Chapter 1

Perl's Place in the Programming World

In This Chapter

- Understanding programming languages
- Comparing Perl to other languages
- Using Perl with emerging technologies

Many people want to know exactly what makes their computer programs tick. They have a nagging itch to find out what's "under the hood" of their computers, beyond the chips and wires. They want to find out how they can create new and better programs and how it is that programmers can make so much money yet look like they dress in the dark.

Don't assume that everyone who programs or writes books about programming is a young whippersnapper. Without getting too specific, let's just say that my college days are way, way behind me. When I worked with computers back then, I often used punched cards and paper tape — ah, those wonderful pre-PC days.

In pre-PC days, BASIC was the easy language to learn, and serious programmers learned FORTRAN or COBOL to do "real work." But many people discovered that you could accomplish a lot with plain old BASIC. Today, many people have discovered that Perl is a great beginning programming language because it's simple to use and yet you can create powerful programs with little effort. Sure, you can immerse yourself in more difficult languages, such as C++ or Java, but you can probably learn Perl faster and find yourself doing what you want to do just as well, if not better.

Understanding the Purpose of Programming

If you're new to the concept of programming, you may be asking yourself, "Why am I learning how to program?" Sure, programs control computers, but that doesn't mean that everyone should become a programmer.

When personal computers appeared on the scene about 25 years ago, almost no commercial software existed, and the software that did exist was pretty primitive. If you wanted to do anything special, you had to learn to program so that you could create programs to meet your specific needs. By 1981, when the IBM PC came out, plenty of commercial programs were available and users no longer needed to program their computers — the software did it for them.

Today, very few computer users have any real need to write programs. Tens of thousands of programs are available, many for free, and you can find one for just about anything that you want to do with your computer.

Some people think that once they learn to program, they can write any kind of program for their computers. In some cases, people may have some big plans:

- "I have this great idea for a word processor that's different from the ones I've seen."
- "I'll create an Internet program that makes Netscape look like a kid's toy!"
- "There are no good programs that convert English into ancient Greek. I'll create one and make a fortune."

Although such programs are possible, each one would take months or years to create. When you learn to program, you begin by creating simple applications. To create complex programs, you have to practice, practice, and practice, usually full time. In this book, I show you how to cook up lots of simple programs using Perl (but nothing on the order of a word processor, Internet browser, or English-to-ancient-Greek translator, although the latter is certainly possible with Perl).

You may not have to program, but learning the art of programming can open doors for you:

- **Learning to program is a good way to understand how computer programs work.** Because everything that happens on your computer is controlled by one program or another, learning what it takes to create programs is a good way to find out what makes your computer tick.

- **Programming is fun, at least to people like us.** If you enjoy computers and being creative, writing programs can provide great amusement. Programming also appeals to tinkerers, because after you create a program, you can change it a bit here, add a few features there, and just keep on going with it.
- **Professional programmers can get paid big bucks, and jobs exist for them all over the world.** Note, however, that most of the programmer jobs you see advertised in the Sunday want ads are for experienced programmers, and the number of entry-level jobs in the field is almost zilch. Still, with a shred of patience and a bit of luck, you can teach yourself programming and turn it into a profession within a few years.

Making Computers Compute

Take a step back to examine what it is that a computer does and what it takes to control a computer. Don't worry, I'll limit the talk about chips and wires and circuit boards to a minimum here. However, you do need a bit of that sort of information because, even though programming is about software, software controls hardware.

A computer is a hardware system that can, well, compute. The main chip in each computer, often called the *central processing unit (CPU)*, takes instructions and data and acts on them, usually to create more data. For example, a CPU can take the data *3* and the data *5* and the instruction *add* and respond with *8*. If, instead, you give it data like *4* and *6* but still give it the instruction *add*, you get the response *10*. Or, if you still used *3* and *5*, but said *multiply* rather than *add*, you see it come back with *15*. This process of taking input, performing some operation, and providing output is really the basis for all computers. Things get fancy when you start talking about the kinds of instructions and data the CPU can handle.

All data given to a computer's CPU is in number form. You may find that fact hard to believe because you've seen your computer produce letters, pictures, and maybe even sounds and movies, but everything you see on-screen is composed in the computer's mind as numbers. When you use a programming language such as Perl, the language lets you think in terms of letters, but the program actually converts letters to numbers for the CPU.

CPUs can do much more than add and multiply. One common CPU operation is the comparison of two numbers. For example, a CPU instruction may say, "If this number is bigger than 0, do this; otherwise, do that instead." By comparing two numbers and choosing what to do next based on the comparison, the CPU can make decisions about how to work when the numbers aren't known ahead of time.

That's the short-and-sweet introduction to the brain of any computer, including a bit about how it "thinks." Of course, a computer doesn't really think: It reacts. And the way that it reacts is based on the instructions and data given to it. The list of instructions that you give to a CPU is a *program,* and you put together a program with a *programming language.* A programming language, such as Perl, is the means by which you, the programmer, can create a set of CPU instructions.

Translating Your Language into Computer Language

In order to make CPUs run efficiently, the instructions and data that you feed them must be compact and concise. As I mentioned in the section "Making Computers Compute," all data given to a computer's CPU is in number form. However, it would be nearly impossible for you to learn a programming language that only used numbers. Instead, programmers use more human-friendly languages like Perl (or C, or COBOL, or Java, and so on). These languages are called *high-level languages,* and when they run, they're converted into instructions and data to let the CPU know what's happening. These instructions and data are called *machine language,* and even though machine language is a kind of programming language, very few people know how to use it.

Interpreters and compilers

Two main types of programming languages exist — *compiled* and *interpreted,* each of which has its advantages and disadvantages. Probably the best-known compiled language is C. Perl, on the other hand, is an interpreted language.

The main difference between the two is that compiled languages generally run faster. Interpreted languages read a program and turn it into machine language before executing it. Compiled languages already have the machine instructions, so they can start executing from the get-go. Many people, particularly C programmers, contend that compiled programs are better because they run faster. In our faster-is-better society, this may be a compelling argument.

The speed difference between a compiler and an interpreter is sometimes infinitesimal, and unnoticeable by mere mortals. For instance, even for large and complex Perl programs, an equivalent C program may take one-half of a second less to run. Another difference is that interpreted languages are a bit easier to debug (that is, to find and fix errors), because they are executed step by step, so you can find exactly where a problem occurred and stop right there to fix it.

An important feature of a programming language is how quickly people can learn to create useful programs with it. Other considerations, such as how fast programs run after you write them and how much energy you have to expend to get particular tasks done, are important; however, of first and foremost importance is a language's ease of use. A difficult language may be okay for the top 10 or 20 percent of the programming population, but for the rest of us, a difficult language simply isn't worth all the struggle in the learning department and, therefore, isn't likely to be popular.

Designing Computer Languages

People sometimes think of human languages as appearing out of nowhere and never changing, but that's clearly not the case. Human languages evolve in a number of ways, and so do computer languages. Computer languages are quite different from human languages:

- The main purpose of human language is person-to-person communication; the main purpose of computer languages is human-to-computer communication.
- All computer languages have been created within the past 50 years. Because they haven't developed over a long period of time like most human languages, they don't present some of the challenges that many human languages do, such as having too many exceptions to the rules.
- Computer languages are usually invented by a single person or a small group of people (although larger groups of people may get together later to decide how to modify a language). Because human languages are developed by entire cultures, they're apt to have more quirks and oddities than computer languages.

When people create a new computer language, they have to think long and hard about what they want to do with it, and why their language will be better than the ones already out there. Convincing people to stop using their favorite computer language and switch to a new one can be difficult, so a designer must craft a language to make it as attractive as possible (in other words, they have to make the language powerful, or easy to use, or both).

Sometimes, a language is created to do just one little thing better than any other language does it, and that one thing is really important to a select group of people. For example, engineers who design robots may want their own computer language to be able to describe how to move an arm or a leg and what direction to move it in, but they don't need the ability to perform calculus. Other languages are designed to do all things fairly well, which makes program writing simpler because the programmer can do everything with just a single tool.

Picky languages versus free-form languages

When you first learn a programming language, you're allowed to make minor mistakes along the way. When it comes to actually creating a workable program, however, some languages are very particular about how you write it and will block your progress unless the program is perfect.

Some programming languages, such as C, demand that you're extremely careful about the types of data you're using; if you make a mistake, the program can cause errors without you noticing them. Other languages, such as Perl, are much more lenient; they let you be less specific about the kind of data you're using.

Professional programmers can argue on and on about whether a language should be picky or not. Some say being picky is good because it forces the programmer to dictate exactly what he or she wants, leaving no guesswork for the language. Others say languages should be lenient so that a programmer can write code quickly without having to dot all the *i*'s and cross all the *t*'s.

A programming language can be stringent or lenient in other ways, as well. Some languages require that every line of a program line up with each other in a certain way, and others are much more free-form. By the way, you'll discover that Perl is overall a pretty lenient language. However, if you want Perl to get picky, you can tell it to check your programs for probable errors in order to find the things that you may have done wrong.

You may think that every language should be simple to learn so that more people can use it. However, that's not really necessary because a fair number of simple languages already exist — Perl being one of them. Instead of creating new, easily mastered languages, language designers are targeting specific groups of programmers who will spend the time to work with special features or interesting language forms.

When creating a new computer language, designers must keep a number of considerations in mind:

- How fast will the language run?
- How easy is it to extend the language to handle tasks unanticipated by the language's authors?
- What sort of background do programmers of that language need?
- What type of computers will run the language?

Because of these design challenges, relatively few computer languages are widely used. Although at least 100 languages have been disseminated over the past 30 years, fewer than a dozen are used by more than a few thousand people.

Appreciating the Benefits of Perl

Computer users can argue *ad nauseum* about which programming language is the best or easiest to use. The fact is, no single language is perfect for every task, and the top three or four languages for most tasks get the job done equally well. For the novice or intermediate programmer, then, the question is not really what is the best program to use, but which is the easiest one to master.

Perl scores high on both the learning-curve and ease-of-use scales. You can write a small Perl program designed to do a simple task after reading just the first few chapters of this book. In fact, if you read through the first part of this book, you'll understand enough about Perl that you can begin to modify programs to your heart's content. One of the common phrases in the world of Perl programmers is "there's more than one way to do it."

Certainly, other languages are easy to learn and use, but they don't have the features that make Perl a great all-around language, such as flexibility with the kinds of data it can handle or its ability to deal with objects. Here's the lowdown on some other languages:

- **BASIC:** Good old BASIC has features similar to the easy features of Perl, but it's not very good for modern programming because of its lack of flexibility.
- **Visual Basic:** Microsoft Visual Basic is not nearly as easy to learn as BASIC, and although it's very powerful, many novices find Visual Basic pretty confusing.
- **C:** Some people think C is easy to learn, but difficult to use unless you're very careful, because you can create C programs that look great on the surface but do harmful things to your computer. C also loses points in the text-handling area — one of the places in which Perl shines.
- **Java:** Boy, it's hard to talk about Java without either the pro-Java or anti-Java zealots attacking you. Speaking frankly, Java is an okay language that suffers from being horribly over-promoted. It has some great features, such as safe execution of programs in secure environments and easy reuse of programs written by other people. Unfortunately, it fails miserably at the feature for which it was most touted; namely, that you can write a program for one type of computer and have the program look and act the same on all other types of computers.

Perl also surpasses other programming languages on some common tasks. With Perl, the tasks of opening a file on your computer, reading it, and making some changes based on what you find in the file is quite easy. Perl can handle text files with aplomb, and has no problems with binary files. Perl is also good at handling text in ways that humans do, such as looking at a sentence and breaking it into words or sorting lines in alphabetical order.

A bit of Perl history

Programming languages have been around for several decades. Perl (short for *Practical Extraction and Report Language*) is one of the youngest of the popular programming languages, debuting in the mid-1980s. The idea for Perl pretty much sprang from the head of one person, Larry Wall, who had a bunch of system-administration tasks to do and no good language to get them all done.

In short, Perl was born out of necessity. Larry Wall needed a language that had the power to open a bunch of text files in different formats, read these different files in one way, and create new files that reported on the original files. He could have used C, a popular language at the time, but C is ornery when dealing with text and is prone to making difficult-to-locate errors if you're not careful.

Perl has been extended to do much more than extract text and then generate a report on it. Thousands of useful system-administration and Internet tools now rely on Perl to perform tasks such as summarizing log files, taking input from Web users, and displaying parts of databases on the Web. In fact, a few Web server programs are written completely in Perl.

Perl now comes bundled with almost every copy of every flavor of UNIX. For a long time, Perl was mostly a UNIX-only language. Until a few years ago, virtually all work on Perl was done on UNIX. The language has recently become popular on PCs and Macintoshes, in part because it is free and easy to get. Indeed, Perl is the most popular free programming language available for Windows systems and Macs, and as these computers need more and more system administration, Perl's acceptance and use is likely to grow even more.

The ten-second summary of Perl's strongest points is this: Perl works great for reading through text files, summarizing processed files or converting files to a different format, and managing UNIX systems. Other languages strive to be elegant or very small; Perl strives to be complete and easy to use.

Another great reason for using Perl is that it's free. This no-cost policy has prompted many people to help out with the Perl development effort and has resulted in widespread usage of Perl over a relatively short period of time. Perl is also *open source software*, meaning that anyone can look at the internal programming in Perl and correct errors or even extend the language itself. Open source software is considered to be more secure and reliable than software for which you can't check the internals.

Many Versions, One Perl

Throughout this book, I talk about Perl as if it were just one programming language. But, like all software, Perl has gone through many changes since the time Larry Wall conceived it. Fortunately, you need to think about only two

major versions of Perl: Perl 5.6 and Perl 5.8. (An older version of Perl, Perl 4, is rarely seen and is mentioned in a few places in this book for those of you using very old copies of Perl.)

Each major version (currently, the major version is Perl 5) has many minor versions; for example, for the last edition of this book, the current version of Perl was 5.6; the current version is now 5.8. All minor versions of a major version act pretty much the same, but higher numbered minor versions usually have a few additions and bug fixes.

Perl version 4 was the stable version of Perl for many years. In fact, a few people still use Perl 4 instead of Perl 5 because they just got so used to it — even though Perl 5 added many advanced features. The final subversion of Perl 4 is "patchlevel 36," but most people just refer to it as version 4.036. Whenever I talk about Perl 4 in this book, I'm referring to Perl version 4.036.

In early 1996, people started making the switch to Perl 5; in early 2000, people started switching to Perl 5.6; now people are switching to Perl 5.8. Perl 5.6 and Perl 5.8 are pretty stable (although not completely bug-free, of course) and can certainly be used for almost any application.

This book is almost exclusively about Perl 5.8, which is not to say that a Perl 5.6 user can't use the programs found here. (Perl 5.7 was only used by the people developing Perl — not by folks like you and me.) In fact, about 95 percent of what's in this book works exactly the same under Perl 5.6 as it does with Perl 5.8, which is a testament to the folks who worked so carefully on the Perl project to engineer a smooth transition from Perl 4 to Perl 5 to Perl 5.6 to Perl 5.8.

However, a few things in Perl 5.8 just plain didn't exist in Perl 4, and of course, I cover them in this book. For example, Chapter 19 covers object-oriented Perl, a feature that doesn't exist in Perl 4. I sprinkle descriptions of a few smaller, but nevertheless handy, new features in Perl 5 throughout the book. Similarly, a few things now exist in Perl 5.8 that didn't exist in Perl 5.6, and I cover those as well.

I urge everyone using Perl 5 to upgrade to Perl 5.6 or, preferably, Perl 5.8. Doing so will make your time with this book much easier, and will also give you a much better (and more bug-free!) experience.

Differentiating Between Perl 5 and Perl 5.6

Perl 5.6 took almost four years to come out, so you may be expecting a zillion new features to have been shoved in all over the place. For many programming languages, that would be a reasonable expectation, but not for Perl. The

good folks who maintain Perl have stuck to their convictions and kept "creeping featureitis syndrome" to a minimum.

The most visible change, and the one that has garnered the most interest around the world, is Perl's embracing of international characters in general and the Unicode standard in particular. People using just the English language have it easy: They have a small set of characters in their alphabet, and none of those characters have diacritic marks such as acute accents (´) and umlauts (¨). Almost every non-English language, however, uses at least a few characters that are not found in the English alphabet.

Until version 5.6, Perl had no standard way of letting you enter these characters or showing you these characters in your output. With version 5.6, that is fairly easy to do. Perl relies on the Unicode standard, which was created and is maintained by the *Unicode Consortium,* a group of companies and individuals who have been working for over a decade to make all the world's languages easy to handle for all computers. You can find out more about the Unicode standard at `www.unicode.org` and more about how Perl 5.6 uses Unicode in Chapter 5 of this book.

The other improvements in Perl 5.6 aren't nearly as visible to most users. In fact, the creators of Perl 5.6 made a successful effort to mostly add "backend" changes in the new version, and to rely on Perl modules (described in Chapter 19) to add the more visible features. Most of these features are well beyond the scope of this book, but a few that appear include binary numbers and closer integration with Microsoft Windows operating systems.

Exploring the New Features of Perl 5.8

The difference between Perl 5.6 and Perl 5.8 is much smaller than the difference between Perl 5 and Perl 5.6. Whereas Perl 5.6 introduced many new and significant concepts to Perl, Perl 5.8 is mostly an evolutionary revision.

That is not to say that Perl 5.8 isn't important — because it is. For one thing, Perl 5.8 is significantly faster for many programs. It is also more stable than earlier versions of Perl. In addition, it has many new features, including:

- Improved support of internationalized text (covered in Chapter 5)
- Additional functions and modules (covered throughout the book)
- Better accuracy for large numbers (covered in Chapter 6)

All of this adds up to a good reason for you to update to Perl 5.8 if you haven't already.