; i <= _characters.Length - 1; i++)
          {
               // ERROR! '_characters' is private!
               Console.WriteLine(_characters[i]);
          }
     }
}
```

Having just `private` and `public` access modifiers clearly isn't enough. A `private` variable or method is inaccessible to descendant classes. That is the purpose of the `protected` modifier. A `protected` item is *visible within the class hierarchy* but not to external code (as per `private` items). So, to fix the code above we can make the following change:

```
// [ Tale parent class ]
// visible to all descendant classes
protected string[] _characters;

. . .

// [ FairyTale descendant class ]
Console.WriteLine(_characters[i]); // no error now!
```

I will finish this section with a simple example illustrating the three key concepts of classes (and objects): inheritance, encapsulation and polymorphism.

```
public class Trickster
{
     // virtual: descendants can override
     public virtual void Deceive() {}
}

public class Spy: Trickster // inheritance !
{
     public override void Deceive() {} // polymorphism !
}

public class Magician: Trickster // inheritance !
{
     protected Device _wand;
     public Device Wand { get => _wand; } // encapsulation !
     public override void Deceive() {} // polymorphism !
}
```

8.4 to Lambdas

Now we are familiar with method members of an object we can look at how C# expands the capabilities of methods with a cluster of language features: the **delegate**, the **anonymous method** and the **lambda**.

First, let's return to a very simple method:

```
int sum = numbers.Add(1, 1);
```

We can also write

```
numbers.Add = 1 + 1;
```

This achieves the same thing and neatly leads us to delegates. We can create a delegate like this:

```
delegate int AddMethod(int: x, int: y);
```

This is a method without a body. It defines a method. Any method having this form can act as a delegate for it. It can be used somewhat like a field:

```
AddMethod Add = new AddMethod( /* see below */ );
```

This line of code says 'assign the memory reference of the delegate we have just created to the 'Add' field variable'. Delegates, we might say, *turn methods into variables*.

Let us create a basic delegate example. Within the body of the implied `Numbers` class used above, we can use our delegate:

```
// create new property with our delegate type
AddMethod Add { get; set; }

// the AddDelegate is compatible with
// the AddMethod definition
int AddDelegate(int: x, int: y)
{
     return x + y;
}

// create a delegate and assign it to the Numbers class
numbers.Add = new AddMethod(AddDelegate);
// call the delegate
numbers.Add(1, 1); // points to AddDelegate
```

This is all quite different to a non-delegated method. The code is the same in this simple example, but an *ordinary* method is static and the code in the class is fixed. The *delegate* method is coded-for *outside of the class*.

> If you need your class to take care of the code, you don't want a delegate and this is generally the case, I'd say always for the core code of any class. But delegates offer the possibility of a good deal of flexibility if needed.

Delegation is made far easier with **anonymous methods**. These are effectively ad hoc delegates that do not need to be declared:

```
numbers.Add = new delegate(int: x, int: y) // no name!!
{
     return x + y;
}
numbers.Add(1, 1); // use the anonymous delegate
```

Lambdas are similar but still more concise:

```
numbers.Add = (int: x, int: y) return x + y; // a lambda
numbers.Add(1, 1); // use the lambda
```

The details of lambdas and their usage are outside the scope of a guide to what code is. But, since their introduction into C#, they are one of the key features of the language.

8.5 Generics

Songs like 'Strawberry Fields Forever' or 'Like A Rolling Stone' are unique. A lot of music though — say Merseybeat or K-Pop or Disco — tends to sound generic. The Mills & Boon publishing company deliberately publishes generic books that ensure its readers make a safe purchase every time. Sometimes the unique is good, but generic is also often the best option. If you want the Mona Lisa hanging in your living room, a generic reprint is your only option.

We have already met with arrays that can store a list of things. In modern programming environments, a list class is much easier to handle than the old array, for it can iterate through its

member via a `foreach` loop without needing an indexer:

```
// old style array: 'for'
for (int i = 0; i <= myarray.Length -1; i++)
{
     /* do something with */ myarray[i];
}

// ArrayList: 'foreach'
// (this is an old pre-generics .NET Framework class)
foreach (int i in myarray)
{
     /* do something with */ i;
}

/*
     // NOTE!
     // arrays in the latest .NET releases *do*
     // allow iteration!
     array[] a // fill
     foreach (int i in a) {}
*/
```

The `ArrayList` class was an improvement, but its initialisation shows that the pre-generics world was not ideal:

```
ArrayList names = new ArrayList();
names.Add("Truman");
names.Add("Eisenhower");
names.Add("Kennedy");
```

```
// etc
ArrayList numbers = new ArrayList();
numbers.Add(1);
numbers.Add(2);
numbers.Add(3);
// etc
```

This code works because the `ArrayList` is being filled with a collection of `object` variables and everything in C#, as we have seen, inherits from `object`. This is error-prone because there is no built-in way of checking the type of an `ArrayList` member.

Enter **generics**. Here is the equivalent generic code:

```
List<string> names = []; // [], minimal list declaration
names.Add("Truman");
// . . .

List<int> numbers = [];
numbers.Add(1);
// . . .
```

This might not look much at first, or seem much different to the `ArrayList` version. The big deal lies within the angular brackets, for this is the generic stuff.

Notice the `<string>` attached to the first list and `<int>` to the second. Here is the definition of a generic list:

```
public class List<T>: ICollection<T>
```

`T` here stands for 'Type'. Translated further, it means 'of any type'. That is, `T` can be replaced by the name of a type such as `string` or `int`. You can write your own class and use it with the list: `List<MyClass>`.

What about the `ICollection<T>` to the right? We have seen that classes inherit and that to express this we write `Mammal: Animal`. The `I` in front of `Collection` tells us this is an **interface**. If the item to right is an interface (not a class) the class being defined must implement the interface. Note that the interface like the class has those angular brackets enclosing a `T`. '`ICollection`' is thus a *generic interface*. Note too how the `<T>` is a match for the class declaration.

But what is an interface?

8.6 Interfaces

In C# an interface can be thought of as *a codeless class-like definition for a single task* (for example *iterating* or *sorting*).

> Note that the latest iteration of C# has for good or ill added the ability to add code to an interface definition, a fact we will conveniently ignore here.

Just as a class defines the form of an object, an interface defines what is effectively a *class fragment*. Any class that implements any interface *must* implement *all* of the interface. This

means that, however large and complex the implementing classes are, they can all be reduced to the members defined in the interface and in this respect *are therefore identical*.

An interface definition is shown below:

```
// create an interface; the 'I' prefix is not enforced
// by the language, but is in universal use
public interface IPlay
{
     void Play();
}
```

It is a typical interface in its minimalism. Interfaces ought to be focussed. Below is its implementation:

```
// implementation 1
public class FootballMatch: IPlay // ':' means 'implements'
{
     // play a game of football
     public void Play() {} // MUST have a Play() method!
}

// implementation 2
public class Child: IPlay
{
     // childsplay
     public void Play() {} // MUST have a Play() method!
}
```

```
// implementation 3
public class Stereo: IPlay
{
     // play a tune
     public void Play() {} // MUST have a Play() method!
}

// create a 'consumer class' for the interface
// all classes implementing IPlay look identical
// so any implementation of the IPlay interface
// is ok here
/*
     Note: 'Player' has a 'primary constructor' (ctor). The
     parameter is passed in at the very top of the class
     definition, so we do not need to create a default
     ctor within the class body.
*/

public class Player(IPlay item) // primary ctor
{
     // 'item' passed in via ctor
     private IPlay _item = item;

     public void Play()
     {
          // play football or childsplay or play music
          _item.Play();
     }
}
```

```
// run the player with each IPlay implementation
Player player = new(new FootballMatch());
player.Play(); // nil-nil draw

// note this replaces the old player
player = new(new Child());
player.Play(); // lot of mess

player = new(new Stereo());
player.Play(); // tuneless warblings
```

Here an `IPlay` interface is declared and implemented by three classes. A consumer class `Player` is created that can run any `IPlay` object. This means that `player.Play()` will perform very different actions depending on whether it is a match or a child or a stereo doing the playing. The only thing it knows is that something is playing.

An important point to emphasise is that an interface has absolute tunnel vision. As stated above, no matter how many properties and methods the `Stereo` class actually defines, as an implementer of `IPlay` and as an instance of `IPlay` it has only the single `Play` method. So,

- a class may have many methods;
- an interface at most a few;
- a class masquerading as an interface has these few only

A well-thought-out interface is worth its weight in `IGold`. The corollary being that a bad interface is certifiably `ILead`.

```
public interface IWhy
{
     void LeafThrough(Book book);
     void Move(int amountX, int amountY);
     void Tidy(Room room);
     object Search(string text);
     void Wait();
     bool IsPerpendicular { get; }
}
```

I mean, why?

8.7 Components

With components we are going up a notch in the hierarchy of complexity:

```
string > array > object (class) > component
```

A component is easiest to understand if we consider user interfaces. UI is buttons and tab panels and check boxes and text boxes etc. To build a UI, we add these to our interface, whether Web or Windows. In fact, a text box is an example of a component. What is a text box? A rectangular region that both displays text and allows text to be entered. The point about a text box is twofold. First, a text box is a text box is a text box. It is a component. It is entirely *predictable*. Every text box is exactly the same. However, a text box has properties. These properties can be set.

Background colour for example. So, to revise our bold statement, 'Every text box has exactly the same properties'. It is the same but it may not look or act the same, which is of course what makes it useful.

That is a component. The text box itself is a built-in component. But you the programmer can create a component too. What about a `Quacker`? A `Quacker` has a button that when pressed emits a quacking sound. It also has an image of a duck. The two basic properties of a `Quacker` component are therefore the image and the quack. What more does a quacker need?

The `Quacker` is a custom component. Once you have written such a component, other applications can use it and add it to their UI.

A component is, therefore, a runnable piece code designed for use in any .NET program. It is created in one .NET project and consumed in other .NET projects. The general principle being that a component is universal. A component is available to all of .NET.

8.8 Assemblies

The last thing to look at is the **assembly**, the end result of all .NET projects. The assembly is the next step up from the component.

The assembly is a synonym for '.NET program', of which there are two types. In Windows terms, an

.exe file is an **executable**. This means that it entitled to start up a *process*, a running Windows program. The other type is a **DLL** ('Dynamic Link Library'). A DLL is basically an executable that cannot start a process. An exe is a primary program that can execute, a DLL is a secondary program that can execute inside an exe. The DLL is *linked* to the exe and executes inside it.

> The latest versions of .NET in fact greatly blur the difference between EXE and DLL and always output an .exe file. But the principle remains that an EXE *runs itself* and a DLL *runs inside*.

The difference between a component and an assembly is that a component is an emphatically discrete thing. If `Quacker` is a component, a related assembly might be `Calls.dll` and contain a set of components (`Quacker` itself, of course, but also `Woofer`, `Meower`, `Brayer`, `Snorter`, `Bleater`, `Mooer`, `Rivetriveter` and many more).

Assemblies though are far too big to provide one-line examples for, as are components. We have therefore advanced as far as we can in our guide to 'what is code'. What is an assembly can only be explained in general terms.

As we have gone as far as we can with C# and code that *does*, we can now move onto the topic of data and code that is *done to*.

9. Data

What is a program without data? and what is data?

It was noted at the beginning of this guide that machine code is all command and data. With this definition, data is almost everything in computing. On the other hand, data can be merely 'what is stored in a database'. This first definition above is too far-reaching to be useful here and the second far too narrow.

To advance our enquiry, we can compare data to memory. Without memory, we humans would move from moment to moment entirely unable to remember anything that had just happened. Just so with a program. We run it, use it, shut it down, and then what? What if even a simple text editor had no memory? We would write a text, close the editor, lose our work.

This gives us a key to a definition of **data** that we can use here. We can agree that the document in both the running program and the saved file is in a sense the same document, but if we keep with the analogy of data as *memory*, we observe it has two states: *active* in a running program and *static* in a stored file. Although we can speak of 'data' in either state or indeed both, this chapter defines data purely in its stored form. Data here is the memory that persists after a program has been shut down.

The main purpose of this chapter is to explore beyond 'active' programming languages like C# and give a brief overview of the equally-important 'static' languages that deal with data, languages quite unlike C#. First, as databases are most obviously associated with data, we turn to SQL ('Structured Query Language').

> SQL is usually pronounced 'sequel' but 'ess-cue-ell' is also in use and won't raise eyebrows.

9.1 SQL

Alternatives to SQL do exist nowadays (NOSQL for example, would you believe?), but for a long time SQL ruled the database roost.

SQL has a mathematical basis in *relational algebra / calculus*. For this reason, a SQL database is often called a **relational database**. To acquire a basic understanding of SQL, you have to imagine a grid of data. This grid is called a **table**. The grid consists of **columns** and **rows** and is where data is stored. Each item of data is contained in a row. It is the job of SQL to fetch data from the grid by **querying** its columns and rows.

Here a table definition in SQL:

```
CREATE TABLE Animal(
     /*
          A NULL column does not have to have a value,
          but a NOT NULL must have a value when a new
          row is inserted into the database.
     */
     Everyday_Name varchar(255) NOT NULL, -- =string
     Scientific_Name varchar(255) NOT NULL,
     Is_Predatory char(1) NOT NULL, -- =bool
     Number_Of_Legs int NOT NULL
)
GO -- run the SQL
```

This creates a table and its grid columns. The table is like a class without any methods. There is no *active data* in SQL and there are no methods.

> There are no methods in SQL, but there are 'stored procedures', which are functions that run SQL commands on a SQL server.

The rows in the grid are the data and they are retrieved by a SQL **query**:

```
-- the asterisk means 'all columns'
SELECT * FROM Animal
```

This will select all the rows in the `Animal` table. A query can, however, select only particular rows and columns:

```
SELECT Everyday_Name -- specify columns here
FROM Animal
WHERE Everyday_Name = 'Lion' -- specify rows here
```

This will select the column `Everyday_Name` and all the rows from the `Animal` table where the row has the exact value of 'Lion' for the `Everyday_Name` column. This should of course be a single row here.

Now take another look at the table definition:

```
Everyday_Name varchar(255) NOT NULL,
Scientific_Name varchar(255) NOT NULL,
Is_Predatory char(1) NOT NULL,
Number_Of_Legs int NOT NULL
```

There are two issues here that usefully illustrate what SQL is and does.

First, the `NOT NULL` declarations. These say that all of the columns *must* have a value. This might be what is wanted. It is difficult to see why any of these should be left blank unless there was some data-input error. However, this is only an option if you have full control over the data. In many cases, a database imports its data from an external source. Here, making any but the `Everyday_Name` column `NOT NULL` would be a very bad design because, with no control over the data, there can be no guarantee there will not be bad data. Only the key column, the `Everyday_Name`, could be `NOT NULL`, for without the key value a record is meaningless. However, even then it might be better to allow bad data to be imported and then fixed rather than refused entry into the database.

The second issue lies with the key, the `Everyday_Name` column. If there were two records with the same

name (for example 'Bear' was entered twice), the database would effectively be corrupted. This can be fixed:

```
Everyday_Name varchar(255) NOT NULL UNIQUE,
Scientific_Name varchar(255) NULL,
Is_Predatory char(1) NULL,
Number_Of_Legs int NULL
```

This would probably be acceptable here, as after all animal names are unique. But if you were storing data about your record collection, and your record collection is very extensive, there would likely be a problem:

```
-- Album_Title is not a good key!
Album_Title varchar(255) NOT NULL UNIQUE,
Artist_Name varchar(255) NOT NULL
```

You look through the thousand or so items in your catalogue and notice you have '*Greatest Hits*' by *The Chancellors* and also '*Greatest Hits*' by *The Shaggs*. You simply cannot have a `UNIQUE` name here. Here, **primary keys** come running to your rescue:

```
-- now we have a proper key!
Album_Id int NOT NULL PRIMARY KEY,
Album_Title varchar(255) NOT NULL,
Artist_Name varchar(255) NOT NULL
```

With a primary key in place, the database will generate a sequence of numbers automatically each time a new record is created. A primary key is therefore guaranteed to be unique and the duplicate

album titles can now be added to your database without any problems.

But there is still a problem with the `Artist` column. As you have every Bob Dylan record ever released and also a number of bootlegs, the value 'Bob Dylan' be duplicated in a great many database records. Such duplication is not the relational database way of doing things. Instead, to be relationally-correct you must create a second `Artist` table with its own primary key and then give the `Album` table a **foreign key**.

```
-- 'Album' table
Album_Id int NOT NULL PRIMARY KEY,
Album_Title varchar(255) NOT NULL,
FOREIGN KEY (Artist_Id) REFERENCES Artist(Artist_Id)


-- 'Artist' table
Artist_Id int NOT NULL PRIMARY KEY,
Artist_Name varchar(255) NOT NULL
```

You can now fetch back data using a **join** in which the `Album` and `Artist` tables are 'joined' via the `Artist_Id` key.

```
SELECT alb.Album_Title, art.Artist_Name
FROM Album alb
INNER JOIN Artist art on alb.Artist_Id = art.Artist_Id
```

Now all your catalogue items are unique and there is only one Bob Dylan. As it should be.

SQL quickly gets very complex, but it always works within the simple concepts illustrated above, so I think that is just enough to enable us to understand what SQL is and does.

As far as deployment into 'active' languages such a C# goes, a longstanding issue with databases has been the disconnect between C# and SQL and how to fetch SQL data into a program. There are now sophisticated means of doing this but nevertheless, SQL is not a lightweight option for storing data. The database must be designed, configured and deployed and of course a SQL server needs to be installed and running. A database is a good choice for a large program, but the smaller a program is the less this is so and a text-file based form of storing data begins to look a better bet.

9.2 XML

XML stands for 'Extensible Markup Language' and it is, coincidentally, a markup language. What is a markup language? Markup structures plain text. In XML terms, markup is achieved using **elements** and **tags**. An element consists of a start tag and an end tag:

```
<!-- this is an element -->
<!-- <tag> is the opening tag -->
<!-- </tag> is the closing tag
<tag>DATA</tag>
```

A tag is enclosed within angular brackets. The name of the tag sits within the brackets. In the closing tag, the name is prefixed with a forward slash (/).

An XML equivalent of the SQL table we defined earlier might look like this:

```
<animals>
     <animal>
          <everyday_name>Tiger</everyday_name>
          <scientific_name>Panthera tigris</scientific_name>
          <is_predatory>Y</is_predatory>
          <number_of_legs>4</number_of_legs>
     </animal>
     <!-- other animals follow -->
     <!-- equivalent to SQL rows -->
</animals>
```

XML is usually stored in a text file (though it can be stored in a database). XML file data can be read by an XML parser and most 'active' programming languages have one of these. XML is very easy to read in C#, for example.

It should be noted that there is a family of XML technologies. XSLT ('Extensible Stylesheet Language Transformations', if you will) transforms one XML document structure into another XML document structure. So,

```
<person>
     <name>Lord Lucan</name>
     <location>
```

```
          <place>Missing</place>
     </location>
<person>
```

might be transformed to the conciser:

```
<person name="Lord Lucan" whereabouts="Missing" />
<!--
    The values in the quotes are known as ATTRIBUTES.

    A lot of XML data can be stored as either tag
    content or in an attribute. It is a fine art
    to decide which is better. In this case, where
    concision is clearly a priority and the data
    is always going to be a short bit of text,
    attributes appear to be good to go.
-->
```

Another important member of the family is XQuery (featuring in the star role XPath). This allows you to 'query' (as per SQL) an XML document.

However, the basic fact is that 'active' languages like C# pretty much exclusively interact with XML itself (though their parsing technologies are likely to use XPath).

To conclude, then, as a simple text file XML is far easier to deploy than a SQL database and its data is easier to read into a program. However, it is clearly less suited for larger datasets.

9.3 JSON

JSON ('JavaScript Object Notation') is a subset of JavaScript, a C-like language that is used ubiquitously for web development. JSON is, basically, the data bits of JavaScript and a JSON file can be thought of one large flub of declarations.

Our data would look like this in JSON:

```
{
  "animal": {
    "everydayName": "Tiger", // a string value
    "scientificName": "Panthera tigris",
    "isPredatory": true, // a boolean value
    "numberOfLegs": 4 // an integer value
  }
}

/*
// for comparison purposes, here is the JSON
// written in 'proper' JavaScript
const animal = {
    "everydayName": "Tiger",
    "scientificName": "Panthera tigris",
    "isPredatory": true,
    "numberOfLegs": 4
  };
*/
```

JSON can be read into any programming language as **serialised** data. To achieve this in C#, you would

create a class that matches the JSON data values, the target for the serialisation:

```
// C# class to match the JSON data
public class Animal
{
     public string everydayName { get; set; }
     public string scientificName{ get; set; }
     public bool isPredatory{ get; set; }
     public int numberOfLegs{ get; set; }
}
```

The JSON data can now be **deserialised** into the C# class. Assuming a JSON file containing a list (array) of animals, the code to deserialise is simple:

```
// the reader method reads the text file
// of JSON data
string jsonAnimalsData =
     MyAnimalsReader.ReadJson(jsonFilePath);
// the JSON data is fed into the deserialiser,
// which converts each item into an object of
// the Animal class and adds it to a Generic
// list of Animal objects
List<Animal> animals =
     JsonSerializer.Deserialize(jsonAnimalsData,
          typeof(List<Animal>)) as List<Animal>;
/*
     NOTES.
     1
     'typeof()'. This is a special C# operator that
     takes in an object and returns its type. C#
```

```
    has a class called Type that expresses type
    information for an object and the Deserialize
    method expects an object of class Type as
    the second parameter.
    2
    ‘as’. Classes can be 'cast' with the 'as'
    operator. The ‘Deserialize’ method has a
    return type of 'object', which means it can
    return any class type. The return value
    therefore needs to be converted (which
    is what 'cast' means) into the type we
    want. The deserialiser, note, will create
    a list of animals. We must cast to this
    type or an error will be thrown.
*/
```

As for serialisation and deserialisation, this refers to the process of creating an object from its class definition (serialisation) or a class definition from the object (deserialisation). So long as language A (here JSON) and language B (here C#) have identical class definitions, an object can be passed from one language to the other.

JSON makes it programmatically very easy to store and read data stored in an object. The downside is that it is less easy for humans to read than XML.

9.4 YAML

Another way of storing data is YAML ('Yet Another Markup Language'). Our data looks like this in YAML:

```
animal:
  - everydayName: Tiger
  - scientificName: Panthera tigris
  - isPredatory: Y
  - numberOfLegs: 4
```

YAML was intended as an easy-to-read format, but nowadays has a bad reputation for a complexity, making it highly error-prone and hard to maintain. However, if the YAML is kept simple it does I think keep its original promise. The YAML is easier to read and write than the JSON, less verbose than the XML and far simpler to deploy than a SQL database.

As with JSON, YAML can easily be serialised and deserialised in C#.

9.5 HTMLF

There are edge cases where the best option might be to roll your own data format. Here is a case study from my personal website.

The site contains a set of overview texts that summarise their page content. To make maintaining the overview content easier, the text is stored in an external file that needs above all to be easily

editable. Evaluating the existing options, XML is simultaneously overkill and fiddly to work with, JSON is not designed for editing data and YAML is quite unsuitable because yamelised stored HTML needs to be carefully indented making editing difficult. So, the HTMLF (HTML Fragment) format was born:

```
@begin/
Tiger
<p>Panthera tigris</p>
/end@
@begin/
Tortoise
<p>Testudinidae</p>
/end@
```

HTMLF simply stores markup fragments and their identifiers between `@begin/` and `/end@` markers. It is an example of a data format that is little, limited and local and it is easy to parse and use:

```
HtmlFragments fragments = new(htmlfPathString);
HtmlFragment fragment = fragments.Find("Tiger");
string tigerFragment = fragment.Html;
```

Summary

All computing and every computer program is a mix and match of action and data. Data can be defined in many ways. One way is to distinguish between its *active* state (when it is in a running program) and its *passive* state (when it is stored after the program is shut down). Without this world-outside-the-

program, computers would have no conceivable use. Stored data is the long-term memory of a program and thus all programming is much-concerned with stored data.

Outside of C# and comparable 'active' languages, languages that deal with data — SQL, XML, JSON, YAML, dare I say HTMLF, etc — are equally part of any programmer's remit.

In a nutshell, coding is as much about *loading* and *storing* data (in XML, say) as *running* data (in C#, say).

10. Web

The World Wide Web entered the world it is named after in the year 1993, though internal development (within particle-colliding CERN) began back in 1989. As this momentous event happened over three decades ago, it seems not quite right to call it a revolution. No revolution lasts that long. Perhaps we can better think of it as an '*evolving revolution*', its basic principles leading to a continuous sequence of revolutionary changes.

Whatever, no one would deny that the www has revolutionised not only computing but its namesake the world.

In computing terms, the www has changed the very nature of computing so that as web sites become more like programs, a program runs less and less *for* a computer but *in* a computer: for web sites run *in* a web browser that runs *on* a computer. A web site can be viewed on a PC, a tablet, a phone, a watch or a TV. Same site, different worlds. In the Overton Window of computing, this is perhaps nowadays the *worlds wide web*.

So what, we ask, were the origins of the www revolution?

10.1 Protocol + Language + Reader

There was an internet before the www. The TTY teletype was an interactive system, but much of the early internet was focused on files. Files could be exchanged via FTP ('File Transfer Protocol'), communications could be exchanged via electronic-mail (in which files were sent to the specified email address) and in message boards, where messages (in other words, files) were sent to a message board server and read by message board members.

This was the environs in which our revolution took place, and the revolution revolved around a **language** (HTML, 'Hypertext Markup Language'), a **protocol** (HTTP, 'Hypertext Transfer Protocol') and a **reader** (the browser). Underlying all these was **hypertext**.

Hypertext is built upon the brilliant insight that computer text is potentially more flexible than everyday text in a book or a newspaper. Computer text can link to other texts and thereby build up an information network. The problem before HTML lay in building up such a network. Hypertext software ran on a computer and the information network had to be contained within the hypertext program. This ensured that the network was always limited.

So to HTML and markup. We have already met with the markup language XML. This is derived from SGML ('Standard Generalized Markup Language'), a markup language used to generate other markup

languages. HTML (like XML) was generated from SGML.

HTML had two main features. First, it contained markup elements with which to create and format fairly sophisticated documents. Second, there was the **anchor** and its **href** ('hypertext reference'), which is where we find the hypertext.

Here is a simple fragment of 'early-HTML':

```
<!--
  The 'table' is a flexible way to structure content.
-->
<table>
   <tr> <!-- table row -->
      <td> <!-- table data (ie table column) -->
        <!-- specify the look and feel of the text -->
        <font color="lime" face="Ariel">
           <!-- paragraph of text -->
           <p>Interesting fact:</p>
        </font>
      </td>
      <td> <!-- second column in this table -->
         <!-- hypertext anchor with 'reference' (href) -->
         <a href="http://www.interesting.org/fact.htm">
            <!-- 'reference' (hyperlink) text -->
            Hypertext!
         </a>
      </td>
   </tr>
</table>
```

The second plank of the web was **HTTP**. This was how .htm pages were delivered from the server to the client.

The third plank was the reader program for the markup code. This is now universally known as a **browser**.

The job of the browser was to read the HTML markup and display the page according to the markup layout and styling instructions. HTML allowed quite sophisticated layouts that were indistinguishable from a basic word-processed document. Above all, of course, the browser handled the hyperlinks within a page's anchor tags. When the user clicked a link, the browser sent an HTTP request to the specified web address and the server duly responded with the requested page. This was read, parsed, dispatched and finally displayed in the browser, replacing the page that requested it.

It took a little while for the web to catch on because at first it suffered from the same restriction as the earlier hypertext programs, a limited information network. But whereas the limitation of the old *programs* was fixed, the web could expand indefinitely.

One phenomenon that quickly grew out of the web was the **site**. Over at 'www.interesting.org' and its facsimiles, it became apparent the hyperlinks could not only connect remote computers but also files on the local server. A traditional web site is in essence

a set of linked files on a local server. Sites themselves could become informational networks devoted to specialist information. The informational network of the web as a whole grew exponentially in a short space of time, so much so that the www became known simply as 'The Internet'.

10.2 Client Revolutions

There was room for improvement, however, and so began the relentless sequences of 'evolutionary revolution' that has characterised The Internet ever since the www began.

First, it quickly became apparent that the `<font>` tag was a mistake. Ideal for an individual web page or the small sites of the early web, it lead to a maintenance nightmare for larger sites. Changing the style of the site, or keeping styles consistent, were needlessly hard tasks.

The giants therefore invented Cascading Style Sheets (CSS). The revolution here was that styling a document was now essentially external to the documents (though styles could still be embedded if preferred).

With CSS, styling text became easy to do and the site was far more maintainable:

```
p {
     color: lime;
}
```

Including this stylesheet in an HTML page would style every <p>aragraph. But as that is somewhat inflexible, stylesheets also have what are called (as per OOP) classes:

```
.lime-colour {
     color: lime;
}

. . .

<p class="lime-colour">Yuck!</p>
```

You can also style individual tags by applying a style to an HTML id (id's must be unique to a page):

```
#yuck {
     color: lime;
}

. . .

<p id="yuck">Yuck!</p>
```

You can even embed styles directly into a tag:

```
<p style="color: lime;">Yuck!</p>
```

The giants also introduced a second revolution with the ability to **script** web pages. Scripting was discussed in chapter two, so suffice it to say here that JavaScript activates web pages. The structure of the HTML is analysed into a **DOM** ('Document Object Model') and scripts operate on this DOM. Nodes can be added to and removed from the DOM using

scripts, which can also modify any existing node. There is also a **BOM** ('Browser Object Model') that allows scripts to interact with the browser itself.

Another significant innovation was **Ajax** ('Asynchronous JavaScript and XML'). This allowed scripts to download data from the web server. The reason this is a big deal is because it made server communication *granular* or perhaps even *modular*. Before Ajax, the browser requested web pages and the context of the web was therefore the page itself. With Ajax, the page could now fetch data for any part of the page from the server *without reloading the page*.

Pages could now be modified via either the client (by using script working purely within the browser itself) or the server (by making an Ajax call). This is illustrated in the code below:

```
<p id="placeholder">Placeholder Text</p>

. . .

<script>
     // this code searches for the above tag in the DOM
     // using its unique id (id's are marked with a hashtag)
     const placeholder =
          document.querySelector('#placeholder');
```

```
    // CLIENT-BASED mod (code runs on client)
    // getReplacementText: local js code
    placeholder.innerHTML = getReplacementText();

    // SERVER-BASED mod (code calls server via Ajax)
    placeholder.innerHTML = getReplacementTextFromAjax();
</script>
```

10.3 Server Revolutions

With Ajax, we arrive at the server. In chapter two we described the revolution introduced by the gateway on the server. This was the early term during the days of the **CGI** ('Common Gateway Interface') and a scripting language called **Perl**. The term 'gateway' is I think still useful as a metaphor, but is rarely used today when it is enough to refer to the 'server' or 'server-side'.

Microsoft's earliest attempt to control the server was **ASP** ('Active Server Pages', an archaic name dating from the time when the company was excited about the COM term 'Active', tying in as it did with the then-new 'ActiveX' technology). ASP was closely tied to Visual Basic. In ASP you would write code like this:

```
<p id="placeholder">
    <%
        Response.Write 'Replaced!'
    %>
</p>
```

This code runs on the server. The page delivered to the client and viewed in the browser would look something like this:

```
<p id="placeholder">
   Replaced!
</p>
```

Long-superseded by **ASP.NET**, in its time ASP marked a significant advance in server-side programming, for unlike the CGI and Perl dynamic duo, ASP was a unified environment, for example tying in with Microsoft's ADO ('ActiveX Data Objects', that 'active' again) tech that allowed ASP pages to easily connect to databases.

In today's world, we have reached the stage of **.NET Core** and ASP.NET **MVC** ('Model View Controller') or API ('Application Programming Interface') **REST** (wait for it, 'Representational State Transfer') sites.

Hold on, I'll repeat that bit without the meaningless 'keys' to the TLA's.

In today's world, we have reached the stage of .NET Core and ASP.NET MVC or API REST sites. Because Core code is compiled from IL into the machine code of the host operating system, ASP.NET sites are no longer tied to Windows. Moreover, there is now a lightweight Core web server called **Kestrel** which can also be executed on the host OS. Kestrel runs in the host's main web server and because Core sites work directly with Kestrel, not the main web server, they will work

with any system Kestrel can. (Before Core and Kestrel, ASP.NET sites ran only on Windows machines and were mostly deployed to Microsoft's main web server software, Internet Information Server.)

A .NET site can also choose the technology to power the site. MVC and API (REST) have been mentioned, but there is also **React.js** (a Facebook language that reimagines JavaScript in components), which in turn is often tied to **TypeScript** (a Microsoft language that reimagines JavaScript into a less wild coding habitus). What a web application is and does is becoming less clearly-defined. API sites do not necessarily have anything to do with either hypertext or HTML — they return data, often in JSON format, but also more sophisticated technologies such as GraphQL (another Facebook language).

MVC creates traditional HTML web sites but even so reimagines their creation. With MVC the actual HTML is now called the View. The Model is a C# class holding the data displayed in the view. The Controller handles requests for pages (that is, views) and is responsible for sending the right page back to the client (that is, the browser).

Here is an example of a controller:

```
public class MontyPythonController: Controller
{
     [Route("[controller]/[action]")]
     public IActionResult MontyPython()
     {
          MontyPythonModel model = new();
          model.Message = "spam spam spam spam spam";
          return View(model);
     }
}
```

The view in this example is called `MontyPython` and illustrates the MVC naming convention. The controller is called `MontyPythonController` and the model `MontyPythonModel`. The view will be named `MontyPython.cshtml` ('C# html').

When a request for the Python page reaches the web server, it will be routed to the controller. This will create the model and pass it to the view, which is another server-side file that mixes client-side code (HTML, CSS, JavaScript) with server-side C#. Microsoft calls this hybrid technology **Razor**. It is Razor files that are given that 'cshtml' extension.

The `MontyPython.cshtml` page/view might look something like this:

```
@model MontyPythonModel
@{
     // '@' denotes Razor code and @{} a Razor block
     <p>@Model.Message</p>
}
```

As you can see, the model has been passed to the view, which outputs the 'spam' message. Using this web application, any number of Python quotes could be sent to the view and the site could easily be expanded to include a `FawltyTowers` view, though please no not a `Yellowbeard` view.

The MVC architecture blurs, from the programmer's point of view, the distinction between server and client:

```
<script>
     var x = 1; // JavaScript code
     @{
          var y = 1; // Razor code
     }
     // ERROR! y here is an undeclared script variable!
     x = y;

     x = @y;
     // OK, but @y is only assigned on the server;
     // on the client, this is equivalent to x = 1
     // which may be what is intended but - the statement
     // looks like a variable assignment (x = y) and
     // client-side it is *not* a variable assignment
</script>
```

The gateway is still there in fact but is now almost invisible in code.

Another curious side-effect of this apparent hybridity is the appearance of the illusorily concise code:

```
@{
   foreach(var stuff in @Model.Stuffs)
   {
      Html.Raw(@stuff.Html);
   }
}
```

This server code looks concise, but what is in that `Html` property? If the HTML is wordy and there are 100 or say 1,000 items — well if you go to 'View page source' (Chrome, or the equivalent in your favoured browser) you will get an unpleasant surprise.

Finally, a word or two about API/REST sites. REST is built around **HTTP requests**. There is a small set of these and here we will concentrate on the two most common: **`GET`** and **`POST`**. REST ingeniously turns these into data-centric functions, effectively 'read' and 'write'. HTTP requests typically send HTML documents, but if we see what is sent as mere data, we can just as easily pass JSON around. A typical API, then, reads and writes JSON over HTTP.

In .NET Core, there is a good deal of similarity between MVC and API. Both are built around

routing. In MVC, stuff is routed to *views* and in API to the API itself.

As a very simple example of an API that 'writes' JSON data via a `GET` request, this code sets up the routing for the API method:

```
app.MapGet("/animals", [HttpGet] () =>
    MyApiController.GetAnimals());
```

The code behind the `GET` would be something like this:

```
public static List<Animal> GetAnimals()
{
   string json = "";
   string path = "c:\json\animals.json"
   using (StreamReader reader = new(path, Encoding.UTF8))
   {
      while (reader.Peek() > -1)
      {
         json += reader.ReadLine() + Environment.NewLine;
      }
   }
   List<Animal>? animals = JsonSerializer.Deserialize(
      json, typeof(List<Animal>)) as List<Animal>;
   return animals ?? [];
}
```

(I leave it as an exercise for the reader to decode the code.)

Finally, a client program can retrieve the data from the excellent site specified in the map:

```
https://www.imaginary.com/animals
```

10.4 Distributed / Cloud

Moving from the apparent merging of client and server, it is appropriate to finish with a brief note on distributed computing and the cloud.

A client/server relationship is traditionally between computer and computer. One computer is the client and the other is the server. In distributed computing, many machines are involved. How many may vary, but by definition it is 'many'.

The latest web trend is programming for this distribution. Let us take for example a process distributed over three machines. X machine, Y machine, Z machine. Let us say that a key piece of data is the Turtle class (it is after all turtles, all the way down). There is a turtle called Bertha and the Turtle class stores all the information about Bertha. Should machines X, Y and Z share information about her? One influential view responds with an emphatic 'No!'. This might lead to a loss of data, or perhaps worse corrupted data. Each machine needs to be separate from each of the others and in that way preserve the integrity of its data. Even if one or even two machines fails, the surviving information about Bertha on the third machine will have retained its integrity.

This loose and unlocalised distribution of data seems to be where at least a great part of computing is

heading, a nebulous mass of machines anchoring The Internet. No one machine can afford to be an independent voice, it must be part of a larger choir. Data must retain its integrity, but it must do so inside the choir. Even to set `x` to `1` is a hard problem, for the simplest of data still needs data integrity.

A cloud, meanwhile, is *precisely* a nebulous mass of machines. Nebulous means 'cloudy'. Data in the cloud is lost in the cloud. Programming is programming for the cloud. Programs run on any machine in the cloud.

What is programming? In the cloud, where the program itself leaps from machine to machine, perhaps this is the beginning of a new answer to the question of what is programming and the beginning of a new computing.

At this new beginning, then — we end.