Microsoft

Start Here!™

FUNDAMENTALS OF

Microsoft®

.NET Programming

Rod Stephens

Ready to learn programming?

Get ready to learn Microsoft .NET programming by exploring the basic concepts that drive all .NET-based languages. If you have absolutely no previous experience, no problem—simply start here! This book introduces must-know concepts and techniques through easy-to-follow explanations, examples, and exercises.

Here's where you start learning how software works

- Understand how the .NET development environment helps you design and run programs
- Delve into basic object-oriented concepts such as properties, methods, and events
- Learn how programs store data in files, object stores, and databases
- Dig into controls—labels, text boxes, menus, scroll bars
- See how programs take advantage of multicore processors
- Get an extensive glossary of key programming terms

NOTE: This title is included as a free companion eBook for the *Start Here!* books for Microsoft Visual Basic® and Visual C#®. This print edition is available for sale for anyone interested in .NET fundamentals.

ISBN: 978-0-7356-6168-4

U.S.A. $19.99
Canada $20.99
[*Recommended*]

Programming/Microsoft Visual Studio

Start Here! Fundamentals of Microsoft® .NET Programming

For Skill Level: Beginner
Prerequisites: None

RESOURCE ROADMAP

Start Here
- Beginner-level instruction
- Easy to follow explanations and examples
- Exercises to build your first projects

Step by Step
- For experienced developers learning a new topic
- Focus on fundamental techniques and tools
- Hands-on tutorial with practice files plus eBook

Developer Reference
- Professional developers; intermediate to advanced
- Expertly covers essential topics and techniques
- Features extensive, adaptable code examples

Focused Topics
- For programmers who develop complex or advanced solutions
- Specialized topics; narrow focus; deep coverage
- Features extensive, adaptable code examples

microsoft.com/mspress

About the Author

Rod Stephens is president of a computer consulting company, has 20 years of experience as a professional developer, and is the author of more than 20 books and 250 articles about programming.

Microsoft®

Microsoft

Start Here!™

Fundamentals of Microsoft® .NET Programming

Rod Stephens

Copyright © 2011 by Rod Stephens
All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-6168-4

1 2 3 4 5 6 7 8 9 LSI 6 5 4 3 2 1

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at *mspinput@microsoft.com*. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Microsoft and the trademarks listed at *http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx* are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions and Developmental Editor: Russell Jones

Production Editor: Jasmine Perez

Editorial Production: S4Carlisle Publishing Services

Technical Reviewer: Debbie Timmins

Indexer: WordCo Indexing Services, Inc.

Cover: Valerie DeGiulio

Contents at a Glance

	Introduction	*xiii*
CHAPTER 1	**Computer Hardware**	**1**
CHAPTER 2	**Multiprocessing**	**15**
CHAPTER 3	**Programming Environments**	**25**
CHAPTER 4	**Windows Program Components**	**33**
CHAPTER 5	**Controls**	**49**
CHAPTER 6	**Variables**	**71**
CHAPTER 7	**Control Statements**	**91**
CHAPTER 8	**Operators**	**105**
CHAPTER 9	**Routines**	**119**
CHAPTER 10	**Object-Oriented Programming**	**141**
CHAPTER 11	**Development Techniques**	**167**
CHAPTER 12	**Globalization**	**183**
CHAPTER 13	**Data Storage**	**191**
CHAPTER 14	**.NET Libraries**	**209**
	Glossary	*215*
	Index	*227*
	About the Author	*241*

Contents

Introduction . . . . . *xiii*

Chapter 1 Computer Hardware **1**

Types of Computers 2
- Personal Computers 2
- Desktops, Towers, and Workstations 2
- Laptops, Notebooks, Netbooks, and Tablets 3
- Minis, Servers, and Mainframes 4
- Handheld Computers 5
- Comparing Computer Types 6

Computer Speed 6

Data Storage 8
- RAM 9
- Flash Drives 9
- Hard Drives 10
- Blu-ray, DVD, and CD Drives 10
- Working with Files 10

Networks 11

Summary 13

Chapter 2 Multiprocessing **15**

Multitasking 16

Multiprocessing 16

Multithreading 17

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Problems with Parallelism 18
Contention for Resources 18
Race Conditions 18
Locks 19
Deadlocks 20
Looking for Parallelism 21
Distributed Computing 22
Task Parallel Library 23
Summary 24

Chapter 3 Programming Environments 25

From Software to Hardware 25
Programming Environments 28
Visual Studio 29
Summary 31

Chapter 4 Windows Program Components 33

Menus 34
Use Ellipses 34
Provide Accelerators 34
Provide Shortcuts 35
Use Standard Menu Items 36
Don't Hide Commands 38
Use Shallow Menu Hierarchies 39
Keep Menus Short 39
A Menu Example 40
Context Menus 40
Toolbars and Ribbons 42
Dialog Boxes 43
User Interface Design 44
Control Order 44

Group Related Controls. 44
The Rule of Seven. 46
Don't Allow Mistakes. 47
Provide Hints. 47
Summary. 48

Chapter 5 Controls 49

Using Controls 51
Windows Forms Controls 52
WPF Controls 57
Properties 60
Windows Forms Properties 60
WPF Properties 63
Methods 66
Events 67
Summary. 69

Chapter 6 Variables 71

Fundamental Data Types 71
Strings 74
Program-Defined Data Types 74
Arrays 75
Enumerations 75
Classes 77
Value and Reference Types 78
Type Conversion 82
Explicit Conversion. 82
Implicit Conversion 83
Scope, Accessibility, and Lifetime. 85
Scope 85
Accessibility 87
Summary. 89

Chapter 7 Control Statements **91**

Pseudocode 92

Looping Statements 93

For Loops 93

For Each Loops 94

Do While Loops 95

While Loops 95

Until Loops 95

Conditional Statements 96

If 96

If Else 97

Else If 97

Case 97

Jumping Statements 99

Go To 99

Exit 101

Continue 101

Return 102

Jumping Guidelines 102

Error Handling 103

Summary 103

Chapter 8 Operators **105**

Precedence 106

Operators 106

Parentheses 107

Operator Precedence 108

Operator Overloading 114

Operator Overloading Overload 115

Conversion Operators 116

Summary 116

Chapter 9 Routines **119**

Types of Routines 120

Advantages of Routines 120

Reducing Duplicated Code 121

Reusing Code 121

Simplifying Complex Code 122

Hiding Implementation Details 122

Dividing Tasks Among Programmers 122

Making Debugging Easier 123

Calling Routines 123

Writing Good Routines 125

Perform a Single, Well-Defined Task 125

Avoid Side Effects 126

Use Descriptive Names 126

Keep It Short 126

Use Comments 127

Parameters 128

Optional Parameters 129

Parameter Arrays 130

Parameter-Passing Methods 130

Reference and Value Types 132

Arrays 134

Routine Overloading 135

Routine Accessibility 136

Recursion 137

Summary 139

Chapter 10 Object-Oriented Programming **141**

Classes 142

Class Benefits 142

Properties, Methods, and Events 143
Properties 143
Methods 145
Events 146
Shared Versus Instance Members 146
Inheritance 147
Polymorphism 148
Overriding Members 149
Shadowing Members 151
Inheritance Diagrams 152
Abstraction and Refinement 154
Abstraction 154
Refinement 156
"Is-A" Versus "Has-A" 158
Multiple Inheritance and Interface Implementation 158
Constructors and Destructors 161
Constructors 161
Destructors 163
Summary 165

Chapter 11 Development Techniques 167

Comments 167
Types of Comments 169
XML Comments 170
Naming Conventions 173
Development Techniques 175
Data-centric Viewpoint 175
User-centric Viewpoint 176
Agile Development 177

Extreme Programming 178
Test-driven Development 179
Summary 181

Chapter 12 Globalization 183

Terminology 184
Culture Codes 184
Locale-Specific Text and Symbols 184
Localizing User Interfaces in Visual Studio 185
Locale-Specific Formats 186
Culture-Aware Functions in .NET 187
Summary 189

Chapter 13 Data Storage 191

Files 191
Text Files 192
Random Access Files 192
INI Files 193
XML Files 194
Config Files 197
The System Registry 199
Relational Databases 200
Other Databases 202
Spreadsheets 202
Object Stores 203
Object-Relational Database 203
Hierarchical Databases 203
Network Databases 205
Temporal Databases 206
Summary 206

Chapter 14 .NET Libraries **209**

Microsoft Namespaces 210

System Namespaces 210

Summary 213

Glossary *215*

Index *227*

About the Author *241*

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Introduction

Programming languages do one very simple thing: they allow you to write programs that tell the computer what to do. You can tell a computer to read a value from the keyboard, add two numbers, save a result in a file on the hard disk, or draw a smiley face on the screen.

No matter what programming language you use, the underlying commands that the computer can execute are exactly the same. Whether you use Java, C#, Microsoft Visual Basic, COBOL, LISP, or any other language, you can make the computer perform roughly the same tasks. Two languages may have very different syntaxes, and some languages make some tasks easier than others, but the fundamental operations they can perform are the same. All these languages can carry out numeric calculations and manipulate files; unfortunately, none of them can reliably pick lottery winners. (If you write a program that can, let me know!)

At a more conceptual level, programming concepts have been refined over the years until most modern languages share a common set of fundamental concepts, such as variables, classes, objects, forms, menus, files, and multiprocessing. Don't worry if you don't know what these are—the purpose of this book is to provide more information about such terms and concepts.

Because programming languages share so many operations and concepts, programming books tend to cover the same topics as well. Books about databases or graphics cover these specialized topics in great detail. Different authors may place emphasis on different subjects, but there's a lot of overlap, particularly in beginning and general "how to program" books. Every one of these books explains what a variable is, how to create objects, and what a text file contains.

All this means that if you want to learn more than one programming language (a practice that I highly recommend), you're going to encounter much of the same material repeatedly. Even if you skim the familiar sections, you still have to pay for the content. You may start with a 600-page book about Visual Basic programming. Later, when you buy a 500-page C# book, you'll discover that 200 of those pages cover things you already know. Next, when your boss decides you need to learn LISP, you'll find that your new 550-page book contains 100 pages that you already know. (There are a lot of differences between LISP and the other two languages, so there will be less overlap if you shift to that language.)

This edition of the *Start Here!* series changes all that. Rather than making each *Start Here!* book cover the exact same topics, those common topics have been moved into this volume for easy reference. Now if you read *Start Here! Learn Microsoft Visual C# 2010 Programming* or *Start Here! Learn Microsoft Visual Basic Programming*, you won't need to rehash the exact same topics. Instead, those books refer you to this one for background information, such as how disk buffering works, freeing the other books to focus on language-specific issues.

There still will be some overlap between any books about different languages. For example, Visual Basic and C# both let you read and write disk files. Although *Start Here! Fundamentals of .NET Programming* explains in general what disk files are and how programs interact with them, the other books still need to explain the syntax for the code that their respective languages use to read and write files.

Note that this book doesn't necessarily cover every last detail of each background topic. It just gives you the information you need to understand how programs fit into a larger context so that you can get the most out of them. For example, this book explains some important programming issues relating to disk drives, but it doesn't explain in detail how disk drives work.

Moving underlying common topics into this separate book provides several benefits, including the following:

- Other *Start Here!* books can spend less time on the background material covered in this book and more time on language-specific issues. Those books can rely on and refer to this book to provide extra detail as needed.
- This book provides more room for, and spends more time on, basic concepts that beginning programming books often must gloss over to make room for language-specific concepts.
- This book provides a single location for learning about general computer topics, without focusing on a particular language. This is important because it can give you a broader understanding of what you can make computers do easily and what might be difficult to make a computer do, regardless of which programming language you choose.
- This book can act as an enhanced glossary, giving you a place to look for explanations of common computer terms. A normal glossary briefly defines key terms, but in addition, the rest of the book provides much more detail about important concepts.

Who Should Read This Book

This book is for anyone who wants a basic understanding of computers and the environments in which programs operate. It provides background information that is useful when you are trying to learn to use any programming language. It also provides information that can help you understand how programs work in general. For example, it explains what multithreading is and why multi-core computers may not always perform much better than single-core systems.

Assumptions

This book does *not* assume that you have any previous programming experience. In fact, it doesn't even assume that you have a computer! Instead, this book is about understanding computers and programs in general, and Microsoft Windows and .NET concepts in particular, not about writing programs in a specific language.

This book is intended for two main audiences: those who want to learn a new programming language, particularly those who are reading one of the other books in the *Start Here!* series, and those who want a better overall understanding of computers.

Although the content of this book is as general as possible, it is not primarily intended as a stand-alone work; instead, it's intended as an accompanying volume for use with other *Start Here!* books, which cover a range of languages and technologies. Most of the information you'll find here applies to computers and programs running Windows, but many of the concepts also apply to other operating systems, such as Unix, Linux, or OS X. Sometimes, however, specificity aids clarity, so in some places this book is targeted toward Windows.

Who Should Not Read This Book

If you're interested in general programming—particularly non-Windows and non–Microsoft .NET Framework programming—this book is not for you. Much of the information in this book applies to programming in general, but a substantial portion of the information applies to .NET Framework topics, and this book does not make any particular effort to distinguish between general and Windows- or .NET Framework–specific information.

What You Need to Use This Book

If you want to use this book, all you'll need is this book. No computer, no software, no programming language, and no programming experience is required!

Organization of This Book

If you just want a better understanding of computers and programming concepts, you can simply read the book.

If you're reading this book along with one of the other *Start Here!* books, you can take a couple of different approaches. First, you can use this book as a reference for the other. When you reach the part of *Start Here! Learn Microsoft Visual C# 2010 Programming* that discusses operators, you may want to read this book's chapter on operators for additional background. *Start Here! Learn Microsoft Visual C# 2010 Programming* may also explicitly refer to places in this book where you can get additional information on a topic.

Another approach to using this book with another *Start Here!* book would be to read this one at odd moments when it's hard to read the other one. For example, I like to use my computer to work through examples and experiment with the code as I'm learning a new language. That makes it hard for me to work through a book like *Start Here! Learn Microsoft Visual C# 2010 Programming* on the bus, waiting at the dentist's office, or while sunning myself on the beach. In contrast, *Start Here! Fundamentals of Microsoft .NET Programming* doesn't require a computer, so it's easy to read just about anywhere (although at the beach, I'd rather play volleyball anyway).

This book is divided into 14 chapters plus a glossary. The chapters are independent, so you can read them in any order. In fact, many of the sections in the chapters are independent, so you can jump around within a chapter to suit your interests and needs.

- **Chapter 1, "Computer Hardware,"** briefly describes the hardware of a computer system. It explains terms such as *computer processing unit (CPU), graphics processing unit (GPU), random access memory (RAM),* and *multi-core*, and explains why those terms are important to programmers. It explains how memory, disk accesses, and other hardware issues can affect a program's performance.
- **Chapter 2, "Multiprocessing,"** summarizes some of the challenges that face programmers writing multiprocessing programs. It explains how the future of programming is likely to be highly parallel and summarizes the Task Parallel Library (TPL) that makes programming for multi-core systems easier.

- **Chapter 3, "Programming Environments,"** explains what a programming environment is and describes some of the features that make Microsoft Visual Studio one of the best programming environments available. It explains how a program's code must be compiled and how a programming environment can make that code transparent to the programmer.
- **Chapter 4, "Windows Program Components,"** describes the pieces of a Windows program from the user's point of view. It describes menus, content menus, accelerators, shortcuts, and dialog boxes. It also mentions several design considerations that beginning programmers should understand if they want to make programs easier to use.
- **Chapter 5, "Controls,"** describes in general terms what controls and components are and how they are used. It also mentions some common properties, such as *Dock* and *Anchor,* which make using many controls easier for the programmer.
- **Chapter 6, "Variables,"** explains the concept of a variable. It explains variable concepts such as data types, conversions, strong and weak type checking, value versus reference types, scope, and accessibility.
- **Chapter 7, "Control Statements,"** describes control statements, such as *If Then* and *For Each,* which a program uses to manage a program's flow. It describes these statements in general terms, provides some examples in pseudocode, and shows a few simple examples in Visual Basic and C# for comparison.
- **Chapter 8, "Operators,"** explains operators. It discusses precedence rules and operator overloading.
- **Chapter 9, "Routines,"** explains what routines are and how they are useful in programming. It describes different kinds of routines, such as methods, subroutines, and functions. It also defines parameters and explains the confusing topic of parameters passed by value or passed by reference.
- **Chapter 10, "Object-Oriented Programming,"** provides an introduction to object-oriented programming. It explains classes, constructors, and destructors. It describes non-deterministic finalization.
- **Chapter 11, "Development Techniques,"** describes basic programming techniques such as using comments, naming conventions, interfaces, and generic classes.

- **Chapter 12, "Globalization,"** explains how to localize a program in Visual Studio so that it works in multiple places. It explains several localization issues, such as different date, number, and currency formats.
- **Chapter 13, "Data Storage,"** describes different methods for storing data such as using the registry, configuration files, and files on disk. It explains different kinds of files, such as Extensible Markup Language (XML) files and databases, and mentions some of the classes that a program would use to work with different kinds of files.
- **Chapter 14, ".NET Libraries,"** summarizes some of the libraries that are most useful in writing NET programs. These libraries let a program encrypt and decrypt information, work with data structures such as stacks and queues, interrogate objects and types to learn about them, and work with multiple threads of execution.
- **The glossary** provides a brief summary of key terms to remind you of their meaning. (In a sense, the whole book acts as a glossary for use by the other *Start Here!* books.) It summarizes key concepts in one or two sentences.

Conventions and Features in This Book

To help you get the most from the text and keep track of what's happening, I've used several conventions throughout the book.

Splendid Sidebars

Sidebars such as this one contain additional information and side topics.

Warning Boxes with a Warning icon like this one hold important, not-to-be-forgotten information that is directly relevant to the surrounding text.

Note The Note icon indicates notes and asides to the current discussion. They are offset and placed in a box like this.

Tip The Tip icon indicates tips, bits and pieces of advice on effective programming. They are offset and placed in a box like this.

More Info The More Info icon indicates somewhere you can go to learn for more information on a particular topic, such as a webpage. They are offset and placed in a box like this.

As for styles in the text:

- New terms and important words are *italicized* when they are introduced. You can also find many of them in the glossary at the end of the book.
- Keyboard keystrokes look like this: Ctrl+A. The plus sign means that you should hold down the Ctrl key and then press the A key.
- Uniform Resource Locators (URLs), code, and email addresses within the text are shown in italics, as in *http://www.vb-helper.com*, *x = 10*, and *RodStephens@vb-helper.com*.

```
Separate code examples use a monofont type with no highlighting.
Bold  text emphasizes code that's particularly important in the current
context.
```

Note The code editor in Visual Studio provides a rich color scheme to indicate various parts of code syntax such as variables, comments, and Visual Basic keywords. The code editor and the Intellisense feature of Visual Studio are excellent tools to help you learn language features in the editor and help you prevent mistakes as you code. However, the colors that you can see in Visual Studio don't show up in the code in this book.

Source Code

Because this book covers concepts that are independent of any particular programming language, it also includes little source code from any particular language. You'll find occasional bits of source code used to contrast the syntaxes of different languages; but more often, this book uses *pseudocode* to demonstrate programming constructs. Pseudocode is an informal high-level "language" that looks sort of like a programming

language, but isn't really. It's intended to describe a situation sufficiently so that you *could* implement the actual code in whatever language you are using.

For example, the following code shows a *for* loop in pseudocode, which repeats a particular operation a specific number of times:

```
For <variable> From 1 To 100
    Do something
```

This pseudocode says the program should make a variable (however you create a variable in the language you're using) and then loop starting at value 1 and finishing at value 100. For each trip through the loop, the program should "Do something."

Contrast this with the following C# code:

```
for (int i = 1; i <= 100; i++)
{
    DoSomething();
}
```

This code does the same thing as the previous pseudocode, but its syntax makes understanding the code harder—unless, of course, you know C# (or some related language, such as C++ or Java). If you don't know C#, you may have trouble understanding the point that this code is illustrating.

Acknowledgments

Thanks to Russell Jones, Diane Kohnen, Dan Fauxsmith, Jasmine Perez, and all the others at O'Reilly Media and Microsoft Press who worked so hard to make this book possible. Also thanks to John Mueller, Evangelos Petroutsos, and authors of the language-centric books in this Start Here! series. Between us I think we've put together a great set of resources!

Errata & Book Support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site:

http://www.microsoftpressstore.com/title/ 9780735661684

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is *not* offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

Other Resources

You can find more information about this book at *http://www.vb-helper.com/start_here_fundamentals.html* or at *http://www.CSharpHelper.com/start_here_fundamentals.html*. Both of these pages provide links to updates, addenda, and other information related to this book.

If you're interested, subscribe to one of my Visual Basic newsletters at *www.vb-helper.com/newsletter.html* or visit my C# blog at *blog.CSharpHelper.com*.

If you have questions, comments, or suggestions, please feel free to email me at *RodStephens@vb-helper.com* or *RodStephens@CSharpHelper.com*. I can't promise to solve all your problems, but I do promise to try to help.

Chapter 1

Computer Hardware

In this chapter:

- What the different kinds of computers are and how the type of computer being used influences the performance of various kinds of programs
- How to assess the speed of a computer and look for potential bottlenecks for different kinds of programs
- The strengths and weaknesses of different data storage devices
- How to ensure that data written to a file is saved and not discarded when a program ends or crashes
- What networks and protocols are

THE MOST ELEGANTLY WRITTEN PROGRAM IN the world is pointless (except as an esoteric work of art) if it doesn't eventually run on some sort of physical device. Often, that device is an ordinary desktop or laptop computer. Having identified the target platform, you might think you don't have to worry about hardware.

To some extent that's true, and—depending on your program—you may be able to ignore much of the computer's hardware. If you have a simple, self-contained, single-user desktop application, you also may be able to ignore the computer's power supply, fans, universal serial bus (USB) ports, Blu-ray drive, Wi-Fi antenna, sound card, microphone, and many other pieces of hardware.

Most programming languages provide high-level access to hardware such as the computer's disk drives, memory, keyboard, and mouse, so you don't necessarily need to know exactly how they work. For example, you usually don't need to know how many disk heads a disk drive has or how many revolutions per second its disks turn to read and write files.

Even though you don't always need to know all these hardware details, you should at least have some understanding of what's going on behind the scenes. For example, if you don't understand how memory use relates to paging, poor memory use can drag your entire system to a grinding halt.

This chapter explains fundamental hardware concepts that can help you get the most out of your programs. This information can help you avoid problems that can be difficult to solve after they occur.

Types of Computers

In the 1960s, computers were warehouse-sized monstrosities costing millions of dollars. The acolytes who worked with these behemoths were engineers who dealt as much with hardware as they did software, so they generally knew what equipment was present.

Today, computers can be small and inexpensive, and they are just about everywhere: on your desk, in your dentist's office, under the hood of your car, and in your phone. Despite being millions of times smaller than the now-ancient computers of 50 years ago, these new devices are millions of times more powerful.

Note The Intel 4004 processor introduced in 1971 could perform 0.07 million instructions per second (MIPS). The fastest processors today are cooled by liquid nitrogen. The IBM z196 processor, which is currently not for sale, reportedly can execute up to 50 billion instructions per second (that is, 50,000 MIPS). The Chinese Tianhe-1A supercomputer can execute 2.57 petaflops (quadrillion floating-point operations per second), although it won't fit on your desktop. See *http://www.top500.org* for the latest information about the world's fastest supercomputers.

When you design a program, you need to consider the device that will run it so you know what kinds of capabilities are likely to be present. The following sections summarize common types of computers that are available today.

Personal Computers

Personal computer (PC) is a general term for a computer intended to be used by a single person at one time, although multiple users may use it at different times. It includes other categories such as desktops, laptops, and personal digital assistants (PDAs).

Desktops, Towers, and Workstations

A *desktop computer* is intended to sit on or beside your desk and not be portable, although most are small enough these days that you can easily pick one up. Because they are not intended to be portable, they typically don't have batteries and integrated screens or keyboards.

The *all-in-one* style desktop has an integrated screen or, to look at it in another way, the computer is attached to a monitor.

A *tower* is similar to a desktop computer, but in a larger case. Its larger size makes it easier to add new hardware, although it may make it hard to fit on a desk.

A *workstation* is a more powerful desktop or tower that may have extra features, such as extra memory and disk space, multiple screens, and fancy graphics hardware for quickly performing three-dimensional (3-D) rendering.

Desktops, towers, and workstations can be quite powerful. They can have fast processors, lots of memory, big hard drives, and large monitors so they can tackle almost any task. If they are connected to a fast network, they also can provide access to centralized databases and servers. Their main disadvantages are that they are not portable, and they may be needlessly expensive for some applications, such as web browsing.

Laptops, Notebooks, Netbooks, and Tablets

A *laptop* is a computer that is intended to be portable and is used just about anywhere (except in the swimming pool). You can literally use a laptop on your lap while you are riding a bus or airplane (legroom permitting).

Because they are intended to run anywhere, laptops have integrated screens and keyboards. They often run on batteries, so power use and battery quality is very important. Heavy use of some pieces of hardware such as the graphics processing unit (GPU) and DVD drives can quickly drain the batteries.

Laptop disk and Blu-ray/DVD/CD-ROM drives are often slower and smaller than those in desktop systems, so programs that use disk files heavily may run more slowly on a laptop. Often, a desktop is faster than a laptop with the same clock speed because of the performance of devices such as these.

Laptops usually have a touchpad, pointing stick, trackball, or other pointing device. Many people find these devices harder to use than a mouse, so they add an external mouse connected to the computer.

Notebooks are basically stripped-down laptops that trade power for portability. They are thin, have relatively small screens, and are ultra-light. They rarely have DVD or CD-ROM drives, and they have fairly limited graphics capabilities.

Because they have no external media such as DVD drives, they typically have integrated network connection hardware so you can load software onto them. Network hardware also means you can use them to access the Internet.

Netbooks are even more stripped down than notebooks. They typically have less powerful processors and are intended primarily for use with networked applications such as web browsers, where most of the processing occurs on a remote server.

Note Other terms for these portable computers include *subnotebook, ultraportable,* and *mini notebook*. Many people use all these terms interchangeably.

A *tablet* computer is a portable computer similar to a laptop that uses a touchscreen or stylus as its primary input device. Tablets may display virtual keyboards on their screens and may use handwriting recognition for text input.

Laptops and tablets can have most of the same features as desktop systems (such as fast processors, lots of memory, and big hard drives) so they can handle many application needs.

Some applications can take advantage of a tablet's touchscreen, and in fact, the lack of a keyboard can be an advantage in some environments, for example at dusty construction sites, where a keyboard might let dirt into the system.

Laptops and tablets tend to be a bit more expensive than desktop systems, and some users don't like their smaller keyboards and lack of a mouse. You can overcome those limitations by adding an external keyboard and mouse if you like, although that adds further to the cost, and of course, reduces portability.

Notebooks and netbooks are often not as powerful as laptops and tablets. They are intended to be both ultra-portable and less expensive. They are often intended for running networked applications, such as web browsers. However, they do include keyboards, making them more suitable than tablets for people who need to type a lot of information.

Minis, Servers, and Mainframes

Mainframes are large centralized computers that can serve hundreds or even thousands of users simultaneously. Each user connects to the mainframe via a "dumb" terminal that has little or no processing power; the terminal simply serves as an input device and displays results generated by the mainframe.

A recent innovation that is similar to mainframe computing is *cloud computing,* where applications, data storage, collaboration services, and other key tools are stored on a centralized server that users access remotely, often through a browser. The users typically connect to the cloud services with a desktop, laptop, or other computer instead of a "dumb" terminal.

The centralized services provided by the mainframe and cloud computing allow a business to upgrade tools without modifying the users' computers. For example, a business can add more disk space or an upgrade to a centralized application on the central servers with no changes to the users' computers.

A *mini* or *minicomputer* is basically a small mainframe that can serve a dozen to a few hundred users simultaneously. Typically, a mainframe might fill a room, whereas a mini might be the size of a filing cabinet.

Supercomputer is a fairly broad term used to describe only the fastest computers. A supercomputer may act as a mainframe and support many simultaneous users, but its focus is on running one program or a few huge programs extremely quickly, rather than on performing many smaller tasks for many users. Typical mainframe applications include massive simulations for weather prediction, fluid dynamics calculations, and nuclear energy research.

Often, a supercomputer uses other computers as "front-ends." Users prepare programs for execution on a second computer. When the program's code and data is ready, it is transferred to the supercomputer for execution. The results are then returned to the secondary computer for analysis.

Server is a generic term for any computer that supports multiple users or client applications simultaneously, so supercomputers, mainframes, and minis are all servers. Mainframes are sometimes called *enterprise servers*.

Minis, servers, and mainframes are useful for applications that have large centralized databases and other resources. For example, suppose you have 100 customer service representatives who may need to interact with any customer's records. In that case, it makes sense to store all the customer information on a server. The users can either use desktop systems to access the information and work with it on their local machines, or "dumb" terminals to work directly on a mainframe.

It's worth noting that there are ways to attach dumb terminals to less powerful servers than mainframes as well.

Handheld Computers

Handheld computers are, as the name implies, computers that you can hold in your hand. These range in size from about the size of a brick to the size of a deck of playing cards.

These devices typically have small screens and may use touchscreens alone or touchscreens with a stylus for input. Some even have integrated barcode scanners and printers.

Palmtops or *pocket* computers are small handheld devices, usually with limited graphics and computing power. They are typically used to store simple information such as contact information, phone numbers, and appointment calendars.

PDAs are similar to palmtops but typically use a stylus and various forms of handwriting recognition for input.

Smartphones running the Windows Phone 7 operating system, iPhone OS (iOS), or Android are basically very small, general-purpose computers with integrated telephone features.

Some of the more powerful palmtops include other features, such as networking capabilities and may even act as music players and phones. Overall, the division between cell phones and palmtops is becoming blurred because many modern cell phones include all the features previously provided by PDAs—and even more.

Handheld computers are useful when portability is essential. For example, a telephone technician would have trouble juggling a laptop at the top of a telephone pole. They also are handy for carrying information that you want available throughout the day, such as phone numbers and appointment calendars. These devices tend to have tiny screens and keyboards, so tasks such as entering data on large forms or viewing large amounts of data can be difficult. More and more these days, smartphones have surprisingly fast processors and good graphics capabilities, however.

Comparing Computer Types

A program's use and needs influence the type of computers you should run it on. Consider the program's processing speed, network bandwidth, and screen size requirements and compare them to the features provided by different computer types.

Conversely, a computer's specific hardware can influence the kinds of programs that you can build effectively. For example, if your company has 75 users who all carry only small handheld computers, your program can't display large forms containing dozens of menus.

Computer Speed

Many people use *clock speed* as a measure of a computer's total computing power, but that term can be very misleading for a couple of reasons. To really understand why this is so, you need to know a little about how the computer processes commands.

The computer keeps all its devices synchronized by using its clock. This isn't a regular clock—it's a "clock in a chip," which keeps highly accurate time and ticks much more rapidly than a wall clock. The faster the computer's clock ticks, the more quickly the device can move on to a new task. The *central processing unit (CPU),* the computer's main processor, needs a certain number of clock ticks to execute each of its instructions. Therefore, the faster the clock ticks (that is, the "clock speed"), the more instructions the CPU can execute per second.

However, that's not the end of the story. Different processors use different instruction sets, each of which can require a different number of ticks. That means different kinds of processors may execute different numbers of instructions per second, even if they have the same clock speed. You can use clock rate to compare two of the same *kinds of processor* (for example, a 2.93-gigahertz (GHz) Intel Pentium 4 and a 3.0-GHz Intel Pentium 4) but not as an accurate comparison between two processors of different types (for example, a 3.0-GHz Intel Pentium 4 and a 3.0-GHz AMD Athlon II).

Even if you could figure out which processor executes more instructions per second, that figure alone doesn't necessarily tell you which computer will be faster *for your program.* Many programs—most, in fact—are limited by factors other than sheer processor speed, including amount and speed of memory, disk space, network speed, graphics or floating-point processor speeds, and *bus* speed.

Note The *bus* is the part of the computer that transfers data between the computer's different components such as its processor and disk drives. The USB lets a computer connect to all sorts of external devices, such as hard drives, DVD drives, keyboards, mice, graphics tablets, flash drives, cameras, and much more.

Many modern computers have multiple processors or multiple cores (execution areas within a processor), so they can perform more than one task at the same time. Whether the computer gets a significant benefit from multiple cores depends on whether the tasks it is performing can be easily

split into separate pieces—and whether the program was written to take advantage of multi-core hardware.

Many programs are limited by disk drive speed. Disk drives spin at anywhere from 3,000 RPM to 15,000 RPM (speeds between 4,200 RPM and 7,200 RPM are most typical), so the time it takes to read and write data can vary dramatically.

Which of these factors is most important for your application depends on what that application does. If your program uses a local database (that is, one stored on a hard disk attached to the computer) heavily, disk speed will be a big factor. If the database is on a remote server accessed via a network, then the speed of the server and the network's speed are probably bigger performance factors for your application than the speed of your local CPU.

The best way to determine how a computer will perform for a given program is to run that program on the computer. Unfortunately, it's often too late to fix problems after you've written the program and bought the computer.

To get an idea of how well the program will run ahead of time, focus on the system's overall performance, running a wide variety of tests rather than looking just at clock speed. To look at one set of tests in the most recent versions of Microsoft Windows, open the computer's Start menu, right-click the Computer entry, and select Properties to see the basic information display shown in Figure 1-1. (You can also right-click the Computer entry in Windows Explorer and select Properties.) The Windows Experience Index gives you a rough idea of the computer's overall performance.

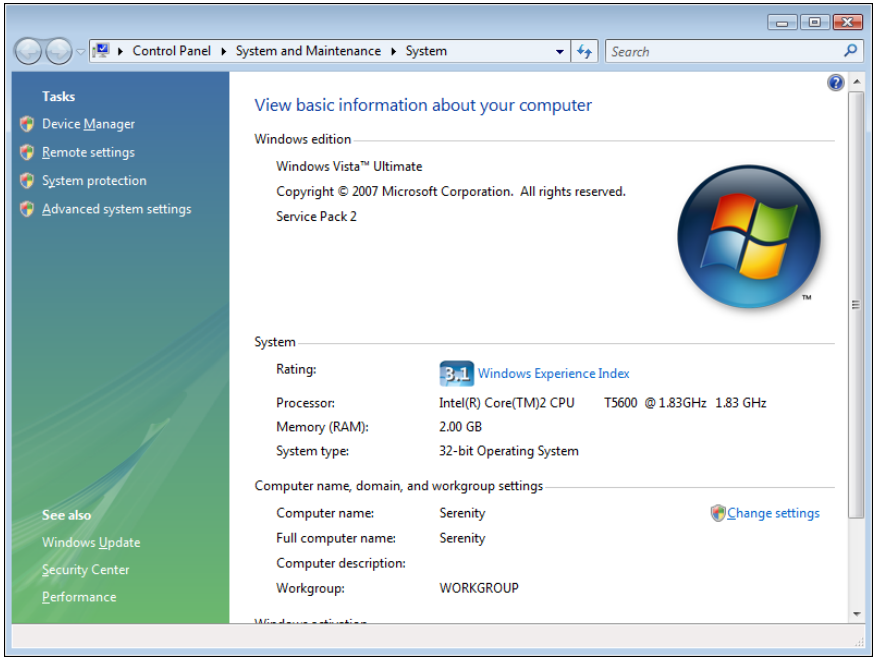

FIGURE 1-1 The Windows Experience Index gives an overview of the computer's performance.

To get more detail, click the Windows Experience Index link to see the display shown in Figure 1-2. This display shows performance scores for several different system features.

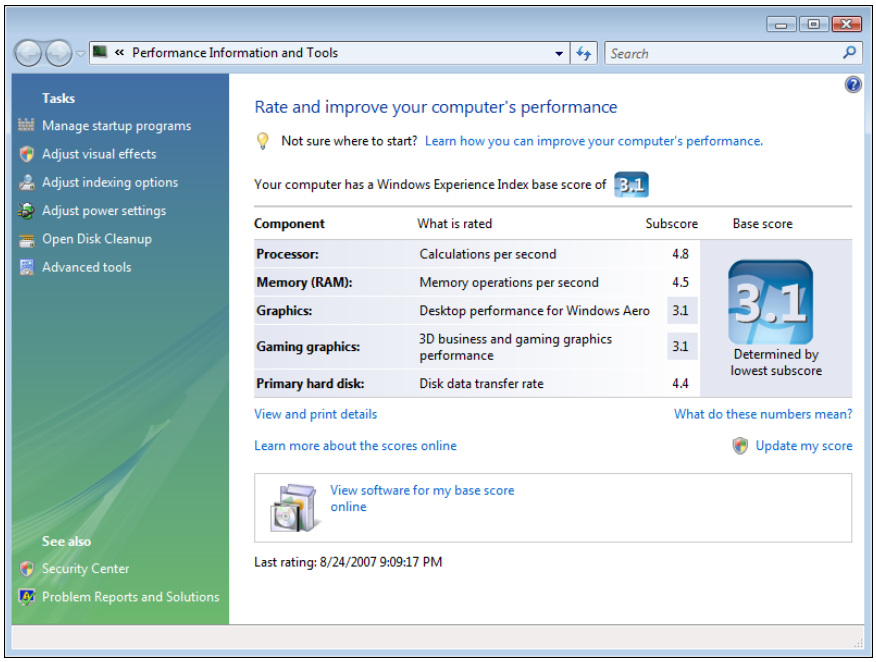

FIGURE 1-2 This display shows how well the computer performs on various tests.

On the system shown in Figures 1-1 and 1-2, the graphics scores are the lowest, so this system may not give the best performance for high-end graphics programs, such as three-dimensional games. But the processor, random access memory (RAM), and disk scores are higher, so this computer may be just fine for applications that are not graphics-intensive. (In fact, this computer works just fine for me on a wide range of applications.)

The Windows Experience Index still doesn't consider your program's particular needs. For example, it doesn't know what kinds of instructions your program will perform the most (such as integer calculations, floating-point calculations, string operations, and so on) and it doesn't consider network bandwidth, but at least it provides a reasonably consistent value that can help you compare different systems. Start there, and then consider the bottlenecks that your program is likely to encounter.

Data Storage

A program can store data in several places, including RAM, flash drives, hard drives, Blu-ray, DVDs, and CDs. The following sections describe the advantages and disadvantages of each so you can match them to your program's needs.

RAM

RAM is extremely fast, but relatively expensive. Moving data between RAM and the processor is lightning-fast, so it's the best place to store frequently used data. Data stored in program variables is generally stored in RAM, which gives them the best performance.

Unfortunately, RAM is also fairly expensive, so computers often have a limited supply. A typical computer might have 2 GB of RAM. That may seem like a lot (and it is), but you need to remember that your program isn't the only one using the RAM. Every program currently running on the system, including the operating system itself, shares it. In fact, the commands that make up the executing programs themselves also take up space in RAM.

Data must move from RAM to the processor and back for anything to occur, so what happens if the programs use up all the RAM? To work around this problem, the computer can page memory to disk.

When *paging* occurs, the computer copies a chunk of its memory onto a hard disk and frees that memory for use by other programs. Later, when a program needs to access the data in the chunk of memory that was copied to the hard disk, the computer pages it back into memory, possibly moving other data to disk to make room in RAM.

Paging lets the computer continue running even if it runs out of RAM—but that capability comes at a heavy performance price. Disk drives are much slower than RAM, so moving data to and from the disk slows the system down greatly.

This is a particular problem with programs that use huge amounts of memory. Suppose you have some complex data analysis program that loads a lot of data into memory. It then jumps around the data, performing comparisons, calculating averages, and so forth. Because the data doesn't all fit in memory at any one time, as the program jumps around in the data, it may cause very frequent paging. (This is sometimes called *thrashing*.) When this happens, you often can hear the disk drive working like crazy, and the computer's performance drops to a crawl.

You can reduce paging and thrashing by buying a computer that has lots of memory (or by adding more memory later). And you can reduce the chances of thrashing by structuring programs so they don't need to jump back and forth across huge amounts of data as often—or at all. If you can redesign a program so it uses the data in chunks, processing one chunk at a time before moving on to the next one, the program may page, but it won't thrash.

Another possibility is to free up chunks of data after using them by disposing of the variables holding the data. That makes their memory available for use by new data. In this case, the program may not page at all.

Flash Drives

Flash drives store data in solid-state memory. They have no moving parts and are *non-volatile,* meaning they don't require power to retain their data. (In contrast, RAM loses its data if it loses power.)

There are two main varieties of flash drives: USB flash drives and solid-state hard drives.

USB flash drives are small and removable. They can fit easily in your pocket, so they are great for quick backups and transferring data from one computer to another. In many ways, USB flash drives (also called USB keys) have taken the place of older floppy drives. Flash drives may last a long time, but people generally use Blu-ray, DVD, and CD drives for permanent storage instead.

Solid-state flash drives are similar to regular disk drives but use flash memory to store data instead of spinning disks. Because they have no moving parts, they are less vulnerable to vibration and shock.

Flash drives also have faster access times than hard drives. Unfortunately, they're still considerably more expensive per gigabyte than regular disk drives.

Hard Drives

Normal hard drives store data in spinning magnetic disks. They are generally slower than flash drives, although they may be faster at transferring large blocks of data. They also have *latency,* a period of time that the computer must wait while the drive is positioning itself to read a particular block of data.

Disk drives have the distinct advantage of being significantly cheaper than flash drives on a per-gigabyte basis. For example, a 240-GB solid-state drive might cost more than $500, whereas a 1-TB normal disk drive might cost only $65.

Because these drives are relatively inexpensive and can be quite large (I've seen up to 3-TB drives), they are the most common form of storage in computers today.

Blu-ray, DVD, and CD Drives

Blu-ray, DVD, and CD devices use removable spinning discs to store data. Although they're less expensive per gigabyte than USB flash drives (typically by a few cents per gigabyte), getting data back from them can be much slower than from either flash drives or hard drives.

The storage capacity of these discs varies depending on the recording format, but typical values are 700 MB for CDs, 4.7 GB for DVDs, and 25 GB for Blu-ray.

Their low cost, high capacity, durability, and removability makes these drive types well suited for backup and long-term storage of large amounts of data.

Working with Files

Before leaving the topic of data storage, I want to briefly mention an important issue related to working with files.

Disk drives naturally read and write data in large blocks. It takes just as much time to read or write an entire block as it does to read or write a single byte. To improve performance, disk drives buffer their data.

If you tell a program to read a few bytes from a file, the disk drive actually reads an entire block and stores it in a *buffer* (a temporary holding location) in memory. As you request other bytes from the same file, they may already be in memory, so the program doesn't need to fetch the new data from the comparatively slow drive.

Similarly, when you write data into a file, the drive actually stores it in a memory buffer until it has enough data to be worth writing to the physical disk.

Because the drive buffers data, it's not obvious when the drive actually writes the data to the disk. An important consequence of this is that you could lose data if a program ends or crashes before the drive has gotten around to writing the data.

To prevent this kind of data loss, your programs should always close files when you're done writing into them. (Closing input files from which the program is reading data is less critical, but still good practice.)

Networks

Computer networks—especially the Internet—play a huge role in many computer applications. Even a typical household may have its own small network connecting computers, printers, and scanners. There isn't room to cover computer networking in great depth here, but it is useful to understand some basic computer terminology.

A *computer network* is a series of connected devices that allow computers to communicate. Those devices include:

- **Network interface card (NIC)** Connects a computer to a network and provides the necessary electronics to send and receive the network's electrical signals. NICs are also called *network interface controllers, network adapters, LAN adapters,* and other similar terms.
- **Hub** A device with several ports that takes the signals that it receives and rebroadcasts them to all the ports other than the one on which it received the signal. Hubs connect multiple computers in a very simple way.
- **Bridge** Similar to a two-port hub, but with more intelligence. A bridge inspects incoming information packets from one port and forwards them to its other port only if the destination of the packet is on the other side of the bridge. This reduces unnecessary traffic on the network.
- **Switch** Similar to a bridge, but with more than two ports. Instead of forwarding signals to every port, they forward signals only to the device that should receive them.
- **Router** Similar to a switch except it can connect multiple networks, possibly using different protocols. The most common routers connect a home computer with an Internet service provider's network via a cable or modem.

Networks are sometimes categorized by their size. Two of the most common terms used to describe networks are *local area network (LAN)* and *wide area network (WAN)*.

Wi-Fi is the trademarked name of a standard for connecting devices wirelessly. *Ethernet* similarly connects devices using wires or cables.

The *Internet* is a global system of connected computer networks. It is the largest WAN, covering the entire world. Often, people use the terms *Internet* and *World Wide Web* (or just *web*) interchangeably, but the World Wide Web (WWW) is only the collection of all hypertext webpages available on the Internet. The Internet contains lots of other information as well, including email, Voice over Internet Protocol (VoIP), and files that are available for download but that are not part of the World Wide Web.

Communication over a network is controlled by various communication protocols. A *communication protocol* is a formal description of the formats and rules for passing information across a network. Protocols often include several layers. The bottommost layers deal with physical signaling and the way the network uses electrical signals to send information. Higher-level layers determine how information is translated to and from electrical signals. Still-higher levels deal with error correction and how to determine whether a message has been received correctly.

The Internet uses the Internet Protocol Suite to define how traffic should work. This suite of protocols is also called TCP/IP, named after the two most important protocols it contains: Transmission Control Protocol (TCP) and Internet Protocol (IP). TCP provides reliable delivery of a stream of bytes from one computer to another. IP provides addressing that lets a network route data packets called *datagrams* to the appropriate destination.

Two of the most common high-level protocols used on the Internet are HTTP and FTP. Hypertext Transfer Protocol (HTTP) is a protocol for hypertext documents that contain links that lead to other documents. This is the protocol that your computer uses when you open a webpage in a browser by using an address that begins with *http://*.

File Transfer Protocol (FTP) is a protocol used to transfer files between computers over a network such as the Internet. This is the protocol that your computer uses when you open a file in a browser by using an address that begins with *ftp://*. Often people use special file transfer programs to upload and download files with the FTP protocol. Addresses that begin with *http://* or *ftp://* are examples of Uniform Resource Locators (URLs).

A Uniform Resource Name (URN) is similar to a URL, but it is intended to be a permanent name for a resource even if the resource is not currently available. The difference between a URL and a URN is minor, and many people use the terms interchangeably.

More Info For more information on networking topics, search online websites such as Wikipedia and About.com, or consult a book on networking.

Summary

This chapter discussed some of the hardware issues that you should consider when you're designing and building an application. It explained how you can use the Windows Experience Index to compare computers running Windows. It also explained how you should match the needs of an application with the capabilities of a particular kind of computer.

For example, if you need to support many users on a centralized database, you might want to plan to use desktop or laptop systems connected to a server on a high-speed network. In contrast, if you want a small, special-purpose calculator to perform calculations throughout the day, you might be better off using a handheld computer or smartphone.

This chapter also described different kinds of storage hardware and explains their strengths and weaknesses. For example, hard drives are slower than RAM, but they are still reasonably quick and much cheaper, so they are a good choice for storing large amounts of data. DVDs are removable and even less expensive, so they're a good choice for backups and long-term storage.

This chapter explained how disk drives buffer input and output, meaning that to prevent data loss, you should always close files when you are done writing into them.

Finally, this chapter briefly explained some common networking concepts and terminology. It won't make you an expert on networks, but it should help you understand normal network discussions, particularly when you deal with networks from the high-level perspective that programmers usually have when writing computer programs.

This chapter also briefly mentioned multi-core systems: systems with processors that have more than one core capable of executing commands. Multi-core systems have great potential to increase performance without requiring faster processors, something that is becoming increasingly difficult to achieve. The next chapter contains more information about multi-core systems in particular and multiprocessing in general.

Chapter 2

Multiprocessing

In this chapter:

- What multiprocessing is and how modern computers can provide it
- The difference between multiprocessing and multitasking
- What processes and threads are
- How to design programs that can take advantage of multiprocessing

MOORE'S LAW, NAMED AFTER INTEL COFOUNDER Gordon E. Moore, says that the number of transistors that can be placed on a chip roughly doubles every two years, and that leads directly to an increase in computer speed. The law has held up remarkably well for more than 40 years and is predicted to continue to hold for at least a few more years, but chip manufacturers are starting to reach the physical limitations of what's possible using current chip fabrication techniques. This might spell the end to large speed improvements for individual chips, but it doesn't necessarily mean the end of performance gains for computers.

Note For more information on Moore's Law, see *http://en.wikipedia.org/wiki/Moore%27s_law*.

Other techniques, such as writing better code and leaner operating systems, can make a computer faster without changing its underlying hardware. One particularly promising approach to improving computer performance is multiprocessing.

This chapter describes multiprocessing and explains how you can take advantage of it to get the best performance possible.

Multitasking

Even the slowest computers are much faster than their human users. A typical computer spends practically all its time sitting around twiddling its electronic thumbs waiting for the user to do something. When the user presses a button or clicks the mouse, the computer springs into action, performs a task, and then goes back to waiting.

For example, the world record for fastest typing was set by Barbara Blackburn at 212 words per minute, or about 18 characters per second. Not even the world's fastest typists can keep up with a computer that can execute millions of instructions per second.

To make better use of the computer's blinding speed, modern operating systems multitask. In *multitasking,* the computer runs several tasks (known as *processes*) in turn. The operating system lets one process execute for a while so it can perform calculations, update its display on the screen, respond to user events such as button clicks, and so on. The operating system then pauses that process and lets another one take a turn. It continues rotating through the processes so they each get to execute.

So long as the operating system can switch the processes quickly enough, they appear to the user as if they are all executing simultaneously, although they are really just taking turns. This works well so long as the system doesn't have too many intensive processes, but if some of the processes are performing really heavy-duty calculations, the computer may have trouble maintaining the illusion that it's running simultaneous tasks.

This is where multiprocessing enters the picture. Multitasking fosters the illusion that the computer is performing several tasks at once. In multiprocessing, the computer really is doing several things simultaneously.

Multiprocessing

In multiprocessing, a computer uses multiple execution elements to perform several tasks at the same time. Those elements could be separate processors running on separate chips or, as is increasingly common these days, they can be separate cores within the same processor. A *core* is the part of a processor that actually executes commands. By putting more than one core on the same chip, a computer can greatly increase its potential computing power.

Today, two or four core computers are common, processors with six or eight cores are also available, and one experimental processor contains more than 1,000 cores! (To learn more about this innovative computer, see *http://www.physorg.com/news/2011-01-scientists-cores-chip.html*.)

With the end of Moore's Law looming over the horizon, these sorts of multi-core systems offer a potential road to increased performance, but multiple cores do not guarantee that applications will run faster. The operating system itself may be able to run different programs on different cores, but a single program could become stuck on a single core and have limited performance. You can allow a single program to run on multiple cores by using multiple threads.

Multithreading

A *process* is an instance of a program running on a computer. (Note that you could have multiple instances of the same program running. For example, you might have two browsers open or two instances of WordPad running.) A *thread* is a sequence of instructions within a single process that may execute in parallel with other threads. Sometimes you can execute multiple threads within the same process at the same time. Each thread keeps track of its position within the program's code and can move through the code as it needs to without interfering with the other threads. This is called *multithreading*.

For example, suppose you write a program that takes a stock's historical prices, performs some sort of complex statistical calculation, and predicts the stock's future price. (If you can get that last part to work reliably, let me know!) Now suppose you want to perform the same task for several stocks. You could have the program perform the calculations sequentially, one after another. If each calculation takes about 30 seconds and you want to predict prices for 10 stocks, the total time will be around 300 seconds, or 5 minutes.

Another approach would be to start 10 threads, one for each stock. A thread would perform the statistical calculation for its stock and display the result.

A single-CPU system will multitask, switching quickly back and forth between the threads to give the illusion that they are all executing at the same time. There is still only one CPU, however, so the total time will still be around 5 minutes. In fact, there is a little bit of overhead in switching between threads so the total run time may be slightly longer.

In contrast, a computer with multiple cores may truly be able to execute more than one thread at a time. In that case, the total time will be roughly the original total time of 5 minutes divided by the number of cores, plus some overhead for setting up the threads and keeping track of what they are all doing. A two-core system might require about 2.5 minutes, whereas a four-core system might need only around 1.25 minutes to finish the calculations.

Unfortunately this speed improvement isn't automatic or free. In addition to a small (but significant) amount of overhead to set up threads, a program may pay a large performance penalty if the threads interfere with each other. Interference can take the form of several different potential problems with parallelism.

Note Some compilers may be able to detect pieces of code that can always execute safely in parallel and in that case you may gain some benefit from multiple cores without any additional work. To get the full benefit, however, you need to structure your program properly.

Problems with Parallelism

At a high level, running threads in parallel is easy to understand. When you look closely at specific tasks, however, you can encounter several problems. Some of these include contention for resources, races, and deadlocks.

Contention for Resources

Sometimes multiple threads need to use the same resources. Consider again the stock calculator example. Suppose the program starts 10 threads to perform calculations for 10 stocks. The first task that each thread must perform is using the Internet to get its stock's price data. If your network bandwidth is limited, this will be a big bottleneck as each thread demands access to the network. Even if your network has plenty of bandwidth, the website that you access to get the stock prices needs to process all the requests and, if it's a slow website, that may cause a bottleneck.

Similarly, multiple threads may need to access the same disk drive, CD or DVD drive, or other limited resource, and performance can be limited as a result. It's bad enough that these sorts of contention can limit performance, but they can also cause incorrect behavior. The most common example of this kind of error is called a *race condition*.

Race Conditions

A *race condition* occurs when the result of a calculation depends on the exact sequence or timing of execution in multiple threads.

For example, suppose you want to compute the total of 2 million numbers. You could loop through the numbers and add them up one at a time, but you want to save time with multithreading, so you break the task into two pieces and solve each piece in a separate thread. The first thread adds the first million numbers to a value called *total* and the second thread adds the second million numbers to *total*. The basic algorithm for each thread looks like this:

```
For i = start To finish
    Get total
    Calculate result = total + value[i]
    Save result In total
```

This code enters a loop where the looping variable *i* starts at the value start and runs to the value *finish*. In other words, it takes the values *start, start* + 1, *start* + 2, . . ., *finish*.

The values *start* and *finish* represent the indices of the values that a thread should process. In this example, the first thread's values for *start* and *finish* would be 1 and 1,000,000, and the values for the second thread would be 1,000,001 and 2,000,000. The two threads run exactly the same code; only the values' *start* and *finish* are different for the two threads.

Inside the loop, each thread reads the current value of the *total* variable, adds the value pointed to by the current value of *i* to *total*, and saves the new result in *total*.

If you're running a single thread to process all the values, this code works perfectly. However, if you use two threads running at the same time, they can enter a race condition. Consider this sequence of events as the two threads execute inside their loops.

```
Thread 1:    Get total
Thread 2:    Get total
Thread 1:    Calculate result = total + value[i]
Thread 1:    Save result In total
Thread 2:    Calculate result = total + value[i]
Thread 2:    Save result In total
```

In this case, both threads start by reading the value *total*. Because thread 1 does this right after thread 2 does it, both threads get the same value.

Next, thread 1 adds a value to *total* and saves the result back in the value *total*. Then thread 2 does the same. Because thread 2 still has the original value for total, it overwrites the new value saved by thread 1.

For a concrete example, suppose *total* starts with the value 100 and the two threads are adding the values 20 and 30, respectively. Both start by reading the value 100. Thread 1 then adds 20 and saves the result 120 in the value *total*. Next, thread 2 adds 30 to the value that it originally read (100), gets the result 130, and saves it in the value *total*. Instead of holding the correct result 150, *total* now holds 130.

Warning Race conditions can be extremely difficult to detect because bugs appear only when events occur in exactly the right order. If the sequence of events that leads to the error is unlikely, you may run a program thousands of times before you encounter the error. When the error does occur, you may be unable to reproduce the exact sequence of events in multiple threads that caused it.

One way to prevent a race condition is to use a lock on the critical section of code.

Locks

A lock guarantees that a thread has exclusive access to a piece of code, memory, or other item that it needs to prevent a race condition or other bug. In the previous example, a thread could use a lock to gain exclusive access to the variable total while performing its calculation. The new code looks like this:

```
For i = start To finish
    Lock total
        Get total
        Calculate result = total + value[i]
        Save result In total
```

Now, if two threads are running at the same time, one cannot read the value of *total* while the other has it locked so it cannot interfere with the other thread. Instead, it waits until the lock is released, and then it locks the *total* value and performs its own calculation without interference.

Unfortunately, locks add considerable overhead to a program because making multiple threads coordinate in this way makes them much slower. The more locks a program must make and release, the more slowly the program will execute. In this example, the threads must lock and unlock the value *total* for each of the 2 million numbers that should be totaled.

In this particular program, the problem is even worse because each thread performs calculations only while it's running code inside the lock; therefore, no two threads can be doing anything significant at the same time, which eliminates all the benefits of multithreading. The result is that this program really performs its calculations one at a time sequentially, just spread across multiple threads in a complicated way using 2 million locks. All those locks guarantee that this program will be *much* slower than the original single-threaded program, which didn't need to use any locks.

Locks solve one problem but sometimes cause another: deadlocks.

Deadlocks

A *deadlock* occurs when two threads are waiting for resources held by each other. For example, suppose thread 1 has resource A locked and is waiting for resource B, but thread 2 has resource B locked and is waiting for resource A. Neither thread can get the second lock it needs, so it cannot continue. Because of this, neither thread will release the lock that it already holds, so they're both stuck.

In this example, the deadlock is simple and easy to avoid by making both threads lock resource A before locking resource B. Then, if thread 1 locks resource A, thread 2 cannot lock resource B until it first locks resource A.

Detecting and breaking deadlocks in more general situations can be harder. If a program has many threads that need exclusive access to lots of resources in complex combinations, it can be difficult to prevent deadlocks.

Looking for Parallelism

Some problems have naturally parallel solutions. For example, consider the Mandelbrot set shown in Figure 2-1. To produce this image, the program considers each pixel in the result separately. For that pixel, it performs a series of calculations that do not involve the other pixels in any way. This program can make as many threads as it wants, and each can work independently to calculate a color for its own pixel.

The only place where the threads need to interact is when they copy their results into the single final image. Even there, the threads don't need to use locks because each thread works with a different pixel and doesn't need to look at or modify the other pixels.

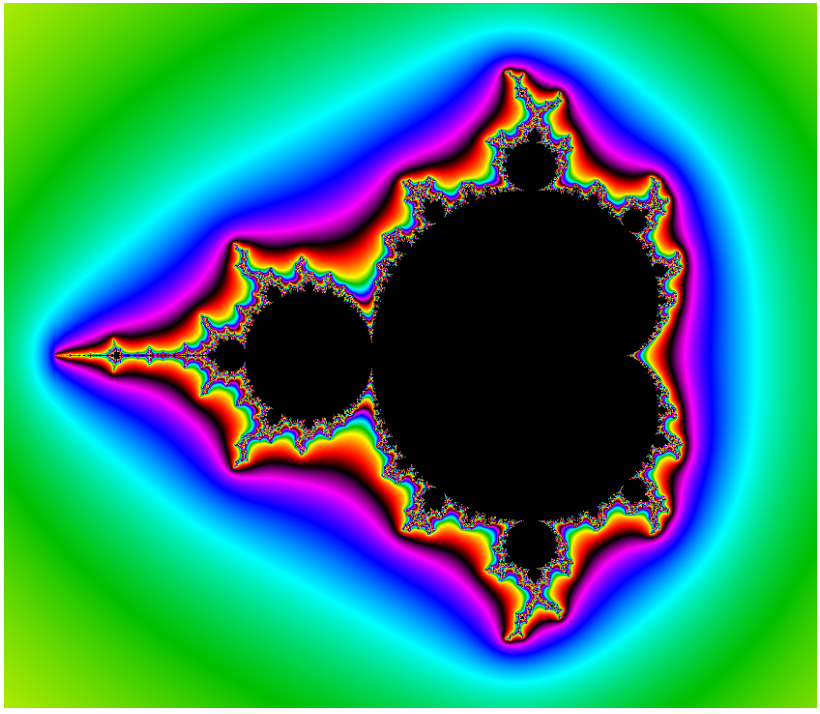

FIGURE 2-1 Displaying the Mandelbrot set is an embarrassingly parallel task.

This kind of algorithm, which is naturally parallel, is sometimes called *embarrassingly parallel*. Other embarrassingly parallel problems include ray tracing, generating frames for an animated movie (which may also involve ray tracing), some artificial intelligence approaches such as genetic algorithms, and random heuristics where the program picks a random solution and evaluates its effectiveness.

Even if a problem isn't embarrassingly parallel, you may be able to come up with a workable parallel solution. For example, consider the earlier problem of adding up 2 million numbers. The simple solution of making two threads each add up half of the numbers doesn't work because they spend a huge amount of time competing for access to the value *total*.

However, suppose each thread added up its own subtotal and only copied the result into *total* when it was finished. The new thread code looks like this:

```
subtotal = 0
For i = start To finish
    Get subtotal
    Calculate result = subtotal + value[i]
    Save result In subtotal
Lock total
    Get total
    Calculate result = total + subtotal
    Save result In total
```

In this version of the program, the loop where most of the work occurs contains no locks, so the threads can work independently. Only at the end do the threads need to lock the value *total*. The previous version of the program required 2 million locks, and those locks prevented the threads from running in parallel. This version uses only two locks and the threads can execute the vast majority of their calculations in parallel, so this version will be much faster.

Note This code was written with the assumption that the threads can access the values they are adding without interfering with each other, and that may not always be the case. If the values are stored on a disk drive, reading one value may move the disk heads, so it takes longer to read other values. If the values are all together on the disk, however, the program can probably read them all into memory at once, and then the threads can work in parallel without disk contention.

This example shows how the approach you use can determine whether a program will benefit from multithreading. The key to making this solution work is avoiding locks. A good multithreaded application doesn't use too many locks, avoids making threads contend for other scarce resources such as disk drives, and generally keeps calculations as separate as possible, as long as possible. Often, a thread's contribution to the overall solution is used at the end of the thread's calculations.

Distributed Computing

In *distributed computing,* multiple computers linked by a network work together to perform a task. You can think of distributed computing as similar to multithreading except that the "threads" run on different computers.

Although coordination among threads on the same computer can be cumbersome, communication among computers in a distributed application is much slower. That means distributed computing is most useful when the problem is embarrassingly parallel. For example, several computers could be assigned the task of generating separate frames for an animated movie. Those computers could then use multithreaded ray tracing programs to calculate the pixels in each frame.

Other examples of distributed computing are "grid computing" applications, which use idle computers scattered across the Internet to perform CPU-intensive calculations while their users aren't using them. Some of these efforts involve thousands of (or even millions of) computers. Examples include *SETI@home,* which tries to detect alien signals in vast amounts of radio signal data; *Folding@home,* which simulates protein folding and molecular dynamics to study diseases; and the Great Internet Mersenne Prime Search (GIMPS), which tries to find Mersenne primes of the form $2^p - 1$ for some number *p*.

Note Currently, only 27 Mersenne primes are known. The largest known prime of any kind is the Mersenne prime $2^{43,112,609} - 1$.

Distributed computing is a specialized subtopic in a specialized field, but some of its basic ideas can be very useful when designing multithreaded (or even single-threaded) programs. One of the most important of those ideas is that each of the cooperating programs should be as independent as possible. If thousands of computers need to communicate frequently with each other or with a central computer, the network's communications needs will quickly outweigh any potential benefits.

Similarly, keeping each thread as separate as possible (avoiding direct communication between them and avoiding locks) makes threads faster, easier to debug, and more scalable so you can easily add more if necessary.

Even if your computer has only a single core, breaking operations up into independent pieces makes writing and debugging the pieces easier. If the pieces are self-contained, then you can debug one without worrying as much about how changes to it will affect other pieces of code.

Keeping the pieces as separate as possible also can help you rewrite the program later if you decide to spread it across multiple threads.

Task Parallel Library

How you create multiple threads depends on the operating system and language you are using. Microsoft's Task Parallel Library (TPL) is a specific library of tools that makes running parallel threads relatively easy for Microsoft .NET applications.

The following list summarizes the main tools provided by the TPL:

- **Parallel.Invoke** Executes several pieces of code at the same time.
- **Parallel.For** Executes the same pieces of code several times in parallel, with different numbers as parameters. For example, it might invoke some code to produce frames in an animated movie where the parameters 1, 2, 3, and so on are passed to the code so it knows which frame number to generate.

- **Parallel.ForEach** Executes the same pieces of code several times in parallel, with different arbitrary values as parameters. This is similar to *Parallel.For* except that the code receives arbitrary values specified by the program as inputs rather than numbers in a sequence. For example, the program could pass each thread a separate image to manipulate. The threads could then perform image processing techniques on the images, creating embossed images.

These three TPL operations provide a relatively simple way for a program to use multiple threads. These may not handle every scenario that you can imagine, and threads still may run into race, lock, deadlock, and other parallel issues, but these are fairly easy to use.

The TPL is also designed to use multiple cores, if they are available, without imposing too much overhead on single-core systems, so you can run the same program on different computers and expect reasonable performance, whether the computer has 1 core or 16.

Summary

This chapter discussed multiprocessing and the ways modern computers provide parallel computing, or at least an illusion of it. All modern computers provide multitasking, quickly switching back and forth among applications to make it seem as if they are all running at the same time.

Some computers have more than one element that can execute instructions, and those computers can perform multiprocessing, truly executing multiple tasks at the same time. Those computers may have multiple processors, or they may have multiple cores within the same processor.

Multi-core systems are becoming increasingly common. To get the best performance from your programs, you need to consider parallel programming issues as you write your programs. If you keep the individual parts of a program as separate as possible, you may be able to execute them on different threads running on separate cores, which will improve performance.

Even if you don't plan to run a program across multiple threads, keeping the various elements of your programs as separate as possible makes them easier to write and debug.

Chapter 1, "Computer Hardware," described a range of computer hardware that you might use, such as desktops, laptops, and smartphones. This chapter described topics in parallel programming that you can use to make computer software take advantage of the processors and cores provided by the computer's hardware. The next chapter bridges the gap between the topics of hardware and software, explaining how programming environments translate the software that you write into commands that the hardware can actually execute.

Index

Symbols

^ operator, 107, 109, 111
^= operator, 110
- operator, 108
-= operator, 110
! operator, 108
!= operator, 109
? operator, 113
?: operator, 109
* operator, 108
*= operator, 110
/ operator, 108
/= operator, 110
\ operator, 108
\= operator, 110
& operator, 109, 112
&& operator, 109
&= operator, 110
% operator, 108
%= operator, 110
+ operator, 108
+= operator, 110
< operator, 109
<< operator, 109, 112
<<= operator, 110
<= operator, 109
<> operator, 109
= operator, 109
== operator, 109
> operator, 109
>= operator, 109
>> operator, 109, 112
>>= operator, 110
| operator, 109, 112
|= operator, 110
|| operator, 109
~ operator, 108, 111
--x operator, 108, 111
++x operator, 108, 111

A

A.B operator, 108
About command, 38
abstract classes, 151
abstraction, 154–156
accelerators (with menus), 34–35
Accept button, 43
accessibility, 87–89
  of routines, 136
accessors, 144
addition, 106
additive operators, 109
agile development, 177–178
A(i) operator, 108
A[i] operator, 108
AL register, 25
Alt key, 34
Anchor property, 61, 62
AndAlso operator, 109
And operator, 109
AppendText method, 66
arrays, 75
arrays, parameter, 130, 134
artificial intelligence, 21
assemblers, 26
assembly language, 26
Assert method, 179
Assignment operators, 110
auto-hiding windows, 30
automatic documentation from, 171
AutoSize property, 61

B

BackColor property, 61
BackgroundImage property, 61
Background property, 63
BackgroundWorker control, 52
backing field, 144
BindingNavigator control, 52
BindingSource control, 52
bits, 72
bit shift operators, 109
bitwise operators, 111–112
Blackburn, Barbara, 16
block comments, 169
Blu-ray, 10
Boolean data type, 73
BorderBrush property, 63
Border control, 57, 60
BorderStyle property, 61
BorderThickness property, 63
break keyword (C#), 101
bridges, 11
buffer, memory, 11
build automation tools, 28
bus, 6
Button control, 52, 57
bytecode, 26
Byte data type, 73
bytes, 72

C

C#, 26
  arrays in, 75
  auto-implemented properties in, 145
  break keyword in, 101
  comments in, 127
  concatenation operators in, 114
  enumerations in, 76
  For loop in, 92
  getter/setter methods in, 144
  hiding implementation details in, 122
  implicit conversion in, 83
  operator overloading in, 114
  parameter arrays in, 130
  parameter-passing methods in, 131
  persistent variables in, 89
  routines in, 120
  structures in, 77
C++, 26
  comments in, 127
calling (of routines), 123–124
call stack, 123
call stack window (Visual Studio), 30
Cancel button, 43
Canvas control, 57
Case statement, 97–99
catching the error, 103
Catch keyword, 103
C, comments in, 127
CD drives, 10
central processing unit (CPU), 6
Char data type, 73
CheckBox control, 52, 57
CheckedListBox control, 52
Checked property, 61
CIL. *See* Common Intermediate Language
CInt function (Visual Basic), 83
class(es), 77–78, 142
  constructors in parent, 163
  as reference types, 79
  structures vs., 78
Clear method, 66
Click event, 67, 68
clock speed, 6
cloud computing, 4
CLR. *See* Common Language Runtime
code
  reducing duplicated, 121
  reusing, 121
  simplifying complex, 122
code editors, 28
code reuse
  inheritance as, 148
  polymorphism as, 149
code tag (Visual Studio), 172
coercion, 83
ColorDialog control, 52
ComboBox control, 52, 57
commands
  disabling vs. hiding, 38
  multiple ways to invoke, 42
  separators between, 40
comma-separated value (CSV) files, 192
comments, 167–173
  block, 169
  for routines, 127–128
  types of, 169
  XML, 170–173

Common Intermediate Language (CIL), 26
Common Language Runtime (CLR), 26
communication protocols, 12
comparison operators, 109
compilers, 28
ComplexNumber class, 116
components, program, 49
compound assignment operators, 114
computer-aided design (CAD), 38
computer hardware, 1–14
  and data storage, 8–11
  and networks, 11–12
  and speed, 6–8
  types of computers, 2–6
computers, 2–6
  comparing types of, 6
  desktops, 2–3
  handheld, 5
  laptops, 3–4
  mainframes, 4–5
  minis, 4–5
  netbooks, 3–4
  notebooks, 3–4
  personal computers, 2
  servers, 4–5
  tablets, 3–4
  towers, 2–3
  workstations, 2–3
computer speed, 6–8, 16
concatenation operator, 109
concatenation operators, 114
conditional logical operators, 112–113
conditional operator, 109
conditional operators, 113
conditional statements, 96–99
  Case statement, 97–99
  Else If statement, 97
  If Else statement, 97
  If statement, 96
configuration files (config files), 197–199
constructors, 161–162
container controls, 64
Content property, 63, 64
ContextMenu property, 61
context menus, 40–41
ContextMenuStrip control, 52
Continue statement, 101–102, 102
controls, 49–70
  custom, 28
  events used with, 67–69
  grouping related, 44–46
  methods used with, 66–67
  order of, 44
  properties of, 60–66
  using, 51–52
  for Windows Forms, 52–56
  WPF, 57–60
control statements, 91–104
  conditional statements, 96–99
  and error handling, 103
  jumping statements, 99–102
  looping statements, 93–96
  and pseudocode, 92–93
conversion, data type, 82–84
conversion operators, 116
Convert class, 188
copying structures, 82
Copy method, 66
cores, 6
  multiprocessing and, 16
CPU. *See* central processing unit
CSV. *See* comma-separated value files
c tag (Visual Studio), 172
culture codes, 184
currency, culture-specific values for, 187
Cursor property, 61
custom controls, 28
Cut method, 66

D

databases
  and computer speed, 7
  hierarchical, 203–205
  network, 205–206
  object-relational, 203
  object stores, 203
  relational, 200–202
  spreadsheets, 202–203
  temporal, 206
data-centric viewpoint, 175–176
DataGridView control, 52
DataSet control, 52
data storage, 8–11, 191–208
  Blu-ray, 10
  CD drives, 10
  DVDs, 10
  files, 192–199

data storage *(continued)*
flash drives, 9–10
hard drives, 10
hierarchical databases, 203–205
network databases, 205–206
object-relational databases, 203
object stores, 203
RAM, 9
relational databases, 200–202
spreadsheets, 202–203
system registry, 199–200
temporal databases, 206
and working with files, 10
data type (of variable), 72
data types
conversion of, 82–84
fundamental, 71–73
program-defined, 74–78
Date data type, 73
DateTimePicker control, 52
deadlocks, 20
Debug class, 179
debuggers, 28
debugging, 123
comments and, 127
Decimal data type, 73
declarations, 72
dependency graphs (Visual Studio), 30
deriving, 147
description tag (Visual Studio), 172
design time, 28
desktop computers, 2–3
destructors, 163–165
development techniques, 167–182
agile development, 177–178
comments, role of, 167–173
data-centric, 175–176
extreme programming, 178
naming conventions, 173–175
test-driven development, 179–180
user-centric, 176–177
Diagnostics namespace, 179
dialog boxes, 43
menu hierarchies vs., 39
DirectoryEntry control, 52
DirectorySearcher control, 52
disk drives
reading from and writing to, 10
speed of, 7
distributed computing, 22–23
division, 106
division operator, 112
DockPanel control, 57
Dock property, 61, 62
documentation, automatic, 171
document object model (DOM), 196
Document Type Definition (DTD), 197
DocumentViewer control, 57
DOM. *See* document object model
DomainUpDown control, 53
DoubleClick event, 68
Double data type, 73
Do While loops, 95
DragDrop event, 68
DragEnter event, 68
DragLeave event, 68
DragOver event, 68
DrawToBitmap method, 66
DTD. *See* Document Type Definition
dumb terminals, 5
duplicated code, 121
DVDs, 10

E

Ellipse control, 57
ellipses (in menus), 34
Else block, 97
Else If block, 97
Else If statement, 97
embarrassingly parallel algorithms, 21
Enabled property, 61
encapsulation, 142
Enter event, 68
enumerations, 75–77
equality operators, 109
error handling, in control statements, 103
ErrorProvider component, 49
ErrorProvider control, 53, 56
errors, 103. *See also* user errors
event handlers, 67
EventLog control, 53
events, 67–69, 146
example tag (Visual Studio), 172
exception tag (Visual Studio), 172
Exit command, 34
Exit Function statement, 102
Exit statement, 101, 102
Exit Sub Function statement, 102

Expander control, 57, 60
explicit conversion, 82–83
exponentiation, 108
Extensible Application Markup Language (XAML), 65, 213
Extensible Markup Language (XML), 170. *See also* XML comments
  file editing in, 30
eXtensible Stylesheet Language for Transformations (XSLT), 197
external keyboard, 4
extreme programming, 178

F

factory methods, 146
fields, 143
File menu, 34
files, 192–199
  config, 197–199
  INI, 193–194
  random access, 193
  text, 192
  working with, 10
  XML, 194–197
FileSystemWatcher control, 53
File Transfer Protocol (FTP), 12
Finalize routine, 163
FindAll method, 67
FindOne method, 67
flash drives, 9–10
floating point data types, 73
FlowLayoutPanel control, 53, 56
Focus method, 66
FolderBrowserDialog control, 53
Folding@home, 23
FontDialog control, 53
FontFamily property, 63
Font property, 61
FontSize property, 63
FontStyle property, 63
FontWeight property, 63
For Each loops, 94–95
ForeColor property, 61
Foreground property, 63
For loops, 93
  C#, 92
  pseudocode for, 92
  Visual Basic, 92
formats, locale-specific, 186–187
FormClosed event, 68
FormClosing event, 68
forms
  control order in, 44
  designing, with Visual Studio, 51
  error flags in, 48
  grouping related controls in, 44
Fortran, 26
fractals, 137
Frame control, 57
Friend keyword, 87, 136
FTP. *See* File Transfer Protocol
functions, 120
F(x) operator, 108

G

garbage collector (GC), 164
GetError method, 67
getter method, 144
GetToolTip method, 67
GIMPS. *See* Great Internet Mersenne Prime Search
global issues, 183–190
  culture-aware functions in .NET, 187–189
  culture codes, 184
  formats, locale-specific, 186–187
  terminology, 184
  text/symbols, locale-specific, 184–185
  Visual Studio, user interfaces in, 185–186
Go To statement, 99–100, 102
  restrictions on, 100
Great Internet Mersenne Prime Search (GIMPS), 23
"grid computing" applications, 23
Grid control, 57, 64
GridSplitter control, 57
GroupBox control, 53, 56, 57

H

handheld computers, 5
hard drives, 10
hardware. *See* computer hardware
Has-A relationships, 158
Height property, 63
Help menu, 38
HelpProvider control, 53

hierachies, menu, 39
hierarchical databases, 203–205
hints, providing, 47–48
HScrollBar control, 53
HTTP. *See* Hypertext Transfer Protocol
hubs, 11
Hypertext Transfer Protocol (HTTP), 12

I

IBM z196 processor, 2
icons, 36
IDisposable interface, 164
If Else statement, 97
If statement, 96
    Continue statement vs., 102
Image control, 58
ImageList control, 53
Image property, 61
immutability, 78
implementation details, hiding, 122
implicit conversion, 83–84
include tag (Visual Studio), 172
inheritance diagrams, 152–154
inheritance(s), 147–148
    multiple, 158–160
INI files, 193–194
instance members, 146–147
instance (of a class), 142
instantiation, 142
Integer data type, 73
integer division operators, 108
integrated development environments (IDEs), 29. *See also* Visual Studio
Intel 4004 processor, 2
IntelliSense, 30, 170–171
interfaces
    localization of, in Visual Studio, 185–186
    multiple, 158–160
internal keyword, 87, 136
internationalization, 184
Internet, 12
Internet Explorer 7
    menus in, 36
Internet Protocol (IP), 12
Internet Protocol Suite, 12
intern pool, 74
Invalidate method, 66
IP. *See* Internet Protocol
irrational numbers, 116
Is-A relationships, 158
Items property, 61
item tag (Visual Studio), 172

J

Java, 26
    comments in, 127
Java bytecode,, 27
Java Virtual Machine (JVM), 27
jumping statements, 99–102
    Continue statement, 101–102
    Exit statement, 101
    Go To statement, 99–100
    Return statement, 102
Just-In-Time (JIT) compilation, 26
JVM. *See* Java Virtual Machine

K

keyboards, external, 4
KeyDown event, 68
KeyPress event, 68
KeyUp event, 68
Kill method, 67

L

Label control, 53, 58
LAN adapters, 11
LANs. *See* local area networks
laptops, 3–4
layout (of forms), 44
LayoutTransform property, 63
Leave event, 68
lifetime, 88–89
LinkLabel control, 53
ListBox control, 53, 58
listheader tag (Visual Studio), 172
lists, Rule of Seven for, 46
list tag (Visual Studio), 172
ListView control, 53, 58
Load event, 68
local area networks (LANs), 12
locales, 184

localization, 184
formats, locale-specific, 186–187
text/symbols, locale-specific, 184–185
Visual Studio, user interfaces in, 185–186
Location property, 61
locks, 19–20
logical operators, 109
Long data type, 73
long date, culture-specific values for, 187
looping statements, 93–96
Do While loops, 95
For Each loops, 94–95
For loops, 93
Until loops, 95–96
While loops, 95
loops, variables in, 94

M

mainframes, 4–5
Mandelbrot set, 21
Margin property, 63
MaskedTextBox control, 53
MaxHeight property, 63
MaximumSize property, 61
MaxWidth property, 63
MediaElement control, 58, 60
members
overriding, 149–151
shadowing, 151–152
memory
copying and, 81
paging of, 9
memory buffer, 11
Menu control, 58
menus, 34–40
accelerators with, 34–35
context, 40–41
disabling vs. hiding commands in, 38
ellipses with, 34
example, 40
length of, 39–40
pop-up, 40
shallow hierarchies for, 39
shortcuts with, 35–36, 41
standard menu items, 36–38
in Visual Studio, 30
MenuStrip control, 53, 56
Mersenne primes, 23
MessageQueue control, 54
methods, 66–67
in object-oriented programming, 145
routines vs., 120
method scope, 85
Microsoft, 50
Microsoft.CSharp, 210
Microsoft namespaces, 210
Microsoft .NET Framework version 3.0, 50
Microsoft.VisualBasic, 210
Microsoft.Win32, 210
Microsoft.Windows.Themes, 210
million instructions per second (MIPS), 2
MinHeight property, 63
minicomputers, 4–5
MinimumSize property, 61
mini notebooks, 3
MinWidth property, 63
MIPS. *See* million instructions per second
modal dialog boxes, 43
modeless dialog boxes, 43
Mod operator, 109
modulus operator, 108, 109
MonthCalendar control, 54
Moore, Gordon E., 15
Moore's Law, 15
mouse, 3
MouseClick event, 68
MouseDown event, 68
MouseEnter event, 68
MouseHover event, 68
MouseLeave event, 68
MouseMove event, 68
MouseUp event, 68
MultiColumn property, 61
multi-core systems, 13, 16
multi-line comments, 127
MultiLine property, 61
multiplication, 106
multiplicative operators, 108
multiprocessing, 15–24
about, 16
and distributed computing, 22–23
multitasking vs., 16
multithreading, 17
and parallel solutions, 21–22
and problems with parallelism, 18–20
Task Parallel Library and, 23–24
multitasking, 16
multithreading, 17

multi-way branch, 97
MustInherit keyword, 151
MustOverride keyword, 151
MyBase keyword, 163
MyClass keyword, 162

N

Name property, 51
names (for routines), 126
namespaces
  Microsoft, 210
  System, 210–213
naming conventions, 173–175
narrow values, 84
netbooks, 3–4
.NET Framework libraries, 209–214
  Microsoft namespaces in, 210
  System namespaces in, 210–213
network adapters, 11
network connection hardware, 3
network databases, 205–206
network interface card (NIC), 11
network interface controllers, 11
networks, 11–12
New (new) keyword, 79
NIC. *See* network interface card
notebooks, 3–4
NotifyIcon control, 54
Not operator, 108
NumericUpDown control, 54
numeric values, converting, 82

O

Object data type, 73
Object Management Group (OMG), 154
object-oriented programming (OOP), 141–166
  abstraction in, 154–156
  classes in, 142
  constructors in, 161–162
  destructors in, 163–165
  events in, 146
  inheritance diagrams in, 152–154
  inheritance in, 147–148
  Is-A vs. Has-A relationships in, 158
  methods in, 145
  multiple inheritances/interfaces in, 158–160
  overriding members in, 149–151
  polymorphism in, 148–149
  properties in, 143–145
  refinement in, 156–158
  shadowing members in, 151–152
  shared vs. instance members in, 146–147
object-oriented tools, 29
object-relational databases, 203
object stores, 203
OMG. *See* Object Management Group
OOP. *See* object-oriented programming
OpenFileDialog control, 54
operating system, multitasking by, 16
operators, 105–118
  bitwise, 111–112
  compound assignment, 114
  concatenation, 114
  conditional, 113
  conditional logical, 112–113
  conversion, 116
  division, 112
  modulus, 112
  overloading of, 114–116
  parentheses with, 107
  post- and pre-increment, 110–111
  precedence of, 106, 108–110
OrElse operator, 109
Or operator, 109
overloading, operator, 114–116
overloading, routine, 135–136

P

PageSetupDialog control, 54
paging, 9
Paint event, 68
pair programming, 178
palmtops, 5
Panel control, 54
Parallel.ForEach tool, 24
Parallel.For tool, 23
Parallel.Invoke tool, 23
parallelism
  problems with, 18–20
  solutions using, 21–22
parameters
  and routine overloading, 135–136
  arrays, 130, 134
  optional, 129–130

passing methods, 130–132
reference types, 132–134
for routines, 128–136
value types, 132–134
paramref tag (Visual Studio), 172
param tag (Visual Studio), 172
para tag (Visual Studio), 172
parent class, constructors in, 163
parentheses (with operators), 107
Parse method, 188
Pascal, 26, 174
passing methods (parameters), 130–132
PasswordBox control, 58
Paste method, 66
PCs. *See* personal computers
PDAs, 5
percentage, culture-specific values for, 187
PerformanceCounter control, 54
permission tag (Visual Studio), 172
persistent variables, 89
personal computers (PCs), 2
petaflops (quadrillion floating-point operations per second), 2
PictureBox control, 54
pocket computers, 5
pointing stick, 3
PointToClient method, 66
PointToScreen method, 66
polymorphism, 143, 148–149
pop-up menus, 40
post-increment operators, 110–111
precedence, operator, 106, 108–110
pre-increment operators, 110–111
primary operators, 108
primary tags (XML comments), 172–173
PrintDialog control, 54
PrintDocument control, 54
PrintPreviewControl, 54
PrintPreviewDialog control, 54
private accessibility, 87
Private (private) keyword, 87, 136
procedure scope, 85
Process control, 54
processes, 16
processors
IBM z196 processor, 2
Intel 4004 processor, 2
program-defined data types, 74–78
arrays, 75
classes, 77–78
enumerations, 75–77
structures, 76–77
programmers, dividing tasks among, 122–123
programming environments, 25–32
languages in, 25–28
tools included in, 28–29
Visual Studio, 29–30
ProgressBar control, 54, 58
properties, 60–66, 143–145
Name, 51
Windows Forms, 60–62
WPF, 63–66
Properties window, 43
property get method, 144
PropertyGrid control, 54
property set method, 144
Protected Friend keyword, 87, 136
protected internal keyword, 87, 136
Protected (protected) keyword, 87, 136
pseudocode, 92–93
Public (public) keyword, 87, 136

R

race conditions, 18–19
RadialGradientBrush control, 65
RadioButton control, 54, 58
RAM, 9
random access files, 193
random heuristics, 21
ray tracing, 21
Rectangle control, 58
recursion, 137–138
Redo method, 66
reference types, 78–82, 132–134
refinement, 156–158
Refresh method, 66
regression testing, 29
relational databases, 200–202
remarks tag (Visual Studio), 172
RenderTransform property, 63
ResizeBegin event, 69
ResizeEnd event, 69
Resize event, 69
resources, parallelism and contention for, 18
returns tag (Visual Studio), 172
Return statement, 102
ribbons, 42

RichTextBox control, 55, 58
routers, 11
routine overloading, 135–136
routines, 119–140
  accessibility of, 136
  advantages of, 120–123
  calling, 123–124
  dividing, among programmers, 123
  parameters for, 128–136
  recursion with, 137–138
  separating out, 125
  types of, 120
  writing good, 125–128
routine scope, 85
Rule of Seven, 46
run time, 28
RunWorkerAsync method, 67

S

Save As command, 34
Save command, 34
SaveFileDialog control, 55
SByte data type, 73
scope, 85–86
scope blocks, 92
ScrollBar control, 58
ScrollBars property, 61
Scroll event, 69
ScrollToCaret method, 66
ScrollViewer control, 58
seealso tag (Visual Studio), 172
see tag (Visual Studio), 173
SelectAll method, 66
SelectedIndexChanged event, 69
SelectedIndex property, 61
SelectionMode property, 61
Select method, 66
Separator control, 58
separators (between commands), 40
sequence diagrams (Visual Studio), 30
SerialPort control, 55
servers, 4–5
ServiceController, 55
SetError method, 67
SETI@home, 23
setter method, 144
SetToolTip method, 67
Shared keyword, 88, 146
shared members, 146–147
shortcuts, menu, 35–36, 41
Short data type, 73
short date, culture-specific values for, 187
side effects, avoiding, 126
Sierpinski curve, 138
signed integer data types, 73
Silverlight, 50
Simple Object Access Protocol (SOAP), 197
Single data type, 73
Size property, 61
sizing/resizing (of modal dialog boxes), 43
Slider control, 58
smartphones, 5
SOAP. *See* Simple Object Access Protocol
solid-state hard drives, 9
source code management tools, 29
spaghetti code, 100
speed, computer, 6–8, 16
SplitContainer control, 55
spreadsheets, 202–203
SQL Server Express, 201
stack frame, 123
StackPanel control, 58
Start method, 67
statements, 72
  indentation in, 92
  rewriting, 111
static keyword, 88
StatusBar control, 58
StatusStrip control, 55, 56, 60
Stop method, 67
StringBuilder class, 74
String data type, 73
String.Format method, 188
strings, 74
  type conversion with, 82
structures, 76–77
  classes vs., 78
  copying, 82
  passing, by value, 134
  as value types, 79
subclassing, 147
subnotebooks, 3
subtraction, 106
summary tag (Visual Studio), 172
supercomputers, 2, 4
supporting tags (XML comments), 172–173

switches, 11
symbols, locale-specific, 184–185
System.CodeDom namespace, 211
System.Collections namespace, 211
System.ComponentModel namespace, 211
System.Configuration namespace, 211
System.Data namespace, 211
System.Deployment namespace, 211
System.Device.Location namespace, 211
System.Diagnostics namespace, 211
System.DirectoryServices namespace, 211
System.Drawing namespace, 211
System.Globalization namespace, 211
System.IO namespace, 212
System.Linq namespace, 212
System.Management namespace, 212
System.Media namespace, 212
System.Messaging namespace, 212
System namespace, 210
System namespaces, 210–213
System.Net namespace, 212
System.Numerics namespace, 212
System.Printing namespace, 212
System.Reflection namespace, 212
system registry, 199–200
System.Resources namespace, 212
System.Runtime namespace, 212
System.Security namespace, 212
System.ServiceProcess namespace, 212
System.Speech namespace, 212
System.Text namespace, 213
System.Threading namespace, 213
System.Timers namespace, 213
System.Transactions namespace, 213
System.Web namespace, 213
System.Windows namespace, 213
System.Xaml namespace, 213
System.Xml namespace, 213

T

TabControl, 55
TabControl control, 59, 60
TabIndex property, 62
TableLayoutPanel control, 55
tablets, 3–4
Tag property, 62
Task Parallel Library (TPL), 23–24
tasks, performing single, well-defined, 125–126
TCP. *See* Transmission Control Protocol
TCP/IP, 12
temporal databases, 206
term tag (Visual Studio), 173
ternary operator, 113
test-driven development, 179–180
testing, regression, 29
testing tools, 29
TextBlock control, 59
TextBox control, 55, 59, 62
TextChanged event, 69
text files, 192
text, locale-specific, 184–185
Text property, 62, 64
thrashing, 9
threads, 17
  distributed computing, 22
Tianhe-1A supercomputer, 2
time, culture-specific values for, 187
timer components, 49
Timer control, 55
ToolBar control, 59
toolbars, 42
  in Visual Studio, 30
ToolStripContainer control, 55
ToolStrip control, 55, 56
ToolTip control, 55
tooltips, 47
ToString method, 86, 188
touchpad, 3
touchscreens, 4
towers, 2–3
TPL. *See* Task Parallel Library
trackball, 3
TrackBar control, 55
track bars, 47
Transmission Control Protocol (TCP), 12
TreeView control, 59
Try Catch Finally block, 103
Try keyword, 103
type conversion
  explicit conversion, 82–83
  implicit conversion, 83–84

U

UInteger data type, 73
ULong data type, 73
ultraportables, 3

UML. *See* Universal Modeling Language
unary operators, 108
Undo method, 66
Unicode, 74
UniformGrid control, 59
Uniform Resource Locators (URLs), 12
Uniform Resource Names (URNs), 12
Universal Modeling Language (UML), 154
Until loops, 95–96
URLs. *See* Uniform Resource Locators
URNs. *See* Uniform Resource Names
USB, 6
USB flash drives, 9
user-centric viewpoint, 176–177
user errors, preventing, 47
user interface design, 44–48
  control order in, 44
  grouping related controls in, 44–46
  hints, providing, 47–48
  Rule of Seven and, 46
  user errors, preventing, 47
UShort data type, 73

V

ValueChanged event, 69
Value property, 62
value tag (Visual Studio), 173
value types, 78–82, 132–134
variables, 71–90
  and accessibility, 87–89
  defined, 72
  and fundamental data types, 71–73
  lifetime of, 88–89
  loop, 94
  persistent, 89
  and program-defined data types, 74–78
  and scope, 85–86
  and strings, 74
  and type conversion, 82–84
  and value/reference types, 78–82
Viewbox control, 59
Visibility property, 63
Visible property, 62
Visual Basic, 26
  arrays in, 75
  auto-implemented properties in, 145
  comments in, 127
  Continue statement in, 101
  destructors in, 163
  Exit Function statement in, 102
  Exit statement in, 101
  Exit Sub Function statement in, 102
  expression evaluation in, 107
  For loop in, 92
  implicit conversion in, 83
  operator overloading in, 114
  optional parameters in, 129
  parameter arrays in, 130
  parameter-passing methods in, 131
  property get/property set methods in, 144
  routines in, 120
  structures in, 77
Visual Studio, 29–30
  accelerators in, 34
  call stack window in, 124
  debugger in, 123
  designing forms with, 51
  localization of user interfaces in, 185–186
  shortcut editor in, 36
  Toolbox window in, 42
  XML comments in, 170
VScrollBar control, 55

W

WaitForExit method, 67
WANs. *See* wide area networks
WebBrowser control, 55
web service, 197
While loops, 95
wide area networks (WANs), 12
wide values, 84
Width property, 63
Wi-Fi, 12
windows, in Visual Studio, 30
Windows Experience Index link, 8
Windows Forms
  about, 50
  Button control in, 146
  controls in, 52–56, 63
  events, 68
  methods, 66
  properties of, 60–62
Windows Phone 7 operating system, 29

Windows Presentation Foundation (WPF), 34, 210
about, 50
controls in, 57–60, 63
properties of, 63–66
Windows program components, 33–48
context menus, 40–41
dialog boxes, 43
menus, 34–40
ribbons, 42
toolbars, 42
user interface design, 44–48
Word 2007, ribbon in, 42
words., 72
workstations, 2–3
World Wide Web (WWW), 12
WPF. *See* Windows Presentation Foundation
WrapPanel control, 59

X

XAML. *See* Extensible Application Markup Language
Xbox 360 game platform, 29
XML. *See* Extensible Markup Language
XML comments, 170–173
automatic documentation from, 171
creating, 172
and IntelliSense, 170–171
primary and supporting tags with, 172–173
XML files, 194–197
XML Path (XPath), 197
XML Schema Definition (XSD), 197
x-- operator, 108, 111
x++ operator, 108, 111
Xor operator, 109
XSLT. *See* eXtensible Stylesheet Language for Transformations

About the Author

Rod Stephens started out as a mathematician, but while studying at MIT, he discovered the joys of algorithms and has been programming professionally ever since. During his career, he has worked on an eclectic assortment of applications in such diverse fields as telephone switching, billing, repair dispatching, tax processing, wastewater treatment, photographic processing, vision diagnostics, cartography, and training for professional football players. Rod has spoken at programming conferences and user's group meetings, and is an experienced instructor.

A Visual Basic Microsoft Most Valuable Professional (MVP), Rod has written 24 books that have been translated into half a dozen different languages, and more than 250 magazine articles covering C#, Visual Basic, Visual Basic for Applications, Delphi, and Java. Rod's popular VB Helper website (*http://www.vb-helper.com*) receives several million hits per month and contains thousands of pages of tips, tricks, and example code for Visual Basic programmers. His new C# Helper website (*http://www.csharphelper.com*) contains similar tips, tricks, and examples for C# developers.

Sign up for his Visual Basic newsletters at *http://www.vb-helper.com/newsletter.html*, visit his blog at *http://blog.csharphelper.com*, or contact him at *RodStephens@vb-helper.com* or *RodStephens@csharphelper.com*.