Introduction to Programming

What Programming Actually Is

This article outlines what programming actually is, with a view to helping aspiring programmers decide if they are making the right career choice.

Scotland Software
http://web.archive.org/web/20060110133319/http://www.scotlandsoftware.com/
Contact the author via the website above for permission to republish or redistribute this article.

Latest version always available at:
http://web.archive.org/web/20060110133319/http://www.scotlandsoftware.com/articles/programming/

An important note

You won't be surprised to hear that just about everyone has an opinion on what programming is, how to get started with it, and so on. This article contains only my own opinions; not necessarily the best or most insightful views, or even correct in any way. By all means seek other viewpoints, and come to a considered, balanced judgment of why mine is far superior.

It's also important to realise the purpose of this article: not to teach you how to program, but rather to teach you **what programming is**. That seems to be a question that's mostly ignored by those who would teach you to program - they assume that you already have an idea of what programming is, why you'd want to do it, what's basically involved, and so on. This article answers all those questions for you, so you can then decide whether you want to start learning **how** to program.

With that in mind, let's begin.

What is programming?

Given the general nature of that question, and the fact that programming is seen as a hideously complex subject, you're probably expecting a highly convoluted and technical answer. But you're not going to get one (sorry about that). In truth, it's quite easy to say what programming is, so I will:

- **Programming is breaking a task down into small steps.**

That's just about the most honest and accurate answer I can give. It also has the added benefit of being concise, and sounding very much like something you'd read in an official book on the topic, thus adding to my credibility.

You're perhaps wondering what exactly I mean by breaking a task down into small steps, so I'll explain the point in more detail. Let me start by giving you a fact about programmers that you'll find very easy to believe:

- **Programmers think in an unnatural way.**

This refers just to the fact that programming is breaking a task into small steps, and that's not the usual way that your mind works. An example will help you to understand what I mean. Here's a task for you to do:

Put these words in alphabetical order: apple, zebra, abacus

I'm going to assume that you managed to put them in alphabetical order, and ended up with abacus, then apple, then zebra. If you didn't manage that, reading this article may soon become one of the worst experiences of your life so far.

Now think about exactly **how** you performed that task; what steps you took to put the words in alphabetical order, and what you required to know in order to do so. The most obvious thing you needed to know was the alphabet; the desired order of the letters. Then, if you're like me, you probably did something like this:

1. Look through the words for one beginning with "A".
2. If you found a word beginning with "A", put that word at the beginning of the list (in your mind).
3. Look for another word beginning with "A".
4. If there's another word beginning with "A", compare its second letter with the second letter of our first "A" word.
5. If the second letters are different, put the two words in alphabetical order by their second letter. If the second letters are the same, proceed to the third letter, and so on.
6. Repeat this whole process for "B" and each other letter in alphabetical order, until all the words have been moved to the appropriate place.

Your method may differ slightly, but probably not by much. The thing to notice (which I noticed particularly, since I'm having to type all this) is how much time it took to explain a process which happens without any real conscious thought. When you saw that you had to put some words in alphabetical order, you certainly didn't first sit down and draw up a plan of what you were going to do, detailing all the steps I listed above. Your mind doesn't need to; you learned to do it once when you were a child, and now it just happens. You have a kind of built-in shortcut to that sequence of steps.

Now let's try another simple example:

Count the number of words in this sentence: "Programming really can be fun."

Hopefully you decided that there were five words. But how did you come to that decision? First you

had to decide what a word is, naturally. Let's assume for the moment that a word is a sequence of letters which is separated from other words by a space. Using that rule, you do indeed get five words when looking at the sentence above. But what about this sentence:

"Sir Cecil Hetherington-Smythe would make an excellent treasurer,minister."

Notice my deliberate mistake: I didn't leave a space after the comma. You might also feel it's a mistake to name anyone Cecil Hetherington-Smythe, but that's a debate for another time. Using our rule about sequences of letters which are separated from other sequences by spaces, you would decide that there were eight words in the new sentence. However, I think we can agree that there are in fact ten words, so our rule clearly isn't working. Perhaps if we revised our rule to say that words can be separated by spaces, commas or dashes, instead of just spaces. Using that new rule, you'd indeed find ten words. Now let's try another sentence:

"Some people just love to type really
short lines, instead of using the full
width of the page."

Although it might not be obvious, there is no space after "really", nor is there a space after "full". Instead, I took a new line by pressing the return key. So, using our newest rule, how many words would you find in that last sentence? I'll tell you: you'd get sixteen, when in fact there are eighteen. This means that we need to revise our rule yet again, to include returns as valid word-separators. And so on, until another sentence trips up our rule, and we need to revise it yet again.

You might wonder why we're doing this at all, because after all, we all **know** what we mean when we say "count the number of words". You can do it properly without thinking about any rules or valid word-separators or any such thing. So can I. So can just about anyone. What this example has shown us is that we take for granted something which is actually a pretty sophisticated "program" in our minds. In fact, our own built-in word counting "program" is so sophisticated that you'd probably have a lot of trouble describing all the actual little rules it uses. So why bother?

The answer comes in the form of an exceptionally important truth which you must learn. It's to do with computers (even your own computer that you're using to read this). Here it is:

- **Computers are very, very stupid.**

To some people, that statement is almost sacrilegious. You can understand that, because computers are really expensive. If you've just bought a Ferrari, you probably don't want your neighbour to come along and say that it's ugly and slow. If you're a Mac user (as I am, so don't send abusive email about this - I'm typing this article on my beloved PowerBook), you might actually kill a person who said your computer was stupid.

Nevertheless, it's true - computers are desperately stupid. Your computer will sit there and do whatever mindless task you tell it to, for days, weeks, months or years on end, without any complaints or any slacking-off. That's not the typical behaviour of something that's even slightly clever. It will

also happily erase it's own hard disk (which is a bit like you deleting all your memories then pulling parts of your brain out), so we're clearly not dealing with an intimidatingly intelligent item.

In fact, computers are so painfully stupid that they require to be told, in **minute detail**, how to do even the most laughably simple of tasks. It's quite pathetic, when you think about it (or perhaps we're pathetic, since we're willing to pay ridiculous amounts of money to own them). In fact, just about the only positive thing about computers is that they're completely obedient. No matter how crazed your instructions might be, your computer will carry them out **precisely**.

By now, hopefully you can see how this is all tying together. Programmers tell computers what to do. Computers require these instructions to be precise and complete in every way. Humans aren't usually good at giving precise and complete instructions since we have this incredible brain which lets us give vague commands and still get the correct answer. Thus, programmers have to learn to think in an unnatural way: they have to learn to take a description of a task (like "count the number of words in a sentence"), and break it down into the fundamental steps which a computer needs to know in order to perform that task.

You may be feeling slightly uneasy at this point. I've admitted that programming is, in a way, unnatural. I've warned you about the spectacular stupidity of computers, so you're probably getting a small idea of the amount of task-description you'd need to do in order to make your computer do anything even vaguely impressive. But don't worry; there are some excellent reasons to become a programmer:

- Programmers make lots of money.
- Programming really is fun.
- Programming is very intellectually rewarding.
- Programming makes you feel superior to other people.
- Programming gives you complete control over an innocent, vulnerable machine, which will do your evil bidding with a loyalty not even your pet dog can rival.

At least some if not all of these points will instantly appeal to you as a human being, and it's none of my business which ones you find most attractive. Additionally, people have been programming for many years, and so have already written many task-descriptions ("programs") for a lot of common tasks, so you can use all their hard work to save yourself some trouble. Which is also a very attractive prospect, I'm sure.

If you're still not sure whether programming is for you, perhaps the next section will help you decide.

Who can be a programmer?

You might be wondering just who can become a programmer (probably because you've read the title of this section, and it got you thinking). The clichéd thing to do would be to say "anyone!", but I won't do that, because it's just not true. There are a few traits which might indicate that the person would be

a good programmer:

- Logical
- Patient
- Perceptive
- At least moderately intelligent
- Enjoys an intellectual challenge
- Star Trek fan
- Female *

* Actually, males and females make equally good programmers. I just said that to address the gender disparity which exists in the programming world; it's true that there are currently more male programmers than female, which is strange given that one of the first ever programmers (Ada Lovelace) was female. Besides, male programmers are especially eager to encourage more females to take up the profession, since they (the male programmers) traditionally have no social lives whatsoever. Or perhaps that's an unfair generalisation, and it's just me who has no social life.

As you can see, it does take a certain type of person to be a successful programmer (in my humble opinion). If you're not logical, you'll have trouble organising and constructing complex programs. If you're not fairly intelligent, you won't be able to break a task down into the necessary individual steps. If you're not patient, you'll deliberately destroy your computer long before you manage to finish all but the most simple programs. If you don't enjoy intellectual challenges, you'll hate every minute of programming. If you're not a Star Trek fan, you'll miss out on a large percentage of the conversation your fellow programmers will have (I'm not saying that's a bad thing). Finally, if you're not perceptive, you won't notice that the twenty consecutive hours you've spent in front of your computer have depleted your body's energy resources to a near-fatal level, and you'll die whilst designing an icon for your new application program (tip: experienced programmers avoid this problem by designing the icon before writing the program).

In other words, if you get annoyed waiting for the five beeps after your microwave finishes cooking a Pot Noodle, and you fly into a rage trying to solve the "coffee break" crossword in the tabloid newspapers, you may want to consider another line of work. That's not to say that you can't be a programmer; you just might find it more difficult than it needs to be.

By now you should at least have an inkling of whether or not programming could be a serious proposition for you. If you're interested in finding out what's actually involved in programming (the actual process of writing programs), read on. If not, then it's clear that you should instead leave the programming to others, and then buy their excellent software (hint hint).

What's actually involved in programming?

In this section I'm going to introduce some technical terms. It's unavoidable. The jargon will always get you in the end (if you choose to become a programmer, soon you'll actually be **talking** in jargon). However, I'll explain each term as we come to it, and there's a list of jargon in the following section

(with explanations, of course).

So, what's actually involved in programming - the actual process of writing programs? I'll answer that myself even though I asked the question, since I'm assuming that you don't know yet. Here's a quick overview of the process:

1. *Write a program.*
2. *Compile the program.*
3. *Run the program.*
4. *Debug the program.*
5. *Repeat the whole process until the program is finished.*

Let's discuss those steps one by one, shall we? Yes, we shall. And here goes.

1. Write a program.

I have a small amount of bad news for you: you can't write programs in English. It would be nice indeed to be able to type "count the number of words in a sentence" into your computer and have it actually understand, but that's not going to happen for a while (unless someone writes a program to make a computer do that, of course). Instead, you have to learn a **programming language**.

You're now probably having nightmarish flashbacks to French class in high school, sweating over which nouns took the verb "avoir" and which didn't. Thankfully, learning a programming language is nothing like that - for one thing, much of a programming language is indeed in English. Programming languages commonly use words like "if", "repeat", "end" and such. Also, they use the familiar mathematical operators like "+" and "=". It's just a matter of learning the "grammar" of the language; how to say things properly. Since there are many possible languages you could choose to learn (see the jargon section for a bit more on some of the languages), I won't go into any specific one here. I just wanted to make you aware of the fact that you won't be typing your programs in English.

So, we said "Write a program". This means: write the steps needed to perform the task, using the programming language you know. You'll do the typing in a **programming environment** (an application program which lets you write programs, which is an interesting thought in itself). A common programming environment is **CodeWarrior**, and another common one is **InterDev**, but you don't need to worry about those just yet. Some programming environments are free, and some you have to buy just like any other application program. Commercial (non-free) programming environments cost anything from $50 to $500+, and you'll almost always get a huge discount if you're a student or teacher of some kind. Again, see the jargon section for where to find some programming environments.

Incidentally, the stuff you type to create a program is usually called **source code**, or just **code**. Programmers also sometimes call programming **coding**. We think it sounds slightly more cool.

2. Compile the program.

In order to use a program, you usually have to **compile** it first. When you write a program (in a programming language, using a programming environment, as we mentioned a moment ago), it's not yet in a form that the computer can use. This isn't hard to understand, given that computers actually only understand lots of 1s and 0s in long streams. You can't very well write programs using only vast amounts of 1s and 0s, so you write it in a more easily-understood form (a programming language), then you convert it to a form that the computer can actually use. This conversion process is called **compiling**, or **compilation**. Not surprisingly, a program called a **compiler** does the compiling (that's why it's called a compiler and not, for example, a banana).

It's worth mentioning that if your program has problems which the compiler can't deal with, it won't be able to compile your program (in this situation, some programmers say that the compiler **puked** or **barfed** - which I'm sure you'll agree is just delightful).

You'll be pleased to hear that your programming environment will include a suitable compiler (or maybe more than one compiler: each different programming language your programming environment allows you to use requires its own compiler). Compilers are just fancy programs, so they too are written by programmers. Programmers who write compilers are a bit like gods; they make it possible for everyone else to program. So if anyone ever tells you that they're a compiler programmer, be sure to buy them a cup of coffee (they will definitely like coffee, you can be sure).

3. Run the program.

Now that you've compiled the program into a form that the computer can use, you want to see if it works: you want to make the computer perform the steps that you specified. This is called **running** the program, or sometimes **executing** it (I'm aware of the potential irony of that term). Just the same as how a car isn't much use if you don't drive it, a program isn't much use if you don't run it. Your programming environment will allow you to run your program too (as you can see, programming environments do rather a lot for you).

4. Debug the program.

You've probably heard the term "**debug**" before (it's pronounced just as you might expect: "dee-bug"). It refers to fixing errors and problems with your program. As I'm sure you know, the term came about because the earliest computers were huge building-sized contraptions, and actual real-life insects sometimes flew into the machinery and caused havoc with the circuits and valves. Hence, those first computer engineers had to physically "debug" the computers - they had to scrape the toasted remains of various kinds of flying insect out of the inner workings of their machines. The term became used to describe any kind of problem-solving process in relation to computers, and we use it today to refer purely to fixing errors in our code. I've never had an insect fly inside my computer (though I hear that cats love to lie on top of them, so be sure to de-cat your computer before debugging your programs).

You may also have heard the phrase "it's not a bug, it's a feature". Programmers sometimes say this when someone points out a problem with their programs; they're saying that it's not a bug, but rather a

deliberate design choice (which is almost always a lie). This is rather like accidentally spilling coffee all over yourself whilst simultaneously falling down some stairs, then getting up and saying "I meant to do that".

Once again, your programming environment will help you to debug your programs (indeed, you'll often find the picture of an insect shown in your programming environment to indicate debugging). You usually debug your program by **stepping** through it. This means just what it sounds like: you go through your program one step at a time, watching how things are going and what's happening. Sooner or later (usually later), you'll see what's going wrong, and slap yourself upside the head at the ridiculously obvious error you've made. Ahem.

5. Repeat the whole process until the program is finished.

And then you repeat the whole process until you're happy with the program. This is more tricky than it might sound, since programmers are never happy with their programs. You see, programmers are perfectionists - never satisfied until absolutely everything is complete and elegant and powerful and just gorgeous (though this pursuit of perfection doesn't always extend to their choice of girlfriends/boyfriends, by necessity). Programmers will commonly release a new version of their program every day for a couple of weeks after the initial release (just ask my friends about this).

As you can imagine, enjoying an intellectual challenge is an important trait to have when you're going back to correct and enhance your code many times over. You'll actually find that you can't wait to get back into your program and fix the bugs, make improvements, and refine the existing code. No, really you will!

And that's the basic process of programming. Note that most programming environments will make a lot of it much easier for you, by doing such things as:

- Warning you about common errors
- Taking you to the specific bit of code which is causing the compiler to puke
- Letting you quickly look up documentation on the programming language you're using
- Letting you just choose to run the program, and compiling it automatically first
- Colouring parts of your code to make it easier to read (for example, making numbers a different colour from other text)
- And many other things

So, don't worry too much about the specifics of compiling then running then debugging or whatever. The purpose of this section was mostly to make you aware of the cyclical nature of programming: you write code, test it, fix it, write more, test it, fix, and so on.

Now, we digress for a moment (or however long it takes you read the next section), to talk briefly about something called scripting.

A brief word about scripting

You've perhaps heard about something called **scripting**, or maybe you've heard of languages like **JavaScript**, **AppleScript**, **Tcl** and others (those languages are called **scripting languages**). You may thus be wondering if scripting is the same as programming, and/or what the differences are, and so on. People get quite passionate about this question, so I'm just going to cover it briefly and technically. Here are some facts:

- Scripting is essentially a type of programming
- Scripting languages have a few minor technical differences which aren't important to discuss at this point
- Scripting languages tend to be **interpreted** rather than compiled, which means that you don't need to compile them - they're compiled "on the fly" (i.e. when necessary, right before they're run). This can make it faster to program in them (since you always have the source code, and don't need to take the deliberate extra step of compiling)
- The fact that scripting languages are interpreted generally makes them slower than programming languages for intensive operations (like complex calculations)
- Scripting languages are often easier to learn than programming languages, but usually aren't as powerful or flexible
- For programming things like applications for personal computers, you'll need to use a programming language rather than a scripting language

Scripting languages can be excellent for beginners: they tend to be easier to learn, and they insulate you from some of the technical aspects of programming (compiling, for one). However, if you're serious about programming, you won't be able to stay with a scripting language forever - you will move on to a programming language at some point. I'd say that it's good to know a scripting language or two, and even to start with a scripting language rather than a programming language. However, there's a point of view which says that, by protecting and "hand-holding" too much, scripting languages don't properly prepare you for "serious" programming, and set you up for a bit of a learning curve when you move on to a programming language. I can't really tell you whether that's true one way or the other; all I can say is that I started with scripting languages like JavaScript and AppleScript, and moved on to programming languages. I'd advise that you try a programming language, and if you find it really hard going, try a scripting language to ease you into thinking like a programmer.

We now move on to the "where to start" section, which is funny considering how much we've talked about already. Such is life.

Where to start

The question of exactly where to start learning to program is a big one, and it's something that everyone has an opinion about (even your postman may have a strongly-held opinion on this subject). I'm going to try to be as practical as possible about answering this, but just be aware that personal preferences (in terms of programming languages, books, programming environments, and so on) will

always creep in somewhere. I do however promise to be as impartial as possible (which isn't too difficult when you're as unquestionably great as I am).

You may think that the obvious first step would be to choose a programming language, probably by weighing up the various pros and cons of each major language. However, when someone begins a sentence with "you may think that", there's a fair chance that they're going to tell you that you were wrong to think whatever it was. I'm not exactly going to do that; instead I'm going to point out something that you must realise before going any further:

- **Programming is programming**

You're probably thinking "you don't say" (or something less kind), but there really is a valid point in there. You see, there are fundamental similarities between all programming languages, so as a beginner it's really not so important which language you begin with; the important thing is to just get some experience with programming. So, you're really free to choose a language, up to a point. Just be aware that there's no "wrong" language to start with.

Having said that, there are of course some other considerations you might want to take into account, for example these:

- Do you want to pay for a programming environment, or use a free one?
- Do you want to learn with a language you'll still be able to use professionally?

You see, each programming environment will only support certain languages. If you're going to use a free programming environment and compiler at first, you may have a limited choice of languages. Also, you may (understandably) want to learn with a language which you'll still be able to use professionally. That's a normal feeling, and it's probably a good idea too, but just don't feel that you **have** to learn with a "commercially viable" language - that's nonsense. As I said, just get started programming, whatever the language may be - your skills will be entirely transferrable to other languages, and you'll find each new language much (**much**) easier after learning your first one.

Let's assume that you want to use a commercial language (meaning one which is used by professional programmers in the current mainstream software industry), and you're willing to pay for a programming environment if need be. The main choices basically boil down to these, in no particular order:

- C
- C++
- Java

(Remember that you can find out more about these languages in the jargon section, later in this article). For now, let's talk about each language very briefly.

• **C**

This is probably the most widely-used, and definitely the oldest, of the three languages I mentioned. It's been in use since the 70s, and is a very compact (i.e. not much "vocabulary" to learn), powerful and well-understood language (by "well understood", I mean that there's a huge body of literature and example source code related to C). It would be an excellent choice. I'd recommend it to you as a starting language if you asked me for my own opinion (but you didn't ask me, so I won't say anything).

• **C++**

This language is a superset of C (that just means that it's C with more stuff added; it's more than C, and includes pretty much all of C). Its main benefit over C is that it's **object oriented**. Don't worry about what that means right now, but feel free to check the jargon section for more about it. The key point is that object oriented languages are more modern, and using object oriented languages is the way things are done now in the programming world.

However, don't take that the wrong way. You'll notice that I said C++ included pretty much all of C (in case you're wondering; there was no "C+"). This means that C is at the core of C++, and you need to know C in order to use C++. I also said that C++ is essentially an object oriented version of C, but it's not the **only** enhanced version of C which is object oriented. You've got basic C, and you've got a few languages which are object oriented, enhanced versions of C, and C++ is only one of those. What I'm trying to convey by saying this is that it's a better idea to learn C, then you can move to whatever object oriented version of C you want to. If you dive in and learn C++ first, you may find it harder to switch to other C-based object oriented languages since you'll have to "forget" parts of C++. I hope that makes sense.

Let me just balance that (which was my own opinion) by saying that C++ is most definitely the second-most used language after C, and may soon become the most used language. I'd say that it's certainly the language of choice for most commercial application software development on Windows and Macintosh, which says a lot. I just feel that you can learn C and not suffer (you can still go on to C++ afterwards, very quickly indeed since you'll already know the vast majority of C++ by virtue of knowing C), and still also leave your options open.

• **Java**

You've almost certainly heard of Java; it's been hyped all the way to the moon and back. Java has a benefit which other programming languages lack: it's **cross-platform**. So what exactly does that mean? Well, it means that it runs on more than one **platform** without needing to be recompiled. A platform is just a particular type of computer system, like Windows or Mac OS or Linux. Normally, if you wanted to use the same program on a different platform from the one it was written on, you'd have to recompile it - you'd have to compile a different version for each different platform. Sometimes, you'd also need to change some of your code to suit the new platform. This probably isn't surprising, since the different platforms work differently, and look different (an alert window on Windows looks different than an alert window on Mac OS, for example, and they both use different

code).

Anyway, Java can run on more than one platform without needing to be recompiled. You can understand why that would be a serious advantage. However, Java also has a disadvantage which is almost as serious: it's slow. Java achieves its cross-platform trick by putting what is essentially a big program on your computer which pretends to be a little computer all of its own. The Java runs inside this "**virtual machine**", which runs on whatever platform you're using (like Windows or Mac OS). Because of this extra layer between the program and the computer's processor chip, Java is slower than a program written and compiled natively for the same platform. Hopefully you can roughly understand why that might be. It's for the same reason that it's quicker to speak French to a French person directly than to speak English to an interpreter, and have the interpreter repeat what you said in French to the French person. It's exactly like that with Java and its Virtual Machine. In fact, I'm rather proud of that analogy (so don't steal it).

Having said all that, remember that I also mentioned how much Java has been marketed and hyped. Because of this, Java has become very popular. Another reason for its popularity is that it runs inside web browsers, letting programmers create little applications which can run on web sites. This is obviously a big attraction, given the explosion of the internet over the past several years. Due to this popularity, there are a lot of jobs out there for Java programmers at the moment. I'm just making you aware of this fact as it's naturally going to be one of your considerations. I wouldn't say that there's more demand for Java programmers than for C or C++ programmers, but there will probably be more demand for Java programmers than for programmers who use languages I've not yet mentioned.

Now let's talk about books, and more generally about reading material, help and tutorials. One of the first things you should do after thinking about a language is to research it a bit. Do a web search, and you'll doubtless find many thousands of relevant web sites. Take a quick look around a few of them, and you'll notice that there are plenty of mailing lists (email lists which you can send mail to, and everyone who's on the list receives it and can reply to it) about programming languages, so those are an excellent place to get help (but do learn at least some of your programming language first). You'll quickly see how popular a language is by how much material you find about it on the internet. You'll also be able to find programming environments, both free and commercial, in the same way. When you're comfortable that you've chosen the right language for you, it's time to get a book.

Don't let anyone tell you otherwise: there's no substitute for being able to hold a book. I'm not claiming that they'll never be replaced with Star Trek-like electronic pads; I'm just saying that they won't be replaced until the pads offer the same light weight, sharpness of text, and portability. At the moment, you can only get that in a good old-fashioned book (and don't say that laptops offer all that; they don't, and I know because I'm typing this article on a very nice one, which is nevertheless not as enjoyable to read as a printed book). But I digress.

You'll want to find a suitable beginner's book on your chosen programming language, and you'll be pleased to know that there will be plenty to choose from (particularly so if you chose one of the three languages I mentioned earlier). Generally speaking, go by the opinions of others, not the blurb on the books themselves. An excellent way to do this is to read the customer reviews at places like amazon.com (and amazon.co.uk, of course). Some of the comments are inevitably semi-literate

nonsense, but you'll get an excellent general impression of the book's quality and relevance. Just one tip about that: when weighing up the reviews, discard the top and bottom 5-10% (in terms of ratings). There will always be overly negative people, and there will most certainly always be those who lavish excessive praise on anything they lay hands upon (perhaps surprisingly, you'll always find more of them than of the overly negative people). But that's just common sense.

I can't give a specific suggestion regarding Java since I haven't read many Java books (though I will say that the book I have here now is *Beginning Java 2* by Ivor Horton, published by Wrox). Regarding the other two languages, I would suggest titles as follows:

• **C**
The C Programming Language by Brian W Kernighan and Dennis M Ritchie, published by Prentice Hall. Make sure you get the 2nd Edition. Let me just say that this was written by the people who created the C language, and is seen as the classic programming text. It's commonly referred to as "K&R", after its authors. However, it's quite challenging, and I wouldn't tend to recommend it for beginners. For beginners I would tend to recommend you **also** get:

The Absolute Beginner's Guide to C by Greg Perry, published by Sams. This is great because it gives a good introduction to programming in general as well as giving you a good grounding in C. It also talks about how to use various popular programming environments, which will be useful if you've never used one before.

• **C++**
The C++ Programming Language by Bjarne Stroustrup, published by Addison-Wesley. This was written by the programmer who created C++, so it's certainly authoritative. I don't have this one myself, but I would presume it will be similar (in terms of its target audience) to *The C Programming Language*, detailed above. Make sure you look for a suitable beginner's C++ book also.

You might feel that this section is a bit dismissive, but you really will benefit hugely from getting hold of an appropriate book and working through it. Let me just finish this section with two tips for when you're going through your new books:

1. Take your time, and don't expect to go too fast.
It may take you an hour to go through a single page and really understand everything (you may particularly find this with *The C Programming Language*, which is focused almost to the point of terseness), but that's fine. The point isn't to get through the book; it's to understand the subject. A programming textbook is like any other textbook; you don't read it, you work through it (and you keep going back to it). It will be an extremely enjoyable and enlightening experience, so enjoy every moment of it.

2. Do the exercises.
That sounds like a really obvious thing to say, but you'd be amazed how many people don't do the exercises - they don't type in the code they see in the book, they don't run it, and then they come away from the book when they're only about a third of the way through, having found that they suddenly ran up against an impossible-to-understand section after they thought they'd been doing so well. It's an

old story indeed (I know, because I have first-hand experience - I never used to do the exercises, and only when I finally tried doing it "by the book" did I suddenly realise how important they were).

Just do it to humour me, or as thanks for writing this huge article. I'll go so far as to say this: if you don't do the exercises, you may still understand every bit of the book, and get through it all, but you won't have learned even half of what you might have. Also, you won't be as good a programmer - I guarantee it. There are so many things you miss by not doing the exercises:

- You miss typing in the code and making typing errors which make the compiler puke
- You miss having to debug your program because you mistyped something in it
- You miss the immersion you get from editing the code yourself
- You miss the feeling of control you get from typing the code in yourself
- You deprive your brain of the chance to assimilate the code line by line

Really, it's **that much** of a loss. You might think that the first two points are reasons not to do the exercises, but you'd be wrong - we learn much more from those kinds of mistakes than from reading some code in a book. If you learn nothing else from this article, just learn that you should always do the exercises. I promise you'll thank me later (or at least you would if you remembered to).

Now, at last, it's time to move on to the dreaded jargon section, where grown men and women have been known to go completely insane after only moments of exposure (though that might just be hearsay). It's not required reading, but it might really help to give you a leg up on some of the terms which you will definitely run into at some point. You can also be fairly sure that nowhere else will you find them explained as sensibly and clearly as they are here (you'd be shocked at the bizarre and convoluted definitions and explanations some people come up with for programming jargon - those people often go on to write documentation for major application programs).

Programming jargon

This section lists various terms related to programming (and computers in general, but only those which are in some way relevant to programming), in alphabetical order. The terms aren't split into categories or topics, because it's presumed that in most cases you wouldn't be sure which category to look at in the first place (people really do categorise glossaries; those people should probably be forced to take part in Star Trek trivia competitions with expert programmers).

Note that terms in **bold** are defined in this section, so if you see a bold term you're not familiar with, just look in this section for an explanation. Now, without further ado, here we go.

• Ada
A **programming language**, sometimes with a number after the name (e.g. Ada95). Used commonly for defense applications (particularly in the UK). A fairly common choice as a first language in university computer science courses (again, particularly in the UK). Named after Ada Lovelace, a very clever lady who was one of the first ever programmers (I told you that gender had nothing to do

with programming ability).

• API
An Application Programming Interface. An API is a kind of predefined standard for how you should write certain types of programs, and commonly refers to a standard for how to write programs which interact with and extend other programs. For example, you probably know that your web browser can use plug-ins to allow it to display more kinds of content. You might have a plug-in that lets you view PDF files, or a plug-in that lets you view Quicktime movies. The reason people can write these plug-ins is because the browser has a "plug-in API" - a predefined set of rules that programmers should follow if they want their plug-ins to work with the web browser.

APIs are everywhere. The reason all Windows applications look much the same (the same style of buttons and checkboxes and menus) is that there's a Windows programming API. Any time you're defining a standard way of writing code to work with something, you're creating an API. If someone asks about the API for something, they're asking for information on how to properly write programs which work with whatever the thing is.

• Apple Computer
A computer **software** and **hardware** company based in Cupertino, California. Apple are the creators of the **Macintosh** (sometimes just called Mac) line of computers, and of the **Mac OS operating system**. Apple invented many of the technologies which all computers use today. People who own Macs are typically intensely fond of their computers, and passionately evangelise them. Many surveys have found that Mac users are amongst the most loyal computer users in the whole computer industry.

• application
A program designed to let you accomplish a specific task. For example, a word processor (like Microsoft Word, or WordPerfect) is an application. So is a graphics program like Photoshop or Paint.

• application framework
A sort of template to help you make **applications**. With some **programming languages**, there's quite a lot of work involved in creating a full application for a **GUI operating system**, so it helps to have the basics already done for you. A popular application framework is PowerPlant, which is included with the **programming environment** called **CodeWarrior**.

• ASCII
American Standard Code for Information Interchange. A way of specifying alphabetical letters, digits, punctuation marks and so on as numbers. Computers deal with numbers by design, so it's easier and more efficient for them to represent things as numbers rather than as anything else. The ASCII system lets you represent letters and other characters as numbers; for example, a capital "A" is represented by the number 65. Sometimes, people use the term "plain text" to refer to text which is encoded as ASCII (notably in email programs).

• barf
Also sometimes **puke**. To run into a programming error; usually used to refer to a **compiler**, in this

form: "I'd left out a semicolon, and the compiler just barfed."

• BASIC
A **programming language**, or more accurately a family of programming languages since there are many versions of BASIC. The term stands for Beginner's All-purpose Symbolic Instruction Code, which sounds technical but really isn't: it's for beginners, it's all-purpose rather than being designed for a specific type of programming (quite a few languages are designed for just one type of programming, like business programming or scientific programming), and it's a "symbolic instruction code" which just essentially means that it's a programming language.

As its name implies, any version of BASIC tends to be good for those who are just starting to program. For that same reason, BASIC tends to have a bad reputation amongst "serious" programmers, for being "too simple" or inflexible. Of course, the real reason they're annoyed is that BASIC makes it even easier for people to take up programming, so there's more competition for all the fabulous **software** products programmers create.

• binary
A way of counting which involves just two digits: 0 and 1. Our normal way of counting (with ten digits) is called decimal. Computers use binary since processors really have millions of little switches, and switches can only be in one of two states: on or off. So, it was sensible to use binary since you can represent off with 0 and on with 1.

Sometimes you'll also hear programmers referring to "a binary". This means the same as **object code**; it's the **compiled** version of a program.

• Borland
A **software** company which makes **programming environments**, notably for **C++** and **Java** programmers.

• C
A very popular **programming language**, created by Dennis M. Ritchie. C is a very compact and quite easy to learn language, and is an excellent choice for those intending to make a career out of programming. It is an especially good first language, since you can then move on to any of various **object oriented** versions of C, or other fairly C-like languages such as **Java**.

• C++
A **programming langauge** which is an **object oriented** version of **C**, and which was created by Bjarne Stroustrup. The modern language of choice for **Windows** and **Mac OS** application development.

• Carbon
An **API** for the **operating system** called **Mac OS X**, which also can be used with **Mac OS** versions 8-9. Carbon is an updated version of the **Macintosh Toolbox**.

• **COBOL**
COmmon Business-Oriented Language; a **programming language**. As its name implies, COBOL was designed to be used for writing business programs. It was very popular during the 1960s and 70s, but is now seen as an essentially dead language.

• **Cocoa**
A very tasty hot drink. In computing, it also refers to an **API** for the **operating system** called **Mac OS X**, which runs on **Macintosh** computers. Cocoa is a very advanced **object oriented API**. Cocoa used to be called NEXTSTEP, back when it was owned by Next Computer Inc.

• **code**
The actual text of the programs you write, before you **compile** them. Sometimes called **source code**, or just **source**.

• **CodeWarrior**
A **programming environment**, including an **editor**, **compiler**, **linker**, **debugger**, **application frameworks** and **RAD** tools. It's called CodeWarrior because some programmers like to call themselves "code warriors", because it's a better title than "pasty-faced geeks". CodeWarrior is available in a full version and a starter version, and both have academic discounts available. You can get versions of CodeWarrior for **Mac OS**, **Windows**, **Linux** and any number of other platforms.

• **compile**
To convert **source code** into **object code**. Sometimes also used as a noun; you might refer to your compiled program as "a compile", or you might say you "did a compile" when you've compiled your program.

• **compile**
A special kind of program which compiles (converts) **source code** (which is easy for humans to read and write) into **object code** (which is what computers need). Compilers are almost always included with your **programming environment**.

• **cross-platform**
Able to **run** on multiple different types of computers and **operating systems** without needing to be re-**compiled** first. **Java** is cross-platform.

• **debug**
To find and correct errors and problems in your programs. You will probably use a **debugger** to help you debug. The term comes from the fact that the earliest computers were huge mechanical devices, and sometimes insects would fly into them and cause damage, thus the computer engineers had to physically remove dead bugs from the internal workings of the computers.

• **debugger**
A special program which lets you **run** your programs one line at a time, and keep track of everything that's going on in them, to help you identify and correct errors and problems. Debugging is an

integral, if sometimes unpleasant, part of programming.

• editor
A program (much like a simple word processor) which you use to write your programs. It can be any text editor, or part of a **programming environment**.

• FOLDOC
The Free OnLine Dictionary Of Computing. A web site at http://www.foldoc.org/ where you can find definitions of pretty much any computing term. Use as a next step after reading this list of common jargon.

• FORTRAN
Also "Fortran" (not all in capital letters). FORmula TRANslation; a **programming language**. FORTRAN is a programming language designed to be used for writing scientific and numerical programs. Still in use, although not really a mainstream language.

• GUI
A Graphical User Interface. The idea of having little pictures (icons) of things instead of just text. You'll be familiar with a GUI, since you're almost certainly using one right now. The concept of having a mouse pointer, menubars, buttons, checkboxes, windows, folders, document icons and so on is all part of a GUI.

• hack
A program which was written very quickly and/or carelessly, or a program which does something unconventional or frivolous. Can also be used as a verb, meaning simply to write computer programs.

• hacker
Traditionally, simply a computer programmer - and this meaning is still in use. More recently, the term also refers to a person who maliciously compromises or damages computer systems (often over the internet).

• hardware
Any physical piece of machinery or electronics. A computer is hardware, as is a mouse or keyboard or printer. A tangible, physical piece of equipment. Compare with **software**.

• InterDev
A **programming environment** from Microsoft. It allows you to program in various different **programming languages**.

• Interface Builder
A **RAD** and **object oriented** programming tool included with **Mac OS X**. Used in conjunction with **Project Builder** to create applications for **Carbon**, **Cocoa** and **Java**.

• Java
A **programming language** which is **object oriented** and can run on many different computers

without needing to be re-**compiled** for each one. Very popular at the moment, but can be slow in comparison to other languages.

• JavaScript
A **scripting language** used within web pages to add basic interactivity and so on. A completely different language from **Java**, despite the similar name. Often confused with Java by those who don't know the difference.

• linker
Usually included with a **compiler**. A special program which links together all the bits of **object code** produced by a compiler. You'll rarely, if ever, have to explicitly use a linker; it's always automatically taken care of by your **programming environment** after your program is **compiled**.

• Linux
An **open source**, **Unix**-like **operating system**. Created by a person called Linus Torvalds, hence it's "Linus' Unix", or Linux.

• Macintosh
Also called Mac. A line of computers created by **Apple Computer**. The first mainstream **GUI** computers (though the first ever GUI computer was created by Xerox Corporation).

• Macintosh Toolbox
The **API** for **Macintosh** computers running the **Mac OS operating system** up to version 9. Now updated for **Mac OS X** and called **Carbon**.

• Mac OS
The **operating system** which runs on **Macintosh** computers. Used to refer to versions up to and including 9 (not including **Mac OS X**).

• Mac OS X
A radically new **operating system** for recent **Macintosh** computers. It combines the power of **Unix** with the friendliness of the Macintosh **GUI**.

• object code
A program which has been **compiled** into a form suitable for the computer to use. Also sometimes called a **binary**.

• object orientation
Also called **OO**. A modern programming concept where the programmer creates "objects" like real-life objects, with both properties and abilities. In traditional programming, the program was very separate from the information it acted upon. That's not very much like real life; objects in real life have both properties (like the colour of your hair, or your height) and abilities (like your ability to read this article, or your ability to tie your shoelaces). Object oriented programming essentially tries to allow programmers to think (and program) in a more natural and familiar way. Popular modern object

oriented **programming languages** include **Java** and **C++** (and my own favourite, **Objective-C**).

• Objective-C
Sometimes also Obj-C or ObjC. A **programming language** which is an **object oriented** version of **C**. Used as the language of choice for developing programs to run in the **Cocoa** environment on **Mac OS X**. A very easy to learn and powerful language. My own language of choice.

• OO
Another way of saying **object orientation** (or object oriented).

• open source
The idea that the **source code** of a program should be available to everyone, as well as the **compiled** version. This allows any programmer to modify and enhance the program as they see fit, and it allows new programmers to see exactly how the program was written. Programs which are open source are free, since anyone can get the source and **compile** it themselves.

• operating system
The special and very important program which makes your computer work. It takes care of things like talking to the screen, printer, keyboard and all the other **hardware**. Without an operating system, your computer would just sit there staring blankly into the distance, like you do after you've had a few beers.

• Pascal
A **programming language** designed for teaching programming; made to be as simple as possible. However, this simplicity is also a weakness, and has lead to Pascal being denounced as a "toy" language, just for hobbyists and rudimentary teaching purposes. There are many variations of Pascal available.

• Perl
A **programming language** originally created for **Unix** computers. Very powerful for manipulating text, and very popular for writing programs to help run web sites.

• programming
The art of writing computer programs (and it is indeed an art). What this article is all about, so you shouldn't need this definition.

• programming language
Any of countless special languages used to write programs. Usually not difficult to learn, and including several English words like "end" and "repeat" and "if".

• programming environment
A special program (or set of programs) which help people to write other programs. Just like you use a word processor program to write letters to your Aunt Mary, you use a programming environment to write programs. Commonly includes a **compiler**, **linker**, **debugger** and sometimes other things too.

• Project Builder
An excellent **programming environment** included with **Mac OS X**. Used in conjunction with **Interface Builder** to create programs for **Carbon**, **Cocoa** and **Java**.

• PROLOG
PROgramming in LOGic; a **programming language**. PROLOG was the first of many languages designed to use so-called "logical programming" techniques. The basic idea is that you define a set of facts, then define a goal, and the computer tries to get to the goal using the facts you've defined.

• puke
Another way of saying **barf**.

• RAD
Rapid Application Development. A concept where you create application programs very quickly by visually dragging controls (like buttons and checkboxes and so forth) into windows, to create the **user interface** for your application. Many **programming environments** now include RAD tools.

• REALbasic
Sometimes also called RB. An **object oriented** version of **BASIC** which includes a **programming environment** with an **editor**, **compiler**, **linker**, **debugger**, and **RAD** tools. REALbasic's programming environment runs on **Mac OS** computers, but can **compile** applications for **Windows** and **Carbon** too. Probably a very good choice for **Macintosh** users who are just starting into programming.

• run
To make a computer perform the steps you've written in your program; to make the program do whatever it does. Sometimes also called "executing" a program.

• SDK
A Software Development Kit. Typically a package of sample **code**, documentation and other items to help you create certain kinds of programs. For example, if you wanted to create a plug-in for Adobe Photoshop (that's a program which lets you edit pictures), you'd probably want to get hold of the Photoshop plug-ins SDK. Sort of like a toolkit for programmers.

• scripting
A type of programming using a **scripting language**.

• scripting language
A special kind of programming language which isn't **compiled** before you run it; it is compiled automatically as needed, right before it's run. Often easier than traditional programming for a few very technical reasons which we won't go into. Might be a good place to start if you find normal **programming languages** too difficult.

• software
Any computer program. It's called software because it's not tangible; not "hard". Compare with

hardware.

• source
Another name for **code**.

• source code
Another name for **code**.

• string
Any piece of text. Computers refer to any amount of alphabetical letters or digits or punctuation marks etc as a "string".

• Unix
A very general group of **operating systems**, known for their power, stability and reliability, but also acknowledged to be more difficult to learn and master than a **GUI** operating system. Extremely popular operating systems for running web servers.

• user interface
The appearance of your program to the person using it. The windows, menus, buttons and so on are all collectively called the user interface. Sometimes just called UI.

• virtual machine
A program which emulates (pretends to be) an entire little computer all of its own. **Java** uses a virtual machine, which allows Java code itself to be **cross-platform**, at the expensive of some speed.

• virtual memory
A technique whereby a computer uses part of its hard disk as temporary memory for a program. A way to use more memory than your computer actually has, but at the expense of considerable speed.

• VM
Either **virtual machine** or **virtual memory**.

• wetware
What programmers sometimes humorously call the human brain; the ultimate computing device. See also **hardware** and **software**.

• Windows
One of several **operating systems** from the company called Microsoft. Collectively they are the most commonly used operating systems in the world. Each version of Windows tends to have a year after it instead of a version number (for example, Windows 98). Windows is a graphical operating system (or **GUI** operating system).

This brings us to the end of the list of jargon (you can wake up now). For a much more extensive list of computer related terms, be sure to look at the **FOLDOC** definition, in the list above - it will give you the address of a web site you can visit to find the meaning of any computer term ever used by

anyone, anywhere (probably).

Further information

This is the final section of the main article, so congratulations (and heartfelt thanks) if you've made it this far. This section gives a brief list of some web sites you might want to visit to find out more about certain topics we've covered previously. I've split the sites into categories, and mentioned which terms or items they're relevant to. So here goes.

• All programming topics

http://www.programmersheaven.com/
Packed full of resources for every kind of programmer (even me).

http://www.programmingjobs.com/
A place for US programmers to post their resumés, or find a programming job. As indeed you might expect from the name.

http://www.seriousprogramming.com/
Lots of information, including tutorials on C and C++, information on how to use a compiler, and more. Covers lots of programming and scripting languages too. Don't be put off by the "serious" name; there's lots of material for beginners.

• Books

http://www.amazon.com/ and http://www.amazon.co.uk/
Go and read all those customer reviews of appropriate books for your chosen programming language.

• C/C++

http://www.cprogramming.com/
A good site for information and tutorials on C and C++.

• CodeWarrior

http://www.metrowerks.com/
The company which makes CodeWarrior, and other programming tools.

• Geeks

http://www.thinkgeek.com/
Essential clothing, beverages, books and toys for the discerning geek. Highly recommended.

http://www.startrek.com/

Assemble an away team and take a look around whilst your ship is docked for repairs. Damn those Borg.

• Java

http://java.sun.com/
The Java site from Sun, the company which created Java. Authoritative, as you might expect.

• Objective-C / Project Builder / Interface Builder / Cocoa

http://www.stepwise.com/
Excellent tutorials and articles on Objective-C and Cocoa (which used to be called NEXTSTEP, hence the name "Stepwise").

http://developer.apple.com/
The developer site from the company who brought us Project Builder, Interface Builder and Cocoa. Lots of good material here, including sample code and technical articles.

http://www.cocoadev.com/
An excellent community-built site for those new to Cocoa and Objective-C. I've contributed material there on a few occasions, and there are some excellent programmers ready to help you out. This site is also particularly interesting since every page of it is editable by anyone who visits. You can actually add your own pages and edit existing pages just using your web browser, without needing to log in or register first. So play nice, please.

http://www.cocoadevcentral.com/
Another great Cocoa site, with an emphasis on new users and question-and-answer type articles.

• REALbasic

http://www.realsoftware.com/
The home of REALbasic. Download a free trial version here, or read all the documentation. Plenty of links to source code and such also.

One final thing

Traditionally, this is the place where the author gives a final, rousing inspirational speech, then goes merrily on his way feeling like a fabulously generous and wise person. I intend to follow that tradition to the letter.

If you're read this far and understood at least most of what I've said, you have sufficient intelligence to pursue a career in programming. It doesn't have to be expensive (there are lots of free programming environments, compilers and so on, and even the commercial ones have starter versions and academic discounts), and it really can be a huge amount of fun. Despite my somewhat deadpan tone, you can

tell by the fact that I wrote all this that I'm passionate about programming, and would dearly like to see even more people taking it up (so that I can steal their code later).

Programming is an extremely rewarding activity. The thrill of thinking your way around a complex problem, and the satisfaction of seeing your program perform the desired task perfectly, are hard to describe. I've always felt that writing computer programs is very like composing music; you take an initial fully-formed idea, and break it down and analyse it until you properly understand how it's formed, then you go about actually building it, piece by piece. So much art and elegance goes into writing programs, and there's considerable room for interpretation and personal style to show through. Programming is seen as an exceptionally technical and boring profession, yet in all the years I've been doing it I've never tired of it. There really is a genuine thrill about finding a new problem, and engaging your mind and all your knowledge and experience to solve it.

Programming is about creating; about making things possible and bringing ideas to life. In that respect it's no different from painting or sculpture or music, but it's in many ways even more rewarding because your creation actually **does** something in itself. Write a program, and watch it go. I'll never tire of seeing my programs go off and solve people's problems and make their lives a little easier. There's something very powerful in the ability to create something which can perform a function, and interact, and process and contribute - to whatever extent you've granted it the ability to. You'll find that you become very fond of, and proud of, your programs. And of course you may well make vast quantities of money. Either way, there's no shortage of satisfaction.

Since you've read all this, you're at least thinking seriously about what programming is, and you may even be considering looking into it further. I urge you to do so. Ever notice how programmers make no apologies about the fact that they're programmers? That's because it really is good to be a programmer, as you'll hopefully be finding out.

Well, that's it; the lecture is over. Naturally I'll now go off and feel like a particularly wonderful person (a justifiable position, I'm sure you'll agree), and hopefully you'll go and start down the road to becoming a programmer. Before you go, or even afterwards, I'd love to hear any comments or relevant anecdotes you might want to share. I'll especially welcome gushingly positive reviews of this article, and free money. But I'll certainly settle for just hearing from you. Send me an email via my web site, which is listed at the very start of the article.

And good luck.