How Computers Think

Computers, explained.

by Sean Timarco Baggaley

16 September 2014 - r2.7

Contents

About this book

This book was written for primarily selfish reasons:

Firstly, I'm one of those people cursed by the phrase "good with computers". This seems to accompany every introduction to new family friends and relatives. As a result, I've had to learn everything from video editing to audio production, programming to graphic design—even copywriting and web design—all thanks to the cardinal sin of knowing which end of a computer the power cord plugs into.

The second reason is to find some way to explain technology in a form even my own mother could understand. The idea being to write something she would *have* to read, because her own son wrote it. If it works, then I'll know I've succeeded. Hopefully, she'll even write a review on Amazon…

The third reason is not selfish: we expect computers to do what we *mean*, not what we *say*...

The Anthropomorphism Problem

Humans have a habit of projecting human traits and attributes onto non-human subjects. This has helped us in the past, particularly with domestication of animals. The same habit has also given us multiple gods of thunder, of the sun and the moon, and more religions than most people can name.

This same habit can also be seen whenever you talk to someone who works with steam locomotives: despite knowing that it really *is* just a machine, many steam engine fans will tell you that these machines have individual characters. Similar anthropomorphic tendencies can be seen even among car owners, who often give their car a pet name and even talk to it as if it can understand them.

The problem with this habit of projecting human traits onto inanimate objects is most obvious with computers and associated technologies. We assume that a computer is self-aware enough to discern meaning and intent from our requests.

This is fundamentally a *design* flaw.

Our brains attempt to build mental models of how stuff works to help us understand how to use them. This process usually works fine: we look at a push-plate on a door and know we must push to open it. If we see a grab-handle on the door rather than a push-plate, we will try to pull on the door instead.

A good designer will know this 'model-making' part of our brain and will design accordingly.

In design, we refer to the handle or push-plate as an *affordance*: a design element chosen not only for its function, but also because it *explains its own function*. There is no need for an instruction manual. We know that a flat brass plate on a door means we need to push the door open, even if it doesn't have "PUSH" stamped on it.

We learn how to recognise such basic affordances as children.

The problem with is that computers are invariably opaque devices. They offer no visible clues as to how they *really* work, so we are forced to rely on the software—the programs—for guidance. But that software isn't designed to explain how the computer works either. It's designed to explain how the *software* works!

So most of us have wildly inaccurate mental models of what a computer actually does all day.

Which is where this book comes in.

Introduction

The electronic computer has been with us in some form or another since the 1940s. While most creatures grow with age, computers have shrunk, from massive, room-filling affairs to tiny little devices we can keep in our pockets or wear on our wrists.

And yet, after all these years, most of us still have no idea how to truly *communicate* with a computer and get it to do *exactly* what we want it to do. All too often, we fumble about in the dark, trying to get our technology to work. We swear at our PC when it crashes no apparent reason. Why are there all these toolbars in my web browser? (What *is* a "web browser"?) Why is my PC getting slower and slower? Why is it asking me about a "driver update"? What *is* a "driver"? There's no car in my PC! Who came up with all this confounding jargon? What does it all mean?

There is only one answer to all this confusion: understanding how computers work. How they *think*. We spend so much time working with computers, yet few of us really understand how to *talk* to them; how to *understand* them.

So… this is a book about how computers *think*. And to understand how computers think, you need to learn what a *computer program* is.

Wait! Don't run away! Programming is nowhere near as difficult as most people think! In fact, computers are very, very simple machines, so programming is fundamentally simple too—almost *too* simple, as I'll explain later.

Most books will try to explain how computers work using mathematics. They will dive straight into binary this, CPU that, buses, RAM, ROM, and so on.

But programming is really just *translation*.

17 May 2014

Yes, there is a little bit of basic maths and arithmetic involved if you want to get very deeply into programming, but this is only because computers are often tasked with doing mathematical jobs.

If you can speak even a few phrases in a foreign language, you can certainly learn how to program, because computers are nowhere near as complicated and hard to understand as an Italian.

Unlike humans, the science behind computers is very well known: we understand how they 'think'—if they can be said to do so—and that's the key to understanding them.

Are you sure about that maths stuff?

I swear, I won't mention lambda calculus or matrix mathematics outside this sentence! Computers are surprisingly *bad* at mathematics out of the box: they have to be taught how to do it, just like we do. And computers are taught by being programmed.

In fact, one of the biggest myths surrounding computers is that programming them requires a deep understanding of mathematics. The thing is, mathematics is *just another language.* (Technically, it's a group of related languages.) So, saying "computers are all about maths" is like saying "computers are all about French": both are true, but both are also meaningless.

> Mathematics is mostly about using language to manipulate data. It is, in some respects, a form of programming language in its own right, albeit not one designed to be easily typed. Different dialects of mathematical language have appeared over the centuries, so we have algebra, differential calculus, vector and matrix mathematics, and so on. But they're all part of the same family of man-made languages.

We *all* learn to translate our own thoughts and ideas into language that others can understand. That's what we spend our first few years of life doing.

Now, I'm not saying translation is for everyone, but many English speakers already know a few French words: the English language is nothing if not a hoarder of vocabulary, and it has hundreds of thousands of words, many of which were derived from ancient Greek, Latin, Old French and bits of old German, not to mention more recent imports from India, Africa and Australia.

> The closest we have to a 'human' programming language is the one colloquially referred to as "Legalese".
>
> A good law is one that makes everything crystal clear with as little ambiguity as is humanly possible. That's why the first part of a complex contract contains long lists of definitions of the terms it uses. They might not be that clear to laypeople, but they're very clear to lawyers.
>
> The laws of a country effectively define its '**operating system**': the foundation on which all else is built. We'll come to operating systems later.

So, yes, if you want to learn—and the mere fact that you are reading this suggests you do—you can learn how to communicate with a computer too. And that means understanding what programming is all about.

You won't become an expert in programming with this book, but programs are 99% of what a computer does, so we do need to look at it.

But first, a bit of history…

The age of clockwork

In a museum in Neuchatel, Switzerland, sit three automatons. These clockwork robots look like three dolls: one is a little girl at an organ; the other two dolls are both small boys. These are spectacular examples of clockwork *automata*, created and built between 1768 and 1774 by three members of the Jaquet-Droz family. They were designed for the ultra-rich of their day. These were very expensive toys and novelties; the 18th century's equivalent of buying your own football team.

The most complex of these automata is *The Writer*, a little boy who sits at his own little desk and, when wound up, writes on a piece of paper in front of him using a goose-feather quill pen. As he writes, he occasionally dips the quill into an inkpot to keep the ink flowing, giving it a couple of shakes to remove the excess. His eyes even follow the movements of his hands.

This particular automaton is considered one of the earliest examples of a *programmable* machine: unlike its two siblings, The Writer can be made to write different words and phrases.

The Writer is a distant mechanical ancestor of the modern computer, right down to how it works. Inside the doll is a complicated mechanism containing some six thousand parts—more than most automobiles—dominated by two major features:

The first is a tall stack of "cams"—small wheels with crooked edges. These edges are very deliberately designed: each cam actually represents a sequence of movements to produce a very specific result.

For example, one cam will contain the sequence for writing the letter 'A', another for drawing a 'B', while yet another cam will contain the sequence for moving the doll's arm over to the inkpot and dipping the quill pen into it.

The cams are 'read' using *cam followers*—three, in this particular example—that follow the cams' edges as they turn. A bump here raises the doll's arm; a dip there might move it slightly to the left, and so on.

The second dominant component is a large wheel containing the phrase to be written. This is *also* a cam, albeit a very big one made of a number of smaller components. There is another *cam follower* that 'reads' this wheel.

The wheel contains letters, instructions to move the paper—like tapping the space bar or performing a carriage return on a typewriter—or to dip the pen in the ink.

In computer parlance, this is the *main program*. And it really is a program, because each letter on this wheel can be changed, so that a different phrase is written out instead.

The stack of cams contains what programmers today would call *subroutines*: small pieces of program that perform a very specific function. Every time the master program wheel presents a "Draw a letter 'A'" command, the automaton simply flicks the stack of cam wheels to the position of the "draw a letter 'A'" cam wheel.

A key point to understand is that the automaton doesn't inherently *know* where each cam wheel is, or what it does. It's just lifting or lowering the cam stack by a fixed amount according to the size of the tooth on the master cam wheel. If someone were to swap the 'A' cam with the 'B' cam, the doll would write a 'B' when it should be writing an 'A'.

Despite appearances, the automaton has absolutely no clue what it's doing. It is not even remotely self-aware. The machinery is just machinery. Jacquet-Droz's *The Writer* is no more aware that it is writing a phrase on paper than a hammer knows it's banging a nail into a wall.

And that is the secret to understanding computers: They only *look* like they know what they're doing.

To make computers do anything remotely useful, you have to stack layer upon layer of programs on top of each other. The lower layers of programming perform basic functions, like "draw a letter 'A'", or "check if a key has been pressed on the keyboard". As you move up to the higher levels, you find more complicated programs, such as "look up the highlighted word in a dictionary", or "open a document".

This is not going to be a heavy, thousand-page tome. 90% of what you *need* to know will sit in the first few chapters. The rest of this book is a series of

essays and primers on all the stuff that has built up around computers: the Internet (and networking in general), security and malware, how websites work, and so on. None of those are obligatory reading. In fact, the idea is that you can dip into any of those later chapters when you need to.

Nor do I have any intention of talking down to you. You are not a dummy or an idiot. You clearly have a desire to learn, or you wouldn't have read this far. And a desire to learn is all you need.

Doesn't IT evolve really quickly?

Yes.

Like all branches of science, Information Technology is a journey, not a destination. Some of what you read in this edition will be out of date before too long so don't get too attached to it.

In fact, the "How computers think" part of this book deliberately simplifies the details of how a computer *actually* works: I don't go into any depth about multicore CPUs, Level 3 caches, PCI buses and the like as it's not really important and these details can and do change without notice. They're best left to separate articles later in the book, which you can read if you want to learn more.

Finally, this is an e-book, so I intend to release updates to it periodically. This isn't possible with printed books, so I won't be selling a print version.

About this edition

This is a preview edition of a longer e-book I'm planning to release later in the year.

This particular version is included with a tutorial written for Unity Technologies' eponymous software development system and has been adapted accordingly. The first draft of the "How computers think" section is

included in this preview, while only a few of the additional articles mentioned are also provided. The rest are still works in progress.

Text with colour highlighting is intended to link to text that explains the word or term in more detail. I'm still working on how to make this work across multiple e-book formats. At the moment, I'm using inline notes on a pale blue background, which aren't ideal, but they do the job.

Why do you spell "colour" with a 'u'?

I was born and raised in the part of the United Kingdom known as "England". You may have heard of it.

The English believe that learning their language should *feel* like a major achievement. *Anyone* can learn French or Italian, but English? That should be on a par with climbing Mount Everest! With surviving a pub-crawl in south London, or reading an entire YouTube comment thread without your brain bleeding out of your ears!

It's *supposed* to be odd, eccentric and annoying. That's the challenge. That's the *point*.

And that's why I spell "colour" with a "u". By rights, it shouldn't be there: it makes no logical sense! But it *is* there, and I remembered that fact, so I feel inordinately proud of myself for ~~writing~~ typing it. It feels ever so good.

I should probably get out more.

And finally…

This document isn't an attempt to teach you every nook and cranny of every programming language as there are simply too many of them. In fact, I'm not trying to teach you how to program computers in detail at all. There are hundreds of books covering that. All I want to do is explain what programming *is*. I believe this is the key to understanding computers and getting the most out of them.

17 May 2014

You should learn enough about general programming theory to be able to pick up most programming languages that you wish to study in more depth afterwards. I strongly recommend doing so: it can be very rewarding!

PART ONE:
How Computers Think

Our Clockwork Friends

Computers don't think like us.

Many of us find computers a bit scary, precisely because we know so little about them, and because they seem so… different.

Luckily, it turns out computers are really, *really* dumb.

All they *do* know how to do is follow sequences of very simple instructions that operate on numbers.

Whole numbers.

And that's it.

That really is all there is to them.

What, exactly, is a computer?

"Computer" is a broad term.

It refers to a combination of a **processor**, and a bunch of **connected devices**. Those devices might be connected within the same physical casing, like the display on a mobile phone. Or they might be physically separate, such as the display and keyboard connected to a PC. (Usually, if the device is connected externally, it's called a **peripheral**, from the same word as 'periphery', meaning 'outside'. This is falling out of favour of late; just think of such devices as "accessories" instead.)

The **processor**—most commonly abbreviated to **CPU**, for "Central Processing Unit"—is the bit that does all the hard work of actually computing stuff. It runs those programs—apps—you like. It shifts data around its **storage areas** and processes it according to the instructions in those programs. And yet more programs also let it talk to any **devices** it's connected to, such as a keyboard or a display.

Since about 2005 or so, it has become increasingly common for a processor *chip* to contain *multiple* processor **cores**, so that it can run more than one program at a time. These cores are usually identical and share some local storage with each other to speed up communication between them.

The reason for this shift in processor design is simple: making chips *faster* makes them run *hotter*. So much so that the fastest processors available at the time required massive fans and heatsinks to dissipate all that unwanted heat.

Furthermore, that heat is wasted energy, and you don't want that in a mobile phone or tablet, because it means you're draining the battery more quickly.

By simply increasing the number of processor cores instead, you gain many of the same advantages of raw speed, because you can share the same load over multiple cores. At the time of writing, there are tablets that run on processors with as many as *eight* processor cores.

However, the number of cores in a computer isn't particularly relevant to understanding how a computer works. All those cores do the same thing, so I'll focus on just the one…

What does a computer *do* all day?

Good question! This is the key to understanding how a computer thinks…

A computer lives for following orders. And it gets them from its system **memory**. The main working memory for a computer is called **Random Access Memory** (or RAM for short). The more a computer has of this kind of memory, the more stuff it can do, and the more quickly it can do it.

RAM is just one form of data storage.

It would be nice if we only had that one kind to worry about, but RAM is relatively expensive. RAM is also **transient**: the data is lost whenever you switch off, or restart, the computer.

So we need **persistent storage** too.

This is what hard drives provide: they store lots of data (relatively) cheaply, and they keep it intact when you switch the computer off, or restart it. Until recently, hard drives were mechanical devices, but today we also have Solid-State Drives that eliminate the spinning disks used in hard drives and replace them with a different kind of memory called flash memory.

But they're still much slower than that system RAM. So we need some of each.

When you switch on a computer, the first thing it does is look at its system memory. This appears to the processor as a very, very long row of boxes stretching far into the distance. Each of these boxes can contain one—and exactly one—whole number.

A **whole number** is any number that doesn't need a decimal point. E.g. 3, 19, -31, and 42 are all whole numbers, but 3.14159 is not.

Such numbers are technically known as **integers**. In fact, as far as the computer is concerned, RAM, hard disks, etc. *only* store integers.

Numbers with fractions, like 3.52, 99.99, and so on, are stored using whole numbers arranged in a specific order.

Everything the computer does revolves around this series of number-filled boxes. And I really do mean *everything*. Graphics? Numbers! Music? Numbers! Desktop publishing and web design? Also numbers!

This is why computers are referred to as 'digital' devices: 'digital' just means *it's all done with numbers*.

It's the programs that tell the computer what the numbers represent, and how they should be processed. **Programs provide the *context* for those numbers.**

Much of the programmer's skill lies in working out how best to arrange and process the data using numbers.

So, how does a computer know what to do?

The first thing the computer does when it wakes up is look at the first number out of the very first box in that system RAM.

Just as graphics, sound, etc. are all done with numbers, so is programming. When you write a program, it all gets boiled down and converted into numbers.

Each number corresponds to a specific instruction. For example, let's consider the following:

```
ADD [contents of box 23] TO [contents of box 24]
```

I've written this out in English so we can understand it, but the computer processor never sees it like this. Instead, it would see a series of numbers, such as these:

```
97
23
24
```

The first number—97—represents the "ADD [contents of Box X] TO [contents of Box Y]" instruction. Now, the processor knows it needs to work out what X and Y are, which is what the next two numbers provide: the 23 and 24 tell it needs to add the contents of box 23 to the contents of box 24.

This is an example of the raw, 'native' language used by computers. It's called **machine code**, and the actual code numbers used for the instructions vary between computer processor families: the Intel processors used in most desktop Windows and Mac computers understand one set of machine instruction codes. The ARM-designed processors used in almost every mobile phone and tablet, on the other hand, have their own, different, set of instruction codes. The two designs can't understand each other's codes directly.

> These differing instruction codes explain why programs written for one system have to be explicitly converted to run on another. The machine code instructions for one manufacturer's processor would be gibberish to another's.

To get around the pain this causes, humans have invented *programming languages* that let us write programs for computers without having to memorise lots and lots of numbers and codes.

To begin with, they started by simply assigning direct, one-to-one names to those raw machine codes. Thus "97" became the *operation code:* ADD [x] TO [y]. The term *operation code* was rapidly shortened to **opcode**.

Because programmers had computers to work with, they wrote programs (in machine code) to "assemble" those opcodes into machine code programs automatically. As this was a very simple job of translation, these assembler programs were themselves very simple. The opcode language that emerged for each processor family became known as an *assembly language*.

But programming languages didn't stop there. Over time, they became more and more complex and powerful, though all are eventually converted into machine code programs. A short, five-line program written in a modern programming language might be translated into a machine code program thousands of lines long.

For a more in-depth look, see the article named **Babel, Squared**.

At this point, I should stress an important point about machine code: it really is very simple stuff: move data from one box to another, add numbers from different boxes together, compare one number to another, and so on.

Every computer program a programmer writes; every application you run—must be converted into those machine code instructions at some point.

And computers have to be taught *everything.*

How does a computer understand that pressing a button on a keyboard should result in a letter appearing on the screen? It doesn't. It's the *programs* it runs that do all that. The programs provide the *context* for all the data. The programmer "knows" what the data is and how it should be processed.

The computer is always just a machine following its machine code instructions to the letter.

The Onion of Context

Context is everything with computers. The processor itself has no clue what it's actually doing. It's just following those simple machine code instructions. The magic comes from the programs the computer processor is running.

For example, when you press a key on the keyboard, the keyboard sends a message to the computer. The computer 'sees' the message as just another box with a number in it. What the number means, the computer does not know. But there is a program called a **device driver** that *does* know!

A device driver is a special kind of program dedicated to interpreting the messages from a device on behalf of other programs, and making their lives easier. This is just one of the many component programs that make up your computer's **operating system**.

The Operating System

Writing a program that does everything itself is rather like constructing a cathedral from LEGO using thousands of the same small selection of bricks. Those bricks represent the small set of machine code instructions made available to us by the computer's processor.

Given sufficient such bricks, you can build *anything*, but you'll need a lot of bricks, laid down layer upon layer, to build that cathedral. It's slow, painstaking work.

Back in the Bad Old Days™ of computers, this was how all programs were built: you had to do *everything* yourself, including read the keyboard, print out the results, run the magnetic (or paper) tape drives, and so on. Luckily, computers of the day were also much more limited than even the ones we now carry in our pockets: There wasn't even any multi-tasking in the sense we know it today.

What if you could get some help? What if you could build your cathedral using pre-built sections of bricks to speed up the process?

This is exactly what happens today: rather than each application having to do everything itself, a special, master control program, called an **operating system**, is used as a starting point instead. An operating system—usually abbreviated to "OS"—provides a foundation to build applications on. It includes a library of many thousands of ready-made programs and functions that programmers can use to build their own cathedrals on top.

Operating systems also typically define a standard method for users to interact with those applications. This is why computers that run Microsoft Windows look and feel different to those that run Apple's OS X.

OS X—the "OS" stands for 'Operating System'—is the operating system that runs on Apple's "Mac"-branded computers. But Apple also make iPhones and iPads. These run a related operating system called *iOS*. In both cases, that operating system does a lot of the hard work for you when building applications: all the standard stuff like windows, icons, etc. are drawn for you, for example. You don't even need to explain to an iPhone what kinds of touch strokes and movements to watch out for in any detail: there are a bunch of ready-made standard ones to pick from, like picking a paint colour from an artist's palette.

The most common *desktop* computer operating system in use today is Microsoft's Windows. On mobile devices, the most common operating system is Google's own 'Android' operating system, which can be found on most Sony, Samsung, LG and HTC smartphones and tablets.

Operating systems mean programmers don't have to reinvent the wheel for every program they write. The operating system hides all the details of working out how to talk to all the devices connected to the computer, so you can concentrate on more important things.

So, back to that keyboard…

The keyboard's device driver is notified whenever the keyboard sends a message to the computer. These messages are usually simple things, like "Key no. 43 pressed," and "Key no. 43 released". As with all things computer, the messages themselves are just numbers. And those numbers appear in one of those boxes the processor sees.

> A **program** is to a computer what the concept of a 'spirit' or 'soul' is to a human. We all have the same physical components: a brain, arms, legs, etc., but it's our mental 'software' that defines who we are as individuals and ensures we don't all think and act alike.
>
> No two people have exactly the same life experiences. All that we learn and experience, all the mental models we create for ourselves are our software. Our spirit is our operating system.

So the device driver program 'knows' that, say, Memory Box no. 430 contains the number of the pressed key. (Box 431 might contain the actual message code, such as "KeyUp" or "KeyDown". Another box will hold the status of the modifier keys—Control, Alt, Shift, etc.)

Over to Box 430 we go. In it, we find "43".

> Remember: the processor itself has absolutely no clue what this number actually represents. It's the *programmer* who defines the context.
>
> This is the key to understanding what it is a computer does.

The keyboard device driver program takes that number and knows it needs to inform the operating system that the user has typed something on the keyboard.

The operating system, which is just a really big collection of programs, includes one that knows what language keyboard is attached to the computer. In this example, it's an English one, so it looks up the

corresponding letter on a table: the table says the number 43 corresponds with the “A” key.

Now we also need to know if it’s a capital or lower-case “A”. The driver program does this by copying a number in another box it knows about, which tells us which, if any, of the *modifier keys* are being pressed.

> **Modifier keys** are keys you press along with another key.
>
> On most Windows PCs, the modifier keys are “Ctrl”, “Alt”, “Shift”, “Alt Gr”, and “Windows logo”.
>
> Macs also have a “Command” key, and most laptops add an “Fn” key for some features, such as brightness and volume control.

This information is passed to the operating system.

Next, the operating system looks for applications that are running and passes this information on to the one the user is actively using.

Let’s say we’re running a word processor: it receives that message and knows it needs to draw the letter “A”.

In fact, this process is more complicated than it used to be—time was you just told the display chip that you wanted an “A” drawn at a specific location and it went away and did it for you. Today, the word processor sends a message back to the operating system that it needs it to draw that letter on its behalf. And it really is *drawn*: in order to make the letter look smooth at any size, modern operating systems use *vector fonts*, which contain instructions for how to draw each letter at any arbitrary size by drawing a series of lines of suitable thickness, shape and colour.

All the above requires coordination between multiple programs.

And that’s just what happens when you type a letter on a keyboard while running a word processor. Operating systems contain layer upon layer of program code.

This is why I referred to an "Onion of Context": at the very lowest levels, we have the bits of the operating system that deal with the very basics of running multiple applications on a computer. This layer is called the **kernel** and does little more than coordinate all other programs that run on it.

On top of that kernel sit the hardware drivers: the programs that monitor all the various components of the computer and provide a consistent way to access them for your applications.

If we didn't have these, every single application would also have to include such code. This program code would be duplicated in every application you ran on your computer, which is a huge waste of time and resources.

> Games developers used to have to do this until the late 1990s, when operating systems started to include standard methods for reading joysticks and joypads, handling multiplayer games, fast 2D and 3D graphics, audio, and so on. I was there. It was no fun at all.

Early computer operating systems had very few layers and concentrated primarily on providing standard code functions for accessing data stored on disks—hence "DOS", for "Disk Operating System". Over time, these operating systems have become more and more complex, gaining more and more layers.

A modern operating system includes multiple programs, and layers upon layers of code, that can handle almost anything: disks, smartphones, multiple displays—all those plugs and sockets on the side or back of your laptop or PC are controlled by dedicated hardware drivers in the operating system too. Plug in a printer, a scanner, a digital camera, an iPod—you name it—it'll *only* work properly if there's a hardware driver for it that's been written for the operating system you're using.

Painting by numbers

We're back to those boxes of numbers again.

You may have noticed that we often define a computer or TV display in terms of its **display resolution**. This is usually a figure like 1920 x 1080 — the standard display resolution for a "Full HD" flat-screen television.

Those numbers refer to **pixels**—an old abbreviation / corruption of "picture element", and a fancy name for 'dots'. (There was a brief period when "Picsel" was considered, but the current spelling was clearly much cooler.)

If you were to grab a magnifying glass and look very closely at your TV, you'd notice tiny dots arranged across the screen. These are pixels. Literally, a tiny dot that can be made to appear in one of millions of colours. Arrange lots of these together, very closely spaced, and you get a picture. Change those dots 50-60 times a second or more and you get the illusion of a moving image. Pack more dots into the same space and you can see more *detail* in your image. (Pixel density is known as "dots per inch"—DPI—a term from the print industry, or "pixels per inch"—PPI. The two are often used interchangeably.)

Meanwhile, back in our computer's RAM, we find a big chunk of those boxes. To the computer's processor there's nothing special about these. They're just boxes with integers—whole numbers—in them, like all the others. But these boxes are special: they're linked to the **graphics chip** that provides the image for the computer's display.

In this example, each pixel on the display corresponds to three of the boxes. For our 1920 x 1080-pixel Full HD display, that means we have 1920 x 1080 x 3 boxes dedicated to that display—that's about 6.2 million boxes! Luckily, RAM is relatively cheap today and that amounts to a little over six megabytes. As even the cheapest smartphone has about 256 megabytes of RAM to play with, six megabytes for the display isn't a big deal.

Measuring Storage

System RAM has historically been measured in bytes. Each byte corresponds to one of the 'boxes' I've been talking about. Today, a byte can hold any whole number between 0 and 255. (There was a time when bytes could be bigger or smaller, but it became standardised in the early '80s.)

Bigger numbers are stored by simply combining bytes together. E.g. two bytes can hold a number up to 65535.

For engineering reasons that make using powers of two easier to work with—see the essay titled **Working with Bits** for a look at why—the "kilo", "mega" and "giga" prefixes used for most other units were held to mean something slightly different: instead of multiples of 1000, engineers found multiples of 1024 easier to use as this number is a power of two. Thus a kilobyte of RAM = 1024 bytes, not 1000; a megabyte of RAM = 1024 x 1024 bytes, and so on.

All this changed recently when the Standards Institute put its foot down and demanded these prefixes be used to refer to multiples of 1000 instead, like every other industry. (Including hard drive manufacturers, who had been doing just that for many years.)

To represent the old 1024-based multiples, the IT industry invented new prefixes: "Kibi", "Mebi", "Gibi", meaning "Kilobinary", "Megabinary", etc.

This new system has taken a while to catch on and the old usage remains in use today, so be prepared for some confusion during the transition period.

For this example, let's assume the special display image boxes start at Box no. 10000.

In programming jargon, this means the image starts at a **memory address** of 10000.

To change a pixel at a desired point on the screen, you only need some very basic arithmetic: take your x and y coordinates, multiply the y by 1920 x 3 (= 5760 bytes) and add the x. Add that to the address of the first box, and you now know the address of the three boxes linked to that pixel.

So, if x is 150 and y is 200, that means we need the three boxes starting at the address of 200 x (1920 x 3) + 150 = 1162150.

Why three boxes per pixel?

Computer displays use an 'additive' system to display colours. This uses three primary colour components: Red, Green and Blue. We need one box—one byte—to store each of these, which is why we need three boxes for a single pixel.

To set a pixel to white, you'd store the maximum possible number in each of the three boxes. As a byte can store whole numbers from zero to 255, that means we store a 255 in each box. The result is that we see a white dot at the pixel's location on the display.

For a bright red dot, we'd store 255 in the first ("Red") byte, and zeroes in the other two. A bright green dot requires a 255 in the second byte, and a bright blue dot requires a 255 in the third byte—again, with the other bytes containing a zero.

In addition to these three bytes for the "RGB" data, many image formats add a fourth: *Alpha*. This stores the level of transparency for the pixel. Alpha is usually abbreviated to just "A".

I know I keep drumming this next point home, but it really is the most important thing to remember when dealing with computers:

> At no point does the processor itself realise that it is messing about with a display. The context is **only** known by the programmer.

This is a double-edged sword: the onus is entirely on the programmer to get that context right. Change the wrong bit of data in that RAM and anything could happen, because **the computer doesn't know it's doing anything wrong**.

Computers have no concept of *anything*. They have no idea of context. They did not evolve from a primordial soup. They have no 'common sense'. They are utterly unaware of their surroundings. All they do is follow their instructions to the letter.

They are—and this bears a lot of repeating—just machines.

A locomotive won't attempt to avoid an accident by itself: if you command it to go forward then leave it to its own devices, it'll go forward and keep on doing so, until it either runs out of fuel, runs out of railway track, or hits something heavy, and probably very expensive.

Similarly, if you try to hit your thumb with a hammer, the hammer won't try and stop you.

Many people have difficulty getting to grips with computers because all those layers and layers of programs make them *appear* to be complex. Because of this, they can give the impression of at least a basic level of sentience or understanding. But this complexity is an illusion: it's the result of running ever more complex *software*.

Homo Sapiens has evolved to view anything demonstrating complex behaviour as at least a little bit sentient, because we've evolved alongside an entire planet's worth of other animals that can, and do, react to their surroundings. There is an inherent expectation that a sentient creature has some ability to understand meaning. A dog or a cat will respond to a whistle or a gesture; a trained falcon will land on an outstretched arm. And, of course, we expect most animals to try and protect themselves.

Many of us still give names and project personalities onto animate objects like ships, cars and locomotives. We even anthropomorphise animals in art and animation, as well as inanimate objects, like toasters and spaceships.

The key to retaining your sanity when working with a computer is to realise that it really is just a hammer. A very complicated hammer, certainly, but a

hammer nevertheless. It won't even know you're *there*, let alone stop you using it to whack your own thumb.

> One other thing computers are *not* good at is the concept of infinity, and similar abstract concepts. Such ideas are hard to represent using simple integers.
>
> Try to divide a number by zero and the computer will complain because the processor doesn't even know how to *store* infinity: those boxes can only hold integers from 0 to 255. To a computer, everything has limits; everything is finite.

So, what *can* computers do?

As it turns out, surprisingly little…

The machine code of a typical computer processor today can:

- Copy numbers between boxes;
- Change the numbers inside a box;
- Compare two numbers and make a decision based on the result;
- Perform some arithmetic on the numbers.

Amazingly, it turns out that's all you need…

Virtual hands, virtual fingers

Computers do something that might surprise you: they do a lot of maths on their own virtual hands. They count on them. They do mathematical operations with them. They pass data around with them. Computer processors refer to these virtual hands as “registers”.

Unlike us humans, different computer processor designs might have dozens of such registers, though some older models had far fewer. These registers are the fastest form of memory the computer has, so the more such registers a processor has available, the faster it can do its stuff.

Another feature of these virtual hands is the number of virtual *fingers* they have: when you see a reference to a 32-bit, or a 64-bit, processor, those ‘bits’ are the maximum number of virtual fingers the processor’s hands can have. (Often, programmers don’t need all 64 or 32 bits for a calculation, so most processors let you use a smaller number of fingers.)

The very first computer processor, the Intel 4004, had only four virtual fingers on each hand. It was used in one of the first pocket calculators.

Most computers in the 1980s had 8-bit processors—a technical term meaning the processor had 8-bit *registers*, or eight virtual fingers per register.

But how, I hear you ask—I really need to talk to a psychiatrist about that—do eight fingers help? After all, even with ten fingers, we can only count to ten, right?

Not so…

How to count to 31 on the fingers of one hand

The smallest useful unit of storage for computers is the *byte*. A byte stores eight virtual fingers' worth of data. However, computer memory used to be very, very expensive, so processors use a much more efficient counting system than we do.

If we were to apply the human approach, a byte could only hold numbers between 1 and 8. This, to an electronics engineer, is simply not good enough: it is inefficient! It is wasteful! Bah!

Hold your right hand over a table, with the fingers just above the surface. Let's call this 'zero'.

Now, lower your little finger to the surface. That's "1".

Now raise your little finger and lower your ring finger to the surface.

That's "2".

Now lower *both* your little finger and ring finger to the surface: that's "3". And we've already saved a finger!

Let's continue the sequence. In the list below, each number, (from left to right), represents the thumb, forefinger, middle finger, ring finger, and little finger. A zero means the finger is raised; a one means the finger is touching the table's surface. We've already done the first three numbers, so…

4: 00100 — just the middle finger
5: 00101 — middle finger plus little finger
6: 00110 — middle finger plus ring finger
7: 00111 — middle, ring and little finger
8: 01000 — just the forefinger
9: 01001 — etc.
10: 01010

— the middle, ring and little fingers are simply repeating the same pattern as before, so let's skip to…

14: 01110
15: 01111

Now the cycle for the fingers repeats again, but this time we finally get to use the thumb…

16: 10000
17: 10001
18: 10010
19: 10011

…etc. Until…

28: 11100
29: 11101
30: 11110
31: 11111

And that's how you count to 31 on just five fingers.

> You might have spotted a couple of patterns here—humans are very good at spotting patterns in things—including the fact that each column, from right to left, flips between 0 and 1 in a fixed sequence: the little finger changes the most often: it changes every step. The ring finger does so every two steps; the middle finger every four… and so on.
>
> Some of you may find it easier to remember how this counting system works if you think of it in terms of such rhythmic sequences.

If that pattern looks a little familiar, there's a good reason for it: it's how you count to 31 in **binary**.

Computers store *everything* like this. They don't do letters or music or art—not directly anyway: They do *binary*. The binary numbers represent on-off switches, which is what all computer memory is actually made from.

Now, binary means 1/0, true/false, on/off, yes/no, up/down, right/left. There are no shades of grey; no maybes, there is *nothing* in between.

You can combine those ones and zeroes together to store integers, and you can do a lot with those integers, but, fundamentally, if your data can't be represented in this form, it can't be stored at all by a computer.

Luckily, almost everything *can* be broken down into whole numbers somehow. Even non-whole numbers! And there are also a number of neat tricks you can do now that you know what binary numbers are.

For calculations involving numbers with fractions, such as "3.14", most processors use what's known as **floating point** arithmetic.

This uses a form of scientific notation that keeps the useful part of the number (e.g. "314") separate from a multiplier. This lets us write the same number as "314×10^{-2}". You could read this as "314, with the decimal point shifted two digits to the left".

Computers do much the same thing, but using powers of two, not powers of ten, as computers use binary numbers under the hood.

There is a downside to this though: accuracy.

Suppose you have a number like: 10110.0101

How do we store this using two bytes?

Answer: we can't. We have to either round up, or chop off that final '1', because there are nine digits here, not eight, or we'd need two bytes just to store the whole number part (the "mantissa").

The same issue arises even with our own preferred decimal number system: how do we represent π — i.e. Pi — which is an irrational, (never-ending), number? Again, we can only store an approximation to it by rounding it as best we can to fit our needs. So this isn't a problem unique to computers.

This kind of problem is known as a "rounding error" and that's a phrase you'll hear often in IT circles.

Rounding errors are a major concern when working with financial data.

A few pennies here or there might not sound like much, but if you're processing all the accounts for a major bank, all those pennies can easily add up to millions.

Hardware Acceleration:
The rise of the specialist processor

Since the 1990s, the traditional computer processor—or CPU—has been increasingly assisted by *specialised* processors that are designed to perform very specific forms of processing. The first such processors were *floating point co-processors* (also known as "maths co-processors"). These are now usually built into the CPU itself as standard. Such processors are common in desktops, laptops, smartphones and tablets.

Some processors, such as those used in, say, washing machines, don't have such features as they don't need them, so this is by no means universal.

Originally, all the graphics stuff was done entirely by the CPU. In the 1980s most microcomputers did it this way, the IBM PC and its clones being no exception, though they did get a little assistance for some duties. A few computers of the day *did*, however, include dedicated custom processors to make graphics processing quicker: these tended to be computers and consoles designed primarily for playing games. Atari, Commodore, Nintendo and Sega made a number of such computers and consoles during the 1980s and 1990s. This technology eventually seeped into mainstream computers, such as the IBM PC and its clones—what we now called Windows PCs as IBM have long since stopped making their own PCs. This happened around the same time graphical user interfaces (GUIs) caught on in a big way: both Microsoft Windows and Apple's Mac systems included such interfaces and that meant they had to move big blocks of data around very quickly and smoothly.

Video cards therefore began to include built-in chips to speed up the moving around of windows, scrolling text, and so on.

Over time, these features started to include other forms of graphics processing, such as 3D graphics rendering, video compression and decompression, and a lot more.

Today, it's a poor computer indeed that lacks any kind of graphics processing support at all. Even the cheapest smartphone includes a graphics processor of some kind now.

But a graphics processor is still just a form of CPU. The only difference is that its *machine code* is designed and extremely optimised for processing graphics data: You feed it 3D model and scene data and tell the GPU to draw it; the CPU then goes off and does something else while the GPU gets on with rendering the images.

But it's all still just numbers under the hood.

So… do I have to program *all* of this stuff myself?

No.

Mercifully, the days when every games program had to include a bunch of code to do even the most basic drawing of graphics, mouse-handling, joystick-reading and so on are now long gone.

This is where **middleware** like Unity comes in: it includes loads of ready-to-use functions that your code can call to do all the hard work.

Platforms & Middleware

Originally, software had to be custom-written for each and every combination of computer hardware and operating system. Each such combination is known as a **platform**. An Intel-powered desktop computer running Windows 8.1 is one platform. An ARM-powered iPhone running iOS 7 is another platform. And so on.

Unity is a software tool that lets you write programs for multiple such platforms. Unity is, in effect, a "virtual platform": it's just software, but it's

software that hides the different kinds of hardware from you. In industry parlance, Unity is middleware.

Unity sits between your program and the computer itself, doing all the boring low-level work: “Draw this image *here*, then that image *there*, then animate that other image, then play that bit of sound, then read the joypad, the mouse, or the touch-screen *now*…” and so on. Unity does all this so you don’t have to. It does all the washing-up, cleaning, vacuuming and other routine chores for you, so you can concentrate on the fun stuff.

Most programming is like this now.

It’s very rare for a programmer to have to deal with low-level chores any more. Operating systems like Windows, OS X, iOS, Linux, etc., all do lots of stuff for you.

Finite State Machines

At the heart of every non-trivial program lives at least one Finite State Machine (FSM). Although the term sounds complicated, it is actually a concept that can be found even in clockwork devices.

Prior to the electronic calculator, mechanical calculators, such as those shown, used clockwork FSMs to perform their calculations. Their programming was embodied in the mechanisms of these machines.

"Mechanical calculators Keyboards" by Ezrdr - Own work. Licensed under Creative Commons Attribution-Share Alike 3.0 via Wikimedia Commons

The Finite State Machine is where you define how your game works. It's the FSM that makes your game a turn-based one, or a real-time one, for example.

In fact, the computer processor is itself just a complex Finite State Machine. When it follows a machine code instruction, it's really just changing its current state to the one the new instruction produces.

What is a Finite State Machine?

A Finite State Machine is a system that can be set to one of a finite (i.e. fixed, known number of) *states*. The FSM's state might be represented by one or

more state variables, but the key point is that each FSM can only hold one *overall* state at a time.

A good illustration of a FSM in the physical world is the humble traffic signal, with its red, amber, and green lights.

In the UK, these have the following sequencing:

1. Red – Stop
2. Red & Amber – Stop (impending green aspect)
3. Green – Proceed with caution
4. Amber – Stop if possible to do so

…and the sequence then returns to step 1.

This system therefore has four *states*. As these are all computer controlled now, this means the computer has a simple FSM that cycles between these four states when requested to do so by the main control program.

Crucially, this FSM *cannot* be set to both "Red" and "Green" states *at the same time*. They are mutually exclusive states for this FSM, and the FSM's design will ensure such a state cannot happen.

In a more complex FSM, such sanity-checks might involve program code, but for a such a simple FSM as this, it's sufficient to just have a single *signalState* variable that can only be set to one of four different states. In C#, this would be done using an **enum**erated data type.

A more complex example is seen at a controlled pedestrian crossing, where multiple traffic signals are also connected to pedestrian "walk" / "don't walk" signals. Pedestrians have buttons they can use to inform the system that they wish to cross.

The computer programs that control such crossings will make use of *multiple* FSMs, each linked to the others, so that the signal sequencing is always correct. (E.g. east-west green; east-west red / north-south green; all roads red / pedestrian signs set to 'walk'… and so on.)

Note the use of the word 'sequencing': FSMs are commonly used to produce patterns and sequences of states.

In programming, an FSM can be as simple or as complicated as you like, but they all boil down to much the same thing: one or more *status* variables and a bunch of '**if** / **then**', (or '**switch** / **case**') code that checks the variables and acts on their status as required.

In Tac-Tac-Toe, the 'master' state is that of the game itself:

1. In main menu, waiting for player to start a game
2. Waiting for Player 1 to play their move
3. Waiting for Player 2 to play their move

As usual, the computer only 'sees' numbers, but it's good practice to use **enum**erated data types for status variables as it makes the code easier to follow.

Another common use for FSMs is for animations. A player character in a platform game might have animations for standing still ("idle"), walking, running, jumping, taking damage, losing a life, and so on. The game will therefore use an FSM to keep track of which state the character is in, and updating that state according to the user's input, which is provided through another FSM. (It's Finite State Machines all the way down!) That same FSM is often where you'll find the code that triggers each of those animations when required.

FSMs are even used to handle individual animations. You may be familiar with Unity's animation timeline feature, used to set key-frames for animated sequences. This is all done using FSMs behind the scenes. In fact, even the graphics rendering is done using FSMs that manage render states.

The Finite State Machine is the heart of programming. Almost all programming revolves around joining FSMs together.

Conclusion

I claimed at the start of this section that programming was fundamentally a process of *translation* and I meant it. The problem is that programming involves *multiple* translation steps, because the programming language(s) you use are translated in turn by another computer program into the final machine code understood by the computer's processor.

On the other hand, that final translation step is effectively automated and need not concern most programmers. There are quite a few professional programmers who have never learned how a computer thinks, for example. Personally, I think it's a lot easier to understand programming with this knowledge, but it's not necessary to remember every tiny detail. The chances of you having to work with an assembly language, let alone raw machine code, are very, very small these days.

*

That's it for the stuff you *need* to know about computers. The next section is a collection of loosely connected essays on various topics related to computers and computing.

For this Unity-specific edition, these essays are primarily concerned with programming.

PART TWO:
Miscellany

What's the best programming language?

Whichever programming language is the best fit for the task. Sometimes, that means multiple programming languages.

A computer requires all our program code be reduced to simple numbers, so numbers are all it sees when it runs our programs. C# programs are *compiled* before they are run. That process takes all the text we've written and turns it into machine language the computer understands. None of those labels whose names we've agonised over remain.

All programming languages are translated to machine code.

Even those that are interpreted instruction by instruction. So, when we're writing a program, it helps to remember that this isn't what the computer processor actually sees when our program is running.

All programming languages, including Javascript, C#, etc., are merely an intermediary between us and the computer. They save us having to translate *directly* into that machine code, because even the hairiest of grizzled programming veterans find working at that level no fun at all, despite their attempts to suggest otherwise.

All programming languages have their pros and cons.

Almost all programming languages were designed to solve a small subset of programming challenges, such as defining business logic, customisation of

various office suite applications, handling complex mathematical formulae, and so on.

So there is no such thing as a "best" programming language. There is only *the best language for the specific task at hand*. Even when working in Unity, you may find it easier to switch occasionally from C# to, say, Javascript or Boo.

All experienced, competent, programmers know more than one programming language.

Many have learned so many, they no longer remember them all—I lost count myself over a decade ago. It's not uncommon for a language to be popular for a while, before fading away into obscurity relatively quickly. One example is Pascal, which was used by Microsoft to create Windows back in the 1980s and early '90s. By the mid-'90s, hardly anyone was using it: a relative of C, called C++, had become the fashionable language about town by that point.

All programming languages share similar features.

That is a Very Good Thing Indeed™, because it means picking up most programming languages is surprisingly easy after you've learned your first. The music might change, the lyrics may be different, but the underlying chord sequences and structure will be very familiar.

User Interfaces

A Brief History

Before we could connect keyboards to a computer, the only way to talk to it in any meaningful sense was by flipping switches on a panel. Remember that endless row of boxes? Those switches were literally used to choose a box to put a number in, while another bank of switches let you set the number itself. As long as computers had memories measured in just a few hundred bytes or so, this was painful, but no so much so as to be worth fixing.

But then computers got more powerful and we needed a better way.

The first step was to connect a keyboard to the computer. The computer's output was usually a teletype printer—essentially a remote-controlled typewriter connected by a phone line to the computer itself: You'd type a command on the keyboard, which would send it to the computer over the phone line. The computer would then send the results of the command back to the teletype, which would print them out. Usually painfully slowly and noisily.

The Command-Line Interface—"CLI" for short—was born.

The next step was to replace the printer bit with a television-style display instead, so you could get the results more quickly and without destroying quite so many trees.

It may surprise many readers to learn that this interface continues in use to this day. In fact, if you open the Terminal app on a Mac, you'll find a blank window with a cryptic prompt in the corner. This is the command-line interface for the Mac. (Macs run a heavily customised version of the venerable Unix operating system, which dates right back to the early 1970s. Linux—which many of you may have heard of—is itself a clone of Unix and thus has a very similar command-line interface under the hood.)

The other major operating system to be found on desktop computers was CP/M. Microsoft created a heavily modified clone of this to produce the IBM PC's own operating system: PC-DOS (more commonly known by Microsoft's brand name for it: MS-DOS.)

This went the way of the dodo back in the late '90s, when Microsoft's later versions of Windows switched to a different operating system core. Nevertheless, many of the commands themselves were retained due to their familiarity, so, in Windows 2000, XP, Vista, 7 and 8, you can use MS-DOS commands too.

The terminal application is called up by tapping the Windows key and R, which will open a box where you can type in a command. Type cmd.exe (or just cmd) and press [Enter]. You'll see a terminal with a similarly cryptic prompt to that Mac one in the corner—usually something like: "C:\>_", with that underscore flashing patiently.

This is what all programmers and computer users of the '70s and '80s used to work with all day. You had to memorise a whole bunch of system commands and know when and how to type them in, along with any arguments or parameters the commands required. When you ran an application, it too tended to use similar typed commands to operate its functions.

And these commands tended to be terse:

Typing ls (thought to be short for "List Stuff") on the Mac's Terminal app, or dir (an abbreviation of "directory"—the old name for a folder) on the Windows one, will perform a similar function: show a list of files in the current folder.

You can usually provide various parameters for each command to tailor its output to your needs. For example:

```
dir *.txt
```

will show only files ending with a ".txt" file extension. This usually means the file is a simple text file, like the files produced by Windows' Notepad.

On the Mac (or any other Unix system), the same result can be had with:

```
ls *.txt
```

Although this suggests the two systems share very similar commands, this isn't the case: Unix has many more commands due to its sheer age, its origins in academia and as a major server operating system used in big corporations.

Windows has fewer commands as more of its functionality was designed to be controlled from its more modern graphical interface…

The Evolution of the Graphical User Interface

The idea of creating a graphical user interface that included windows, icons, menus and pointers dates right back to the 1960s, but it wasn't until the 1970s that the technology to make it possible appeared. That technology was expensive, so we had to wait until 1984 for the first successful home computer to appear with such an interface: Apple's Macintosh.

The **W**indows, **I**cons, **M**enus and **P**ointer ("WIMP") concept, with its now-traditional office desktop metaphor, on-screen trash cans and the like, is well over 40 years old. It has changed somewhat over the years, but it's still very recognisable.

Microsoft built their own version, called Windows, that finally caught on in the early '90s. If you were to compare Windows 7 with Apple's OS X, you'd see the similarities: both have a bunch of icons on a virtual 'desktop' area; both have a docking area at the bottom of the screen; both let you open up windows that can overlap; both let you select commands from menus rather than having to type them in, and so on.

There are some differences in detail however: on a Mac, all programs share a single menu bar at the top of the screen, with only the active program's menu shown. This saves space—an important feature in the early days of the Apple Mac as they had very small screens at the time. That single menu bar is also easier for users to point to with the mouse as they don't have to be too accurate: you can't send the pointer off the top of the screen, so you don't have to worry about overshooting.

Microsoft, on the other hand, nailed its menus directly into the windows instead. They're not quite as easy to navigate to—you can easily overshoot if you're new to using a mouse—but it can be less confusing as the menus don't change whenever you switch from one program to another.

WIMP: The keyboard is still king

Microsoft's Windows arguably has better support for keyboard users. This might seem like an odd priority, but there's a valid reason for it: the traditional WIMP type of graphical user interface was never intended to be used solely by the mouse. Its purpose was to let new users explore the system and applications more easily without having to resort to lengthy manuals or memorising of long lists of commands.

Instead, the idea was for users to *learn the keyboard shortcuts*, which would be displayed next to each command, so that, over time, they'd use those instead of taking a hand off the keyboard every few seconds to move the mouse and click a button.

An expert user hardly ever touches the mouse, unless they're doing graphics work.

However, there is a new kind of graphical user interface that does not follow the WIMP model…

Multi-touch interfaces

17 May 2014

The advent of touch-screen devices like the iPhone and iPad have brought about a revolution in graphical user interface design. At the time of writing, the traditional desktop PC has already become a niche market, with only the laptop / notebook designs selling in any great numbers. These come with trackpads that offer similar features to a touch-screen display, so companies like Microsoft and Apple have been focusing their efforts on these at the expense of the traditional mouse.

But it's the touch-screen devices themselves that have stolen the desktop computer's thunder. They've rewritten the rules of graphical user interfaces: with no keyboards as standard, there's no emphasis on learning keyboard shortcuts, for example. Instead, that touch-screen interface has to work well for both newcomers and experts alike.

These interfaces are still very new. Unlike the WIMP approach, they've only been around in the consumer market since 2007 and will no doubt evolve greatly over the following years.

Design Patterns

Common features of programming languages

There are *hundreds* of programming languages out there today. All of them were designed to make it easier to write certain kinds of program. Some are very old—the earliest date back to the 1950s—while others are much more recent. Some very old languages, like COBOL and BASIC are still in use today, if not as frequently as they once were.

As programs became bigger and more complicated, higher-level languages were invented that combined multiple machine code instructions into a single high-level instruction.

One example of such a language is BASIC, the initials of which stand for "Beginners' All Purpose Instruction Code". This was created in the 1960s primarily as an educational tool, and it has been updated substantially over the intervening years as it found work outside academia.

Some BASIC instructions performed very complex operations, such as loading program code from cassettes, or (later) drawing complex shapes on a graphical display.

Microsoft, Inc., actually started life creating versions of BASIC for the dozens of microcomputers produced in the 1970s and early '80s.

BASIC is still used today, but it is used today mainly to customise Microsoft's suite of office applications.

The C family

One language that made it to the big time is C. (Yes, you read that right: a programming language that has a single letter as its name.)

C was designed to be a portable take on those assembler programs mentioned earlier. Each processor manufacturer—and there were many back then—had their own machine code and assembler, so whenever a programmer wanted to create a version of their program for another system, they had to rewrite it. C got around this by acting as a 'universal' assembler that meant you could write your program once in C, then *compile* it for any system that supported the C compiler.

Assembling, Interpreting, Compiling

All programming languages have to be converted into numeric codes that the computer understands.

Assembly languages are the easiest as there's a one-to-one relationship between each instruction and its machine code. I.e. each instruction corresponds directly with exactly one machine code instruction.

An **interpreted** programming language converts the program into machine code *as it runs*. This has some advantages for programmers as you can halt the program to inspect what it's doing, but such languages can be quite slow as the interpreted instructions aren't cached: they're re-interpreted every time. Such languages are increasingly rare now as hybrid systems are taking their place.

A **compiled** language converts the entire program into machine code before it runs. This results in a much faster program, but it can be much harder to find and kill bugs.

Today, we often see a *hybrid* version of compilation and interpretation, called **Just-In-Time Compilation**. (Often shortened to "**JITC**", or "**JIT**".) This involves interpreting the program code as it runs, but also storing the resulting machine code, rather than re-interpreting it every time.

C inspired a number of languages, including C++, C#, Java, Javascript (no relation), and many more. All took some design cues from C. For example, C#, C++ and Java all retain C's requirement to end each program statement with a semicolon.

Unity makes use of Just-In-Time Compilation and supports two languages from the C family: **C#** and **Javascript**. (The third language, Boo, is not a member of the C family, which is why it looks quite different.)

All programming languages share common features, because they all have to be converted down to machine code at some point.

Generally, you will find commands for:

- storing data in **variables**—basically, storage boxes with names;
- defining more complex **data structures**;
- performing **basic arithmetic** on the data;
- defining **functions** that do stuff to data;
- **testing** and **comparing** data stored in variables…
    - …and **making decisions** based on the results of such tests;
- **looping** and **iteration** features that let you step through arrays or other sequences of data;
- **organising** the program code and data to make it easier to work with.

This list defines a basic design pattern: a *template*, if you will, that is common to all major programming languages. In fact, most languages these days are more concerned with the organisational aspects than anything else.

Two decades ago, the *what* was very much the hardest part of programming. Today, almost everything you might want to translate into a form a computer can understand has *already been translated before*. It's extremely rare to find yourself pushing the programming envelope and creating something utterly new and untried. (The games industry is one of the few fields where this can still happen, but even there, it's still unusual.)

Most programming today is therefore primarily concerned with finding suitable algorithms, and working out how to express them in your chosen programming language.

Algorithms

An algorithm is like a cooking recipe.

It describes a procedure you can implement in a programming language, but it is not written in a form specific to any one programming language.

Algorithms are therefore written in a human language rather than a computer programming language. They describe the steps your program must perform in detail, but it's up to you to convert each step into actual program code.

These are the building blocks of modern programs. You'll find thousands of algorithms online and they can save you a lot of time and head-scratching.

All the major programming languages share the same core features. Where they will often differ is in the details of how the features are presented, and—most of all—in how they *organise* your code.

If you read the following sections, it'll be easier to follow any program code written in any of the major languages.

Variables

Variables are similar to the boxes the processor works with. You can give a name to a variable, to make them easier to remember.

Some programming languages have a generic "variable" type that lets you store any kind of value—text, a filename, a phone number, an integer, or a fraction.

Other programming languages require you to specify what **type** of data you want to store in each variable.

Unity offers examples of both:

Javascript

```
var someText = “This is some text stored in a variable.”
```

C#

```
string someText = “This is some text stored in a string variable.”
```

Note how the C# version requires you to specify that the variable *someText* will hold a string. And *only* a string. This particular variable will not be allowed to hold anything else.

> A ‘string’ is a common programming term for text variables. The name comes from the fact that each letter is stored by the computer as a ‘string’ of numbers, each representing an individual letter, number, or other character.

The Javascript code uses the generic “var”. This is just an abbreviation of “variable”.

This means the same variable can then be reassigned something else:

```
someText = 3.14159
```

Now *someText* contains a floating point number.

There are pros and cons to both approaches and, in fact, Javascript will let you specify the data type if you prefer.

The main advantage to specifying the data type is that it can help reduce certain kinds of bugs. If you were to accidentally assign the wrong kind of data to someText in Javascript, there’d be no error or warning given as the system has no idea if the assignment was intentional or not.

If you were to try to do the same thing in C#, you’d get an error from the compiler as it knows *someText* is supposed to hold only text strings.

The disadvantage of explicit data types is that you may have to explicitly **cast** a variable from one type to another if you need to copy it.

Note: *Most of the code examples in this document are written in C#.*

Data Structures

All the major programming languages support a basic set of data types: integers, floating point numbers, text strings, arrays (i.e. groups of identical data types), etc.

Sometimes, these data types aren't enough on their own. Perhaps you want to keep a database of contact details. You could create separate arrays for the first and last names, phone number, email address, and so on, but this would get very messy and unwieldy.

Arrays

Let's say you want to store a series of names. You could create a bunch of variables with names like "Name1", "Name2" and so on. Or you could use an *array*. This lets you use a single variable to store multiple chunks of data.

For example, your programming language might use "string names[10]" to define an array called "names" that can hold up to 10 name strings. You access the entry you require by using names[index], where 'index' is the number of the string you want to access.

Most programming languages let you define collections of basic data types in a *structure* of some kind. In C#, this is done using either *classes* or *structs*.

For our example, we might create a structure called contactRecord that contains two strings for the first and last names, another for the phone number (which should never be stored as integers), as well as dates of birth, anniversaries, etc.

We could then create a *contactRecord* array and access individual records by selecting the record number we want, like this:

```
    contactRecord[currentContact].firstName = newFirstName;
    contactRecord[currentContact].lastName = newLastName;
```

In this example, (which would work in either C# or Javascript), I'm using a second variable, *currentContact* to store the record number I want to access, while *newFirstName* and *newLastName* contain the new first and last names I want to store in this record.

You will be using arrays and data structures a lot in Unity.

Basic operations

All programming languages worthy of the name let you perform basic assignment, arithmetic and similar operations on variables:

```
    numTrousers = trousersPurchased;    // simple assignment
```

This simply copies the data in the variable on the right to that on the left.

In C# and Javascript—in fact, in many programming languages that are members of the 'C' family—any text placed after a double-slash is considered a comment and is ignored by the compiler and, consequently, the computer.

```
    totalPrice = numTrousers * priceOfTrousers;   // multiply
```

Multiplication and division symbols

In programming, asterisks are used to represent multiplication. Similarly, a forward slash is used for division, rather than the conventional '÷' symbol we learned in school. There are a few reasons for this. For example, most keyboards even today still lack a '÷' key.

The multiply symbol we normally use is an 'x', but, to a computer, that symbol is ambiguous: it might refer to the letter 'x', which could be the name of a variable in your program. Hence the asterisk is used in almost every programming language.

Some programming languages also let you use shortcuts to save on typing. The C family of programming languages have a lot of these:

```
    totalPrice += subTotal;    // adds subTotal to totalPrice
```

Here, we see the "+=" shortcut, which saves us having to type this instead:

```
    totalPrice = totalPrice + subTotal;
```

Increasing a variable's value by one is even easier in the C family of languages:

```
    counter++;       // increments counter by 1
```

Decreasing by one is, as you'd expect, very similar:

```
    counter--;       // decrements counter by 1
```

One caveat with these however: programming languages are very picky about the order of such operations. We'll come back to this later…

Functions

A function is a blob of program code that does something specific. Usually, the function is named after what the code it contains actually does.

For example:

```
    public int AddTwoIntegers(int a, int b)
    {
     return (a+b);
    }
```

Not a particularly useful function, but it illustrates all the key features of one:

The keyword **public** means "let other functions outside this organisational construct access this one." Almost every programming language has some

kind of organisational system used to group related functions (and, often, related data as well) together.

In C#, and all other *Object Oriented* programming languages, that organisational construct is the Class where the function is defined. We'll look at this in more detail later.

The keyword **int** means "this function returns an integer to the code that calls it".

Next we see the **function name** and the **list of parameters** it takes. (Parameters are optional, but you still need the parentheses.)

In this example, the function takes two integer parameters. The names "a" and "b" are basically placeholders for whatever variables are passed to the function by its caller.

Next, we see the C family's traditional use of **curly braces**. These are used to define a distinct block of code. In this case, the block of code is tied to the function name.

Finally, we come to the useful part of code that actually does something: the **return** keyword tells the function to 'return' the value it produces to the code that called it. In this case, we're simply returning the sum of the two parameters: a + b.

Note the **semicolon** at the end: this is another C family tradition: It means "this instruction ends here" and is a bit like the full stop at the end of this sentence.

To call such a function, we might write something like this:

```
int num1 = 10;
int num2 = 22;
int sum = AddTwoIntegers (num1, num2);
```

At this point, sum would contain 32.

> Note that **sum** is an **integer**, because we defined **AddTwoIntegers()** to **return** an **integer**.
>
> The sentence above puts a pair of parentheses after the function's name. This is another 'C family' programming tradition: it makes it easier to tell that this is a function name and distinguishes it from a variable name.

A note on naming conventions

As the box over there to the right explains, all these variable and function names are there purely for the benefit of us humans. So it makes sense to pick names we can both type and understand easily.

When using programming languages that are members of the C family of languages, a common convention is to use nouns for variables, and verbs for function names. Another convention is to start function names with a capital letter, while variables start with a lower-case letter.

> **What's in a name?**
>
> In modern programming languages like C#, all these names, "num1", "sum", "AddTwoIntegers", etc. are just labels for **our** convenience; when the code is compiled, they're all replaced by numbers.
>
> Remember: all this stuff gets reduced to that machine code we saw at the start of this document. It really **is** all just numbers.
>
> It's this conversion from human-readable words to numbers that gave programming its nickname: "coding".

Aside from that, the rest is open to personal preference and the limits of the language itself. For example, I'm not aware of any programming language that lets you include spaces in variable or function names (or, indeed, any kind of label), because this would confuse the compiler. Consider the following:

```
int this is a very long variable name = 42;
```

The above has multiple problems:

1. Both **this** and **long** are actually reserved keywords in C#: the first is a shorthand for "this object" (or "the object this code is defined in"), while the second defines a type of integer variable that uses more than the usual number of bytes for storage.

2. What if there's already a variable or function called "a" or "name"? In either case, the compiler would complain, because it doesn't know if this is an intentional definition, or a simple typo.

The upshot is that you can't use spaces in such labels. Some other symbols are also not permitted, but this varies from language to language. In C#, you also shouldn't use dots, semicolons, commas, colons, or parenthesis, for example. These all have specific meanings to the C# compiler.

I therefore use a system called "CamelCase", which, as its name and orthography suggests, uses a capital letter for each word to visually separate them from each other. ThisMeansMyVariableNamesLookLikeThis. Though I prefer to keep them shorter than that example.

Choosing the right name

As the computer never sees our label names when running our programs, we should pick names that are convenient for **us**.

Today's program editing tools, such as MonoDevelop (included with Unity) and Microsoft's Visual Studio, have a feature called "auto-complete". This means you only have to type a label out in full once. After that, you only need to type in enough letters for the program editor to work out which label you're referring to, at which point it'll offer to type the rest of it out. This is a huge timesaver. It also means you can use long labels if you like.

The golden rule, then, is to pick a name that *helps explain the program* to the reader—in technical parlance: to make the program *self-documenting*.

Avoid single-letter labels like "a" or "i", unless you're using them purely as temporary counters or placeholders.

As stated above, pick label names that *help document your program*. The more you do this, the fewer explanatory comments you'll need to include to explain what your program code is doing.

Testing, comparing and decision-making

A programming language that doesn't let you run different bits of code depending on the results of comparisons and tests is a poor language indeed. (Or, more likely, it's called HTML and is not, in fact, a programming language—despite what some of the more ignorant members of the mainstream media believe.)

> **HTML** is a **mark-up language** used to define a web page's appearance.
>
> It uses tags to specify how to display a web page's text, images and other content. It does **not** let you do fancy programming: that's done using an embedded programming language called **JavaScript**. Unity supports a slightly customised dialect of this language called **JavaScript.Net**.
>
> Confusingly, **JavaScript** is unrelated to the programming language known as **Java**; the share only the "Java" name. (JavaScript and its relatives are specific implementations of **ECMAScript**, but few people use that name.)

There is, in fact, a test for whether a programming language is a real one or not. The test is known as Turing Completeness, named after Alan Turing.

Data comparison and decision-making are key features of any programming language worthy of the name, and C# is no exception…

```
if ( x++ > 50)
{
      x = 0;
      ++y;
}
```

Here, we the variable x being compared against 50 *before* its value is increased by one.

Is the contents of x greater than 50? If so, we reset x to zero and add 1 to y.

Note that the "++" appears before the "y" here; the position of the increment operator does make a difference: If the comparison test had been written like this instead…

```
if ( ++x > 50)
… etc…
```

…then the value of x would have been increased by 1 **before** the comparison test was made.

In the first example, the position of the ++ before the y makes no difference as we're not doing anything else with the variable.

This kind of handy little shortcut is great and all, but it can also be a source of very annoying, and very subtle, bugs. This particular operator is a holdover from the original C programming language.

All programming languages have their little 'Gotcha!' features like this one. C#, despite being relatively modern, is no exception; this feature actually comes from its spiritual ancestor: the C programming language.

Things get a bit more interesting when you start testing for equality. For example, let's consider this line from an old BASIC dialect:

```
10 LET text$ = "This is some text stored in a variable."
```

Not all BASIC dialects require the **LET** keyword, or line numbers; this particular dialect was used by the Sinclair ZX Spectrum series of computers around 30 years ago.

One advantage of requiring that LET keyword to assign data to a variable is that it lets you make comparisons more easily, without leaping through semantic and syntactic hoops the way C# and Javascript do. Because the C family of languages uses the "=" operator to assign data to a variable, it needs to use another operator to test for equality: "==":

```
if (text$ == "")
```

```
    {
        printf(“text$ is empty!”);
    }
```

For our old BASIC dialect, you can test for equality like this:

```
    120 IF (text$ = “”) THEN PRINT “text$ is empty!”
```

Because the LET keyword isn't used, the BASIC interpreter knows that this is a comparison test, not an assignment.

There are a couple of other comparison operators in the C family of languages that can catch you out. For example, to test for non-equality, you'd use:

C family: **!=**

BASIC: **<>**

However, the real danger is to use “=“.

This is because, in the C family, *any* operator can also work as a comparison operator. This is because these languages treat the conditional statement as any other program statement. To be valid, all the comparison statement has to do is return a valid boolean (i.e. “true” / “false”) value. If the value is true, the condition is said to be ‘true’; if the value returned is false, the condition is said to be ‘false’.

Boolean Variables

A **boolean** variable is just a switch that can represent any yes/no concept, like “on”/“off”, “up”/“down”, and so on.

These variables are most commonly used to differentiate “true” from “false”.

Programming languages differ in how they represent a “true” state. In the C family, “true” is taken to mean “any value that isn't zero”. In other languages, “true” might be explicitly defined as ‘-1’, or ‘1’. In all cases, “false” is always zero.

The **if** keyword expects a condition that evaluates to either **true** or **false**. Only if the condition evaluates to true does the code in the **if** statement's code block run.

You can find out more about this subject in the chapter entitled **Working with Digits.**

Loops

We often need to run through a series or sequence of data items, processing each one in turn. Programming languages therefore include *loop* constructs, like this one:

```
foreach (int i in myIntArray)
{
      i *= 3;   // multiply each int by three
}
```

The above (C#) code multiplies each integer in an array called "myIntArray" by three. Not particularly deep, granted, but it illustrates the process.

An more common form of loop construct is the "for" loop:

```
for (int i = 0; i < myIntArray.Length(); ++i)
{
      myIntArray[i] *= 3;
}
```

This does exactly the same thing as the previous foreach loop type. Note how this construct is a little more involved: you have to manually extract the length of the array from the array itself. This kind of loop construct is more common in older languages, while the foreach type is a little less common.

The *foreach* construct relies on something called an *iterator* function being defined for the data type you're processing. C# and Javascript already include such iterators for their built-in data types, but you can define such iterators for your own custom data types if desired.

The *for* loop has no such requirement.

An iterator function defines how its associated data type can be stepped through, one element at a time. For example, an array's iterator function

might simply return the array element requested of it. More complex iterator functions can be written if required.

The **for** construct is one of the oldest looping constructs and is identical to the one found in the 1970s-era C programming language. This predates the concept of iterator functions and object-oriented design. Instead, the iteration calculations are done explicitly in the loop construct itself.

In some older languages, the **for** construct *always* uses a counter of some sort. C (and C#) do it a bit differently:

The **for** construct in C# is surprisingly powerful: it comprises three statements, the first of which initialises a variable; the second defines the 'exit' condition for the loop, so it doesn't run forever; the third, final element defines how the variable is changed after each loop cycle.

Mostly, the **for** loop will use a counter, and all three of those statements defined in the **for()** construct will relate to that counter in some way. But it's possible to create some very unusual loops with it that don't even use a counter at all.

Other loop constructs include the "while–do" and "do–until" forms, but these old constructs are rarely used. In both cases, a conditional test is performed and the code within the conditional code block is only run if the condition is true. The only difference is that the "do–until" form puts the condition after the conditional code block, so that code *always* runs at least once. These are the oldest loop constructs in programming.

The biggest differences between programming languages are those due to that last, ***organisational***, element in the list…

Program Organisation

In the early days of computers, they were so basic and simple, with such tiny amounts of storage and RAM, that their software had to be equally basic and simple. Most were programmed directly in machine code—often by flipping toggle switches on a front panel covered in a Christmas tree's worth of blinking lights. *Star Trek* got at least that bit right.

As computers became more powerful, so did the software that ran on them. This led to higher-level programming languages: first *assembly language*, then languages like FORTRAN, COBOL (possibly one of the most verbose programming languages ever invented), and BASIC. The 1970s saw the birth of C—itself a descendant of an equally tersely named language called BCPL, which stood for the "B Computer Programming Language". (You can see why so few programmers have ever won awards for literature.)

C was a **procedural** language. This was in contrast to the older **BASIC** language, which more closely resembled assembly languages at the time.

In early dialects of BASIC, you could do stuff like this:

```
10 PRINT "HELLO, WORLD!"
20 GOTO 10
```

This was usually the first program most people write in BASIC. School-kids would often type this into computers on display in shops—often changing the text to taste—and annoy said shop owners.

Note the line numbers. These remained in BASIC well into the 1980s. (Unlike many other programming languages, BASIC saw little standardisation and there have been hundreds of incompatible 'dialects' of the language over its lifetime.)

At this time, the concept of closed, self-contained *functions* was still a mere glint in a programming language designer's eye. If you had some code you needed to call a few times, you'd store any data it would need in some

variables, then use the **GOTO [line number]** instruction to run it, and that code would then have to GOTO back again. (You could use GOTO with a variable in many versions of BASIC.)

As programs got more complicated, this tended to result in a horrible mess of "spaghetti" code that was very difficult to follow. So much so that a famous essay was written entitled "GOTO considered harmful", triggering the creation of programming languages that did a better job of structuring the code.

Procedural languages introduced the concept of self-contained functions. Originally called subroutines, these even appeared in BASIC:

```
10 GOSUB 2000
20 GOTO 10
…
2000 PRINT "HELLO, WORLD!"
2010 RETURN
```

Admittedly, early dialects of BASIC implemented it rather crudely, but you can already see the appearance of the RETURN keyword we came across in our earlier C# examples. (Later BASIC dialects drop the line numbers and allow you to use labels instead of line numbers. Microsoft's own Visual Basic.Net is an example. You'd have a hard time spotting the family resemblance though.)

The C programming language did away with line numbers entirely, using named labels instead, but note that the effect under the hood is identical: those labels are still just location markers once the C program is compiled:

```
public void main()
{
      PrintGreeting();
}

public void PrintGreeting()
{
```

```
        printf("Hello, World!");
    }
```

Every program has to start running from somewhere: in C, that "somewhere" is the **main()** function.

(In line-numbered BASIC programs, the program always starts from the lowest-numbered line.)

The **PrintGreeting()** function prints the traditional programmer's greeting. As this function doesn't return a value to the calling function—hence the "void" keyword—there's no need to use the **return** keyword here; the function returns automatically to the caller when the computer reaches its closing curly bracket.

This very short program also ends all by itself after the PrintGreeting() call, because the main() function itself ends at that point. Like the PrintGreeting() function, the main() entry function doesn't return a value either, hence the 'void' keyword in its own definition.

This was fine for a while, but as programs got bigger and bigger, it became harder for teams of programmers to work on them. Organisation of large software projects became a real headache, because code and data were still mixed together in an increasingly nasty mess.

The next evolutionary step, then, was to split code and data *object orientation*. This approach combined code and its related data into a single **object**.

The idea was to treat each such object as a 'black box' that only offered a few functions to other objects. The actual workings of that object—its code and data structures—were kept hidden from other objects.

Another (mostly theoretical) advantage was the idea of being able to reuse objects written for another program, or even to buy ready-made objects off the shelf.

> This did add some programming overhead to convert data structures suitable for one object into structures suitable for a different object, but it was considered a small price to pay. The tutorial that accompanies this book includes a slightly contrived example of this.

C# is an object-oriented programming language. So are Javascript and Boo. This approach is still very popular and Unity itself is also entirely designed around the concept.

In C#, classes are defined explicitly. Here's part of the Player class from the Tic-Tac-Two tutorial:

```
public class Player
{
	private bool _isX = false;
	public pieces PlayingAs {
		get {
			return _isX ? pieces.X : pieces.O;
		}
		set {
				if (value == pieces.X)
					_isX = true;
				else if (value == pieces.O)
					_isX = false;
		}
	}
	private string _name = string.Empty;
	public string PlayerName {
		get {
				return _name;
		}
		set {
				_name = value;
```

```
            }
        }
        private bool _isHuman = false;
        public beings PlayerIs {
            get {
                    return _isHuman ? beings.Human :
beings.AI;
            }
            set {
                    if (value == beings.Human)
                            _isHuman = true;
                    else if (value == beings.AI)
                            _isHuman = false;
                }
        }
        … [some code omitted] …
}
```

(Note: I've removed the comments from the code as they just get in the way of this example.)

At first glance, this appears to contain only a data structure and no program code as such, but the **get** and **set** definitions are actually small functions that pass data into and out of the object's variables. These are **accessor** functions for the variables the class maintains.

Accessor functions—**accessor methods** in object-oriented programming jargon—are often used to sanity-check data passed to the object to make sure it isn't out of the accepted limits in some way. The variable **value** is a special variable supplied by C# itself: it refers to the value the calling object wants to pass into this one.

This is why you can see pairs of variables. The first begins with an underscore and is tagged as **private**—this variable is hidden from view by other objects and can *only* be accessed by the code defined in this class. This variable is immediately followed by a **public** variable that also defines the accessor

functions. These accessor functions either **set** that hidden variable according to the value passed in, or **get** the value from that hidden variable and return the desired information to the caller.

The public variables *aren't actually variables at all*: no storage space is defined for them. We only have those accessor functions. But to the functions that use them they look *exactly* like ordinary variables.

The advantage of this approach is that you can do additional processing of the data passed in (e.g. to check if it's valid), or create "read-only" variables that simply return a value calculated from the data stored in other variables, or even create "write-only" variables if desired. As far as the functions that access these variables are concerned, there's no difference between them and an ordinary variable.

Classes vs. Instances: When is an object not an object?

In most Object-Oriented Programming ("OOP") languages, you *define classes*. However, a Class definition—like the one shown earlier—is *not* in itself an object: it is a *recipe* for ***instantiating*** objects. Just as a recipe in a recipe book is not, in itself, a lasagne you can eat, so a class merely tells the computer how to cook up an object of that class' type.

To create an object of the class' type, you have to **instantiate** it. In our Player example, that process looks like this:

```
    Player player1 = new Player();
```

Here, **player1** is our new object. It's created using the **new** instruction, which you'll find in all Object-Oriented relatives of the C programming family (and in quite a few others).

This instruction tells the computer to set aside a block of memory just big enough to store the Player object's own variables.

The functions defined inside a class are shared across all classes, so there's no need to set memory aside for those. Behind the scenes, the compiler includes a memory pointer that points to the relevant data when the function is called. This pointer is passed as a hidden parameter to each function, so it always knows where its data is.

The key takeaway here is that a class definition *explains how to create* an object, just as a cookbook's recipe for lasagna *explains how to create* a lasagna. When you want to make a lasagna, you look up the recipe and follow its instructions—a process called "cooking". In programming, that same process is called **instantiation**, but it means much the same thing.

Enumerations & Constants

A **constant** value is a value that never changes. In C#, there are two kinds of constant. The first is a single, fixed constant, e.g.:

```
public const float PI = 3.1415927;
```

The second is the enumeration.

Missing from the earlier long code snippet are the definitions of both *pieces* and *beings*. These are enumerations, defined in the same file as follows:

```
public enum pieces
{
      X,
      O }
;
public enum beings
{
      Human,
      AI }
;
```

These two enumerations define **pieces.X** and **pieces.O**, and **beings.Human** and **beings.AI**. Both **enum**erations are really just integers under the hood, so each pair of enumerations is defined as numbers 0 and 1 respectively. The more values you define in the enumeration, the higher the count goes.

In some programming languages, an enumeration is stored as a simple integer. In C#, an enumeration is a distinct data type. You can't just store these in an ordinary **int** variable: you have to define the variable as being of the enumerator's own type. E.g. this is okay…

```
    public pieces thePiece = pieces.0;
```

…but this is not:

```
    public int thePiece = pieces.0;      // error!
```

An enumeration is a set of *related* constant values (e.g. days of the week, country names, etc.) that avoid the need to use numbers in your code. C# lets you use these to make your code easier to read, and also to reduce the potential for bugs: it will report an error if you try and store anything else in a datatype defined as using an enumeration.

One-off **const**ant values can be **int**egers, **float**ing points, **string**s, and so on. For example, I use two constants in the Tic-Tac-Two project that define the board width and board height.

I use all-caps to label my one-off constants, so these two are named BOARDWIDTH and BOARDHEIGHT respectively. You'll see them defined in the BoardManager.cs file. This naming convention comes from the original C programming language, but isn't as common in C#.

As the computer never sees these labels, it doesn't make any difference what naming conventions you use, as long as you're consistent. The only major exception is when you're working in a team, in which case you should all stick to an agreed naming convention to avoid serious headaches later on.

Conclusion

These *paradigms*—and there are rather more of them than I've listed here—are fundamentally about *organising* your program. The only other differences between programming languages are their syntaxes: whether they use lots of brackets and semicolons, or rely on white space (e.g. indentation, etc.). These differences aside, they're almost all the same.

Working with Digits

Computers know nothing but a world of numbers. Every item of data they are tasked with processing must be given in the form of numbers, so it helps to understand some of the ways we can do this…

Binary

At the physical level, computers work with **binary** numbers. This is exactly like our own counting system except there are only two digits, not ten. (Mathematicians refer to binary as a **Base 2** system.)

The reason for this is simple: computers are electronic devices and are basically just collections of millions upon millions of tiny on-off switches.

Programmers today rarely work directly with binary numbers, but there are a few occasions when it comes in handy to understand them. It's particularly handy when dealing with networking, for example.

Most of us are familiar with the **Base 10** system we use every day. We call this system "decimal" and it's based around the use of ten digits: 0 – 9.

Decimal was not our first number system. Before we standardised on this, it wasn't uncommon to find other systems in use, usually based on 20 or even 12 digits (the former is technically known as "vigesimal").

> Some traces of both systems remain in the English language today: note how we don't settle into a consistent counting sequence until we hit 'twenty'. And the 'teens' don't start until 'thirteen'.
>
> In Abraham Lincoln's day, it was still common to count in 'scores', as in 'Four-score and seven years ago…'
>
> In France, they still use multiples of twenty in their counting system: e.g. the number '90' is given as 'quatre-vingt-dix', or "four-twenty-ten". (In

neighbouring Belgium, they've dropped this and use "novante" instead. Ditto for French-speaking parts of Canada.)

The Welsh language also retains its ancestor's vigesimal system.

Just as our own decimal system has a spatial component, so does binary.

A decimal number, like 237, can be read as "two times 100, plus 3 times 10, plus 7 times 1". The "100", "10" and "1" come from this:

10000000	1000000	100000	10000	1000	100	10	1
0	0	0	0	0	2	3	7

Note how each column heading is ten times that of the one to its right. That's because each column is ten raised to the power of the column number. Thus, the 'hundreds' column is 10^2, while the 'thousands' column is 10^3. And so on. (For the units, or '1', column, it's 10^0. This is a rare occurrence of humans counting from zero, like computers, rather than one!)

Binary has a similar system, except instead of raising ten to a power, we raise *two* to the same power:

128	64	32	16	8	4	2	1
1	1	1	0	1	1	0	1

The above shows the same number, 237, in binary. To convert to decimal, add together all the columns with a '1' in them: 128+64+32+8+4+1 = 237.

(Mercifully, for people like myself who are terrible at arithmetic, there's no multiplication involved here!)

Of note is that binary numbers only *look* big, because they need more digits to represent the same number that our decimal system can represent in just three.

Hexadecimal

This is another counting system that is a little more common in programming. This is a **Base 16** system, which uses *more* numbers than our own decimal (base 10) system. As we only have symbols for ten numbers, 1–10, the additional 'numbers' are actually the letters A through F.

The reason for hexadecimal's popularity in programming is that it lets us represent an entire 4-bit binary number with a single digit:

128	64	32	16	8	4	2	1
1	1	1	0	1	1	0	1

Above we see the same binary number we saw earlier: 237.

In hexadecimal, this would be: ED. Just two symbols to type, instead of three.

The key point here is that hexadecimal lets us represent an entire 4-bit binary number with just one symbol, 0–F. The letters A through F are used to represent the numbers 11, 12, 13, 14, and 15. This is very convenient because this:

8	4	2	1
1	1	1	1

Can be represented as either **1111**, or, more simply, as **F**, in hexadecimal.

The main reason is that it's a lot easier to visualise the binary representation of a number when it's written out in hexadecimal as each digit corresponds exactly to four binary digits. (Another reason is that many programmers simply dislike typing!)

Finally, a note on notations: C# doesn't even let you type in a binary number directly. (Its ancestor, C, can. As can some of its older relatives, like Objective-C.) So we're stuck with using decimal or hexadecimal notation.

In C#, hexadecimal numbers must be prefixed with an '0x'—a zero followed by a lower-case 'x'—so we can type either 237 (decimal), or 0xED (hexadecimal). The latter clearly requires more typing than the decimal form, but it does make it easier to work out the binary form. You'll see hexadecimal used more often for longer numbers as the typing advantage improves the larger the number becomes.

And now, this…

Boolean Logic

Most of us are familiar with basic arithmetic operations like addition, subtraction, multiplication, division and so on. However, this is not the only kind of mathematics out there.

Boolean Logic is a language for the mathematics of *logic*, rather than sums. It was invented by George Boole in the mid-1800s, hence the name.

I'm not going to explain the formal mathematical notation George Boole invented for describing logic operations as few programming languages support it.

Boolean logic is all about true and false. (Or, more accurately, "on" and "off", given that computers are made almost entirely of on-off switches.)

In fact, many languages support a **bool**ean data type that can be set only to *true* or *false*. This requires only a single bit of data to store, though most

compilers will cheerfully use an entire 32-bit **int** behind the scenes, because it's usually quicker.

Boolean logic is something we use frequently in everyday life without realising it: I might want either Item A **or** Item B, but not both. Or I might want both Item A **and** Item B.

Both **OR** and **AND** are Boolean operators. We use them in programming all the time, where they are used in one of two ways: either to combine *conditional tests*, or—much less frequently—to manipulate and combine individual binary digits, or *bits*, together directly.

If we want to do something only if a variable, **aValue**, is both greater than 10 **and** less than 20, we would write something like this:

```
if ( (aValue > 10) && (aValue < 20) )
{
     // do something here
}
```

If we want to do something if **either** that same variable is greater than 10 **or** less than 20 (i.e. it could be set to 5), we'd write something like this:

```
if ( (aValue > 10) || (aValue < 20) )
{
     // do something here
}
```

Note that the only difference is the conjunction symbol: two vertical bars rather than two ampersands. (These operators are a trait of the C family of languages; many other languages simply use "AND" and "OR" keywords instead.)

The **NOT** operator is even easier: if we only want to do something if **aValue** is **not** greater than 10, we'd write something like this:

```
if ( !(aValue > 10) )
{
     // do something here
}
```

That exclamation mark flips the condition's result, so if the condition comes up as false, it becomes true, and vice-versa. The upshot of which is that the code between the curly braces only runs if "aValue > 10" is **NOT** true. (Note that this means the code will also be run if aValue *equals* 10.)

All conditional tests must return a **bool**ean value of either true or false. There's no "maybe": computers are very bad at handling maybes. They have to have shades of grey explained to them very carefully.

Bitwise logic

The double-ampersand and double-vertical bar operators are used only when working with conditional tests. However, if we make them single symbols, we can work directly with individual ***bi****nary digi****ts***—or ***bits***.

Suppose we have some true / false values we want to store. If storage space is scarce, it might not be sensible to store each bit of data in its own **integer** variable as that variable might soak up as many as four whole bytes. Why waste 32 bits of storage for a single bit of actual data? Often, such values are *packed* into a single variable instead.

> This sort of data compression is less useful in reducing storage space as storage is so cheap now. However, compression is still very much in use today as a means to speed up transmission times when sending data over a network.
>
> Storing and extracting the individual bits of data can be a little more involved than just reading an integer variable, but the added processing time is a tiny, tiny fraction of the time you'll spend sending and receiving the data, so it's processing time well spent.

But it's not just single-bit data we can store. A Tic-Tac-Toe game board could be stored using just 18 bits of data: 9 for "X" and another 9 bits for "O". (Where both sets have a '0' bit, the corresponding square is empty.)

So how do we pack data into a single variable?

It turns out that the C# programming language has inherited some features from its ancestor, C, that are very useful when manipulating bits. These are the ***bitwise* operators**: AND, OR, XOR and NOT.

The differences between these operators are best illustrated using what are called "truth tables", which show which combination will result in a 'true' (or, in C#'s case, "1") value:

X	*AND*	*Y*	=
0	*0*	*0*	
0	*1*	*0*	
1	*0*	*0*	
1	*1*	*1*	

Note that we only get a '1' (or 'true') result if both X and Y are '1'. (It may help to read '0' as 'false'.)

The OR operator works like this:

X	*OR*	*Y*	=
0	*0*	*0*	
0	*1*	*1*	
1	*0*	*1*	
1	*1*	*1*	

Here you can see that if *either* of the two input variables is 'true', then the result is also 'true'.

The less-used XOR operator is slightly different:

X	*XOR*	*Y*	=
0	*0*	*0*	
0	*1*	*1*	
1	*0*	*1*	

1	*1*	*0*

Here, the result is 'true' only if *one or the other* of the two inputs is 'true', but if *both* are true, the result is 'false'. This operator is used most often in low-level programs, like device drivers.

Finally, there's the simple NOT operator, which only takes one input and flips it:

NOT X	***=***
0	*1*
1	*0*

Simple!

The next set of operators are to do with *shifting* the bits along to the left or the right. These are the **logical shift** operators (**<<** and **>>** operators respectively).

The logical shift operators simply shift all the bits in a variable along by the specified amount. Any bits that fall off the end are simply discarded, while all new bits are zeroes.

(There is another form of logical shift, called the *circular shift*, or *rotation*, but there's no dedicated operator for these in C#.)

Typically, packing data into an **byte** variable is done using the bitwise **OR** operator, which is just a single vertical bar.

For example, let's say we want to pack the following **bool**ean variables into just one **byte** for later transmission over a network:

```
    bool isPlayerRunning;
    bool isPlayerJumping;
    bool isPlayerStanding;
```

We'll assume these variables have been assigned their data elsewhere. Now we have to pack them into a single variable:

```
byte PlayerStatus;
```

Our boss has told us that we need to store the bits in the following positions:

This shows the first eight bits of the **PlayerStatus** variable. The three bit positions we're interested in are 1, 2 and 4.

> **Note:**
>
> The right-most bit in the diagram above is labelled 0 because it has an *offset* of zero from the start of the byte.
>
> You'll see this often in programming. Some programming languages start counting from '1' to make life easier for us, but this hides what's really happening under the hood.
>
> All the programming languages in the "C" family use indexing from zero, not one.

Our boss has also told us that we can't assume that there isn't already valuable data in that byte. So we can't clear it first.

Unfortunately, we can't just tell C# which bit to store each of our three boolean variables in directly. We'll have to use **logical shifts** and the **OR** operator to store our three boolean variables.

Let's start with **isPlayerRunning**:

```
PlayerStatus |= (byte)isPlayerRunning << 4;
```

Here, we're **OR**ing the isPlayerRunning data into PlayerStatus, after first *shifting it to the left* by four bits.

The bitwise OR operator means we're not going to change any of the other values.

> Note the '**(byte)**' cast: this tells the compiler that we know the two data types aren't the same, and that **isPlayerRunning** should be converted to a byte before shifting it. There's no point in applying a logical shift to a single-bit variable like a **bool**ean, but it does make sense to do so if it's converted to a byte first.

But wait: There's a step missing here! What if we're reusing PlayerStatus repeatedly without resetting it between game cycles? This makes sense as a game player's status usually needs to be retained for much longer periods than a 50th of a second or less.

So we must first clear the target bit in PlayerStatus. If we don't, we'll hit a problem: look back at the truth table for the OR operator and you'll see that if **isPlayerRunning** is false (zero), then we can't just OR that into place because we don't know if the relevant bit in **PlayerStatus** is true. If it is, the OR won't set it to zero, because 0 **OR** 1 = 1.

First, then, we need to clear the relevant bit of **isPlayerRunning** to zero, so we can then use OR to store **isPlayerRunning** correctly:

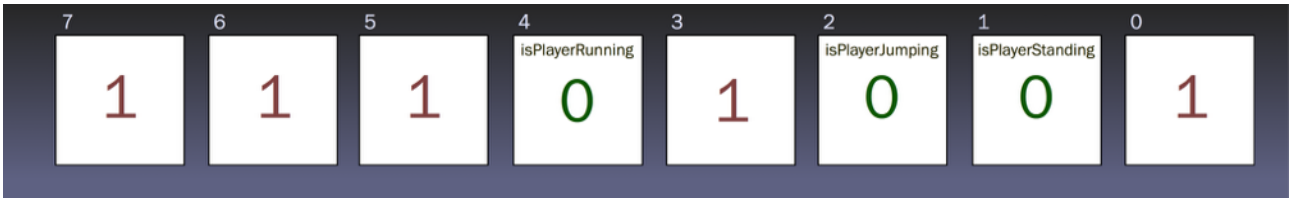
```
PlayerStatus &= (byte)0xEF;
PlayerStatus |= (byte)isPlayerRunning << 4;
```

The second line is the same as before. It's the first one that interests us: this applies what's known as a "bit mask". It has a '1' for every bit we're *not* interested in changing, and a '0' for bit 4, which is where we want to store **isPlayerRunning**'s value. The "mask" term comes from this operation's result: it effectively punches a hole in **PlayerStatus** at bit 4, ensuring it's set to zero.

In fact, we can punch such holes for *all three* of our variables at the same time:

```
    PlayerStatus &= (byte)0xE9;
```

0xE9 is the hexadecimal representation of this binary number:



(That's 233 in decimal.)

The above line of code will clear our three target bits, while leaving the other bits untouched. This is because either 1 or 0 **AND**ed with 0 is always 0.

Now we have everything we need to store all three of our variables:

```
    PlayerStatus &= (byte)0xE9;
    PlayerStatus |= (byte)isPlayerRunning << 4;
    PlayerStatus |= (byte)isPlayerJumping << 2;
    PlayerStatus |= (byte)isPlayerStanding << 1;
```

Conclusion

That's about it for this chapter on playing with bits and bytes.

Working directly with binary numbers is becoming rarer in programming today as more and more of this stuff is being provided for us by ready-made code libraries, operating systems and other platforms, but there are times when knowing about techniques like these can come in very handy.