"When I look at my bookshelf, I see eleven books on Perl programming. *Perl by Example, Third Edition*, isn't on the shelf; it sits on my desk, where I use it almost daily. I still think it is the best Perl book on the market for anyone—beginner or seasoned programmer—who uses Perl daily."

—Bill Maples, Enterprise Network Support, Fidelity National Information Services

PRENTICE HALL

PERL by EXAMPLE

FOURTH EDITION

CD-ROM contains all source code and data files used in the book

Ellie Quigley

Praise for Ellie Quigley's Books

"I picked up a copy of *JavaScript by Example* over the weekend and wanted to thank you for putting out a book that makes JavaScript easy to understand. I've been a developer for several years now and JS has always been the "monster under the bed," so to speak. Your book has answered a lot of questions I've had about the inner workings of JS but was afraid to ask. Now all I need is a book that covers Ajax and Coldfusion. Thanks again for putting together an outstanding book."

—Chris Gomez, Web services manager,
Zunch Worldwide, Inc.

"I have been reading your *UNIX® Shells by Example* book, and I must say, it is brilliant. Most other books do not cover all the shells, and when you have to constantly work in an organization that uses tcsh, bash, and korn, it can become very difficult. However, your book has been indispensable to me in learning the various shells and the differences between them…so I thought I'd email you, just to let you know what a great job you have done!"

—Farogh-Ahmed Usmani, B.Sc. (Honors), M.Sc., DIC,
project consultant (Billing Solutions), Comverse

"I have been learning Perl for about two months now; I have a little shell scripting experience but that is it. I first started with *Learning Perl* by O'Reilly. Good book but lacking on the examples. I then went to *Programming Perl* by Larry Wall, a great book for intermediate to advanced, didn't help me much beginning Perl. I then picked up *Perl by Example, Third Edition*—this book is a superb, well-written programming book. I have read many computer books and this definitely ranks in the top two, in my opinion. The examples are excellent. The author shows you the code, the output of each line, and then explains each line in every example."

—Dan Patterson, software engineer,
GuideWorks, LLC

"Ellie Quigley has written an outstanding introduction to Perl, which I used to learn the language from scratch. All one has to do is work through her examples, putz around with them, and before long, you're relatively proficient at using the language. Even though I've graduated to using *Programming Perl* by Wall et al., I still find Quigley's book a most useful reference."

—Casey Machula, support systems analyst,
Northern Arizona University, College of Health and Human Services

"When I look at my bookshelf, I see eleven books on Perl programming. *Perl by Example, Third Edition,* isn't on the shelf; it sits on my desk, where I use it almost daily. When I bought my copy I had not programmed in several years and my programming was mostly in COBOL so I was a rank beginner at Perl. I had at that time purchased several popular books on Perl but nothing that really put it together for me. I am still no pro, but my book has many dog-eared pages and each one is a lesson I have learned and will certainly remember.

"I still think it is the best Perl book on the market for anyone from a beginner to a seasoned programmer using Perl almost daily."

—Bill Maples, network design tools and automations analyst,
Fidelity National Information Services

"We are rewriting our intro to OS scripting course and selected your text for the course. It's an exceptional book. The last time we considered it was a few years ago (second edition). The debugging and system administrator chapters at the end nailed it for us."

—Jim Leone, Ph.D., professor and chair, Information Technology,
Rochester Institute of Technology

"Quigley's book acknowledges a major usage of PHP. To write some kind of front end user interface program that hooks to a back end MySQL database. Both are free and open source, and the combination has proved popular. Especially where the front end involves making an HTML web page with embedded PHP commands.

"Not every example involves both PHP and MySQL. Though all examples have PHP. Many demonstrate how to use PHP inside an HTML file. Like writing user-defined functions, or nesting functions. Or making or using function libraries. The functions are a key idea in PHP, that take you beyond the elementary syntax. Functions also let you gainfully use code by other PHP programmers. Important if you are part of a coding group that has to divide up the programming effort in some manner."

—Dr. Wes Boudville, CTO,
Metaswarm Inc.

Perl by Example

Fourth Edition

This page intentionally left blank

Perl by Example

Fourth Edition

Ellie Quigley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

The publisher offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U.S. Corporate and Government Sales
(800) 382-3419
corpsales@pearsontechgroup.com

For sales outside the United States please contact:

International Sales
international@pearsoned.com

Visit us on the Web: informit.com/ph

Library of Congress Cataloging-in-Publication Data

Quigley, Ellie.
Perl by example / Ellie Quigley. — 4th ed.
p. cm.
Includes index.
ISBN 978-0-13-238182-6 (pbk. : alk. paper) 1. Perl (Computer program language) I. Title.
QA76.73.P22Q53 2007
005.13'3—dc22

2007029600

ISBN-13: 978-0-13-238182-6
ISBN-10: 0-13-238182-6

Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
Second printing, January 2009

Editor-in-Chief
Mark L. Taub

Managing Editor
John Fuller

Full-Service Production Manager
Julie B. Nahil

Production Editor
Dmitri Korzh,
Techne Group

Copy Editor
Techne Group

Indexer
Larry Sweazy

Proofreader
Evelyn Pyle

Publishing Coordinator
Noreen Regina

Cover Designer
Alan Clements

Composition
Techne Group

Contents

Preface

You may wonder, why a new edition of *Perl by Example*? Perl 5 hasn't really changed that much; in fact, it's changed very little at all since the third edition of this book was published. And since Perl 6 hasn't been officially released, why not wait? Well, consider this. Let's say you bought a new Whirlpool washing machine six years ago. It's running perfectly. But since then, the mounds of laundry washed by that machine have come and gone. Now you're sporting a new trendy fashion, you have designer sheets and towels, and the detergent brand you use is hypoallergenic, nontoxic, and biodegradable, not available when you bought the washer. Even though Perl 5 has changed very little, the computer world has. It is always in a flux of new innovations, technologies, applications, and fads, and programs are being written to accommodate those changes. Whether analyzing data from the GenBank sequence database, writing applications for an iPhone, creating a personal blog on "myspace," or adjusting to the changes in a new Vista version of Windows, some computer program is involved, and very possibly it is a Perl program. Whatever the case, we like to keep up with the times. This new edition of *Perl by Example* was written for just that purpose.

As we speak, I am teaching Perl at the UCSC[1] extension in Sunnyvale, California, to a group of professionals coming from all around the Silicon Valley. I always ask at the beginning of a class, "So why do you want to learn Perl?" The responses vary from, "Our company has an auction site on the Web and I'm the webmaster. I need to use Perl and Apache to process our order information and send it to Oracle," or "I work in a genetics research group at Stanford and have to sift through and analyze masses of data, and I heard that if I learn Perl, I won't have to depend on programmers to do this," or "I'm a UNIX/Linux system administrator and our company has decided that all admin scripts should be converted to Perl," or "I just got laid off and heard that it's an absolute must to have Perl on my resume." And I am always amazed at the variety of people who show up: engineers, scientists, geneticists, meteorologists, managers, salespeople, programmers, techies, hardware guys, students, stockbrokers, administrators of all kinds,

1. University of California, Santa Cruz.

librarians, authors, bankers, artists—you name it. Perl does not exclude anyone. Perl is for everyone and it runs on everything.

No matter who you are, I think you'll agree that a picture is worth a thousand words, and so is a good example. *Perl by Example* is organized to teach you Perl from scratch with examples of complete, succinct programs. Each line of a script example is numbered, and important lines are highlighted in bold. The output of the program is then displayed with line numbers corresponding to the script line numbers. Following the output is a separate explanation for each of the numbered lines. The examples are small and to the point for the topic at hand. Since the backbone of this book was used as a student guide to a Perl course, the topics are modularized. Each chapter builds on the previous one with a minimum of forward referencing and a logical progression from one topic to the next. There are exercises at the end of the chapters. You will find all of the examples on the CD at the back of the book. They have been thoroughly tested on a number of major platforms.

Perl by Example is not just a beginner's guide but a complete guide to Perl. It covers many aspects of what Perl can do, from regular expression handling, to formatting reports, to interprocess communication. It will teach you about Perl and, in the process, a lot about UNIX and Windows. Since Perl was originally written on and for UNIX systems, some UNIX knowledge will greatly accelerate your learning curve, but it is not assumed that you are by any means a guru. Anyone reading, writing, or just maintaining Perl programs can greatly profit from this text.

Perl has a rich variety of functions for handling strings, arrays, the system interface, networking, and more. In order to understand how these functions work, background information concerning the hows, whys, and what-fors is provided before demonstrating functional sample programs. This eliminates continually wading through manual pages and other books to understand what is going on, what the arguments mean, and what the function actually does.

The appendices contain a complete list of functions and definitions, command-line switches, special variables, popular modules, and the Perl debugger; a bioinformatics tutorial to introduce *BioPerl*, and a tutorial covering *mod_perl*, the fast way to create server side Perl scripts that replace the need for the Common Gateway Interface.

I have been teaching for the past thirty years and am committed to understanding how people learn. Having taught Perl now for more than 14 years, all over the world, I find that many new Perlers get frustrated when trying to teach themselves how to program. Most people seem to learn best from succinct little examples and practice. So I wrote a book to help myself learn and to help my students, and now to help you. As Perl has grown, so have my books. This latest, fourth, edition includes a new chapter on Perl and DBI with MySQL, a revised chapter on Perl objects, and new examples and explanations for the rest of the chapters to keep things current and interesting. The appendix material has been revised to include BioPerl and *mod_perl*. In this book, you will not only learn Perl, but also save yourself a great deal of time. At least that's what my students and readers have told me. You be the judge.

Acknowledgments

I'd like to acknowledge the following people for their contributions to the fourth edition.

Thanks to Dmitri Korzh and Techne Group for their skill in editing, formatting, and indexing that turned my attempts at using FrameMaker from a rough chunk of raw text into a real professional, polished book.

I'd like to acknowledge Oleg Orel, a brilliant student from NetApp, who wrote the initial program to illustrate "closures" in the chapter on objects, and who helped me with the problems I was having downloading modules from CPAN.

Thank you, Mark Taub, the editor-in-chief to be praised for being very cool in every step of the process from the signing of the contract to the final book that you have now in your hand. Mark has a way of making such an arduous task seem possible; he soft talks impossible deadlines, keeps up a steady pressure, and doesn't get crazy over missed deadlines, quietly achieving his goal and always with a subtle sense of humor. Thank you, Mark, for being the driving force behind this new edition!

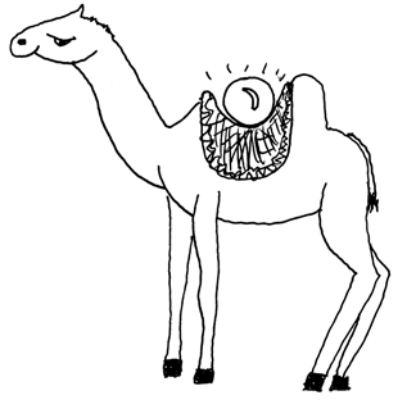
Of course, none of this would have been possible without the contributions of the Perl pioneers—Larry Wall, Randal Schwartz, and Tom Christiansen. Their books are must reading and include *Learning Perl* by Randal Schwartz and *Programming Perl* by Larry Wall, Tom Christiansen, and Jon Orwant.

And last, but certainly not least, a huge thanks to all the students, worldwide, who have done all the real troubleshooting and kept the subject alive.

This page intentionally left blank

chapter

1



The Practical Extraction and Report Language

1.1 What Is Perl?

> "Laziness, impatience, and hubris. Great Perl programmers embrace those virtues."
>
> —Larry Wall

Perl is an all-purpose, open source (free software) interpreted language maintained and enhanced by a core development team called the Perl Porters. It is used primarily as a scripting language and runs on a number of platforms. Although inititally designed for the UNIX operating system, Perl is renowned for its portability and now comes bundled with most operating systems, including RedHat Linux, Solaris, FreeBSD, Macintosh, and more. Due to its versatility, Perl is often referred to as the Swiss Army knife of programming languages.

Larry Wall wrote the Perl language to manage log files and reports scattered over the network. According to Wikipedia.org, "Perl was originially named "Pearl" after the "Parable of the Pearl" from the "Gospel of Matthew." The parable is brief: A merchant is seeking pearls. He finds one that is so valuable and beautiful that he is willing to sell everything he has to purchase it. And in the end he is even wealthier than he was before. However you interpret this, it has very positive implications.

Before its official release in 1987 the "a" in "Pearl" was dropped and the language has since been called "Perl," later dubbed the Practical Extraction and Report Language, and by some, it is referred to as the Pathologically Eclectic Rubbish Lister. Perl is really much more than a practical reporting language or eclectic rubbish lister as you'll soon see. Perl makes programming easy, flexible, and fast. Those who use it, love it. And those who use it range from experienced programmers to novices with little computer background at all. The number of users continues to grow at a phenomenal rate.[1]

1. Perl is spelled "Perl" when referring to the language, and "perl" when referring to the interpreter.

Perl's heritage is UNIX. Perl scripts are functionally similar to UNIX *awk*, *sed*, shell scripts, and *C* programs. Shell scripts consist primarily of UNIX commands; Perl scripts do not. Whereas *sed* and *awk* are used to edit and report on files, Perl does not require a file in order to function. Whereas *C* has none of the pattern matching and wildcard metacharacters of the shells, *sed,* and *awk*, Perl has an extended set of characters. Perl was originally written to manipulate text in files, extract data from files, and write reports, but through continued development, it can manipulate processes, perform networking tasks, process Web pages, talk to databases, and analyze scientific data. Perl is truly the Swiss Army knife of programming languages; there is a tool for everyone.

The examples in this book were created on systems running Solaris, Linux, Macintosh UNIX, and Win32.

Perl is often associated with a camel symbol, a trademark of O'Reilly Media, which published the first book on Perl, called *Programming Perl* by Larry Wall and Randal Schwartz, referred to as "the Camel Book."

1.2 What Is an Interpreted Language?

To write Perl programs, you need two things: a text editor and a Perl interpreter, which you can download very quickly from any number of Web sites, including *perl.org*, *cpan.org*, and *activestate.com*. Unlike with compiled languages, such as C++ and Java, you do not need to first compile your program into machine-readable code before it can be executed. The Perl interpreter does it all; it handles the compilation, interpretation, and execution of your program. Advantages of using an interpreted language like Perl is that it runs on almost every platform, is relatively easy to learn, and is very fast and flexible.

Languages such as Python, Java, and Perl are interpreted languages that use an intermediate representation, which combines both compilation and interpretation. It compiles the user's code into an internal condensed format called bytecode, or threaded code, which is then executed by the interpreter. When you run Perl programs, you need to be aware of two phases: the compilation phase and then the run phase, where you will see the program results. If you have syntax errors, such as a misspelled keyword or missing quote, the compiler will send an error. If you pass the compiler phase, you could have other problems when the program starts running. If you pass both of these phases, you will probably start working on formatting to make the output look nicer or improving the program to make it more efficient, etc.

The interpreter also provides a number of command-line switches (options) to control its behavior. There are switches to check syntax, send warnings, loop through files, execute statements, turn on the debugger, etc. You will learn about these options throughout the following chapters.

1.3 Who Uses Perl?

Because Perl has built-in functions for easy manipulation of processes and files, and because Perl is portable (i.e., it can run on a number of different platforms), it is especially popular with system administrators, who often oversee one or more systems of different types. The phenomenal growth of the World Wide Web greatly increased interest in Perl, which was the most popular language for writing CGI scripts to generate dynamic Web pages. Even today, with the advent of other languages, such as PHP and ASP.net, focused on processing Web pages, Perl continues increased popularity with system and database administrators, scientists, geneticists, and anyone who has a need to collect data from files and manipulate it.

Anyone can use Perl, but it is easier to learn if you are already experienced in writing UNIX shell scripts, Perl, or languages derived from *C*, such as *C++* and *Java*. For these people, the migration to Perl will be relatively easy. For those who have little programming experience, the learning curve might be a little steeper, but after learning Perl, there may be no reason to ever use anything else.

If you are familiar with UNIX utilities such as *awk*, *grep*, *sed*, and *tr*, you know that they don't share the same syntax; the options and arguments are handled differently, and the rules change from one utility to the other. If you are a shell programmer, you usually go through the grueling task of learning a variety of utilities, shell metacharacters, regular expression metacharacters, quotes, and more quotes, etc. Also, shell programs are limited and slow. To perform more complex mathematical tasks and to handle interprocess communication and binary data, for example, you may have to turn to a higher-level language, such as *C*, *C++*, or *Java*. If you know *C*, you also know that searching for patterns in files and interfacing with the operating system to process files and execute commands are not always easy tasks.

Perl integrates the best features of shell programming, *C*, and the UNIX utilities *awk*, *grep*, *sed*, and *tr*. Because it is fast and not limited to chunks of data of a particular size, many system administrators and database administrators have switched from the traditional shell scripting to Perl. *C++* and *Java* programmers can enjoy the object-oriented features added in Perl 5, including the ability to create reusable, extensible modules. Now Perl can be generated in other languages, and other languages can be embedded in Perl. There is something for everyone who uses Perl, and for every task "there's more than one way to do it" (*http://www.oreilly.com/catalog/opensources/book/larry.html*).

You don't have to know everything about Perl to start writing scripts. You don't even have to be a programmer. This book will help you get a good jump-start, and you will quickly see some of its many capabilities and advantages. Then you can decide how far you want to go with Perl. If nothing else, Perl is fun!

1.3.1 Which Perl?

Perl has been through a number of revisions. There are two major versions of Perl: Perl 4 and Perl 5. The last version of Perl 4 was Perl 4, patchlevel 36 (Perl 4.036), released in 1992, making it ancient. Perl 5.000 (ancient), introduced in fall 1994, was a complete

rewrite of the Perl source code that optimized the language and introduced objects and many other features. Despite these changes, Perl 5 remains highly compatible with the previous releases. (Examples in this book have been tested using both versions, and where there are differences, they are noted.) As of this writing, the current version of Perl is 5.8.8. Perl 6 is the next generation of another Perl redesign and does not have an official release date. It will have new features, but the basic language you learn here will be essentially the same.

1.3.2 What Is Perl 6?

> "Perl 5 was my rewrite of Perl. I want Perl 6 to be the community's rewrite of Perl and of the community."
>
> —Larry Wall, State of the Onion speech, TPC4

Perl 6 is essentially Perl 5 with many new features. The basic language syntax, features, and purpose will be the same. If you know Perl, you will still know Perl. If you learn Perl from this book, you will be prepared to jump into Perl 6 when it is released. Perl 6 has been described as learning Australian English if you speak American English, rather than trying to switch from English to Chinese.

To get information about everything happening with Perl 6, go to:
http://www.perl.com/pub/a/2006/01/12/what_is_perl_6.html?page=2

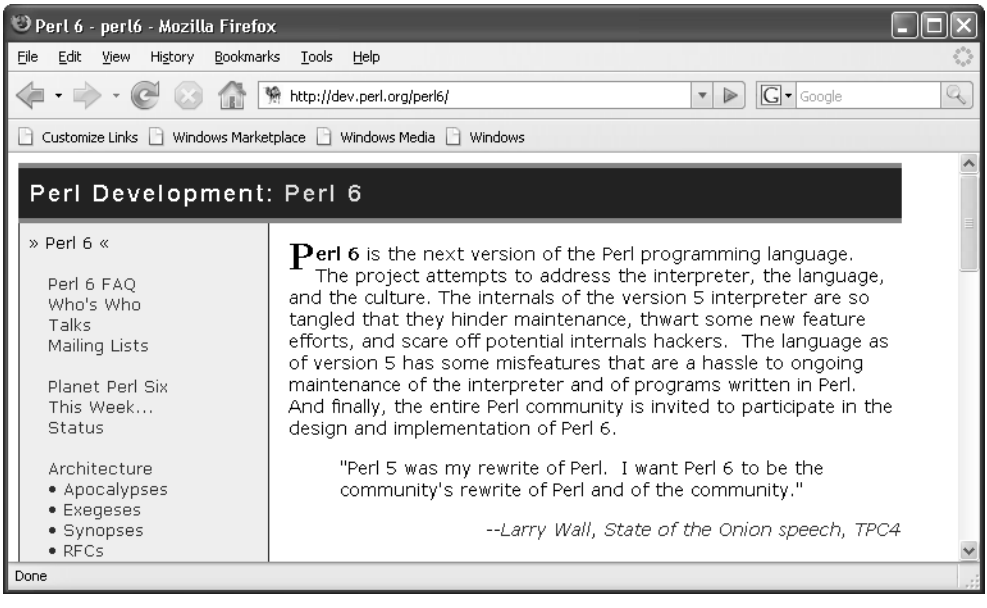


Figure 1.1 Perl 6 development Web page.

And for a sketch of Larry Wall and history of Perl, go to:
http://www.softpanorama.org/People/Wall/index.shtml#Perl_history

1.4 Where to Get Perl

Perl is available from a number of sources. The primary source for Perl distribution is CPAN, the Comprehensive Perl Archive Network (*www.cpan.org*).

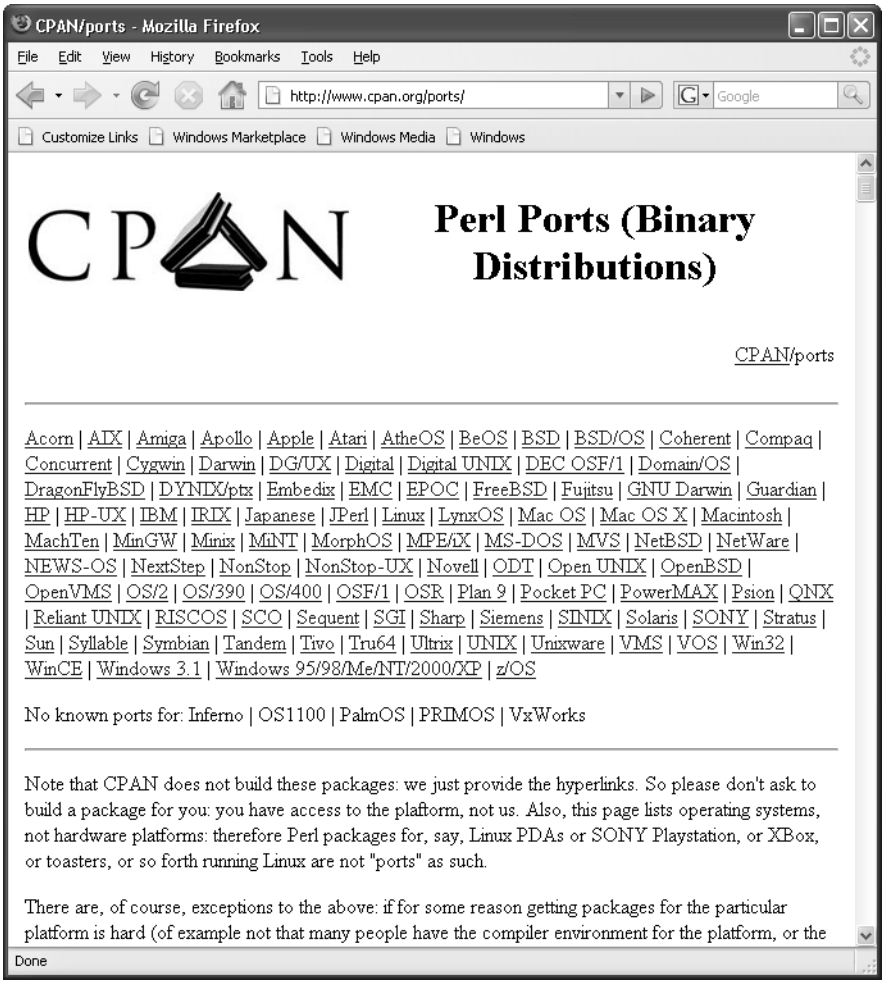


Figure 1.2 CPAN ports for binary distribution.

Go to *http://www.cpan.org/ports/* to find out more about what's available for your platform. If you want to install Perl quickly and easily, ActivePerl is a complete, self-installing distribution of Perl based on the standard Perl sources for Windows, Mac OS X, Linux, Solaris, AIX, and HP-UX. It is distributed online at the ActiveState site (*www.activestate.com*). The complete ActivePerl package contains the binary of the core Perl distribution and complete online documentation.

Here are some significant Web sites to help you find more information about Perl:

- The official Perl home page, run by O'Reilly Media, Inc.: *www.perl.com*
- The Perl Directory, run by the Perl Foundation, with the aim of being "the central directory of all things Perl": *www.perl.org*
- The Comprehensive Perl Archive Network, where you will also find "All Things Perl": *http://www.cpan.org/*
- The site where you will find the essential tools for Perl development: *http://www.activestate.com/*

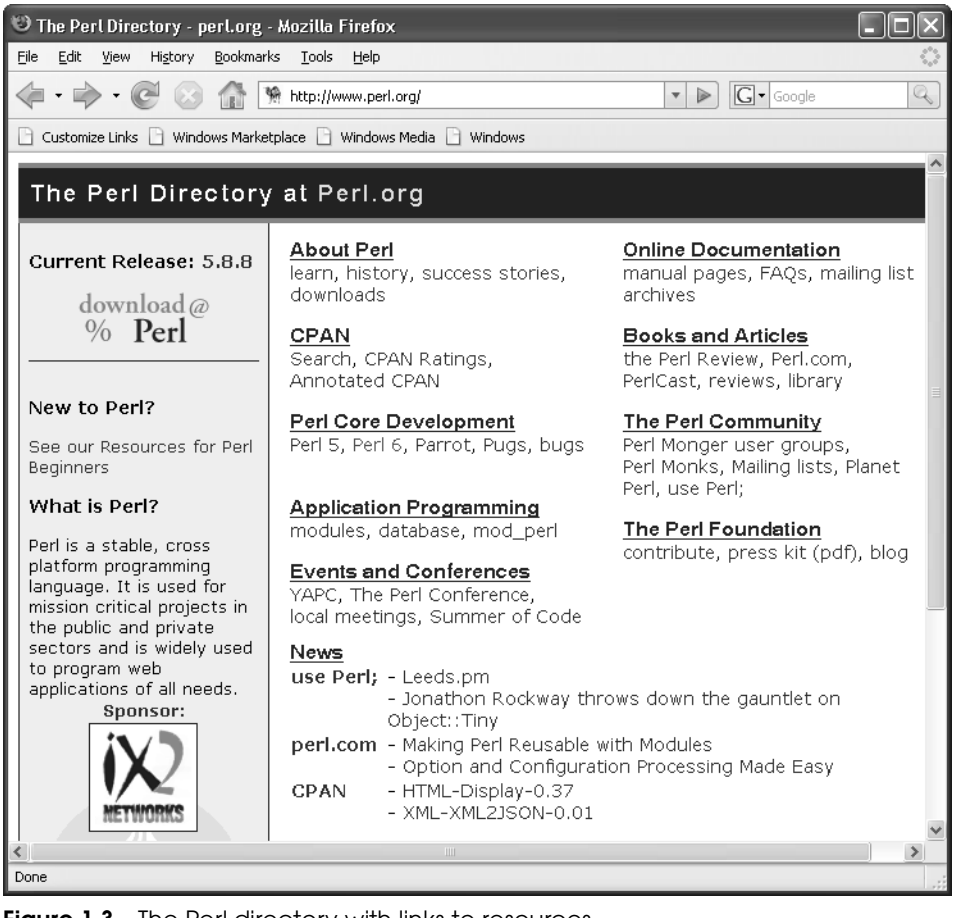


Figure 1.3 The Perl directory with links to resources.

Figure 1.4 The official Perl home page (run by O'Reilly Media).

1.4.1 What Version Do I Have?

To obtain your Perl version, date this binary version was built, patches, and some copyright information, type the following line shown in Example 1.1 (the dollar sign is the shell prompt):

EXAMPLE 1.1

```
$ perl -v
1   This is perl, v5.8.8 built for MSWin32-x86-multi-thread
    (with 50 registered patches, see perl -V for more detail)

2   Copyright 1987-2006, Larry Wall

3   Binary build 820 [274739] provided by ActiveState
    http://www.ActiveState.com
    Built Jan 23 2007 15:57:46
```

EXAMPLE 1.1 (CONTINUED)

```
4   Perl may be copied only under the terms of either the Artistic
    License or the GNU General Public License, which may be found in
    the Perl 5 source kit.
       Complete documentation for Perl, including FAQ lists, should be
    found on this system using "man perl" or "perldoc perl".  If you
    have access to the Internet, point your browser at
    http://www.perl.org/, the Perl Home Page.

    This is perl, v5.8.8 built for MSWin32-x86-multi-thread
    (with 1 registered patch, see perl -V for more detail)

5   Perl may be copied only under the terms of either the Artistic
    License or the GNU General Public License, which may be found in
    the Perl 5.0 source kit.
      Complete documentation for Perl, including FAQ lists, should be
    found on this system using man perl or perldoc perl. If you have
    access to the Internet, point your browser to www.perl.com/, the
    Perl home page.

    ------------------------------------------------------------

6   perl -v
    This is perl, v5.8.3 built for sun4-solaris-thread-multi
    (with 8 registered patches, see perl -V for more detail)

    Copyright 1987-2003, Larry Wall

    Binary build 809 provided by ActiveState Corp.
    http://www.ActiveState.com
    ActiveState is a division of Sophos.
    Built Feb  3 2004 00:32:12

    Perl may be copied only under the terms of either the Artistic
    License or the GNU General Public License, which may be found in
    the Perl 5 source kit.

    Complete documentation for Perl, including FAQ lists, should be
    found on this system using `man perl' or `perldoc perl'.  If you
    have access to the Internet, point your browser at
    http://www.perl.com/, the Perl Home Page.
```

EXPLANATION

1 This version of Perl is 5.8.8 from ActiveState for Windows.
2 Larry Wall, the author of Perl, owns the copyright.
3 This build was obtained from ActiveState.
5 Perl may be copied under the terms specified by the Artistic License or GNU. Perl is distributed under GNU, the Free Software Foundation, meaning that Perl is free.
6 This version of Perl is 5.8.3 for Solaris (UNIX).

1.5 What Is CPAN?

CPAN, the "gateway to all things Perl," stands for the Comprehensive Perl Archive Network, a Web site that houses all the free Perl material you will ever need, including documentation, FAQs, modules and scripts, binary distributions and source code, and announcements. CPAN is mirrored all over the world, and you can find the nearest mirror at

www.perl.com/CPAN
www.cpan.org

CPAN is the place you will go to if you want to find modules to help you with your work. The CPAN search engine will let you find modules under a large number of categories. Modules are discussed in Chapter 12, "Modularize It, Package It, and Send It to the Library!"

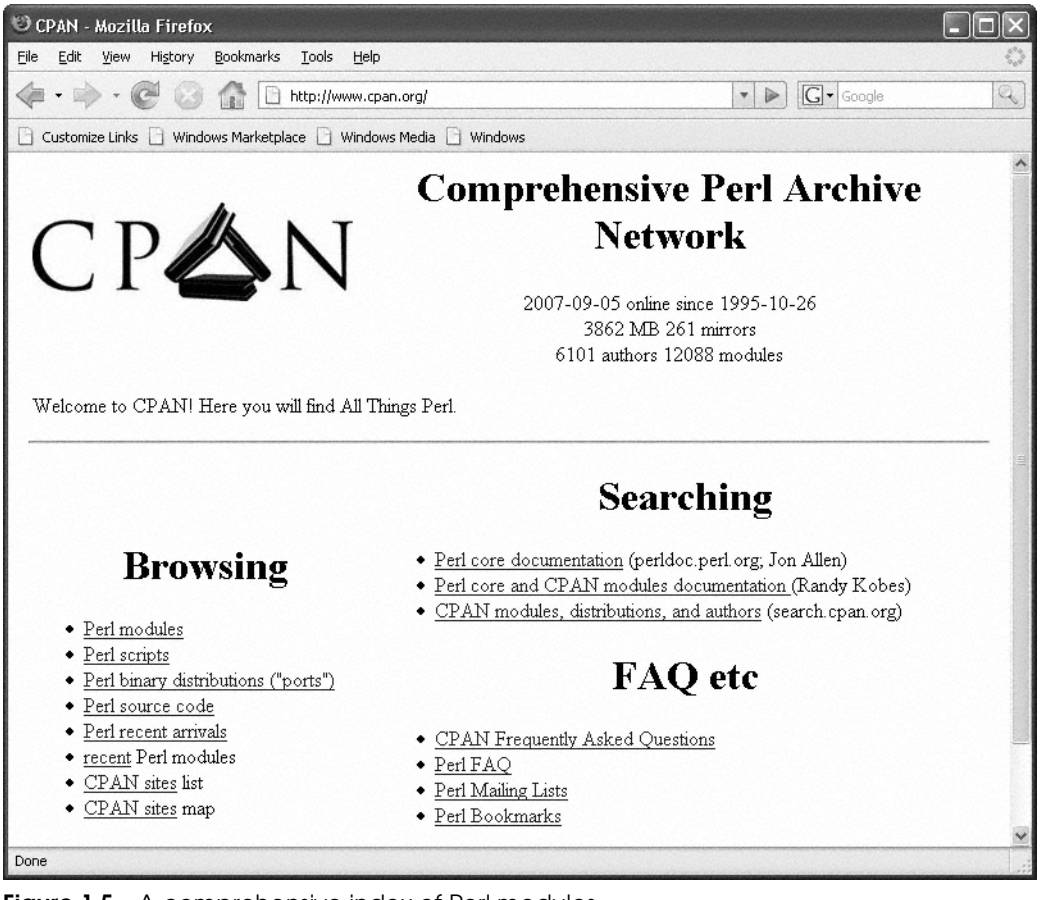


Figure 1.5 A comprehensive index of Perl modules.

1.6 Perl Documentation

1.6.1 Perl Man Pages

The standard Perl distribution comes with complete online documentation called *man* pages, which provide help for all the standard utilities. (The name derives from the UNIX *man* [manual] pages.) Perl has divided its *man* pages into categories. If you type the following at your command-line prompt:

```
man perl
```

you will get a list of all the sections by category. So, if you want help on how to use Perl's regular expresssions, you would type

```
man perlre
```

and if you want help on subroutines, you would type

```
man perlsub
```

The Perl categories are listed as follows, with the following sections available only in the online reference manual:

perlbot	Object-oriented tricks and examples
perldebug	Debugging
perldiag	Diagnostic messages
perldsc	Data structures: intro
perlform	Formats
perlfunc	Built-in functions
perlipc	Interprocess communication
perllol	Data structures: lists of lists
perlmod	Modules
perlobj	Objects
perlop	Operators and precedence
perlpod	Plain old documentation
perlre	Regular expressions
perlref	References
perlsock	Extension for socket support
perlstyle	Style guide
perlsub	Subroutines
perltie	Objects hidden behind simple variables
perltrap	Traps for the unwary
perlvar	Predefined variables

If you are trying to find out how a particular library module works, you can use the *perldoc* command to get the documentation. For example, if you want to know about the *CGI.pm* module, type at the command line

```
perldoc CGI
```

and the documentation for the *CGI.pm* module will be displayed. If you type

```
perldoc English
```

the documentation for the *English.pm* module will be displayed.

To get documentation on a specific Perl function, type *perldoc -f* and the name of the function. For example, to find out about the *localtime* function, you would execute the following command at your command-line prompt. (You may have to set your UNIX/DOS path to execute this program directly.)

```
perldoc -f localtime
localtime EXPR
localtime
        Converts a time as returned by the time function to a 9-element
        list with the time analyzed for the local time zone. Typically
        used as follows:
           #    0    1    2     3     4    5     6     7     8
            ($sec,$min,$hour,$mday,$mon,$year,$wday,$yday,$isdst) =
                                                        localtime(time);
<continues>
```

1.6.2 HTML Documentation

ActivePerl provides execllent documentation (from ActiveState.com) when you download Perl from its site. As shown in Figure 1.6, there are links to everything you need to know about Perl.

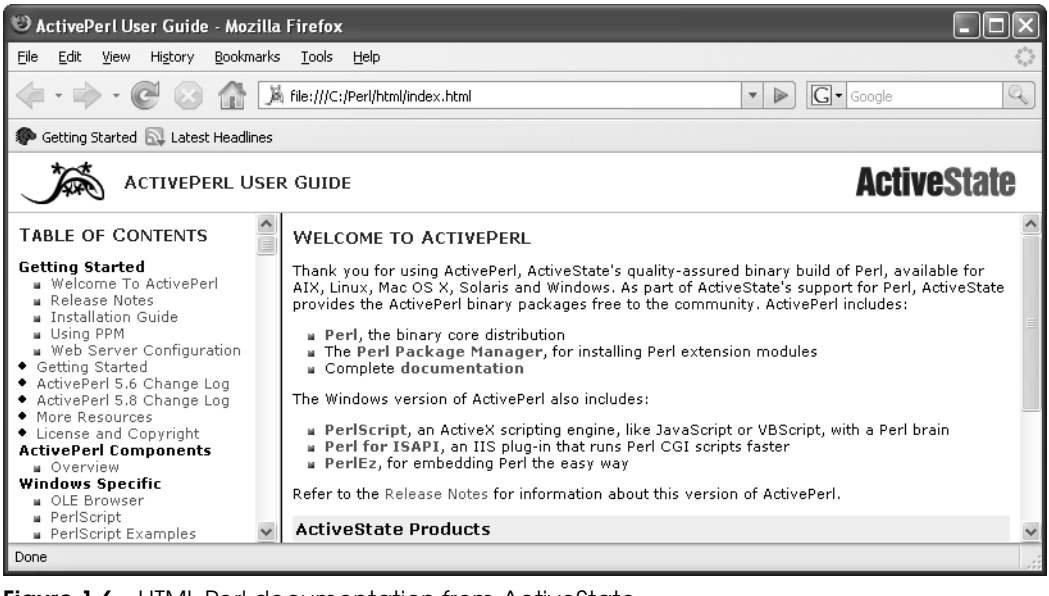


Figure 1.6 HTML Perl documentation from ActiveState.

1.7 What You Should Know

1. Who wrote Perl?
2. What does Perl stand for?
3. What is the meaning of "open source"?
4. What is the current release?
5. What is Perl used for?
6. What is an interpreter?
7. Where can you get Perl?
8. What is ActivePerl?
9. What is CPAN?
10. Where do you get documentation?
11. How would you find documentation for a specific Perl function?

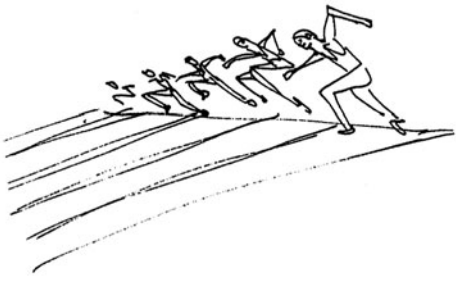
1.8 What's Next?

In the next chapter, you will learn how to create basic Perl scripts and execute them. You will learn what goes in a Perl script, about Perl syntax, statements, and comments. You will learn how to check for syntax errors and how to execute Perl at the command-line with a number of Perl options.

chapter

2

Perl Quick Start



2.1 Quick Start, Quick Reference

2.1.1 A Note to Programmers

If you have had previous programming experience in another language, such as Visual Basic, C/C++, Java, ASP, or PHP, and you are familiar with basic concepts, such as variables, loops, conditional statements, and functions, Table 2.1 will give you a quick overview of the constructs and syntax of the Perl language.

At the end of each section, you will be given the chapter number that describes the particular construct and a short, fully functional Perl example designed to illustrate how that constuct is used.

2.1.2 A Note to Non-Programmers

If you are not familiar with programming, skip this chapter and go to Chapter 5. You may want to refer to this chapter later for a quick reference.

2.1.3 Perl Syntax and Constructs

Table 2.1 Perl Syntax and Constructs

The Script File	A Perl script is created in a text editor. Normally, there is no special extension required in the filename, unless specified by the application running the script; e.g., if running under Apache as a *cgi* program, the filename may be required to have a *.pl* or *.cgi* extension.
Free Form	Perl is a free-form language. Statements must be terminated with a semicolon but can be anywhere on the line and span multiple lines.

Comments	Perl comments are preceded by a # sign. They are ignored by the interpreter. They can be anywhere on the line and span only one line.
	EXAMPLE
	`print "Hello, world";  # This is a comment`
	`    #  And this is a comment`

Printing Output	The *print and printf* functions are built-in functions used to display output.
	The *print* function arguments consist of a comma-separated list of strings and/or numbers.
	The *printf* function is similar to the C *printf()* function and is used for formatting output.
	Parentheses are not required around the argument list.
	(See Chapter 3.)
	`print value, value, value;`
	`printf ( string format [, mixed args [, mixed ...]] );`
	EXAMPLE
	`print "Hello, world\n";`
	`print "Hello,", " world\n";`
	`print ("It's such a perfect day!\n");  # Parens optional;.`
	`print "The the date and time are: ", localtime, "\n";`
	`printf "Hello, world\n";`
	`printf("Meet %s%:Age 5d%:Salary \$10.2f\n", "John", 40, 55000);`
	(See Chapter 4.)

Data Types/Variables	Perl supports three basic data types to hold variables: scalars, arrays, and associative arrays (hashes). Perl variables don't have to be declared before being used. Variable names start with a "funny character," followed by a letter and any number of alphanumeric characters, including the underscore. The funny character represents the data type and context. The characters following the funny symbol are case sensitive. If a variable name starts with a letter, it may consist of any number of letters (an underscore counts as a letter) and/or digits. If the variable does not start with a letter, it must consist of only one character. (See Chapter 5.)

Table 2.1 Perl Syntax and Constructs (continued)

Scalar

A scalar is a variable that holds a single value, a single string, or a number. The name of the scalar is preceded by a "$" sign. Scalar context means that one value is being used.

EXAMPLE

```
$first_name = "Melanie";
$last_name = "Quigley";
$salary = 125000.00;
print $first_name, $last_name, $salary;
```

Array

An array is an ordered list of scalars; i.e., strings and/or numbers. The elements of the array are indexed by integers starting at 0. The name of the array is preceeded by an "@" sign.

```
@names = ( "Jessica", "Michelle", "Linda" );
print "@names";  # Prints the array with elements separated by a space
print "$names[0] and $names[2]";  # Prints "Jessica" and "Linda"
print "$names[-1]\n";  # Prints "Linda"
$names[3]="Nicole";    # Assign a new value as the 4th element
```

Some commonly used built-in functions:

pop	removes last element
push	adds new elements to the end of the array
shift	removes first element
unshift	adds new elements to the beginning of the array
splice	removes or adds elements from some position in the array
sort	sorts the elements of an array

Hash

An associative array, called a hash, is an unordered list of key/value pairs, indexed by strings. The name of the hash is preceded by a "%" symbol. (The % is not evaluated when enclosed in either single or double quotes.)

EXAMPLE

```
%employee =  (
   "Name"      => "Jessica Savage",
   "Phone"     => "(925) 555-1274",
   "Position"  => "CEO"
);

print "$employee{"Name"};  # Print a value
$employee{"SSN"}="999-333-2345";  # Assign a key/value
```

Some commonly used built-in functions:

keys	retrieves all the keys in a hash
values	retrieves all the values in a hash
each	retrieves a key/value pair from a hash
delete	removes a key/value pair

Continues

Table 2.1 Perl Syntax and Constructs (continued)

Predefined Variables

Perl provides a large number of predefined variables. The following is a list of some common predefined variables:

$_	The default input and pattern-searching space.
$.	Current line number for the last filehandle accessed.
$@	The Perl syntax error message from the last *eval()* operator.
$!	Yields the current value of the error message, used with *die*.
$0	Contains the name of the program being executed.
$$	The process number of the Perl running this script.
$PERL_VERSION / $^V	The revision, version, and subversion of the Perl interpreter.
@ARGV	Contains the command-line arguments.
ARGV	A special filehandle that iterates over command-line filenames in *@ARGV*.
@INC	Search path for libarary files.
@_	Within a subroutine the array @_ contains the parameters passed to that subroutine.
%ENV	The hash *%ENV* contains your current environment.
%SIG	The hash *%SIG* contains signal handlers for signals.

Constants (Literals)

A constant value, once set, cannot be modified. An example of a constant is PI or the number of feet in a mile. It doesn't change. Constants are defined with the *constant pragma* as shown here.

EXAMPLE

```
use constant BUFFER_SIZE => 4096;
use constant PI => 4 * atan2 1, 1;
use constant DEBUGGING => 0;
use contstant ISBN => "0-13-028251-0";
PI=6;  # Cannot modify PI; produces an error.
```

Numbers

Perl supports integers (decimal, octal, hexadecimal), floating point numbers, scientific notation, Booleans, and null.

EXAMPLE

```
$year = 2006;  # integer
$mode = 0775;  # octal number in base 8
$product_price = 29.95;  # floating point number in base 10
$favorite_color = 0x33CC99;  # integer in base 16 (hexadecimal)
$distance_to_moon=3.844e+5;  # floating point in scientific notation
$bits = 0b10110110;  # binary number
```

Table 2.1 Perl Syntax and Constructs (continued)

Strings and Quotes

A string is a sequence of bytes (characters) enclosed in quotes.
When quoting strings, make sure the quotes are matched; e.g., "string" or 'string'. Scalar and array variables ($x, @name) and backslash sequences (\n, \t, \", etc.) are interpreted within double quotes; a backslash will escape a quotation mark, a single quote can be embedded in a set of double quotes, and a double quote can be embedded in a set of single quotes. A *here document* is a block of text embedded between user-defined tags, the first tag preceded by <<.

The following shows three ways to quote a string:
Single quotes: 'It rains in Spain';
Double quotes: "It rains in Spain";
Here document:

```
print <<END;
    It
    rains in
    Spain
END
```

EXAMPLE

```
$question = 'He asked her if she wouldn\'t mind going to Spain';
       # Single quotes
$answer = 'She said: "No, but it rains in Spain."';  # Single quotes
$line = "\tHe said he wouldn't take her to Spain\n";
$temperature = "78";
print "It is currently $temperature degrees";
       # Prints: "It is currently 78 degrees.". Variables are
       # interpreted when enclosed  in double quotes, but not
       # single quotes
```

Alternative Quotes

Perl provides an alternative form of quoting. The string to be quoted is delimted by a nonalphanumeric character or characters that can be paired, such as ( ), { }, [ ].
The constructs are: **qq**, **q**, **qw**, **qx**

EXAMPLE

```
print qq/Hello\n/;   # same as: print "Hello\n";
print q/He owes $5.00/, \n"; #  same as: print 'He owes $5.00', "\n";
@states=qw( ME MT CA FL );  # same as ("ME", "MT", "CA","FL")
$today = qx(date);  # same as $today = `date`;
```

Continues

Table 2.1 Perl Syntax and Constructs (continued)

Operators

Perl offers many types of operators, but for the most part they are the same as C/C++/Java or PHP operators. Types of operators are (see Chapter 6):

Assignment	=, +=, -=, *= , %=, ^=, &=, |=, .=
Numeric equality	= =, !=, <=>
String equality	eq, ne, cmp
Relational numeric	> >= < <=
Relational string	gt, ge, lt, le
Range	5 .. 10 # range between 5 and 10, increment by 1
Logical	&&, and, ||, or, XOR, xor, !
Autoincrement/decrement	++ --
File	-r, -w, -x,-o, -e, -z, -s, -f, -d, -l, etc.
Bitwise	~ & | ^ << >>
String concatenation	.
String repetition	x
Arithmetic	* / - + %
Pattern matching	=~, !~

EXAMPLE

```
print "\nArithmetic Operators\n";
print ((3+2) * (5-3)/2);

print "\nString Operators\n";  # Concatenation
print "\tTommy" . ' ' . "Savage";

print "\nComparison Operators\n";
print 5>=3 , "\n";
print 47==23 , "\n";

print "\nLogical Operators\n";
$a > $b && $b < 100
$answer eq "yes" || $money == 200

print "\nCombined Assignment Operators\n";
$a = 47;
$a += 3;       # short for $a = $a + 3
$a++;          # autoincrement
print $a;  # Prints 51

print "\nPattern Matching Operators\n"
$color = "green";
print if $color =~ /^gr/;     # $color matches a pattern
                              # starting with 'gr'
$answer = "Yes";
print if $answer !~ /[Yy]/;  # $answer matches a pattern
                              # containing 'Y' or 'y'
```

Table 2.1 Perl Syntax and Constructs (continued)

Conditionals

The basic *if* construct evaluates an expression enclosed in parentheses, and if the condition evaluates to true, the block following the expression is executed.
(See Chapter 7.)

if statement

```
if ( expression ){
     statements;
}
```

EXAMPLE

```
if ( $a == $b ){ print "$a is equal to $b"; }
```

if/else statement

The *if/else* block is a two-way decision. If the expression after the *if* condition is true, the block of statements is executed; if false, the *else* block of statements is executed.

```
if ( expression ){
     statements;
else{
     statements;
}
```

EXAMPLE

```
$coin_toss = int (rand(2 )) + 1; # Generate a random
                                 # number between 1 and 2
if( $coin_toss == 1 ) {
   print "You tossed HEAD\n";
}
else {
   print "You tossed TAIL\n";
}
```

if/elsif statement

The *if/elsif/else* offers multiway branch; if the expression following the if is not true, each of the *elsif* expressions is evaluated until one is true; otherwise, the optional *else* statements are executed.

```
if ( expression ){
     statements;
elsif ( expression ){
     statements;
}
elsif (expression){
     statements;
else{
     statements;
}
```

Continues

Table 2.1 Perl Syntax and Constructs (continued)

EXAMPLE

```
#  1 is Monday, 7 Sunday
$day_of_week = int(rand(7)) + 1;
print "Today is: $day_of_week\n";
if ( $day_of_week >=1 && $day_of_week <=4 ) {
    print "Business hours are from 9 am to 9 pm\n";
}
elsif ( $day_of_week == 5) {
    print "Business hours are from 9 am to 6 pm\n";
}
else {
    print "We are closed on weekends\n";
}
```

Conditional Operator

Like C/C++, Perl also offers a shortform of the *if/else* syntax, which uses three operands and two operators (also called the ternary operator). The question mark is followed by a statement that is executed if the condition being tested is true, and the colon is followed by a statement that is executed if the condition is false.

(condition) ? statement_if_true : statement_if_false;

EXAMPLE

```
$coin_toss = int (rand(2 )) + 1; # Generate a random number
                                 # between 1 and 2
print ( $coin_toss == 1 ? "You tossed HEAD\n" : "You tossed TAIL\n" );
```

Loops

A loop is a way to specify a piece of code that repeats many times. Perl supports several types of loops: the *while* loop, *do-while* loop, *for* loop, and *foreach* loop. (See Chapter 7.)

while/until Loop

The while loop:
The *while* is followed by an expression enclosed in parentheses, and a block of statements. As long as the expression tests true, the loop continues to iterate.

while (conditional expression) {
    code block A
}

EXAMPLE

```
$count=0;  # Initial value
while ($count < 10 ){  # Test
    print $n;
    $count++;  # Increment value
}
```

Table 2.1 Perl Syntax and Constructs (continued)

The until loop:
The *until* is followed by an expression enclosed in parentheses, and a block of statements. As long as the expression tests false, the loop continues to iterate.

until (conditional expression) {
    code block A
}

EXAMPLE

```
$count=0;  # Initial value
until ($count == 10 ){  # Test
   print $n;
   $count++;  # Increment value
}
```

do-while Loop

The do-while loop:
The *do-while* loop is similar to the *while* loop except it checks its looping expresssion at the end of the loop block rather than at the beginning, guaranteeing that the loop block is executed at least once.

do {
    code block A
} while (expression);

EXAMPLE

```
$count=0;  # Initial value
do {
   print "$n ";
   $count++;  # Increment value
   while ($count < 10 );  # Test
}
```

for Loop

The for loop:
The *for* loop has three expressions to evaluate, each separated by a semicolon. The first inititializes a variable and is evaluated only once. The second tests whether the value is true, and if it is true, the block is entered; if not, the loop exits. After the block of statements is executed, control returns to the third expression, which changes the value of the variable being tested. The second expression is tested again, etc.

for(initialization; conditional expression; increment/decrement) {
    block of code
}

Continues

Table 2.1 Perl Syntax and Constructs (continued)

EXAMPLE

```
for( $count = 0; $count < 10; $count = $count + 1 ) {
    print "$count\n";
}
```

foreach Loop

The **foreach loop**:
The *foreach* is used only to iterate through a list, one item at a time.

```
foreach  $item ( @list ) {
     print $item,"\n";
}
```

EXAMPLE

```
@dessert =  ( "ice cream", "cake", "pudding", "fruit");

foreach $choice (@dessert){
    # Iterates through each element of the array
    echo "Dessert choice is: $choice\n";
}
```

Loop Control

The *last* statement is used to break out of a loop from within the loop block. The *next* statement is used to skip over the remaining statements within the loop block and start back at the top of the loop.

EXAMPLE

```
$n=0;
while( $n < 10 ){
    print $n;
    if ($n == 3){
        last;  # Break out of loop
    }
    $n++;
}
print "Out of the loop.<br>";
```

EXAMPLE

```
for($n=0; $n<10; $n++){
    if ($n == 3){
        next;  # Start at top of loop;
               # skip remaining statements in block
    }
    echo "\$n = $n<br>";
}
print "Out of the loop.<br>";
```

Table 2.1 Perl Syntax and Constructs (continued)

Subroutines/ Functions

A function is a block of code that peforms a task and can be invoked from another part of the program. Data can be passed to the function via arguments. A function may or may not return a value. Any valid Perl code can make up the definition block of a function. Variables outside the function are available inside the function. The *my* function will make the specified variables local.
(See Chapter 11.)

sub function_name{
block of code
}

EXAMPLE

```
sub greetings() {
   print "Welcome to Perl!<br>";  # Function definition
}
&greetings;  # Function call
greetings(); # Function call
```

EXAMPLE

```
$my_year = 2000;

if ( is_leap_year( $my_year ) ) {   # Call function with an argument
   print "$my_year is a leap year\n";
}
else {
   print "$my_year is not a leap year";
}

sub is_leap_year {   # Function definition

   my $year = shift(@_);  # Shift off the year from
                          # the parameter list, @_
   return (((($year % 4 == 0) && ($year % 100 != 0)) ||
   ($year % 400 == 0)) ? 1 : 0;   # What is returned from the function
}
```

Continues

Table 2.1 Perl Syntax and Constructs (continued)

Files

Perl provides the *open* function to open files, and pipes for reading, writing, and appending. The *open* function takes a user-defined filehandle (normally a few uppercase characters) as its first argument and a string containing the symbol for read/write/append followed by the real path to the system file. (See Chapter 10.)

EXAMPLE

To open a file for reading:

```
open(FH, "<filename");      # Opens "filename" for reading.
                            # The < symbol is optional.
open (DB, "/home/ellie/myfile") or die "Can't open file: $!\n";
```

To open a file for writing:

```
open(FH, ">filename");      # Opens "filename" for writing.
                             # Creates or truncates file.
```

To open a file for appending:

```
open(FH, ">>filename");     # Opens "filename" for appending.
                            # Creates or appends to file.
```

To open a file for reading and writing:

```
open(FH, "+<filename");     # Opens "filename" for read, then write.
open(FH, "+>filename");     # Opens "filename" for write, then read.
```

To close a file:

```
close(FH);
```

To read from a file:

```
while(<FH>){ print; }       # Read one line at a time from file.
@lines = <FH>;              # Slurp all lines into an array.
print "@lines\n";
```

To write to a file:

```
open(FH, ">file") or die "Can't open file: $!\n";

print FH "This line is written to the file just opened.\n";
print FH "And this line is also written to the file just opened.\n";
```

EXAMPLE

To Test File Attributes

```
print "File is readable, writeable, and executable\n" if -r $file and
-w _ and -x _;
            # Is it readble, writeable, and executable?
print "File was last modified ",-M $file, " days ago.\n";
            # When was it last modified?
print "File is a directory.\n " if -d $file;    # Is it a directory?
```

Table 2.1 Perl Syntax and Constructs (continued)

Pipes	Pipes can be used to send the output from system commands as input to Perl and to send Perl's output as input to a system command. To create a pipe, also called a filter, the *open* system call is used. It takes two arguments: a user-defined handle and the operating system command, either preceded or appended with the "|" symbol. If the command is preceded with a "|", the operating system command reads Perl output. If the command is appended with the "|" symbol, Perl reads from the pipe; if the command is prepended with "|", Perl writes to the pipe.
	(See Chapter 10.)
	
	EXAMPLE
	
	Input filter
	`open(F, " ls |") or die;  # Open a pipe to read from`
	`while(<F>){ print ; }  # Prints list of UNIX files`
	
	Output filer
	`open(SORT, "| sort" ) or die;  # Open pipe to write to`
	`print SORT "dogs\ncats\nbirds\n"`
	`  # Sorts birds, cats, dogs on separate lines.`

Regular Expressions. A regular expression is set of characters enclosed in forward slashes. They are to match patterns in text and to refine searches and substitutions. Perl is best known for its pattern matching. (See Chapter 8.)

Table 2.2 Some Regular Expression Metacharacters

Metacharacter	*What It Represents*
^	Matches at the beginning of a line
$	Matches at the end of a line
a.c	Matches an 'a', any single character, and a 'c'
[abc]	Matches an 'a' or 'b' or 'c'
[^abc]	Matches a character that is not an 'a' or 'b' or 'c'
[0-9]	Matches one digit between '0' and '9'
ab*c	Matches an 'a', followed by zero or more 'b's and a 'c'
ab+c	Matches an 'a', followed by one or more 'b's and a 'c'
ab?c	Matches an 'a', followed by zero or one 'b' and a 'c'
(ab)+c	Matches one or more occurrences of 'ab' followed by a 'c'
(ab) (c)	Captures 'ab' and assigns it to $1, captures 'c' and assigns it to $2.

EXAMPLE

```
$_ = "looking for a needle in a haystack";
print if /needle/;
     If $_contains needle, the string is printed.

$_ = "looking for a needle in a haystack"; # Using regular expression metacharacters
print if  /^[Nn]..dle/;
     # characters and "dle".

$str = "I am feeling blue, blue, blue..."
$str =~ s/blue/upbeat/;  # Substitute first occurrence of "blue" with "upbeat"
print $str;
I am feeling upbeat, blue, blue...

$str="I am feeling BLue, BLUE...";
$str = ~ s/blue/upbeat/ig;  # Ignore case, global substitution
print $str;
I am feeling upbeat, upbeat...

$str = "Peace and War";
$str =~ s/(Peace) and (War)/$2, $1/i;  # $1 gets 'Peace', $2 gets' War'
print $str;
War and Peace.

$str = "He gave me 5 dollars.\n"
s/5/6*7/e; #  Rather than string substitution, evaluate replacement side
print $str;
He gave me 42 dollars."
```

Passing Arguments at the Command Line. The *@ARGV* array is used to hold command-line arguments. If the ARGV filehandle is used, the arguments are treated as files; otherwise, arugments are strings coming in from the command line to be used in a script. (See Chapter 10.)

EXAMPLE

```
$ perlscript filea fileb filec

(In Script)
print "@ARGV\n";  # lists arguments: filea fileb filec
while(<ARGV>){  # filehandle ARGV -- arguments treated as files
   print;  # Print each line of every file listed in @ARGV
}
```

References, Pointers. Perl references are also called pointers. A pointer is a scalar variable that contains the address of another variable. To create a pointer, the backslash operator is used. (See Chapter 13.)

EXAMPLE

```
# Create variables
$age = 25;
@siblings = qw("Nick", "Chet", "Susan","Dolly");
%home = ("owner" => "Bank of America",
         "price" => "negotiable",
         "style" => "Saltbox",
);

# Create pointer
$pointer1 = \$age;   # Create pointer to scalar
$pointer2 = \@siblings;  # Create pointer to array
$pointer3 = \%home; # Create pointer to hash
$pointer4 = [ qw(red yellow blue green) ]; # Create anonymous array
$pointer5 = { "Me" => "Maine", "Mt" => "Montana", "Fl" => "Florida" };
           # Create anonymous hash

# Dereference pointer
print $$pointer1; # Dereference pointer to scalar; prints: 25
print @$pointer2; # Dereference pointer to array;
        # prints: Nick Chet Susan Dolly
print %$pointer3; # Dereference pointer to hash;
        # prints: styleSaltboxpricenegotiableownerBank of America
print $pointer2->[1];   # prints "Chet"
print $pointer3->{"style"}; # prints "Saltbox"
print @{$pointer4}; # prints elements of anonymous array
```

Objects. Perl supports objects, a special type of variable. A Perl class is a package containing a collection of variables and functions, called properties and methods. There is no "class" keyword. The properties (also called attributes) are variables used to describe the object. Methods are special functions that allow you to create and manipulate the object. Objects are created with the *bless* function. (See Chapter 14.)

Creating a Class

EXAMPLE

```
package Pet

sub new{ # Constructor
    my $class = shift;
    my $pet = {
         "Name"  => undef,
         "Owner" => undef,
         "Type"  => undef,
    };
    bless($pet, $class);
# Returns a pointer to the object

    sub set_pet{    # Accessor methods
         my $self = shift;
         my ($name, $owner, $type)= @_;
         $self->{'Name'} = $name;
         $self->{'Owner'}= $owner;
         $self->{'Type'}= $type;
    }
    sub get_pet{
    my $self = shift;
    while(($key,$value)=each($%self)){
         print "$key: $value\n";
    }
}
```

Instantiating a Class

EXAMPLE

```
$cat = Pet->new();  # alternative form is: $cat = new Pet();
# Create an object with a constructor method
$cat->set_pet("Sneaky", "Mr. Jones", "Siamese");
# Access the object with an instance
$cat->get_pet;
```

Perl also supports method inheritance by placing base classes in the *@ISA* array.

Libraries and Modules. Library files have a *.pl* extenson; modules have a .pm extension. Today, *.pm* files are more commonly used than *.pl* files. (See Chapter 12.)

Path to Libraries

@INC array contains list of path to standard Perl libraries.

To include a File

To load an external file, use either *require* or *use*.

```
require("getopts.pl"); # Loads library file at run time
use CGI;  # Loads CGI.pm module at compile time
```

Diagnostics. To exit a Perl script with the cause of the error, you can use the built-in *die* function or the *exit* function.

EXAMPLE

```
open(FH, "filename") or die "Couldn't open filename: $!\n";
if ($input !~ /^\d+$/){
       print STDERR "Bad input. Integer required.\n";
       exit(1);
}
```

You can also use the Perl pragmas:
use *warnings*; *# Provides warning messages; does not abort program*
use *diagnostics*; *# Provides detailed warnings; does not abort program*
use *strict*; *# Checks for global variable, unquoted words, etc.; aborts program*
use *Carp*; *# Like the* **die** *function with more information about program's errors*

2.2 Chapter Summary

This chapter was provided for programmers who need a quick peek at what Perl looks like, its general syntax, and programming constructs. It is an overview. There is a lot more to Perl as you'll see as you read through the following chapters.

Later, after you have programmed for awhile, this chapter can also serve as a little tutorial to refresh your memory without having to search through the index to find what you are looking for.

2.3 What's Next?

In Chapter 3, we will discuss Perl script setup; i.e., how to name a script, execute it, and add comments, statements, and built-in functions. We will also see how to use Perl command-line switches and how to identify certain types of errors.

This page intentionally left blank

chapter

3

Perl Scripts

3.1 Script Setup

A Simple Perl Script

EXAMPLE 3.1

```
(The Script)
#!/usr/bin/perl
print "What is your name? ";
chomp($name = <STDIN>);  # Program waits for user input from keyboard
print "Welcome, $name, are you ready to learn Perl now? ";
chomp($response = <STDIN>);
$response=lc($response);  # response is converted to lowercase
if($response eq "yes" or $response eq "y"){
     print "Great! Let's get started learning Perl by example.\n";
}
else{
   print "O.K. Try again later.\n";
}
$now = localtime;  # Use a Perl function to get the date and time
print "$name, you ran this script on $now.\n";

(Output)
What is your name? Ellie
Welcome, Ellie, are you ready to learn Perl now? yes
Great! Let's get started learning Perl by example.
Ellie, you ran this script on Wed Apr  4 21:53:21 2007.
```

Example 3.1 is an example of a Perl script. In no time, you will be able to write a similar script. Perl scripts consist of a list of Perl statements and declarations. Statements are terminated with a semicolon (;). (Since only subroutines and report formats require declarations, they will be discussed when those topics are presented.) Variables can be created

anywhere in the script and, if not initialized, automatically get a value of 0 or "null," depending on their context. Notice that the variables in this program start with a $. Values such as numbers, strings of text, or the output of functions can be assigned to variables. Different types of variables are preceded by different "funny symbols," as you'll see in Chapter 4.

Perl executes each statement just once, starting from the first to the last line.

3.2 The Script

3.2.1 Startup

UNIX/Mac OS. If the **first line** of the script contains the *#!* symbols (called the *shbang* line) followed by the full pathname of the file where your version of the Perl executable resides, this tells the kernel what program is interpreting the script. An example of the startup line might be

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

It is extremely important that the path to the interpreter is entered correctly after the *shbang* (*#!*). Perl may be installed in different directories on different systems. Most Web servers will look for this line when invoking CGI scripts written in Perl. Any inconsistency will cause a fatal error. To find the path to the Perl interpreter on your system, type at your UNIX prompt[1]:

```
which perl
```

If the *shbang* line is the first line of the script, you can execute the script directly from the command line by its name. If the *shbang* **is not the first line** of the script, the UNIX shell will try to interpret the program as a shell script, and the *shbang* line will be interpreted as a comment line. (See "Executing the Script" on page 36 for more on how to execute Perl programs.)

Mac OS is really just a version of UNIX and comes bundled with Perl 5.8. You open a terminal and use Perl exactly the same way you would use it for Solaris, Linux, *BSD, HP-UX, AIX OSX, etc.

Windows. Win32 platforms don't provide the *shbang* syntax or anything like it.[2] For Windows XP and Windows NT 4.0[3] you can associate a Perl script with extensions such as *.pl* or *.plx* and then run your script directly from the command line. At the command-line prompt or from the system control panel, you can set the *PATHEXT* environment

1. Another way to find the interpreter would be: *find / -name '*perl*' -print;*
2. Although Win32 platforms don't ordinarily require the *shbang* line, the Apache Web server does, so you will need the *shbang* line if you are writing CGI scripts that will be executed by Apache.
3. File association does not work on Windows 95 unless the program is started from the Explorer window.

variable to the name of the extension that will be associated with Perl scripts. At the command line, to set the environment variable, type

```
SET PATHEXT=.pl;%PATHEXT%
```

At the control panel, to make the association permanent, do the following:

1. Go to the Start menu.
2. Select Settings or just select Control Panel.
3. Select Control Panel.
4. In the control panel, click on the System icon.
5. Click on Advanced.
6. Click on Environment Variables.
7. Click on New.
8. Type *PATHEXT* in the Variable Name box.
9. In the Variable Value box, type the extension you want, followed by a semicolon and *%PATHEXT%*.
10. OK the setting.

From now on when you create a Perl script, append its name with the extension you have chosen, such as *myscript.pl* or *myscript.plx*. Then the script can be executed directly at the command line by just typing the script name without the extension, e.g., *myscript.pt*. (See "Executing the Script" on page 36 for more on script execution.)

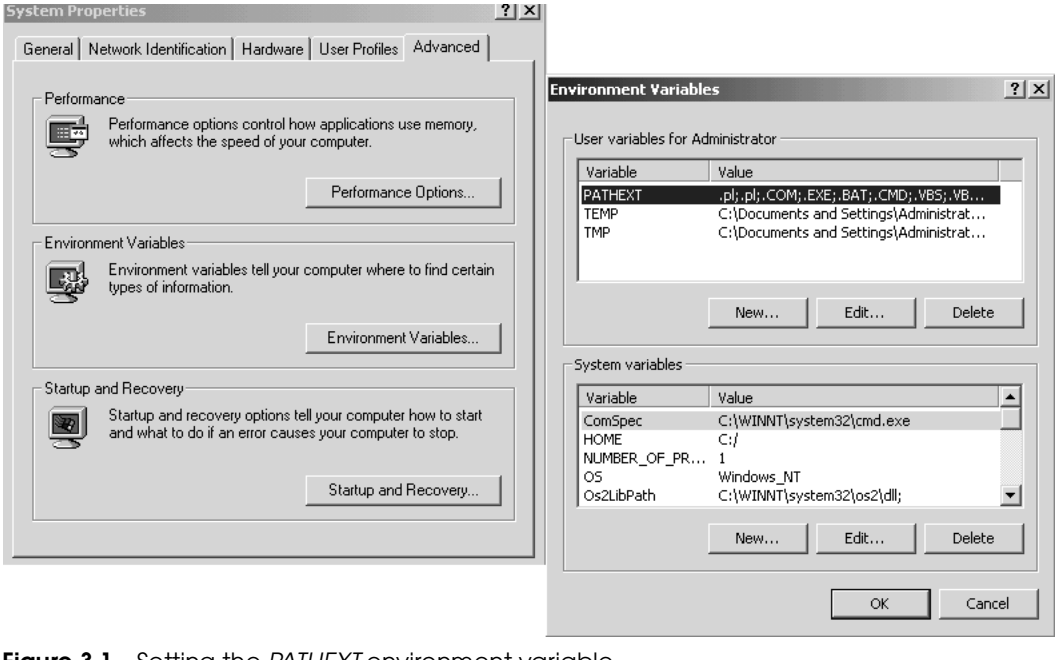


Figure 3.1 Setting the *PATHEXT* environment variable.

3.2.2 Finding a Text Editor

Since you will be using a text editor to write Perl scripts, you can use any of the editors provided by your operating system or download more sophisticated editors specifically designed for Perl, including third-party editors and Integrated Development Environments (IDEs). Table 3.1 lists some of the editors available.

Table 3.1 Types of Editors

BBEdit, JEdit	Macintosh
Wordpad, Notepad, UltraEdit, vim, PerlEdit, JEdit, TextPad	Windows
pico, vi, emacs, PerlEdit, JEdit	Linux/UNIX
Komodo	Linux, Mac OS, Windows
OptiPerl, PerlExpress	Windows
Affus	Mac OS X

3.2.3 Naming Perl Scripts

The only naming convention for a Perl script is that it follow the naming conventions for files on your operating system (upper-/lowercase letters, numbers, etc.). If, for example, you are using Linux, filenames are case sensitive, and since there are a great number of system commands, you may want to add an extension to your Perl script names to make sure the names are unique. You are not required to add an extension to the filename unless you are creating libraries or modules, writing CGI scripts if the server requires a specific extension, or have set up Windows to expect an extension on certain types of files. By adding a unique extension to the name, you can prevent clashes with other programs that might have the same name. For example, UNIX provides a command called "test". If you name a script "test", which version will be executed? If you're not sure, you can add a *.plx* or *.perl* extension to the end of the Perl script name to give it its own identity.

And of course, give your scripts sensible names that indicate the purpose of the script rather than names like "foo", "foobar", or "testing".

3.2.4 Statements, Whitespace, and Linebreaks

Perl is called a free-form language, meaning you can place statements anywhere on the line and even cross over lines. Whitespace refers to spaces, tabs, and newlines. The newline is represented as "\n" and must be enclosed in double quotes. Whitespace is used to delimit words. Any number of blank spaces are allowed between symbols and words. Whitespace enclosed in single or double quotes is preserved; otherwise, it is ignored. The following expressions are the same:

5+4*2 is the same as 5 + 4 * 2;

And both of the following Perl statements are correct even though the output will show that the whitespace is preserved when quoted.

```
print "This is a Perl statement.";

   print "This
              is
                also
                  a Perl
                      statement.";
```

Even though you have a lot of freedom when writing Perl scripts, it is better to put statements on their own line and to provide indentation when using blocks of statements (we'll discuss this in Chapter 5). Of course, annotating your program with comments, so that you and others will understand what is going on, is vitally important. See the next section for more on comments.

3.2.5 Comments

You may write a very clever Perl script today and in two weeks have no idea what your script was trying to do. If you pass the script on to someone else, the confusion magnifies. Comments are plain text that allow you to insert documentation in your Perl script with no effect on the execution of the program. They are used to help you and other programmers maintain and debug scripts. Perl comments are preceded by a # mark. They extend across the line, but do not continue onto the next line.

Perl does **not** understand the *C* language comments /* and */ or *C++* comments //.

EXAMPLE 3.2

```
1   # This is a comment
2   print "hello";  # And this is a comment
```

EXPLANATION

1 Comments, as in UNIX shell, *sed*, and *awk* scripts, are lines preceded with the pound sign (#) and can continue to the end of the line.

2 Comments can be anywhere on the line. Here the comment follows a valid Perl *print* statement.

3.2.6 Perl Statements

Perl executable statements make up most of the Perl script. As in *C*, the statement is an expression, or series of expressions, terminated with a semicolon. Perl statements can

be simple or compound, and a variety of operators, modifiers, expressions, and functions make up a statement, as shown in the following example.

```
print "Hello, to you!\n";
$now = localtime();
print "Today is $now.\n";
$result = 5 * 4 / 2;
print "Good-bye.\n";
```

3.2.7 Using Perl Built-in Functions

A big part of any programming language is the set of functions built into the language or packaged in special libraries (see Apendix A.1). Perl comes with many useful functions, independent program code that performs some task. When you call a Perl built-in function, you just type its name, or optionally you can type its name followed by a set of parentheses. All function names must be typed in lowercase. Many functions require arguments, messages that you send to the function. For example, the *print* function won't display anything if you don't pass it an argument, the string of text you want to print on the screen. If the function requires arguments, then place the arguments, separated by commas, right after the function name. The function usually returns something after it has performed its particular task. In the script shown at the beginning of this chapter, we called two built-in Perl functions, *print* and *localtime*. The *print* function took a string as its argument and displayed the string of text on the screen. The *localtime* function, on the other hand, didn't require an argument but returned the current date and time. Both of the following statements are valid ways to call a function with an argument. The argument is "Hello, there.\n"

```
print("Hello, there.\n");
print "Hello, there.\n";
```

3.2.8 Executing the Script

A Perl script can be executed at the command line directly by its name when the *#!* startup line is included in the script file and the script has execute permission (see Example 3.3) or, if using Windows, filename association has been set as discussed in "Startup" on page 32. If the *#!* is **not** the first line of the script, you can execute a script by passing the script as an argument to the Perl interpreter.
Perl will then compile and run your script using its own internal form. If you have syntax errors, Perl will let you know. You can check to see if your script has compiled successfully by using the *-c* switch as follows:

```
$ perl -c scriptname
```

To execute a script at either the UNIX or MS-DOS prompt, type

```
$ perl scriptname
```

3.2.9 Sample Script

The following example illustrates the five parts of a Perl script:

1. The startup line (UNIX)
2. Comments
3. The executable statements in the body of the script
4. Checking Perl syntax
5. The execution of the script (UNIX, Windows)

EXAMPLE 3.3

```
    $ cat first.perl       (UNIX display contents)
1   #!/usr/bin/perl
2   # My first Perl script

3   print "Hello to you and yours!\n";

4   $ perl -c first.perl   # The $ is the shell prompt
    first.perl syntax OK

5   $ chmod +x first.perl  (UNIX)

6   $ first.perl or ./first.perl
7   Hello to you and yours!
```

EXPLANATION

1 The startup line tells the shell where Perl is located.

2 A comment describes information the programmer wants to convey about the script.

3 An executable statement includes the *print* function.

4 The *-c* switch is used to check for syntax errors. Hopefully, everything is "OK."

5 The *chmod* command turns on execute permission.

6 The script is executed (as long as your UNIX path includes the "." directory). If you get "Command not found" (or a similar message), precede the script name with a dot and a forward slash.

7 The string *Hello to you and yours!* is printed on the screen.

EXAMPLE 3.4

```
    $ type first.perl      (MS-DOS display contents)
1   # No startup line; This is a comment.
2   # My first Perl script
3   print "Hello to you and yours!\n";
```

EXAMPLE 3.4 (CONTINUED)

```
4   $ perl first.perl  (Both UNIX and Windows)
5   Hello to you and yours!
```

EXPLANATION

1 The startup line with *#!* is absent. It is not necessary when using Windows. If using ActiveState, you create a batch file with a utility called *pl2bat*.

2 This is a descriptive line; a comment explains that the startup line is missing.

3 An executable statement includes the *print* function.

4 At the command line, the Perl program takes the script name as an argument and executes the script. The script's output is printed. You can execute a Perl script this way with any operating system.

3.2.10 What Kinds of Errors to Expect

Expect to make errors and maybe lots of them. You may try many times before you actually get a program to run perfectly. Knowing your error messages is like knowing the quirks of your boss, mate, or even yourself. Some programmers make the same error over and over again. Don't worry. In time, you will learn what most of these messages mean and how to prevent them.

When you execute a Perl script, it takes just one step on your part, but internally the Perl interpreter takes two steps. First, it compiles the entire program into bytecode, an internal representation of the program. After that, Perl's bytecode engine runs the bytecode line by line. If you have compiler errors, such as a missing semicolon at the end of the line, misspelled keyword, or mismatched quotes, you will get what is called a syntax error. These types of errors are picked up by using the *-c* switch and are usually easy to find once you have become acquainted with them.

EXAMPLE 3.5

```
    (The Script)
   print "Hello, world";
1  print "How are you doing?
2  print "Have you found any problems in this script?";

(Output)
Bareword found where operator expected at errors.plx line 3, near
"print "Have"
  (Might be a runaway multi-line "" string starting on line 2)
        (Do you need to predeclare print?)
syntax error at errors.plx line 3, near "print "Have you "
Search pattern not terminated at errors.plx line 3.
```

EXPLANATION

1 This line should have a closing double quote and a terminating semicolon.

2 This Perl statement is correct, but Perl is still looking for the closing quote on the previous line and is confused by the word "print" on this line because this line is still part of the last line. Why? Because the previous line is missing a double quote and was not terminated with a semicolon. Whenever you see the word "runaway" in the error message, it usually means a quote that has "run away"; i.e., missing. If you see "Bareword," it means that a word has no quotes surrounding it.

After the program passes the compile phase (i.e., you don't get any syntax errors or complaints from the compiler), then you may get what are called runtime, or logical, errors. These errors are harder to find and are probably caused by not anticipating problems that might occur when the program starts running. Or it's possible that the program has faulty logic in the way it was designed. Runtime errors may be caused if a file or database you're trying to open doesn't exist, a user enters bad input, you get into an infinite loop, or you try to illegally divide by zero.Whatever the case, these problems, called "bugs," are harder to find. Perl comes with a debugger that is helpful in determining what caused these logical errors by letting you step through your program line by line. (See "Debugger" on page 858.)

3.3 Perl at the Command Line

Although most of your work with Perl will be done in scripts, Perl can also be executed at the command line for simple tasks, such as testing a function, a print statement, or simply testing Perl syntax. Perl has a number of command-line switches, also called command-line options, to control or modify its behavior. The switches discussed next are not a complete list (see Appendix A) but will demonstrate a little about Perl syntax at the command line.

When working at the command line, you will see a shell prompt. The shell is called a "command interpreter." UNIX shells such as *Korn* and *Bourne* display a default $ prompt, and *C* shell displays a % prompt. The UNIX, Linux (*bash* and *tcsh*), Mac OS shells are quite similar in how they parse the command line. By default, if you are using Windows XP or Vista, the MS-DOS shell is called *command.com,* and if you are using Windows NT, the command shell is a console application residing in *cmd.exe*. It too displays a $ prompt.[4] The Win32 shell has its own way of parsing the command line. Since most of your Perl programming will be done in script files, you will seldom need to worry about the shell's interaction, but when a script interfaces with the operating system, problems will occur unless you are aware of what commands you have and how the shell executes them on your behalf.

4. It is possible that your command-line prompt has been customized to contain the current directory, history number, drive number, etc.

3.3.1 The *-e* Switch

The *-e* switch allows Perl to **execute** Perl statements at the command line instead of from a script. This is a good way to test simple Perl statements before putting them into a script file.

EXAMPLE 3.6

```
1   $ perl -e 'print "hello dolly\n";'      # UNIX/Linux
    hello dolly
2   $ perl -e "print qq/hello dolly\n/;"    # Windows and UNIX/Linux
    hello dolly
```

EXPLANATION

1 Perl prints the string *hello dolly* to the screen followed by a newline *\n*. The dollar sign ($) is the UNIX shell prompt. The single quotes surrounding the Perl statement protect it from the UNIX shell when it scans and interprets the command line. This will fail to execute on a Windows system.

2 At the MS-DOS prompt, Perl statements must be enclosed in double quotes. The *qq* construct surrounding *hello dolly* is another way Perl represents double quotes. For example, *qq/hello/* is the same as *"hello"*. An error is displayed if you type the following at the MS-DOS prompt:

```
$ perl -e 'print "hello dolly\n";'
Can't find string terminator "" anywhere before EOF at -e line 1.
```

Note: UNIX systems can use this format as well.

3.3.2 The *-n* Switch

If you need to print the contents of a file or search for a line that contains a particular pattern, the *-n* switch is used to implicitly loop through the file one line at a time. Like *sed* and *awk*, Perl uses powerful pattern-matching techniques for finding patterns in text. Only specified lines from the file are printed when Perl is invoked with the *-n* switch.

Reading from a File. The *-n* switch allows you to loop through a file whose name is provided at the command line. The Perl statements are enclosed in quotes, and the file or files are listed at the end of the command line.

EXAMPLE 3.7

```
(The Text File)
1   $ more emp.first
    Igor Chevsky:6/23/83:W:59870:25:35500:2005.50
    Nancy Conrad:6/18/88:SE:23556:5:15000:2500
    Jon DeLoar:3/28/85:SW:39673:13:22500:12345.75
    Archie Main:7/25/90:SW:39673:21:34500:34500.50
    Betty Bumble:11/3/89:NE:04530:17:18200:1200.75
```

EXAMPLE (CONTINUED) 3.7 (CONTINUED)

```
2   $ perl -ne 'print;' emp.first     # Windows: use double quotes
    Igor Chevsky:6/23/83:W:59870:25:35500:2005.50
    Nancy Conrad:6/18/88:SE:23556:5:15000:2500
    Jon DeLoar:3/28/85:SW:39673:13:22500:12345.75
    Archie Main:7/25/90:SW:39673:21:34500:34500.50
    Betty Bumble:11/3/89:NE:04530:17:18200:1200.75

3   $ perl -ne 'print if /^Igor/;' emp.first
    Igor Chevsky:6/23/83:W:59870:25:35500:2005.50
```

EXPLANATION

1 The text file *emp.first* is printed to the screen. Perl will use this filename as a command-line argument in line 2.

2 Perl prints all the lines in the file *emp.first* by implicitly looping through the file one line at a time. (Windows users should enclose the statement in double quotes instead of single quotes.)

3 Perl uses **regular expression** metacharacters to specify what patterns will be matched. The pattern *Igor* is placed within forward slashes and preceded by a caret (^). The caret is called a "beginning of line anchor." Perl prints only lines beginning with the pattern *Igor*. (Windows users should enclose the statement in double quotes instead of single quotes.)

Reading from a Pipe. Since Perl is just another program, the output of commands can be piped to Perl, and Perl output can be piped to other commands. Perl will use what comes from the pipe as input, rather than a file. The *-n* switch is needed so Perl can read the input coming in from the pipe.

EXAMPLE 3.8

```
(UNIX)
1   $ date | perl -ne 'print "Today is $_";'
2   Today is Mon Mar 12 20:01:58 PDT 2007


(Windows)
3   $ date /T | perl -ne "print qq/Today is $_/;"
4   Today is Tue 04/24/2007
```

EXPLANATION

1 The output of the UNIX *date* command is piped to Perl and stored in the $_ variable. The quoted string *Today is* and the contents of the $_ variable will be printed to the screen followed by a newline.

EXAMPLE 3.8 (CONTINUED)

2 The output illustrates that today's date was stored in the $_ variable.
3 The Windows *date* command takes */T* as an option that produces today's date. That ouput is piped to Perl and stored in the $_ variable. The double quotes are required around the print statement.

Perl can take its input from a file and send its output to a file using standard I/O redirection.

EXAMPLE 3.9

```
1   $ perl -ne 'print;' < emp.first
    Igor Chevsky:6/23/83:W:59870:25:35500:2005.50
    Nancy Conrad:6/18/88:SE:23556:5:15000:2500
    Jon DeLoar:3/28/85:SW:39673:13:22500:12345.75
    Archie Main:7/25/90:SW:39673:21:34500:34500.50
    Betty Bumble:11/3/89:NE:04530:17:18200:1200.75
2   $ perl -ne 'print' emp.first > emp.temp
```

EXPLANATION

1 Perl's input is taken from a file called *emp.first*. The output is sent to the screen. For Windows users, enclose the statement in double quotes instead of single quotes.
2 Perl's input is taken from a file called *emp.first*, and its output is sent to the file *emp.temp*. For Windows users, enclose the statement in double quotes instead of single quotes.

3.3.3 The *-c* Switch

As we demonstrated earlier in this chapter, the *-c* switch is used to check the Perl syntax without actually executing the Perl commands. If the syntax is correct, Perl will tell you so. It is a good idea to always check scripts with the *-c* switch. This is especially important with CGI scripts written in Perl, because error messages that are normally sent to the terminal screen are sent to a log file instead. (See also the *-w* switch in Chapter 4.)

EXAMPLE 3.10

```
1   print "hello';   Search pattern not terminated at  line 1.
 Can't find string terminator '"' anywhere before EOF at test.plx
2   print "hello";
    test.plx syntax OK
```

EXPLANATION

1 The string *hello* starts with a double quote but ends with a single quote. The quotes should be matched; i.e., the first double quote should be matched at the end of the string with another double quote but instead ends with a single quote. With the *-c* switch, Perl will complain if it finds syntax errors while compiling.

2 After correcting the previous problem, Perl lets you know that the syntax is correct.

3.4 What You Should Know

1. How do you set up a script?
2. How are statements terminated?
3. What is whitespace?
4. What is meant by free form?
5. What is a built-in function?
6. What is the #! line in UNIX?
7. How do you make the script executable?
8. Why use comments?
9. How do you execute a Perl script if not using the *shbang* line.
10. What comand-line option lets you check Perl syntax?
11. What is the *-e* switch for?

3.5 What's Next?

If you can't print what your program is supposed to do, it's like trying to read the mind of a person who can't speak. In the next chapter, we discuss Perl functions to print output to the screen (*stdout*) and how to format the output. You will learn how Perl views words, whitespace, literals, backslash sequences, numbers, and strings. You will learn how to use single, double, and backquotes and their alternative form. We will discuss *here documents* and how to use them in CGI scripts. You will also learn how to use warnings and diagnostics to help debug your scripts.

EXERCISE 3
Getting with It Syntactically

1. At the command-line prompt, write a Perl statement that will print

   Hello world!!
   Welcome to Perl programming.

2. Execute another Perl command that will print the contents of the *datebook* file. (The file is found on the accompanying CD.)

3. Execute a Perl command that will display the version of the Perl distribution you are currently using.

4. Copy the program sample in Example 3.1 into your editor, save it, check the syntax, and execute it.

chapter

4

Getting a Handle on Printing

4.1 The Filehandle

By convention, whenever your program starts execution, the parent process (normally a shell program) opens three predefined streams called *stdin*, *stdout,* and *stderr*. All three of these streams are connected to your terminal by default.

stdin is the place where input comes from, the terminal keyboard; *stdout* is where output normally goes, the screen; and *stderr* is where errors from your program are printed, also the screen.

Perl inherits *stdin*, *stdout*, and *stderr* from the shell. Perl does not access these streams directly but gives them names called *filehandles*. Perl accesses the streams via the filehandle. The filehandle for *stdin* is called *STDIN*; the filehandle for *stdout* is called *STDOUT*; and the filehandle for *stderr* is called *STDERR*. Later, we'll see how you can create your own filehandles, but for now we'll stick with the predefined ones.

The *print* and *printf* functions by default send their output to the *STDOUT* filehandle, your screen.

4.2 Words

When printing a list of words to *STDOUT*, it is helpful to understand how Perl views a word. Any unquoted word must start with an alphanumeric character. It can consist of other alphanumeric characters and an underscore. Perl words are case sensitive. If a word is unquoted, it could conflict with words used to identify filehandles, labels, and other reserved words. If you see the error "Bareword," it means that the word has not been surrounded by quotes. If the word has no special meaning to Perl, it will be treated as if surrounded by single quotes.

4.3 The *print* Function

The *print* function prints a string or a list of comma-separated words to the Perl filehandle *STDOUT*. If successful, the *print* function returns 1; if not, it returns 0.

The string literal \n adds a newline to the end of the string. It can be embedded in the string or treated as a separate string. To interpret backslashes, Perl requires that escape sequences like \n be enclosed in double quotes.

EXAMPLE 4.1

```
(The Script)
1   print "Hello", "world", "\n";
2   print "Hello world\n";

(Output)
1   Helloworld
2   Hello world
```

EXPLANATION

1 Each string passed to the *print* function is enclosed in double quotes and separated by a comma. To print whitespace, the whitespace must be enclosed within the quotes. The \n escape sequence must be enclosed in double quotes for it to be interpreted as a newline character.

2 The entire string is enclosed in double quotes and printed to standard output.

EXAMPLE 4.2

```
(The Script)
1   print Hello, world, "\n";

(Output)
1   No comma allowed after filehandle at ./perl.st line 1
```

EXPLANATION

1 If the strings are not quoted, the filehandle *STDOUT* must be specified, or the *print* function will treat the first word it encounters as a filehandle (i.e., the word *Hello* would be treated as a filehandle). The comma is not allowed after a filehandle; it is used only to separate strings that are to be printed.

EXAMPLE 4.3

```
(The Script)
1   print STDOUT Hello, world, "\n";

(Output)
1   Helloworld
```

EXPLANATION

1 The filehandle *STDOUT* must be specified if strings are not quoted. The \n must be double quoted if it is to be interpreted. It is not a good practice to use unquoted text in this way. Unquoted words are called "Barewords."
Note: There is **no** comma after *STDOUT*.

4.3.1 Quotes

Quoting rules affect almost everything you do in Perl, especially when printing a string of words. Strings are normally delimited by a matched pair of either double or single quotes. When a string is enclosed in single quotes, all characters are treated as literals. When a string is enclosed in double quotes, however, **almost** all characters are treated as literals with the exception of those characters that are used for variable substitution and special escape sequences. We will look at the special escape sequences in this chapter and discuss quoting and variables in Chapter 5, "What's in a Name."

Perl uses some characters for special purposes, such as the dollar sign ($) and the (@) sign. If these special characters are to be treated as literal characters, they may be preceded by a backslash (\) or enclosed within single quotes (' '). The backslash is used to quote a single character rather than a string of characters.

EXAMPLE 4.4

```
(The Script)
1  $name="Ellie";
2  print "Hello, $name.\n";# $name and \n evaluated
3  print 'Hello, $name.\n';# String is literal; newline not
                           # interpreted

4  print "I don't care!\n";# \n is interpreted in double quotes
5  print 'I don\'t care!', "\n";# Backslash protects single quote
                                # in string "don\'t"

(Output)
2  Hello, Ellie.
3,4  Hello, $name.\nI don't care!
5 I don't care!
```

It is so common to make mistakes with quoting that we will introduce here the most common error messages you will receive resulting from mismatched quotes and bare words.

Think of quotes as being the "clothes" for Perl strings. If you take them off, you may get a "Bareword" message such as:

***Bareword** "there" not allowed while "strict subs" in use at try.pl line 3. Execution of program.pl aborted due to compilation errors.*

Also think of quotes as being mates. A double quote is mated with a matching double quote, and a single quote with a matching single quote. If you don't match the quotes, if one is missing, the missing quote has "run away." Where did the mate go? You may receive an error like this:

*(Might be a **runaway** multi-line "" string starting on line 3)*

Breaking the Quoting Rules

EXAMPLE 4.5

```
(The Script)
    #!/usr/bin/perl
    # Program to illustrate printing literals
1   print "Hello, "I can't go there";  # Unmatched quotes
2   print "Good-bye";

(Output)
Bareword found where operator expected at qtest.plx line 2, near
""Hello, "I"
        (Missing operator before I?)
Bareword found where operator expected at qtest.plx line 3, near
"print "Good"
  (Might be a runaway multi-line "" string starting on line 2)
        (Do you need to predeclare print?)
String found where operator expected at qtest.plx line 3, at end of line
        (Missing semicolon on previous line?)
syntax error at qtest.plx line 2, near ""Hello, "I can't "
Can't find string terminator '"' anywhere before EOF at qtest.plx line 3
```

EXPLANATION

1 The string "*Hello* starts with an opening double quote but is missing the ending quote. This cascades into a barrage of troubles. Perl assumes the double quote preceding the word "I" is the mate for the first quote in "Hello." That leaves the rest of the string "I can't go there" exposed as a bare string. The double quote at the end of the line will be mated with the double quote on the next line. Not good.

2 The word "Good_bye" is considered a bareword because Perl can't find an opening quote. The double quote at the end of "there" on line 1 has been matched with the double quote at the beginning of "Good-bye," leaving "Good-bye" exposed and bare, with an unmatched quote at the end of the string. Ugh!

4.3.2 Literals (Constants)

When assigning literal values[1] to variables or printing literals, the literals can be represented numerically as integers in decimal, octal, or hexadecimal or as floats in floating point or scientific notation.

Strings enclosed in double quotes may contain string literals, such as \n for the newline character, \t for a tab character, or \e for an escape character. String literals are alphanumeric (**and only alphanumeric**) characters preceded by a backslash. They may be represented in decimal, octal, or hexadecimal or as control characters.

Perl also supports special literals for representing the current script name, the line number of the current script, and the logical end of the current script.

Since you will be using literals with the *print* and *printf* functions, let's see what these literals look like. (For more on defining constants, see the "*constant*" pragma in Appendix A.)

Numeric Literals. Literal numbers can be represented as positive or negative integers in decimal, octal, or hexadecimal (see Table 4.1). Floats can be represented in floating point notation or scientific notation. Octal numbers contain a leading *0* (zero), hex numbers a leading *0x* (zero and x), and numbers represented in scientific notation contain a trailing *E* followed by a negative or positive number representing the exponent.

Table 4.1 Numeric Literals

Example	***Description***
12345	Integer
0b1101	Binary
0x456fff	Hex
0777	Octal
23.45	Float
.234E–2	Scientific notation

String Literals. Like shell strings, Perl strings are normally delimited by either single or double quotes. Strings containing string literals, also called **escape sequences**, are delimited by **double quotes** for backslash interpretation (see Table 4.2).

1. Literals may also be called constants, but the Perl experts prefer the term "literal," so in deference to them, we'll use the term "literal."

Table 4.2 String Literals

Escape Sequences	***Descriptions (ASCII Name)***
\t	Tab
\n	Newline
\r	Carriage return
\f	Form feed
\b	Backspace
\a	Alarm/bell
\e	Escape
\033	Octal character
\xff	Hexadecimal character
\c[	Control character
\l	Next character is converted to lowercase
\u	Next character is converted to uppercase
\L	Next characters are converted to lowercase until \E is found
\U	Next characters are converted to uppercase until \E is found
\Q	Backslash all following nonalphanumeric characters until \E is found
\E	Ends upper- or lowercase conversion started with \L or \U
\\	Backslash

EXAMPLE 4.6

```
print "This string contains \t\t two tabs and a newline.\n" # Double quotes
(Output)
This string containstabs and a newline.

print 'This string contains\t\t two tabs and a newline.\n; #Single quotes
(Output)
This string contains\t\t two tabs and a newline.\n
```

Special Literals. Perl's special literals _ _*LINE*_ _ and _ _*FILE*_ _ are used as separate words and will **not** be interpreted if enclosed in quotes, single or double. They represent the current line number of your script and the name of the script, respectively. These special literals are equivalent to the predefined special macros used in the *C* language.

The _ _*END*_ _ special literal is used in scripts to represent the logical end of the file. Any trailing text following the _ _*END*_ _ literal will be ignored, just as if it had been commented. The control sequences for end of input in UNIX is *<Ctrl>-d* (*\004*), and *<Ctrl>-z* (*\032*) in MS-DOS; both are synonyms for _ _*END*_ _.

The _ _*DATA*_ _ special literal is used as a filehandle to allow you to process textual data from within the script instead of from an external file.

EXAMPLE 4.7

```
print "The script is called", _ _FILE_ _, "and we are on line number ",
_ _LINE_ _,"\n";
(Output)
The script is called ./testing.plx and we are on line number 2
```

Note: There are two underscores on either side of the special literals (see Table 4.3).

Table 4.3 Special Literals

Literal	*Description*
_ _*LINE*_ _	Represents the current line number
_ _*FILE*_ _	Represents the current filename
_ _*END*_ _	Represents the logical end of the script; trailing garbage is ignored
_ _*DATA*_ _	Represents a special filehandle
_ _*PACKAGE*_ _	Represents the current package; default package is *main*

4.3.3 Printing Literals

Now that you know what the literals look like, let's see how they are used with the *print* function.

Printing Numeric Literals

EXAMPLE 4.8

```
(The Script)
    #!/usr/bin/perl
    # Program to illustrate printing literals
1   print "The price is $100.\n";
2   print "The price is \$100.\n";
3   print "The price is \$",100, ".\n";
4   print "The binary number is converted to: ",0b10001,".\n";
5   print "The octal number is converted to: ",0777,".\n";
6   print "The hexadecimal number is converted to: ",0xAbcF,".\n";
7   print "The unformatted number is ", 14.56, ".\n";
8    $now = localtime(); # A Perl function
9    $name = "Ellie"; # A string is assigned to a Perl variable
10  print "Today is $now, $name.";
11  print 'Today is $now, $name.';

(Output)
1   The price is .
2   The price is $100.
3   The price is $100.
4   The binary number is converted to: 17.
5   The octal number is converted to: 511.
6   The hexadecimal number is converted to: 43983.
7   The unformatted number is 14.56.
10  Today is Sat Mar 24 15:46:08 2007, Ellie.
11  Today is $now, $name.
```

EXPLANATION

1 The string *The price is $100* is enclosed in double quotes. The dollar sign is a special Perl character. It is used to reference scalar variables (see Chapter 5, "What's in a Name"), not money. Therefore, since there is no variable called *$100*, nothing prints. Since single quotes protect all characters from interpretation, they would have sufficed here, or the dollar sign could have been preceded with a backslash. But when surrounded by single quotes, the \n will be treated as a literal string rather than a newline character.

2 The backslash quotes the dollar sign, so it is treated as a literal.

3 To be treated as a numeric literal, rather than a string, the number *100* is a single word. The dollar sign must be escaped even if it is not followed by a variable name. The *\n* must be enclosed within double quotes if it is to be interpreted as a special string literal.

4 The number is represented as a binary number because of the leading *0b* (zero and b). The decimal value is printed.

EXPLANATION (CONTINUED)

5 The number is represented as an octal value because of the leading *0* (zero). The decimal value is printed.
6 The number is represented as a hexadecimal number because of the leading *0x* (zero and x). The decimal value is printed.
7 The number, represented as *14.56*, is printed as is. The *print* function does not format output.
8 Perl has a large set of functions. You have already learned about the *print* function. The *localtime()* function is another. (The parentheses are optional.) This functions returns the current date and time. We are assigning the result to a Perl variable called *$now*. You will learn all about variables in the next chapter.
9 The variable *$name* is assigned the string "Ellie".
10 When the string is enclosed in double quotes, the *print* function will display the value of the variables *$now* and *$name*.
11 When the string is enclosed in single quotes, the *print* function prints all characters literally.

Printing String Literals

EXAMPLE 4.9

```
(The Script)
    #!/usr/bin/perl
1   print "***\tIn double quotes\t***\n";   # Backslash interpretation
2   print '%%%\t\tIn single quotes\t\t%%%\n'; # All characters are
                                              # printed as literals
3   print "\n";

(Output)
1   ***     In double quotes        ***
2   %%%\t\tIn single quotes\t\t%%%\n
3
```

EXPLANATION

1 When a string is enclosed in double quotes, backslash interpretation is performed. The *\t* is a string literal and produces a tab; the *\n* produces a newline.
2 When enclosed within single quotes, the special string literals *\t* and *\n* are not interpreted. They will be printed as is.
3 The newline *\n* must be enclosed in double quotes to be interpreted. A "*\n*" produces a newline.

EXAMPLE 4.10

```
(The Script)
    #!/usr/bin/perl
1   print "\a\t\tThe \Unumber\E \LIS\E ",0777,".\n";

(Output)
1   (BEEP)          The NUMBER is 511.
```

EXPLANATION

1 The *\a* produces an alarm or beep sound, followed by *\t\t* (two tabs). *\U* causes the string to be printed in uppercase until *\E* is reached or the line terminates. The string *number* is printed in uppercase until the *\E* is reached. The string *is* is to be printed in lowercase, until the *\E* is reached, and the decimal value for octal *0777* is printed, followed by a period and a newline character.

Printing Special Literals

EXAMPLE 4.11

```
(The Script)
    #!/usr/bin/perl
    # Program, named literals.perl, written to test special literals
1   print "We are on line number ", _ _LINE_ _, ".\n";
2   print "The name of this file is ",_ _FILE_ _,".\n";
3   _ _END_ _
    And this stuff is just a bunch of chitter-chatter that is to be
     ignored by Perl.
    The _ _END_ _ literal is like Ctrl-d or \004.[a]

(Output)
1   We are on line number 3.
2   The name of this file is literals.perl.
```

a. See the *-x* switch in Appendix A for discarding leading garbage.

EXPLANATION

1 The special literal _ _*LINE*_ _ cannot be enclosed in quotes if it is to be interpreted. It holds the current line number of the Perl script.

2 The name of this script is *literals.perl*. The special literal _ _*FILE*_ _ holds the name of the current Perl script.

3 The special literal _ _*END*_ _ represents the logical end of the script. It tells Perl to ignore any characters that follow it.

EXAMPLE 4.12

```
(The Script)
    #!/usr/bin/perl
    # Program, named literals.perl2,
    # written to test special literal _ _DATA_ _
1   print <DATA>;
2   _ _DATA_ _
    This line will be printed.
    And so will this one.

(Output)
This line will be printed.
And so will this one.
```

EXPLANATION

1 The *print* function will display whatever text is found under the special literal _ _*DATA*_ _. Because the special literal _ _*DATA*_ _ is enclosed in angle brackets, it is treated as a filehandle opened for reading. The *print* function will display lines as they are read by <*DATA*>.

2 This is the data that is used by the <*DATA*> filehandle. (You could use _ _*END*_ _ instead of _ _*DATA*_ _ to get the same results.)

4.3.4 The *warnings* Pragma and the *-w* Switch

The *-w* switch is used to warn you about the possibility of using future reserved words and a number of other problems that may cause problems in the program. (Often, these warnings are rather cryptic and hard to understand if you are new to programming.) Larry Wall says in the Perl 5 *man* pages, "Whenever you get mysterious behavior, try the *-w* switch! Whenever you don't get mysterious behavior, try the *-w* switch anyway."

You can use the *-w* switch either as a command-line option to Perl, as

```
perl -w <scriptname>
```

or after the *shbang* line in the Perl script, such as

```
#!/usr/bin/perl -w
```

A pragma is a special Perl module that hints to the compiler about how a block of statements should be compiled. You can use this type of module to help control the way your program behaves. Starting with Perl version 5.6.0, *warnings.pm* was added to the standard Perl library; similar to the *-w* switch, it is a pragma that allows you to control the types of warnings printed.

In your programs, add the following line under the *#!* line or, if not using the *#!* line, at the top of the script:

```
use warnings;
```

This enables all possible warnings. To turn off warnings, simply add as a line in your script

```
no warnings;
```

This disables all possible warnings for the rest of the script.

EXAMPLE 4.13

```
      (The Script)
         #!/usr/bin/perl
         # Scriptname: warnme
      1  print STDOUT Ellie, what\'s up?;

(Output) (At the Command Line)
$ perl -w warnme
  Unquoted string "what" may clash with future reserved word at warnme line 3.
  Backslash found where operator expected at warnme line 3, near "what\"
  Syntax error at warnme line 3, near "what\"
  Can't find string terminator "'" anywhere before EOF at warnme line 3.
```

EXPLANATION

1 Among many other messages, the *-w* switch (see Appendix A) prints warnings about ambiguous identifiers, such as variables that have been used only once, improper conversion of strings and numbers, etc. Since the string *Ellie* is not quoted, Perl could mistake it for a reserved word or an undefined filehandle. The rest of the error message results from having an unmatched quote in the string.

EXAMPLE 4.14

```
      (The Script)
         #!/usr/bin/perl
         # Scriptname: warnme
      1  use warnings;
      2  print STDOUT Ellie, what\'s up?;

(Output)
Unquoted string "what" may clash with future reserved word at warnme line 3.
Backslash found where operator expected at warnme line 3, near "what\"
Syntax error at warnme line 3, near "what\"
Can't find string terminator "'" anywhere before EOF at warnme line 3.
```

EXPLANATION

In Perl versions 5.6 and later, the *warnings* pragma is used instead of the *-w* switch. The *use* function allows you to use modules located in the standard Perl library. The *warnings* pragma sends warnings about ambiguous identifiers. Since the string *Ellie* is not quoted, Perl could mistake it for a reserved word or an undefined filehandle. The compiler complains because the string is not terminated with a closing quote.

4.3.5 The *diagnostics* Pragma

This special pragma enhances the warning messages to a more verbose explanation of what went wrong in your program. Like the *warnings* pragma, it affects the compilation phase of your program, but unlike the warnings pragma, it attempts to give you an explanation that doesn't assume you are an experienced programmer.

EXAMPLE 4.15

```
(The Script)
use diagnostics;
print "Hello there';  # Unmatched quote
print "We are on line number ", _ _LINE_ _,"\n";

(The output)
Bareword found where operator expected at test.plx line 3, near "$now
= "Ellie"
  (Might be a runaway multi-line "" string starting on line 2) (#1)
    (S syntax) The Perl lexer knows whether to expect a term or an
operator.
    If it sees what it knows to be a term when it was expecting to see
    an operator, it gives you this warning.  Usually it indicates that
    an operator or delimiter was omitted, such as a semicolon.

        (Missing operator before Ellie?)
String found where operator expected at test.plx line 3, at end of
line (#1)
        (Missing semicolon on previous line?)

syntax error at test.plx line 3, near "$now = "Ellie"
Can't find string terminator '"' anywhere before EOF at test.plx line
3 (#2)
    (F) Probably means you had a syntax error.  Common reasons
include:

        A keyword is misspelled.
        A semicolon is missing.
        A comma is missing.
        An opening or closing parenthesis is missing.
print "hello there';
print "We are on line number ", _ _LINE_ _,"\n";
```

EXPLANATION

In Perl versions 5.6 and later, the *diagnostics* pragma is used instead of the *-w* switch or the *warnings* pragma. This special Perl module sends detailed messages about the problems that occurred in the script. Since the string *Hello there* does not contain matched quotes, the *diagnostics* pragma issues a list of all the potential causes for the failed program. The compiler expects the string to be terminated with another double quote.

4.3.6 The *strict* Pragma and Words

Another pragma we will mention now is the *strict* pragma. If your program disobeys the restrictions placed on it, it won't compile. If there is a chance that you might have used "bare," i.e., unquoted, words[2] as in Example 4.15, the *strict* pragma will catch you and your program will abort. The *strict* pragma can be controlled by giving it various arguments. (See Appendix A for complete list.)

EXAMPLE 4.16

```
(The Script)
    #!/usr/bin/perl
    # Program: stricts.test
    # Script to demonstrate the strict pragma
1   use strict "subs";
2   $name = Ellie;              # Unquoted word Ellie
3   print "Hi $name.\n";


(Output)
$ stricts.test
  Bareword "Ellie" not allowed while "strict subs" in use at
./stricts.test line 5.
  Execution of stricts.test aborted due to compilation errors.
```

EXPLANATION

1 The *use* function allows you to use modules located in the standard Perl library. When the *strict* pragma takes *subs* as an argument, it will catch any barewords found in the program while it is being internally compiled. If a bareword is found, the program will be aborted with an error message.

2. Putting quotes around a word is like putting clothes on the word—take off the quotes, and the word is "bare."

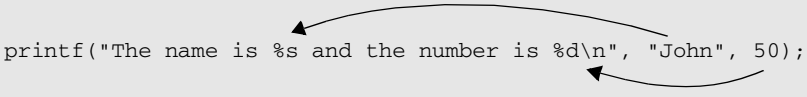
4.4 The *printf* Function

The *printf* function prints a formatted string to the selected filehandle, the default being *STDOUT*. It is like the *printf* function used in the *C* and *awk* languages. The return value is *1* if *printf* is successful and *0* if it fails.

The *printf* function consists of a quoted control string that may include format specifications. The quoted string is followed by a comma and a list of comma-separated arguments, which are simply expressions. The format specifiers are preceded by a % sign. For each % sign and format specifier, there must be a corresponding argument. (See Tables 4.4 and 4.5.)

Placing the quoted string and expressions within parentheses is optional.

EXAMPLE 4.17



```
printf("The name is %s and the number is %d\n", "John", 50);
```

EXPLANATION

1 The string to be printed is enclosed in double quotes. The first format specifier is *%s*. It has a corresponding argument, *John*, positioned directly to the right of the comma after the closing quote in the control string. The *s* following the percent sign is called a **conversion character**. The *s* means *string* conversion will take place at this spot. In this case *John* will replace the *%s* when the string is printed.

2 The *%d* format specifies that the decimal (integer) value *50* will be printed in its place within the string.

Table 4.4 Format Specifiers

Conversion	*Definition*
%b	Unsigned binary integer
%c	Character
%d, i	Decimal number
%e	Floating point number in scientific notation
%E	Floating point number in scientific notation using capital *E*
%f, %F	Floating point number
%g	Floating point number using either *e* or *f* conversion, whichever takes the least space
%G	Floating point number using either *e* or *f* conversion, whichever takes the least space
%ld, %D	Long decimal number

Continues

Table 4.4 Format Specifiers (continued)

Conversion	*Definition*
%lu, %U	Long unsigned decimal number
%lo, %O	Long octal number
%p	Pointer (hexadecimal)
%s	String
%u	Unsigned decimal number
%x	Hexadecimal number
%X	Hexadecimal number using capital *X*
%lx	Long hexidecimal number
%%	Print a literal percent sign

Flag modifiers are used after the *%* to further define the printing; for example, *%-20s* represents a 20-character left-justified field.

Table 4.5 Flag Modifiers

Conversion	*Definition*
%-	Left-justification modifier
%#	Integers in octal format are displayed with a leading *0*; integers in hexadecimal form are displayed with a leading *0x*
%+	For conversions using *d*, *e*, *f*, and *g*, integers are displayed with a numeric sign, + or -
%0	The displayed value is padded with zeros instead of whitespace
%number	Maximum field width; for example, if number is *6*, as in *%6d*, maximum field width is six digits
%.number	Precision of a floating point number; for example, *%.2f* specifies a precision of two digits to the right of the decimal point, and *%8.2* represents a maximum field width of eight, where one of the characters is a decimal point followed by two digits after the decimal point

When an argument is printed, the **field** holds the value that will be printed, and the **width** of the field is the number of characters the field should contain. The width of a field is specified by a percent sign and a number representing the maximum field width, followed by the conversion character; for example, *%20s* is a right-justified 20-character string; *%-25s* is a left-justified 25-character string; and *%10.2f* is a right-justified 10-character

floating point number, where the decimal point counts as one of the characters and the precision is two places to the right of the decimal point. If the argument exceeds the maximum field width, *printf* will **not** truncate the number, but your formatting may not look nice. If the number to the right of the decimal point is truncated, it will be rounded up; for example, if the formatting instruction is *%.2f*, the corresponding argument, *56.555555*, would be printed as *56.6*.

EXAMPLE 4.18

```
(The Script)
   #!/usr/bin/perl
1  printf "Hello to you and yours %s!\n","Sam McGoo!";
2  printf("%-15s%-20s\n", "Jack", "Sprat");
3  printf "The number in decimal is %d\n", 45;
4  printf "The formatted number is |%10d|\n", 100;
5  printf "The number printed with leading zeros is |%010d|\n", 5;

6  printf "Left-justified the number is |%-10d|\n", 100;
7  printf "The number in octal is %o\n",15;
8  printf "The number in hexadecimal is %x\n", 15;
9  printf "The formatted floating point number is |%8.2f|\n",
      14.3456;
10 printf "The floating point number is |%8f|\n", 15;
11 printf  "The character is %c\n", 65;

(Output)
1  Hello to you and yours Sam McGoo!
2  Jack                 Sprat
3  The number in decimal is 45
4  The formatted number is |       100|
5  The number printed with leading zeros is |0000000005|.
6  Left-justified the number is |100     |
7  The number in octal is 17
8  The number in hexadecimal is f
9  The formatted floating point number is |   14.35|
10 The floating point number is |15.000000|
11 The character is A
```

EXPLANATION

1 The quoted string contains the *%s* format conversion specifier. The string *Sam Magoo* is converted to a string and replaces the *%s* in the printed output.

2 The string *Jack* has a field width of 15 characters and is left-justified. The string *Sprat* has a field width of 20 characters and is also left-justified. Parentheses are optional.

3 The number *45* is printed in decimal format.

4 The number *100* has a field width of 10 and is right-justified.

EXPLANATION (CONTINUED)

5 The number *5* has a field width of 10, is right-justified, and is preceded by leading zeros rather than whitespace. If the modifier *0* is placed before the number representing the field width, the number printed will be padded with leading zeros if it takes up less space than it needs.
6 The number *100* has a field width of 10 and is left-justified.
7 The number *15* is printed in octal.
8 The number *15* is printed in hexadecimal.
9 The number *14.3456* is given a field width of eight characters. One of them is the decimal point; the fractional part is given a precision of two decimal places. The number is then rounded up.
10 The number *15* is given a field width of eight characters, right-justified. The default precision is six decimal places to the right of the decimal point.
11 The number *65* is converted to the ASCII character *A* and printed.

4.4.1 The *sprintf* Function

The *sprintf* function is just like the *printf* function, except it allows you to assign the formatted string to a variable. *sprintf* and *printf* use the same conversion tables (Tables 4.4 and 4.5). Variables are discussed in Chapter 5, "What's in a Name."

EXAMPLE 4.19

```
(The Script)
1   $string = sprintf("The name is: %10s\nThe number is: %8.2f\n",
                      "Ellie", 33);
2   print "$string";

(Output)
2   The name is:      Ellie
    The number is:    33.00
```

EXPLANATION

1 The *sprintf* function follows the same rules as *printf* for conversion of characters, strings, and numbers. The only real difference is that *sprintf* allows you to store the formatted output in a variable. In this example, the formatted output is stored in the scalar variable *$string*. The *\n* inserted in the string causes the remaining portion of the string to be printed on the next line. Scalar variables are discussed in Chapter 5, "What's in a Name." Parentheses are optional.
2 The value of the variable is printed showing the formatted output produced by *sprintf*.

4.4.2 Printing without Quotes—The *here document*

The Perl *here document* is derived from the UNIX shell *here document*. It allows you to quote a whole block of text enclosed between words called user-defined terminators. From the first terminator to the last terminator, the text is quoted, or you could say "from *here* to *here*" the text is quoted. The *here document* is a line-oriented form of quoting, requiring the << operator followed by an initial terminating word and a semicolon. There can be no spaces after the << unless the terminator itself is quoted. If the terminating word is not quoted or double quoted, variable expansion is performed. If the terminating word is singly quoted, variable expansion is not performed. Each line of text is inserted between the first and last terminating word. The final terminating word must be on a line by itself, with no surrounding whitespace.

Perl, unlike the shell, does not perform command substitution (backquotes) in the text of a *here document*. Perl, on the other hand, does allow you to execute commands in the *here document* if the terminator is enclosed in backquotes. (Not a good idea.)

Here documents are used extensively in CGI scripts for enclosing large chunks of HTML tags for printing.

EXAMPLE 4.20

```
(The Script)
1   $price=1000;   # A variable is assigned a value.
2   print <<EOF;
3   The consumer commented, "As I look over my budget, I'd say
4   the price of $price is right. I'll give you \$500 to start."\n
5   EOF

6   print <<'FINIS';
    The consumer commented, "As I look over my budget, I'd say
7   the price of $price is too much.\n I'll settle for $500."
8   FINIS

9   print << x 4;
    Here's to a new day.
    Cheers!
10
    print "\nLet's execute some commands.\n";
    # If terminator is in backquotes, will execute OS commands
11  print <<`END`;
    echo Today is
    date
    END

(Output)
3   The consumer commented, "As I look over my budget, I'd say
    the price of 1000 is right. I'll give you $500 to start."
```

EXAMPLE 4.20 (CONTINUED)

```
6   The consumer commented, "As I look over my budget, I'd say
    the price of $price is too much. \n I'll settle for $500."
9   Here's to a new day.
    Cheers!
    Here's to a new day.
    Cheers!
    Here's to a new day.
    Cheers!
    Here's to a new day.
    Cheers!
11  Let's execute some commands.
    Today is
    Fri Oct 27 12:48:36 PDT 2007
```

EXPLANATION

1 A scalar variable, *$price*, is assigned the value *1000*.

2 Start of *here document*. *EOF* is the terminator. The block is treated as if in double quotes. If there is any space preceding the terminator, then enclose the terminator in double quotes, such as *"EOF"*.

3 All text in the body of the *here document* is quoted as though the whole block of text were surrounded by double quotes.

4 The dollar sign has a special meaning when enclosed in double quotes. Since the text in this *here document* is treated as if in double quotes, the variable has special meaning here as well. The $ is used to indicate that a scalar variable is being used. The value of the variable will be interpreted. If a backslash precedes the dollar sign, it will be treated as a literal. If special backslash sequences are used, such as *\n*, they will be interpreted.

5 End of *here document* marked by matching terminator, *EOF*. There can be no space surrounding the terminator.

6 By surrounding the terminator, *FINIS*, with single quotes, the text that follows will be treated literally, turning off the meaning of any special characters, such as the dollar sign or backslash sequences.

7 Text is treated as if in single quotes.

8 Closing terminator marks the end of the *here document*.

9 The value *x 4* says that the text within the *here document* will be printed four times. The *x* operator is called the *repetition operator*. There must be a **blank line** at the end of the block of text, so that the *here document* is terminated.

10 The blank line is required here to end the *here document*.

11 The terminator is enclosed in backquotes. The shell will execute the commands between *`END`* and *END*. This example includes UNIX commands. If you are using another operating system, such as Windows or Mac OS, the commands must be compatible with that operating system.

***Here Documents* and CGI.** The following program is called a CGI (Common Gateway Interface) program, a simple Perl program executed by a Web server rather than by the shell. It is just like any other Perl script with two exceptions:

1. There is a line called the MIME line (e.g., *Content-type: text/html*) that describes what kind of content will be sent back to the browser.
2. The document consists of text embedded with HTML tags, the language used by browsers to render text in different colors, fonts faces, types, etc. Many CGI programmers take advantage of the *here document* to avoid using the *print* function for every line of the program.

CGI programs are stored in a special directory called *cgi-bin,* which is normally found under the Web server's root directory. See Chapter 16, "CGI and Perl: The Hyper Dynamic Duo," for a complete discussion of CGI.

To execute the following script, you will start up your Web browser and type in the Location box: *http://servername/cgi-bin/scriptname*.[3] See Figure 4.1.

EXAMPLE 4.21

```
    #!/bin/perl
    # The HTML tags are embedded in the here document to avoid using
    # multiple print statements
1   print <<EOF;      # here document in a CGI script
2   Content-type: text/html
3
4   <HTML><HEAD><TITLE>Town Crier</TITLE></HEAD>
    <H1><CENTER>Hear ye, hear ye, Sir Richard cometh!!</CENTER></H1>
    </HTML>
5   EOF
```

EXPLANATION

1 The *here document* starts here. The terminating word is *EOF*. The *print* function will receive everything from *EOF* to *EOF*.

2 This line tells the browser that the type of content that is being sent is text mixed with HTML tags. This line **must** be followed by a blank line.

4 The body of the document consists of text and HTML tags.

5 The word *EOF* marks the end of the *here document*.

3. You must supply the correct server name for your system and the correct filename. Some CGI files must have a *.cgi* or *.pl* extension.

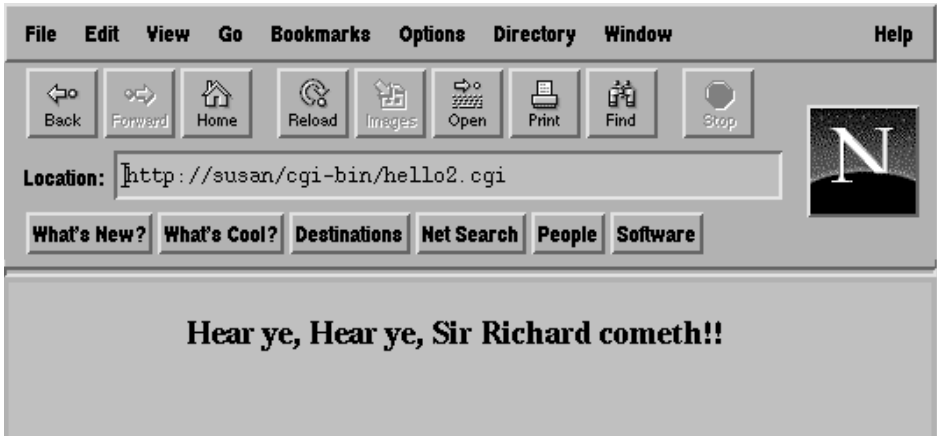


Figure 4.1 The Web browser in Example 4.21.

4.5 What You Should Know

1. How do you define *stdin*, *stdout*, and *stderr*?
2. What is meant by the term "filehandle"?
3. How do you represent a number in octal? Hexadecimal? Binary?
4. What is the main difference between the *print* and *printf* functions?
5. How do double and single quotes differ in the way they treat a string?
6. What are "literals"?
7. What is the use of _ _END_ _?
8. What are backslash sequences?
9. What is the purpose of the *sprintf* function?
10. What is a pragma?
11. How can you check to make sure your syntax is ok?
12. What is a *here document*? How is it useful in CGI programs?

4.6 What's Next?

In the next chapter, you will learn about Perl variables and the meaning of the "funny symbols." You will be able to create and access scalars, arrays, and hashes understand context and namespaces. You will also learn how to get input from a user and why we need to "chomp." A number of array and hash functions will be introduced.

EXERCISE 4
A String of Perls

1. Use the *print* function to output the following string:

   "Ouch," cried Mrs. O'Neil, "You musn't do that Mr. O'Neil!"

2. Use the *printf* function to print the number $34.6666666 as $34.67.

3. Write a Perl script called *literals.plx* that will print the following:

```
$ perl literals
Today is Mon Mar 12 12:58:04 PDT 2007  (Use localtime())
The name of this PERL SCRIPT is literals.
Hello. The number we will examine is 125.5.
The NUMBER in decimal is 125.
The following number is taking up 20 spaces and is right justified.
|                        125|
                The number in hex is 7d
                The number in octal is 175
The number in scientific notation is 1.255000e+02
The unformatted number is 125.500000
The formatted number is 125.50
My boss just said, "Can't you loan me $12.50 for my lunch?"
I flatly said, "No way!"
Good-bye (Makes a beep sound)
```

   Note: The words PERL SCRIPT and NUMBER are capitalized by using string literal escape sequences.

   What command-line option would you use to check the syntax of your script?

4. Add to your literals script a *here document* to print:

   Life is good with Perl.
   I have just completed my second exercise!

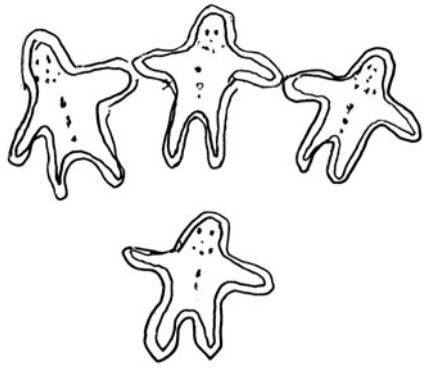
5. How would you turn on warnings in the script? How would you turn on diagnostics?

This page intentionally left blank

chapter

5

What's in a Name



5.1 About Perl Variables

Before starting this chapter, a note to you, the reader. Each line of code in an example is numbered. The output and explanations are also numbered to match the number in the code. These numbers are provided to help you understand important lines of each program. When copying examples into your text editor, don't include these numbers, or you will generate many unwanted errors! With that said, let's proceed.

5.1.1 Types

Variables are fundamental to all programming languages. They are data items whose values may change throughout the run of the program, whereas literals or constants remain fixed. They can be placed anywhere in the program and do not have to be declared as in other higher languages, where you must specify the data type that will be stored there. You can assign strings, numbers, or a combination of these to Perl variables. For example, you may store a number in a variable and then later change your mind and store a string there. Perl doesn't care.

Perl variables are of three types: scalar, array, and associative array (more commonly called hashes). A scalar variable contains a single value (e.g., one string or one number), an array variable contains an ordered list of values indexed by a positive number, and a hash contains an unordered set of key/value pairs indexed by a string (the key) that is associated with a corresponding value. (See "Scalars, Arrays, and Hashes" on page 77.)

5.1.2 Scope and the Package

The scope of a variable determines where it is visible in the program. In Perl scripts, the variable is visible to the entire script (i.e., global in scope) and can be changed anywhere within the script.

The Perl sample programs you have seen in the previous chapters are compiled internally into what is called a **package**, which provides a **namespace** for variables. Almost all variables are **global** within that package. A global variable is known to the whole package and, if changed anywhere within the package, the change will permanently affect the variable. The default package is called *main*, similar to the *main()* function in the *C* language. Such variables in *C* would be classified as **static**. At this point, you don't have to worry about naming the *main* package or the way in which it is handled during the compilation process. The only purpose in mentioning packages now is to let you know that the scope of variables in the *main* package, your script, is global. Later, when we talk about the *our*, *local,* and *my* functions in packages, you will see that it is possible to change the scope and namespace of a variable.

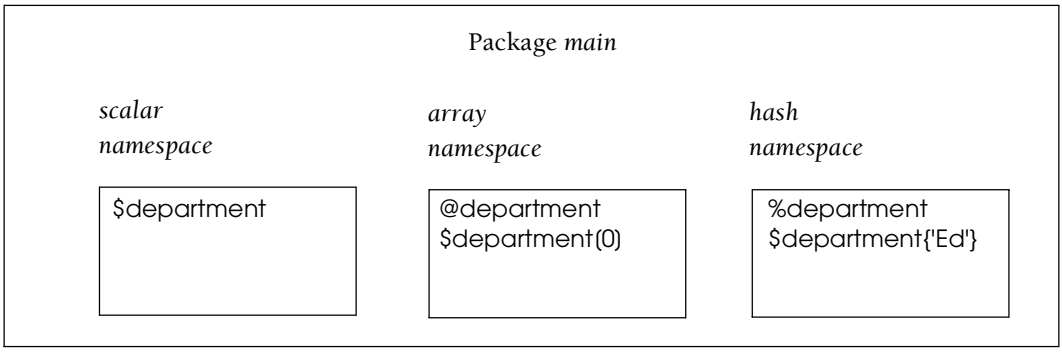


Figure 5.1 Namespaces for scalars, lists, and hashes in package *main*.

5.1.3 Naming Conventions

Unlike *C* or *Java*, Perl variables don't have to be declared before being used. They spring to life just by the mere mention of them. Variables have their own namespace in Perl. They are identified by the "funny characters" that precede them. Scalar variables are preceded by a $ sign, array variables are preceded by an @ sign, and hash variables are preceded by a % sign. Since the "funny characters" indicate what type of variable you are using, you can use the same name for a scalar, array, or hash and not worry about a naming conflict. For example, *$name*, *@name*, and *%name* are all different variables; the first is a scalar, the second is an array, and the last is a hash.[1]

Since reserved words and filehandles are not preceded by a special character, variable names will not conflict with reserved words or filehandles. Variables are **case sensitive**. The variables named *$Num*, *$num*, and *$NUM* are all different.

If a variable starts with a letter, it may consist of any number of letters (an underscore counts as a letter) and/or digits. If the variable does not start with a letter, it must consist of

1. Using the same name is allowed but not recommended; it makes reading too confusing.