Chapter 1: Programming is Like Juggling

Computing as a Fact of Life

This millennium is an exciting time to be involved in computing. The Web, which did not exist 25 years ago (!), is an important new part of our society, our culture. Ordinary people, with no training in computing, can sit at a personal computer and access millions of other computers across the planet. Anyone with patience and the ability to follow simple instructions can make a Web page and post information that is immediately accessible from around the world. This is a revolutionary development. Java is a programming language designed for programming on the Web.

Computing as Information Processing

Computing is information processing. Always. Only. Although many people experience computing as Web browsing, or chatting, or game playing, the underlying programs are always processing information by executing instructions.

A program is a series of instructions that is executed by a processor. Each instruction does some small task like moving a little information from one place to another, or adding one number to another and storing the result, or comparing one number to another and taking different actions depending on which is bigger. Although each individual action is small, a modern processor executes billions of instructions a second and so can accomplish an amazing amount in a short time.

Information is stored mainly in two places, temporarily in memory, and permanently in files on disks (or flash RAM). A file is a series of numbers. This is the same “file” you might store from a word processor or an email program. When you save a file in a word processor, the information in your document (i.e. the text you have typed, along with formatting information) is converted to a series of numbers and written on the disk along with the name of the file that you select. Later, when you load that file again, the numbers are converted back into text to be displayed. The files you save (and the information in them) remain on the disk indefinitely unless you delete them. Information in memory, by contrast, lasts only so long as the machine is turned on and the program that is using that memory is running. When the program terminates, that memory may be used for other processes. When the machine is turned off, all the information in memory is lost.

State and Its Representation

In an ordinary digital computer, memory is measured in bytes. It is common, in 2012, for machines to have 4 or 8 gigabytes of memory; a gigabyte is $2^{30}$ bytes. Each byte is composed of 8 bits, and is just enough memory to hold a single letter. As you have likely heard, all information in a digital computer is ones and zeros; since there are 2 possible values, it is called binary. A single binary digit is called a bit. Memory is a sequence of numbered locations, each of which can hold one byte. The good news is, programming in Java, you will almost never need to deal directly with bits, bytes or memory locations.
You will, though, need to understand that information in your program is stored in variables, which are associated with particular locations in memory, and that a particular variable may take on different values at different times. The current value of a variable is referred to as its “state”. This is an ordinary usage of the word state, but perhaps not a common one. If it seems confusing, you might think of the “state of a light bulb” (i.e. either on or off), or the “State of the Union”.

**Definition of Algorithm**

A program is an algorithm written in a particular programming language. That’s fine, but what’s an algorithm?

Rough Definition: *An algorithm is a step-by-step description of a process to solve a problem.*

Thus a recipe is, roughly, an algorithm. It lists the various ingredients that should be added in what order, and the cooking or baking process. Most recipes are not quite algorithms because they require judgment to carry out correctly.

Better definition: *An algorithm is a step-by-step description of a process, where each step is described explicitly and requires no judgment, to solve a problem, or a class of problems.*

The reason many recipes do not fit this definition is that they include directions like, “Bake until done”, or “Cook until just tender”. While any experienced cook understands these instructions, the cook must exercise judgment to follow them. People do this very well, but computers do not.

So, when presented with a problem to be solved by a program, a programmer’s job is first to formulate an algorithm which can solve that problem. The second job is to convert the algorithm into a programming language so that it can be executed (i.e. carried out) by a computer.

**Action! The Only Three Statements That Do Anything**

The remainder of this book concerns techniques that allow you to solve problems and implement algorithms in Java. There will be several hundred pages explaining control structure, class structure, objects, expressions, methods and data structures. Oddly, in spite of all that, there are only three actions that actually accomplish anything. Input statements and assignment statements change the state (or value) of variables; and output statements send information out of the program (usually for a human to read). That’s it, three things. Input, assignment, output.

1. **Input statement** -- brings information into the program

2. **Assignment statement** -- changes the information in (state of) a variable

3. **Output statement** -- sends information out of the program

Everything else in a program, all the hours that a programmer spends designing, coding and debugging, only arranges for those three things to happen the right number of times and in the right order. This may seem strange, but it is true.
Structure: Everything Else
The parts of a program besides input, assignment and output include three categories:

- Control structure -- selects which statements are executed, in what order, and how many times
- Data structures -- organize data (information) so that it is more convenient to access
- Class structure -- organizes classes so that they are easy to understand, modify and work with.

These three will be the subject of much of the rest of this text.

Computing as a Revolution
Computing, viewed from the inside, is always about information; inputting information, processing information, and outputting information. Viewed from the outside, it is an exciting and revolutionary development. It is transforming our world, our ways of communicating, learning, playing, and our understanding of ourselves. The changes computing will bring are mostly in the future; the computing revolution is only beginning.

The Web as a New Cultural Phenomenon
Some people think of “culture” as meaning opera or art galleries. There is a broader meaning that includes language, education, technology and even dating. Many everyday activities are being transformed by the Web. How we shop; how we communicate with our friends, family, and coworkers; how our cars and televisions work; how we plan and spend our time; these and more are done differently with the internet.

People Have Never Had a Tool for Processing Information Before
We are only just beginning to learn how to use computing. Nothing is settled yet. The leading hardware and software producers come out with steady updates. But the way we deal with information has been changed forever. A good example is Google. It sends bots out to collect information in the dead of night (i.e. when the net is not so busy), then catalogs and indexes it so when people type queries it can respond quickly and direct them to relevant Web sites. If you’re used to Google this may seem like no big deal. But it is. Search engines and the Web allow information to be disseminated orders of magnitude faster. Even research scientists use search engines instead of spending long hours in libraries searching for paper copies of research articles. What will this mean for our culture? Who can say? But, it will certainly change it.

Automated Reasoning and Process
Computing also allows us to study process. A program is a mechanization of information processing. Before computers existed, information processing was only done by people (and other natural systems). In the short time that mechanical computers have existed we have just begun to learn about process; who knows what the future will bring?

Artificial Intelligence?
Artificial intelligence is an exciting and alarming possibility. Might intelligent machines replace us all? There have been many projects starting in the 1950s to build programs to do language un-
derstanding, automatic translation, scene analysis, and more recently, build autonomous vehicles. Thus far, successes have been extremely limited, although this year Google has made self-driving cars! Although computers are everywhere, blindingly fast, and never forget; to call them stupid would be a compliment; they have no intelligence. Nevertheless, clever programmers can make them do some amazing things.

One reason artificial intelligence is so difficult is that while computers can add numbers billions of times a second, and store and access millions of facts or rules without ever forgetting even one, they do not learn as people do, they do not form and apply concepts flexibly. People are the product of billions of years of evolution and we have highly developed and highly specialized information processing capacities. Additionally, people are born with a set of unconscious patterns corresponding to important relationships and ways of thinking. Important how? Evolutionarily. Certain ways of processing information are more adaptive than others, and individuals must have survived long enough and done the right things to have and raise children, or they are not our ancestors. Our fascination with sex and violence is not accidental. Neither is our love for children. Our abilities to communicate with metaphors, to ascribe meaning, and to discern pattern in noise are still far beyond the reach of any existing computer.

The enterprise of attempting to create intelligence in a digital computer has been likened to trying to climb a tree to the moon. Early, preliminary attempts seem to yield good progress (“Look! See how high I’ve climbed already?”), but sooner or later frustrating impasses always seem to loom (“Huh, the branches are getting pretty skinny up here.”), and eventually the project is abandoned (“But... maybe if I found a tree on a very high hill?”). There’s no way to know how apt that metaphor is (and that’s always a question you should ask when encountering a metaphor, otherwise you risk being led badly astray!). Everyone knows you cannot climb a tree to the moon, whereas whether a digital computer could have intelligence remains to be seen.

**Societal Impacts**

Our culture is changing in many ways because of computing; here are just three examples.

1) Around the middle of the second millennium kingdoms arose in Europe. In the days before computing, to raise an army to make war on your neighbor, or defend yourself from them, required first assembling a bureaucracy. A small army of clerks and functionaries was needed to coordinate the calling up, feeding, housing and supplying of any army. This meant that to wage war a ruler had to enlist or compel the cooperation of many many civilians. Nowadays, one person with a thousand dollar machine plus access to appropriate databases and software could coordinate much of that without assistance. Will this change warfare? Has it already? Look around. Watch and see.

2) Many Americans put a premium on their privacy. They hate the idea of anyone compiling and/or selling data about them without their permission and cherish the notion of relative anonymity in transactions on the Web. But the nature of computing and the Web means that privacy is essentially an illusion. If you send email across the country there are copies of it on at least several
mail servers; worse, there are logs. Admittedly, only system administrators can access them, but if there were a reason to, they could.

Uninformed people may imagine they can do things anonymously on the Web. It is true that your personal identity is hidden, but where you are sitting and your ISP, are definitely not a secret. If they were, there would be no way for the Web server you are receiving information from to send it to you. Even though you can’t see them, there are servers and routers relaying the packets across the planet to your screen; and they are all logging all the transactions, just in case someone needs to find out who was accessing what.

3) Some people predict a collapse of traditional brick and mortar commerce. As more and more goods are sold over the Web, there will be less and less need for physical stores. Of course there will always be stores, but perhaps soon there will be considerably fewer of them.

**The Past: A (Very) Brief History of Computing**

It is not strictly necessary to know the history of computing before learning to program, but there are some interesting perspectives one may develop by doing so. Therefore, this brief history.

**Living in Scaffolding -- A Cautionary Tale**

A cathedral took many years to build. Imagine building a gigantic stone structure without power tools. Huge blocks of stone, high walls of stone. To place a block of stone on a wall, you must first lift it up and then set it in place, making sure that it fits tightly. This requires solid, strong, more or less permanent scaffolding.

It was common for the workers to construct living quarters in the lower levels of the scaffolding (alongside the already completed walls). Perhaps this was simply a convenient place to build a shelter to cook for the workers, and for the workers to eat during inclement weather. As years turned into decades and the walls (and the vacated scaffolding next to them) grew higher, it was simply easier for the families of the workers to take up residence there. As decades turned to centuries, and construction was interrupted by wars or plagues, people were born, grew, raised children and died, living in the scaffolding, knowing no other life. There were even cases where the original project was abandoned and the workers settled into the scaffolding as a permanent residence.

There is a danger, in any endeavor, that temporary measures, adopted as a means to an end, remain in effect for so long that practitioners no longer remember that they were not the goal of the project. This is not, generally, a good situation. Digital computers are a case in point.

**Why Are We Still Using This Prototype?**

This text will emphasize generic problem solving principles, in addition to Java programming. They will be used in service of programming, but will typically be applicable to many other areas. The phrase “problem solving” is used in a special sense here. It is the activity that one engages in when one is trying to accomplish something and runs into a problem that stops them.
These problem solving principles are for when you don’t know how to cope with a problem. Here’s the first (and perhaps the most general).

Problem Solving Principle #1 -- Build a Prototype

When attempting to solve a difficult problem, first build a prototype that solves a similar problem that is simpler or smaller. This is useful for several reasons. If you’re stuck on a problem, sometimes a smaller or simpler version of the same problem will be easy. Plus, sometimes the reason a problem stops you is that you’ve never thought carefully about anything like it. The cognitive structure that is generated (sometimes referred to as “expertise”) by solving the simpler version sometimes allows you to see through the more complex problem.

Here’s a problem that may stop you. Each of 32 students is asked whether they have ever been snowboarding; they write their answers on a piece of paper; the first on the first line, the second on the second line, and etc. How many possible different sequences of yeses and noes are there? If you’ve been studying combinatorics and know, skip to the next section. If not, read on.

What’s the smallest we could make the problem? How many people? Right, one. One person can write one of two things; yes or no -- so there are two possible sequences (if you can call one thing a sequence). The good thing about solving a problem with one thing instead of $n$ things is that it is usually trivial. The bad thing is that it may not tell you much.

So, try two people. The first can write either yes or no. Then the second can write either yes or no. If the first person writes yes, the possible pairs are {yes, yes} and {yes, no}. If the first person writes no, they are {no, yes} and {no, no}. So, four total.

The trick, now, is to generalize to $n$ people, or in this case, 32. For one person the answer was two, for two people it was four; can you discern the pattern? Commit to a pattern. Then see if it is true for three people. Here’s how to do the analysis for three.

As we just saw, for the first two people there were four possible states of the list:
1. {yes, yes}
2. {yes, no}
3. {no, yes}
4. {no, no}

The third person can add yes or no, giving 8 possibilities (the third person’s answer is in bold):
1. {yes, yes, yes}
2. {yes, yes, no}
3. {yes, no, yes}
4. {yes, no, no}
5. {no, yes, yes}
6. {no, yes, no}
7. {no, no, yes}
8. {no, no, no}
There are twice as many possibilities as with two people. So the number of possible states is not twice the number of people, but rather, it starts at two and *doubles* with each additional yes or no added to the list. Thus, you can see the answer with 32 people is, $2 \times 2 \times 2 \times \ldots \times 2$, 32 times, which is written $2^{32}$ (that’s somewhat more than 4 billion). Remember this number, $2^{32}$, and how it was derived; you will meet it again later.

**Von Neumann’s Prototype**

Most digital computers we use are said to use the "von Neumann architecture". John von Neumann was among the designers of the first working digital computers; his name is attached to them because he took the time to write up and publish the ideas that would lead to working computers. The second example of building a prototype of a system involves the digital computer. In the 1930s, von Neumann set out to build a thinking machine. Since the only things we know of that think are brains, he decided to study brain function first (an eminently reasonable idea!). He concluded that brains had both analog and digital properties, and considered designing a machine with both. A digital machine would have elements with digital values; in binary, either 0, or 1. An analog machine might have elements with values that ranged between 0 and 1, where 0 represents all the way off, 1 all the way on, 0.5 half way on, 0.75 three quarters on, and etc. He started trying to settle on the details of this thinking machine, and found himself stuck on how exactly to implement the analog portion. The way he conceptualized it, the accuracy of the analog portions was going to be very poor. So, he decided to build a strictly digital machine first; namely one with only two states, 0 and 1 -- a binary computer.

It turned out that even this simpler machine was not trivial to implement, but with time and persistence they got it to work. Von Neumann died before he constructed an analog computer, or even started working on making a computer think; and here we are living in the scaffolding. Perhaps mechanical intelligence will require analog computers. We shall see.

**Evolution of Computing**

The rate of evolution in computing is astounding. Digital computing is changing so rapidly for a number of reasons: a) it is new, b) hardware is improving rapidly, c) software is improving rapidly, and, d) more and more people are getting involved, both as users and programmers. Any human endeavor, at its inception evolves rapidly. Early, simple, clumsy, poorly conceived systems give way to more sophisticated, better debugged, easier to use systems.

**Hardware**

In the 70 years that computing has existed, computers have been utterly transformed. Early computers were made with relays, and vacuum tubes, like the radios and televisions of the time. The invention of the transistor allowed a truly digital device. The first transistors were hand-made and large. They were mounted on circuit boards and wired together. Soon hundreds of tiny transistors were being packaged on a chip; these were called integrated circuits (ICs) and were mounted on similar boards. For example, in a 1960s computer the central processing unit (CPU), or processor, which does the arithmetic and logic, was a board perhaps 18” square, packed with chips on one side and festooned with more wires than you’d want to count on the other. Before
long, tens of thousands of transistors were being packed into a chip, a technique, then called very large scale integration (VLSI). One day, someone managed to pack all the functionality for a CPU onto one chip, and the age of the microprocessor began.

A microprocessor can run much faster than any processor spread out on a board for one simple reason; the information in the processor has a shorter distance to travel. Several facts will help illuminate this. Electrical signals travel at about 0.6c; where c is the speed of light (approximately 3x10^{10} cm/sec). Modern processors commonly have clock speeds of 4gHz or more. A clock speed of 1gHz means the clock ticks a billion times a second, the time from one tick to another is a billionth of a second, a nanosecond. Light travels about a foot in a nanosecond. So, if components of the CPU were a foot apart, there is no way a signal could get from one to the other in time for the next clock cycle. That would slow down the processor. Thus, every modern processor is a microprocessor and every modern computer a microcomputer.

**Education, Language and Culture**

In spite of that, there are books and people who still talk about “mainframes”. Some out of date (but still in publication) books include typologies of computers including: mainframes, mid-sized computers, mini-computers and microcomputers, as if it is a spectrum from large and powerful to small and not, when the reverse is typically true. Why is this?

This is a characteristic of the slow evolution of culture and language. Language in a rapidly changing culture lags behind the phenomena it describes. An example is “floppy disks” as the name of 3 1/2 inch removable disks. They replaced the 5 1/4 inch removable disks, which were actually flexible (i.e. floppy). Another computing example is RAM. This is the acronym for random access memory, the main memory in computers. It was called “random access” because it supplanted tape memory, which was strictly “serial access”. If the information you needed was on the other end of the tape, you had to wait for it to rewind; this is why computers in old movies had so many whirling tape drives, before there was RAM, memory was mostly serial, on tape. In time, the language (and movies) will catch up; assuming computing stops changing so quickly!

**Software: Programming the Hardware**

As hardware has evolved, so has software. Software is the programs that control the hardware. It is pure information, the stuff of dreams, and seemingly as difficult to control. The first computers were programmed and debugged by physically connecting and reconnecting wires to components. The reason was that there was no memory to store the programs. Once memory was invented (tape, and then RAM), programs were written in binary, the machine language that the processor could execute directly. Each tiny instruction for the processor was laboriously entered by setting toggle switches (right, before keyboards). Even the simplest programs were very long and tedious to debug; there were no screens. Later, paper tape, card punches and card readers made it possible to type programs, store them on, and feed them in as many times as was necessary.
Programming in machine code is a big headache and unbelievably slow. Before long, some enterprising systems programmer wrote a symbolic assembler that eliminated some of the mind-numbing tedium (computers are excellent for mindless, repetitive tasks!). Assume the add instruction, in machine code, were 2; then to add the number at location 143 you might write: 2143. An assembler allows the programmer to move a little away from machine code, and write “add 143” instead. (By the way, 2, 3, and 4 are not binary digits; the actual binary instruction would have looked more like 0010000101000011).

Assembly coding is tedious, and is ridiculously inefficient to write. Once programmers had good assemblers, they realized they could write compilers. Compilers input some higher level language (like Fortran, or C) and output assembly code. The assembly code then goes into the assembler, which emits machine code.

Fortran and C were great advances over assembly code, but before long more powerful languages were invented. There are functional languages, database languages, logic languages and object languages (among others). C++ was an early object language based on C. Java is a later language based on C; it was designed for programming on the Web and eliminates some of the worst shortcomings of C++. There will be many other new languages as time goes on. Perhaps you will design one.

**Why It’s Not Quite That Simple**

Perhaps you noticed that the distinction between hardware and software was not quite as clean as it might be. Hardware is stuff you can hold in your hand. Software is information. Plugging and unplugging wires was the first example of software, and wires are definitely physical; their arrangement is logical, but still...

People like to create simple categories. Hardware/software. Us/them. Nature/nurture. But things are not quite so simple. A hundred years ago a debate raged over how much of what animals (including people) do is determined by genetics and how much by experience. The blank slate faction said it was all experience; the instinct school claimed it was mostly built in; some cooler heads argued it was about 50-50. Now most everyone knows that a better answer is 100-100 -- what we become is determined by our genetic heritage and our experiences. To neglect either would be a mistake.

Another common and natural false dichotomy is to divide the world into us and them. There are people who blame “them” for whatever goes wrong. “Them” could be teachers or students, government or citizens, parents or children, our race or theirs, our country or theirs, our religion or theirs. If people are not paying attention they seem to do this automatically; there’s reason to think it might be an innate proclivity. But we can learn not to.

The hardware/software dichotomy is also murky. It is possible to bake programs into silicon; there are hardware Java chips. Processors have microcode inside that governs their function. It is even possible to build virtual machines; software simulations of hardware. Perhaps the most revolutionary aspect of Java is the Java Virtual Machine (JVM).
The Java Virtual Machine

Different computers have different processors. For a program to run on a particular machine, it must first be translated into the machine code of the processor on that machine. Thus, if you have a program that must run on a dozen different CPUs, it must first be translated into a dozen different “binary” files. When a new CPU is invented, the program must be retranslated for that CPU. If you hope to distribute a program across the Web, this is a major obstacle.

Java solves this problem by using an intermediate form, byte-code. Byte-code is a machine independent code that runs on the Java virtual machine. The Java virtual machine (VM) for a particular machine interprets byte-code and executes instructions on that machine to accomplish what the programmer intended. A Java program is compiled into byte-code and can then be run on any machine that has a current Java VM installed. The VM must still be written for every type of processor, but it’s a huge improvement to only distribute one version of a program.

Juggling (!)

Programming is like juggling, you can’t learn to do it by watching!

If you’ve been juggling for many years you can easily keep three balls in the air for long periods of time without dropping them. You already know how, so it’s easy. If you’re just trying juggling for the first time, you will drop the balls often. If, each time you miss one, you curse yourself, or the ball, or whatever distracted you, or think to yourself, “I’m just no good at juggling!”, you are not helping yourself learn. The appropriate response is, “Oops! Missed!”; then pick up the balls and keep practicing.

Programming is similar in that almost every program has mistakes initially; these are called “bugs”. Only novice programmers imagine that any nontrivial program will be right the first time. Experienced programmers make fewer mistakes than neophytes, and they are better at finding them (having done it before); nevertheless, they expect their programs to have bugs and program in a way that makes it easier to find them. When you run into bugs, if you get angry or imagine it reflects on you personally it doesn’t help you learn to program. It’s normal for programs to have bugs, especially initially. But after you’ve practiced for a few dozen hours, you will drop the balls much less frequently, and accept that it is a natural part of the process.

Learning to Program

Programming is also rather like writing in many respects. They both are creative, iterative processes without any one correct way. They both have syntax (grammar) and semantics (meaning), and grammatical errors can obscure the meaning in either. Of course, no one reads programs for pleasure; on the other hand, novels do not deliver email, control automated factories, ATMs, or iPads (or call up armies to invade foreign countries).

There are many different ways to teach Java programming. Some books start by teaching the old C constructs, and only introduce user defined classes after 6 or 10 weeks. Modern approaches introduce classes earlier. This text begins with classes and adds a cognitive component. A pro-
gramming environment always includes a programmer; teachers and students of programming ignore the characteristics of programmers, and in particular, novice programmers, at their peril. Good software development methodology minimizes cognitive overhead, and thus leaves the programmer, at whatever level of expertise, with enough cognitive capacity to solve the problems that will inevitably arise -- this is especially important for beginners.

Learning to program is not easy. The first several weeks, before you can do even the simplest things, can be especially frustrating. Fortunately, after five or six weeks, when you have mastered the basics of input, output, classes, and calculation, the ability to satisfaction of being able to make a machine do what you want balances out the initial difficulty of doing so.

One of the reasons programming is difficult initially is that there are so many details one must learn before one can construct even the simplest program. There are a number of facts and concepts (a few dozen) that one must grasp before programming begins to make sense -- unfortunately, they are all inter-related, and understanding (or explaining) one, requires understanding (or explaining) all the rest. So, at first, the whole enterprise can seem hopelessly confusing -- even daunting. But, with patience, persistence, and practice, you can surmount this obstacle; and once you learn to program it can be very rewarding and remunerative. If, given these cautions, you wish to continue, read on!

**The Approach Used Here: Less is More; More is Less**

Computing is different from other fields. First, it is brand new. Mathematics, rhetoric, psychology, physics and philosophy are thousands of years old. The first digital computer (which was as big as a house and less powerful than a modern day low end calculator) was developed in the 1940s. Java, the language this book teaches, was released in 1995 (by Sun Microsystems). It extends the old C language from the 1970s. Computing is evolving at an unheard of pace. Better hardware makes possible better software; better software allows programmers to build even better software; as programmers mature who have been educated in the new paradigms, they can invent better paradigms. The synergy between faster, cheaper hardware, better programming tools, and better educated programmers will transform of our lives, our world. And you can quote me.

Second, because computing is so new there is no consensus on how to teach it. In biology, or mathematics, or any of the long established disciplines, introductory courses have been taught to millions of students over the past hundred years or so. As a result, these disciplines have well entrenched examples and excellent textbooks, tested on class after class after class. Some Java textbooks, by contrast, are rewritings of C++ textbooks. Others are a hodgepodge of hints and techniques. It’s not that computer scientists don’t know how to write, it’s more that there hasn’t been time, and it’s not yet clear how to help people learn to program well in Java.

This text takes a different approach, the less is more and more is less approach. Less is more, in that there are many extraneous details of Java that are best avoided in the beginning, so that the student may focus on what really matters. On the other hand, it is imperative that students begin
by learning to write classes, even though this is difficult until they grasp a certain set of the basics. So even though it is not easy to learn to write classes, that’s where we will start; this seems like more, but in fact, after a few chapters it will result in there being much less to learn later.

**Conclusion**

This brief (and somewhat opinionated) introduction to programming, computing, culture and human nature has introduced many of the themes that will run through this course. With any luck it has whetted your appetite for an introduction to Java programming which follows immediately.