# 2 Getting started

For the purposes of this textbook, I shall assume that you have access to a computer system on which Perl is available, and that you know how to log on to the system and get to a point where the system is displaying a prompt and inviting you to enter a command. Perl is free, and versions are available for all the usual operating systems, so if you are working in a multi-user environment such as a university computer centre then Perl is almost sure to be on your system already. (It would take us too far out of our way to go through the details of installing Perl on a home computer which does not already have it; though, if the home computer is a Mac running OS X, it *will* already have Perl – available from the Terminal utility under Applications → Utilities.)

Assuming, then, that you have access to Perl, let us get started by creating and running a very simple program.[2] Adding two and two is perhaps as simple as it gets. This could be a very short Perl program indeed, but I'll offer a slightly longer one which illustrates some basics of the language.

First, create a file with the following contents. Use a text editor to create it, not a word-processing application such as Word – files created via WP apps contain a lot of extra, hidden material apart from the wording typed by the user and displayed on the screen, but we need a file containing just the characters shown below and no others.

```
$a = 2;
$b = $a + $a;
print $b;
```

Save it under some suitable name – `twoandtwo.pl` is as good a name as any. The `.pl` extension is optional – Perl itself does not care about the format of filenames, and it would respond to the program just the same if you called it simply `twoandtwo` – but some operating systems want to see filename extensions in some circumstances, so it is probably sensible to get in the habit of including `.pl` in the names of your Perl programs.

Your `twoandtwo.pl` file will contain just what is shown above. But later in this book, when we look at more extended examples of Perl code I shall give them a label in brackets and number the lines, like this:

(1)

```
1   $a = 2;
2   $b = $a + $a;
3   print $b;
```

These labels will be purely for convenience in discussing the code, for instance I shall write "line 1.3" to identify the line `print $b`. The labels are not part of what you will type to create a program. However, when your programs grow longer you may find it helpful to create them using an editor which shows line-numbers; the error messages generated by the Perl interpreter will use line numbers to identify places where it finds problems.

In (1), the symbols `$a` and `$b` are *variables* – names for pigeonholes containing values (in this case, numbers). Line 1.1 means “assign the value 2 to the variable `$a`”. Line 1.2 means “assign the result of adding the value of `$a` to itself to the variable `$b`”. Line 1.3 means “display the value of `$b`”. Note that each instruction (the usual word is *statement*) ends in a semicolon.

To run the program, enter the command

```
perl twoandtwo.pl
```

to which the system will respond (I’ll show system responses in italics) with

```
4
```

Actually, if your system prompt is, say, `%`, what you see will be

```
4%
```

– since nothing in the `twoandtwo.pl` program has told the system to output a newline after displaying the result and before displaying the next prompt. For that matter, nothing in our little program has told the system how much precision to include in displaying the answer; rather than responding with *4*, some systems might respond with *4.00000000000000* (which is a more precise way of saying the same thing). In due course we shall see how to include extra material in a program to deal with issues like these. For now, the point is that the job in hand has been correctly done.

ie business school
#1 EUROPEAN BUSINESS SCHOOL
FINANCIAL TIMES 2013
#gobeyond
MASTER IN MANAGEMENT
Because achieving your dreams is your greatest challenge. IE Business School’s Master in Management taught in English, Spanish or bilingually, trains young high performance professionals at the beginning of their career through an innovative and stimulating program that will help them reach their full potential.
Choose your area of specialization.
Customize your master through the different options offered.
Global Immersion Weeks in locations such as London, Silicon Valley or Shanghai.
Because you change, we change with you.
www.ie.edu/master-management
mim.admissions@ie.edu

If you have typed the code exactly as shown and Perl does not respond correctly (or at all) when you try running it, various system-dependent problems may be to blame. I assume that, where you are working, there will be someone responsible for telling you what is needed to run Perl on your local system. But meanwhile, I can offer two suggestions. It may be that your program needs to tell the system where the Perl interpreter is located (this is likely if you are seeing an error message suggesting that the command `perl` is not recognized). In that case it is worth trying the following. Include as the first line of your program this "magic line":[3]

```
#!/usr/bin/perl
```

This will not be the right "magic line" for every system, but for many systems it will be. Secondly, if Perl appears to run without generating error messages, but outputs no result, or outputs material suggesting that it stopped reading your program before the end, it may be that your editor is supplying the wrong newline symbols – so that the sequence of lines looks to the system like one long line. That will often lead to problems; for instance, if the first line of your program is the above "magic line", but Perl sees your whole program as one long line, then nothing will happen when you run it, because the Perl interpreter will only begin to operate on the line following the "magic line". Set your editor to use Unix (decimal 10) newlines.

If neither of these solutions works, then, sorry, you really will need to find that computer-support staff member to tell you how to run Perl on the particular system you are working at!

Let's now go back to the contents of program (1). One point which may have surprised you about our first program is the dollar signs in the variable names `$a` and `$b`. Why not simply name our variables `a` and `b`? In many programming languages, these latter names would be fine, but in Perl they are not. One of the rules of Perl is that any variable name must begin with a special character identifying what kind of entity it is, and for individual variables – names for single separate pigeonholes, as opposed to names for whole sets of pigeonholes – the identifying character is a dollar sign.

If you ask *why* variable names in this particular language should have this strange requirement, the answer has to do with ensuring that the Perl interpreter – the software which "understands" your lines of code and translates them into actions within the workings of the computer – can resolve any line mechanically and without ambiguity. Any programming language has to make compromises between allowing users to write in ways that feel clear and natural to human beings, and imposing constraints so as to make things easy for the computer, which cannot read the programmer's mind and has to operate mechanically. Requiring dollar signs on variable names is a constraint which gives such large clues to the Perl interpreter that it frees the language up to be easygoing and tolerant of humans' preferred usage in other respects. Although many other programming languages have no similar requirements on variable names, overall they are more rigid than Perl about forcing users to code in unnatural ways.

After the dollar sign, a variable name can be any mixture of letters, numbers, and the underline symbol "_", beginning with a letter. (The possibilities are in fact a bit wider than this in some complicated ways, but I am keeping things simple; you will never go wrong by choosing variable names which conform to that pattern.) So e.g. `$fern`, `$fern23`, or `$Fern` would all be good variable names. (Case matters: `$fern` and `$Fern` are two different variables.) In principle there is a limit on the length of a variable name, but you are never likely to bump up against the limit.

The reason for allowing the underline character is so that it can be used to represent a written space when the obvious name for something is a multi-word phrase. If we need a variable to represent, say, roof tiles, we cannot call it `$roof tiles` (which the interpreter would see as a variable `$roof` followed by an unknown word), but we could call it `$roof_tiles`. Alternatively, for the sake of brevity some programmers prefer to run words in variable names together and use capitals to show where they join: `$roofTiles`. It is a good idea to pick one of these two styles which suits you, and to stick to it consistently as your Perl programs grow longer and more complex.

Saving a Perl program in a named file and running it by giving your system prompt the command `perl` *`program-name`*, as we did above, is not the only way to run Perl. If we don't want to take the time to save a short program to a file before testing it, we can simply enter the command `perl` at the system prompt, and then type the program in line by line. In that case, we need to tell the system when we have finished typing; we indicate that by entering `__END__` as the last line, whereupon the system will run the program.

This direct way of running Perl is a good, low-effort method of deepening your mastery of the language by quickly testing brief examples of constructions you are not sure about. To learn Perl, or any other programming language, you have to use the language. No-one ever really *taught* anyone else to program; we all have to teach ourselves, and the most a teacher or a textbook can achieve is to put learners in a position to teach themselves by doing. If you feel unsure how some piece of Perl works, try it, and if it doesn't work the way you expect first time, experiment until it does what you have in mind. That way you will remember it far better than by reading the information in a book.

When you have a program which is thoroughly debugged, so that you are likely to want to run it repeatedly, it is possible to save the effort of typing `perl` on the command line by making the program name itself a recognized command – that is, rather than entering `perl twoandtwo.pl` at the system prompt, you can just enter `twoandtwo.pl`. However, the methods of achieving that vary from system to system, so we shall not look into them here. It does not take much effort to type the word `perl`, after all.