Microsoft

Microsoft® Visual Basic® 2010 Developer's Handbook

Klaus Löffelmann
Sarika Calla Purohit

Microsoft® Visual Basic® 2010 Developer's Handbook

Your expert guide to building modern applications with Visual Basic 2010

Take control of Visual Basic 2010—for everything from basic Windows® and web development to advanced multithreaded applications. Written by Visual Basic experts, this handbook provides an in-depth reference on language concepts and features, as well as scenario-based guidance for putting Visual Basic to work. It's ideal whether you're creating new applications with Visual Basic 2010 or upgrading projects built with an earlier version of the language.

Discover how to:

- Use Visual Basic 2010 for Windows Forms and Windows Presentation Foundation projects
- Build robust code using object-oriented programming techniques, such as classes and types
- Work with events and delegates—and add your own events to custom classes
- Program arrays, collections, and other data structures in the Microsoft .NET Framework
- Solve problems quickly and easily using the My namespace in Visual Basic
- Dive into Microsoft LINQ, including LINQ to XML and LINQ to Entities
- Tackle threading, multitasking, and multiprocessor development and debugging

Get code samples on the web
Ready to download at
http://go.microsoft.com/FWLink/?Linkid=223982

*For **system requirements**, see the Introduction.*

About the Authors

Klaus Löffelmann, a Microsoft MVP for Visual Basic, has been a professional software developer for more than 20 years. He's the author of several books about Visual Basic, and owns a company that provides training and coaching in the use of Microsoft technologies.

Sarika Calla Purohit is lead software design engineer in test for the Visual Studio Languages team at Microsoft. A member of the team for more than eight years, she was responsible for testing the Visual Basic IDE in Visual Studio 2010.

RESOURCE ROADMAP

Developer Step by Step
- Hands-on tutorial covering fundamental techniques and features
- Practice exercises
- Prepares and informs new-to-topic programmers

Developer Reference
- Expert coverage of core topics
- Extensive, pragmatic coding examples
- Builds professional-level proficiency with a Microsoft technology

Focused Topics
- Deep coverage of advanced techniques and capabilities
- Extensive, adaptable coding examples
- Promotes full mastery of a Microsoft technology

See inside cover

microsoft.com/mspress

ISBN: 978-0-7356-2705-5

U.S.A. $59.99
Canada $68.99
[*Recommended*]

Programming/Microsoft Visual Basic

Microsoft® Visual Studio® 2010

Microsoft®

Microsoft®

Microsoft® Visual Basic® 2010 Developer's Handbook

Klaus Löffelmann
Sarika Calla Purohit

ISBN: 978-0-7356-2705-5

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

Printed and bound in Canada.

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.

Acquisitions and Developmental Editor: Russell Jones
Production Editor: Kristen Borg
Production Services: Octal Publishing Services.
Technical Reviewer: Evangelos Petroutsos
Copyeditor: Bob Russell
Indexer: Lucie Haskins
Cover Design: Twist Creative • Seattle
Cover Composition: Karen Montgomery

To Adriana, in love. Thanks for always letting me be myself, and taking me the way I am.

— Klaus

Contents at a Glance

Table of Contents

Foreword

Visual Studio 2010 is an exciting version for the Visual Basic language, which reaches a double digit version in Visual Basic 10. This is a phenomenal achievement for a programming language, and it demonstrates the enormous utility that the language continues to provide, year after year. Visual Basic has always been a premier tool for making Microsoft platforms accessible and easy to use. And even though the specific technologies and devices have changed over time, the core mission of Visual Basic has remained the same. Starting in 1991 with Visual Basic 1 and continuing through to Visual Basic 3, Visual Basic revolutionized Windows application development by making it accessible in a way that simply wasn't possible before its arrival. Moving forward to Visual Basic 4 through Visual Basic 6, the language greatly simplified component programming with the Component Object Model (COM), Object Linking and Embedding (OLE) automation, and ActiveX controls. Finally, with Visual Basic 7 and beyond, the language has enabled developers to take advantage of the Common Language Runtime (CLR) and many .NET Framework technologies. This book covers examples of this, using Visual Basic to access .NET Framework data types, Language Integrated Query (LINQ), Windows Presentation Foundation (WPF), and the Task Parallel Library. LINQ in particular has had a significant impact on the language, providing a unified way to access data from objects, XML, or relational data sources. One of the most revolutionary features introduced as part of LINQ is XML literals, which makes Visual Basic the most productive language for programming with XML.

Looking ahead, there are three major development trends that we see influencing the Visual Basic language, now and in the future: declarative, dynamic, and concurrent programming.

Declarative programming lets developers state what the program should do, rather than requiring them to specify in great detail how the compiler should do it. This has always been a design principle for Visual Basic, in which we strive to increase the expressiveness of the language so that you can "say more with less code." Some recent examples of this in Visual Basic 9 are LINQ and type inference. Visual Basic 10 introduces similar efficiencies with multi-line lambdas, array literals, collection initializers, autoimplemented properties, and implicit line continuation—all of which are covered in this book.

Dynamic programming is another style that has influenced the design of Visual Basic. Late binding is an important feature that has made Visual Basic a great language for Microsoft Office development and COM programming. In Visual Basic 10, we extended Visual Basic's late-binding support to work with other dynamic type environments, such as JavaScript and IronPython. This was made possible by the Dynamic Language Runtime (DLR), which was introduced in .NET Framework 4.

Finally, concurrency is an undeniable trend that we see influencing many forms of development. Whether your application is running on a multicore machine, a clustered environment on premises, via distributed computing in the Cloud, or even on a single-core computer performing IO-bound operations, concurrency can help speed up its execution. .NET Framework 4 provides some great tools for concurrent programming, such as the Task Parallel Library and Parallel LINQ. Part VI of this book shows how to use these technologies in Visual Basic.

Visual Basic is a vibrant environment, and we invite you to dive into it in Visual Studio 2010. Whether you've used previous versions of Visual Basic or other object-oriented programming (OOP) languages, or you are new to OOP altogether, this book has the information you need to quickly become productive. It explains programming concepts, Visual Basic, Visual Studio, and the .NET Framework from the bottom up, and it establishes a strong foundation. For the more experienced reader, this book also goes deeply into these topics and includes dedicated sections on what's new in the 2010 release of Visual Studio. The book covers a variety of topics; some of them are technology-specific (such as WPF), while others are application agnostic (such as garbage collection and serialization). This book establishes a solid foundation that you can leverage when developing applications for any platform that Visual Studio 2010 targets, including Microsoft SharePoint, the Web, and the Cloud.

As Visual Studio Community Program Manager, I always enjoy meeting members of the Visual Studio community. One of the first times Klaus wrote to me, he quoted a motto he had learned from his grandmother: "the worst attempt is the one that you'll never make." I knew at that point that he was an ambitious person! Klaus has been writing computer books for more than 20 years. The subjects of those books include, Commodore 16, Commodore 64, Commodore 128, Atari ST, Amiga, Visual Basic 1, Visual Basic 3, Visual Basic 4, Visual Basic 5, Visual Basic 6, Visual Studio 2003, Visual Studio 2005, Visual Studio 2008, and now Visual Studio 2010. I've met with Klaus in various cities around the world: Antwerp, Berlin, and Seattle. His first trip to Seattle was for the Microsoft Most Valuable Professional (MVP) Summit. The MVP program honors Microsoft technology experts for their impact in the community, and Klaus was recognized as a Visual Basic MVP for the great support he's provided through writing books, delivering webcasts, and reviewing German content on MSDN. Klaus made his second trip to Seattle while writing this book. He found great inspiration in being at the place where "the magic happens." I took Klaus on a tour through the Microsoft offices, where he was able to connect with other members of the Visual Basic product team. It was a really exciting visit and I could see how passionate Klaus is about Visual Basic.

I was happy to connect Klaus with Sarika, who has been a great partner in writing this book. As a Test Lead on the Microsoft Visual Studio Professional team, Sarika has extensive expertise in Visual Studio. She joined the team in 2002, and has worked on the releases for Visual Studio 2003, Visual Studio 2005, Visual Studio 2008, and Visual Studio 2010. During this time, she's been deeply involved in the evolution of the Visual Basic language and Integrated Development Environment (IDE). As Test Lead, Sarika spends a lot of time thinking about how Visual Basic developers use Visual Studio and the .NET Framework to create client, web, and other types of applications. She also interacts with the Visual Basic community at various phases of the product cycle to gather feedback, review bugs, and present to customers.

Sarika's and Klaus's backgrounds were key assets in writing this well-thought-out book. Sarika has spent nearly a decade working on the Visual Studio IDE, which was a significant area of investment in this release. The Visual Studio 2010 user interface was rewritten in Windows Presentation Foundation, which enabled richer user experiences in Visual Studio itself as well as greater extensibility capabilities for third-party add-ins. Sarika shares her insight on this topic in the "sightseeing tour" of the Visual Studio 2010 IDE in Chapter 4. Klaus's experience draws from many years of work as a consultant on Visual Basic .NET and Visual Basic 6 migrations. He's worked with many development

teams on a variety of projects, and he knows which concepts can be the most challenging for developers to pick up. He has a true passion for compiling the most useful information for his readers and presenting it in a way that's easy to understand. While writing a book can be an arduous task, Klaus tackles it with enthusiasm. He has a great sense of humor that shines through in the playful writing style and makes it fun to follow along.

So as Klaus's grandma would advise, give this book a try! I'm sure you will find that the 10th version of Visual Basic helps you to tackle your software development projects with greater ease and productivity than ever before.

Lisa Feigenbaum
Community Program Manager
Microsoft Visual Studio

August, 2010

Introduction

When someone asks me what I do for a living, I don't really know what to say. That might be because some 25 years ago I wrote my first book, *Programming Graphics for the Commodore 128*, (in German, and out of print) at age 16, when it was uncool, freaky, geeky, and nerdy to even work with computers at all, much less to write about programming them. Perhaps that feeling is etched on my memory and helps explain why the answer still causes me some embarrassment. "I write technical books about software development," also sounds a bit out-of-touch with the real world, doesn't it?

It isn't quite that bad nowadays, because the truth is, it's not just about writing anymore. The small company I own (ActiveDevelop—there are ten of us working there now, located in the only high-rise office building in Lippstadt, Germany) doesn't just write about *developing* software, it also actually *develops* software. We also help other companies with their software development efforts, bringing their teams up to speed on the latest technologies localizing software from English to German and vice versa, and helping them to migrate from Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET (or reluctantly, and often unnecessarily, to C#). One of us has even been the recipient of a Microsoft MVP award three times—and, er, that would be me. And to capitalize on this promotional opportunity, if you live in a German or English-speaking country, and need competent support for .NET, training, and project coaching in Visual Basic or C#, localization expertise, and a motivated team with good connections (hey, my co-author even works on the Visual Studio team), you now know where to find us: just send an email to info@activedevelop.de. Oh, and you can always follow me on Twitter @loeffelmann.

Because writing about software development has always been my passion, every once in a while I write a new book. Usually, this happens when Microsoft releases new products or new versions. That's the only way to explain why the book you are holding right now is the thirtieth book I have either written or co-authored. While I still find it exciting (just as it was back in the time of the Commodore 64) to learn new technologies and to receive beta versions of the latest Microsoft software, writing books has become more of a routine. In any case, that was true until my last book, *The Visual Basic 2008 Developer Handbook*, came out in Germany.

When writing this book, however, I experienced a second spring because it fulfilled a long-held wish. Remember? I live in a comparatively small town in the northwestern part of Germany. And I *always* wanted to write a programming book in the Microsoft metropolis, Seattle. And so I did. In June 2010 I flew about 6000 miles from Frankfurt to Seattle, where I spent almost four weeks writing a large part of this book at the "origin location." I wrote most of this material in an apartment in Bellevue, in the Japanese restaurant Blue Fin, at the Northgate Mall (I can recommend the large sushi selection); on the Boeing air field (OK, not really *on* the air field); in the hundreds of Starbucks in and around Seattle; at the pier

overlooking Puget Sound; at Pike Place Market; in pubs overlooking Lake Washington; in the cafeteria between building 41 and 42 on the Microsoft campus; and on the lawn in front of Microsoft's Building 41 (home of the compiler teams and of my co-author, Sarika Calla). I wrote everywhere. One time I even travelled to Whidbey Island, from which Visual Studio 2005 got its code name. I wrote there, too. It was a lot of fun getting to know all the talented and competent people at Microsoft. They answered my questions even when they were really busy. I'm still impressed when I remember back to that time.

Getting the assistance of Sarika Calla was the icing on the cake; not even in my wildest dreams did I imagine I might work with someone from the Visual Studio team. Among other things, Sarika took care of the completely new Visual Studio user interface—and not only in this book! I can't think of anyone who could have done better: Sarika was the test lead for the new WPF-supported user interface of Visual Studio 2010. And finally, at this very moment, I'm reviewing the English translation of this book, which I originally wrote in German. All this is so exciting that I wanted to tell you about it at the beginning of this book. And in case we ever meet in person: since you already know what I do for a living—don't bother to ask… I still don't really know what to say.

Who Should Read This Book

Visual Basic has always had a special target audience. Typically, a Visual Basic programmer expects his favorite programming language to allow him to focus primarily on domain-specific knowledge and achieve a great solution in an exceptionally short time. That's the reason Visual Basic 6 became so popular to begin with, and why so many great business solutions are still programmed in older Visual Basic versions. Now, Visual Basic has grown up: what was missing from Visual Basic 6 is here now, and is often better and easier to use than in any other .NET language. Yet the typical Visual Basic developer can still expect Visual Basic to help him provide an architecture for his domain-specific application in a comparatively short time. Version 2010 is—in terms of OOP and being team enabled—as powerful as C#. BASICally, it provides developers the best of both worlds.

This book is for those developers who want to reach the high bar Visual Basic sets. The book doesn't start at square one, but it doesn't require a lot of previous knowledge, either. It leads you and teaches you the things you need to know to become as skilled in modern software development methodologies and object-oriented programming as you already are in your domain-specific area. You'll get results that are as fast as is possible with Visual Basic 6, but at the same time you'll develop quality applications that don't need to hide behind the C# or C++ competitors.

Assumptions

This book expects that you have at least a minimal understanding of procedural programming concepts. If you have not yet picked up the basic principles of Visual Basic programming, you might consider reading Michael Halvorson's *Microsoft Visual Basic 2010 Step by Step* (Microsoft Press, 2010).

Other than that, you're good to go!

Who Should Not Read This Book

Not every book is aimed at every possible audience. If you don't want to become an expert in Visual Basic and have fun learning at the same time, this book is not for you! Just kidding. But honestly, if you (as already stated in the previous section) don't have a basic knowledge about what programming is, or maybe have only had some basic (or, even better—BASIC) classes in high school or college, you should consider starting with a book that teaches the BASIC language from scratch. This book focuses on the Visual Basic language itself; it only scratches the surface of topics like Windows Forms programming or Windows Presentation Foundation. While this book provides sufficient information for you to build your first applications based on those technologies, it doesn't focus on them; there are whole books written about those topics alone, so don't expect this book to cover those subjects in depth.

Organization of This Book

This book is divided into six sections.

- Part I, Beginning with Language and Tools, provides an introduction to the Visual Basic language and the Visual Studio Integrating Development Environment. It also shows you how to develop applications based on Windows Forms or Windows Presentation Foundation (WPF) in practical step-by-step lessons.
- Part II, Object-Oriented Programming, lets you become an expert software developer and provides you with all the tools and techniques for building professional and robust .NET business applications that can compete with industrial standards.
- Part III, Programming with .NET Framework Data Structures, shows the important details that you need to hone your Visual Basic skills to perfection. It covers topics like programming with generic data types, Nullables, Tuples, Events, Delegates, and Lambdas. Most of all, this part provides you with the in-depth knowledge of arrays and collections that you need.

- Part IV, Development Simplification in Visual Basic 2010, shows you how to use features which are unique to Visual Basic, and provides shortcuts for many of the tasks you need to solve in your daily programming routine.
- Part V, Language-Integrated Query—LINQ, is all about querying data stored in various data source types. It demonstrates how to construct queries that filter, order, and group information from internal lists and object collections, as well as from data that comes from external data sources like SQL Server or XML documents.
- Part VI, Parallelizing Applications, is another important part of this book. Have you noticed that the clock speeds of modern processors haven't increased much over the last years? Well, the processor core counts certainly have. So, to really get all the performance you need (even from smaller computers like tablet PCs or netbooks), you need to parallelize your applications. This final section shows you how to do that—and what pitfalls might result.

Conventions and Features in This Book

This book presents information by using conventions designed to make the information readable and easy to follow.

- Boxed elements with labels such as "Note" provide additional information or alternative methods for completing a step successfully.
- Text that you type (apart from code blocks) appears in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.
- When the constraints of the printed page require code lines to break where they normally wouldn't, an arrow icon (➥) appears at the beginning of the new line.
- A vertical bar between two or more menu items (such as File | Close), means that you should select the first menu or menu item, then the next, and so on.

System Requirements

You will need the following software to complete the practice exercises in this book:

- One of Windows XP with Service Pack 3, Windows Vista with Service Pack 2 (except Starter Edition), Windows 7, Windows Server 2003 with Service Pack 2, Windows Server 2003 R2, Windows Server 2008 with Service Pack 2, or Windows Server 2008 R2.
- Visual Studio 2010, any edition (multiple downloads may be required if using Express Edition products)
- Computer that has a 1.6GHz or faster processor (2GHz recommended)
- 1 GB (32 Bit) or 2 GB (64 Bit) RAM (Add 512 MB if running in a virtual machine or SQL Server Express Editions, more for advanced SQL Server editions)
- 3.5GB of available hard disk space
- 5400 RPM hard disk drive
- DirectX 9 capable video card running at 1024 x 768 or higher-resolution display
- DVD-ROM drive (if installing Visual Studio from DVD)
- Internet connection to download software or chapter examples

Depending on your Windows configuration, you might require local administrator rights to install or configure Visual Studio 2010 and SQL Server 2008 products.

Code Samples

Most of the chapters in this book include exercises that let you interactively try out new material learned in the main text. All sample projects are available for download here:

http://www.microsoftpressstore.com/title/9780735627055

Follow the instructions to download the VbDevBook2010Samples.zip file.

Alternatively, you can download the files from the author's company website:

http://www.activedevelop.de/download/VbDevBook2010Samples.zip

Note In addition to the code samples, your system should have Visual Studio 2010 and SQL Server 2008 installed. The instructions below use SQL Server Management Studio 2008 to set up the sample database used with the practice examples. If available, install the latest service packs for each product.

Installing the Code Samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book.

1. Unzip the VbDevBook2010Samples.zip file that you downloaded from the book's website.
2. If prompted, review the displayed license agreement. If you accept the terms, select the accept option, and then click Next.

Using the Code Samples

The folder created by the Setup.exe program is structured by chapters. In the chapters of this book when the text refers to a certain sample, it shows the relevant part of the folder where you unzipped the samples to. In every sample folder you'll find a Visual Basic solution file with the file extension .sln. Open this solution from within Visual Studio and run the sample according to what is stated in the text.

Acknowledgments

From Klaus Löffelmann:

First I would like to thank Lisa Feigenbaum. She put a lot of effort into the concept of this book and not only helped me to work out the relevant topics but also to perfect the English version. As community manager, Lisa is the primary contact for MVPs, and provides us with first-hand information. I'm sure I can speak for all MVPs: we are lucky to work with Lisa. Lisa, you rock!

Next I want to thank Sarika Calla, who not only agreed to reveal many aspects of the new Visual Studio user interface but who also was always at hand with help and advice while I was writing this book. It is great to be able to ask the real experts—those who developed Visual Basic and Visual Studio—while researching new topics for a book this size. Thank you for co-authoring this book!

I would also like to thank Ramona Leenings, our IT specialist trainee. Ramona not only created more than 90 percent of the screen shots of the original book (even though she hated it; it's not really a great job) but also edited and converted many examples, and wrote the practical WPF examples in Chapter 5. At age 20, Ramona is already a first-class developer, and has a knack for aesthetics and design. She is ambitious and linguistically able. I expect the Visual Basic and .NET communities to encounter her name more often in the upcoming years. Ramona, I'm your fan!

I want to thank the ActiveDevelop lead developer Andreas Belke (our database expert), who helped me with the LINQ part. It is always a lot of fun to work with Andreas. We are united by the fact that—like me—Andreas is a big fan of the Pacific Standard time zone.

Thomas Irlbeck, as the technical German editor,had the thankless task of checking the book to ensure the content was both correct and plausible. He attacked this task bravely, and his efforts ensured the quality of this book. His eye for detail is incredible, and he catches discrepancies that the authors overlook even after reading the text ten times.

Also, thanks to the folks at Octal Publishing, Inc., who handled the production of this book. I appreciate their commitment to get the first page proofs of this book to the printing press on time, and I enjoyed working with you, and I really had fun playing a Word-comment match while reviewing this book!

A big thank you also goes to Russell Jones from O'Reilly whose task was to convert my more-German-than-English-speech into a readable form. And thanks to this book's production editor, Kristen Borg, for giving me five more days for the review—I really needed the sleep! ;-)

And of course I want to thank my parents. First, I literally couldn't be here without you, and for that alone, I have to thank you! You had to put up with me while I was writing. I won't say anything too personal here. Instead I'll give you each a big kiss on the cheek.

And finally, there is my girlfriend: Adriana, I know I can be difficult, especially when I'm swamped with writing and contemplating chapters, screenshots, and debugging samples, and therefore, often trapped in a parallel universe. But your patience with me seems endless, and I so appreciate the extraordinary care you give to me. Thanks for always letting me be myself and for taking me the way I am—I love you so much! Please know, this book is dedicated to you.

From Sarika Calla:

In co-authoring this book, I have been helped by many people. I would like to thank Manish Jayaswal, Microsoft, and my father-in-law, Mr. R.K. Purohit for the care with which they reviewed my original manuscript. I am deeply indebted to Klaus Löffelmann, co-author of this book, and to Lisa Feigenbaum, who provided invaluable advice from inception to conclusion of this project. Last, but not least, I owe a great debt of gratitude to my husband Bhanu, my children, Shubham and Soham, my parents (Dr. S.K. Calla and Dr. Sudha Calla), my mother-in-law (Mrs. Sharda Purohit), and other family members (Surabhi, Nitesh, Dr. Veena, Dr. Rajesh, Yogi) and friends for their tremendous support and encouragement.

Thanks,
Sarika

Errata and 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/9780735627055

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*.

Part I

Beginning with Language and Tools

In this part:

Chapter 1

Beginners All-Purpose Symbolic Instruction Code

In this chapter:

Looking at the chapter title, you're probably thinking, "What a strange name for a chapter!" Why this name? Well, if you take the first letter of each word in the phrase "*Beginners All-purpose Symbolic Instruction Code*" they form the acronym, "BASIC." Developed in 1964 by John George Kemeny and Thomas Eugene Kurtz at Dartmouth College (they also came up with the name), BASIC, as it was originally conceived, had very little to do with the programming language we know today as Microsoft Visual Basic 2010. It was as far removed from the object-oriented programming we use today as Columbus was from India at the end of his famous voyage of discovery.

However, the modern version of the language contains fundamental linguistic elements, such as variable declarations and the use of structural commands, that are still very much "basic-esque," according to the original definition of BASIC. In this chapter, you will learn all that you need to know about these fundamental language elements.

Don't roll your eyes now, and say, "Oh, come on! I already know all that stuff!" It's possible that you really *do* already know everything contained in this chapter, in which case, by all means, you can pat yourself on the back and praise yourself, saying "Man—I'm good! I'm going to continue with object oriented-programming right away!" And then, highly motivated, you apply yourself to those much more challenging topics elsewhere in the book.

Or...you can read through the following sections and maybe catch yourself once in a while saying, "What? That works too?"

Either way, let me point out here that this chapter is not meant as a beginner's handbook, explaining the language at length, and it certainly doesn't start at square one. You should already be familiar with basic programming–preferably *in* BASIC; the following sections are meant to summarize Visual Basic for you, while showing you the differences between the BASIC dialects that you might have worked with so far—all in as concise a format as possible. It is not the purpose of this chapter to teach Visual Basic from scratch.

Starting Visual Studio for the First Time

These days, programming in Visual Basic means that you are very likely to spend 99.999 percent of your time in Microsoft Visual Studio. The rest of the time you probably spend searching for code files from other projects and binding them into your current project—or rebooting Visual Studio after it has crashed, which, thankfully, has become extremely rare after Service Pack 1 became available.

The integrated development environment (IDE) in Visual Studio 2010 provides tools in a user interface that help you to design your programs. Sorted according to importance these tools are:

- The Visual Basic 2010 Compiler, which becomes active when you use a command to start the compilation (in the *Create* menu or the corresponding Toolbar).

Note The compiler translates programs that you write into Microsoft Intermediate Language (MSIL), which then is converted into processor code at runtime, taking the specific machine characteristics into account. You will learn more about this in Chapter 2, "Introduction to the .NET Framework." In the interest of being thorough, Visual Studio also provides other compilers for C++ or C#, but we're not worried about those in this context. Visual Basic Express provides a leaner version of Visual Studio, which only contains the Visual Basic Compiler. Of course, you have the option of adding Visual C# Express or C++ Express. You can find the link to Express downloads at *http://www.microsoft.com/Express/*.

- The Visual Studio Editor, which provides syntax highlighting support, IntelliSense, and other aids while you are editing the source code of your program.
- Various designers with corresponding tool dialogs, which support you as you create forms and other visual objects.
- The Solution Explorer, which manages and organizes the code files in your project.

However, starting Visual Studio 2010 for the first time doesn't take you directly to this IDE. Instead, the Choose Default Environment Settings dialog box appears, as presented in Figure 1-1.

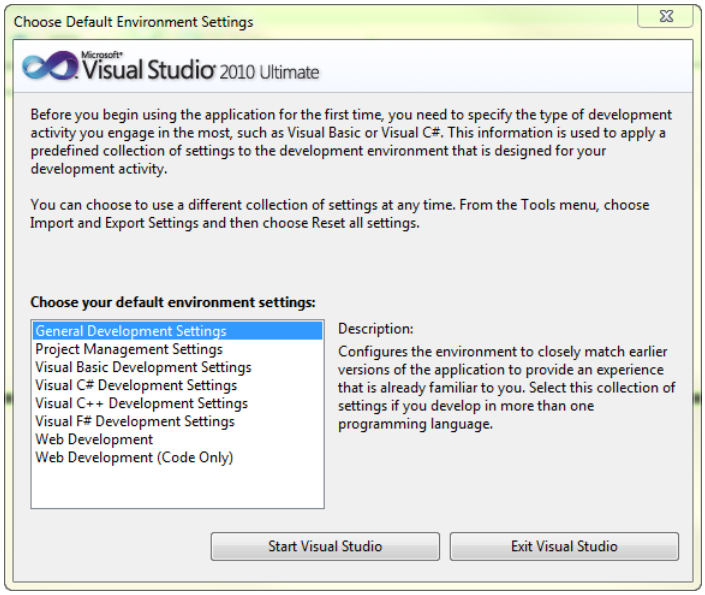

FIGURE 1-1 When starting the program for the first time, you need to customize the default settings for the development environment.

In this dialog, you decide which default settings to use to configure the development environment. Your best bet is to select General Development Settings.

Tip General Development Settings is the default for most Visual Studio 2005/2008 and 2010 installations. Visual Basic Development Settings include specific customizations, which provide the possibility to quickly adjust certain Visual Basic commands for dialog layout, command menus, and shortcuts. For example, the dialog for creating a new project is limited to Visual Basic projects only; some options, such as creating a solution directory along with a new project, are automatically hidden when you create a new project. You are able to create projects without naming them, and then later, at the end of your development session, you can choose whether to save them with a specific name. The same concepts apply to commands, which you can call by using drop-down menus: "Customized" means that many functions, which could also affect Visual Basic projects, are simply hidden.

Try different variations to see what works best for you. If, later on, you are no longer happy with the default settings you selected here, see Chapter 3, "Sightseeing," for help on how to reset your Visual Studio settings.

Console Applications

If you are developing for end users, you are probably creating programs that use the Windows graphic user interface. In Microsoft .NET jargon, such applications are called *Windows Forms applications*, or *WinForms applications* for short. For end users, this is currently the simplest and most familiar way to navigate a program; however, that project type is not necessarily appropriate when it comes to teaching developers, because the bells and whistles of the Windows UI, with all its graphic elements (such as buttons, dialog boxes, mouse control, and so on) can distract learners from focusing on each particular language element.

In .NET, you can use another project type, as well, which results in applications that older programmers will remember fondly (I'll leave it up to you to determine what "older" is) and server administrators will know about, even today. They're called *console applications*. These are programs that start with a minimalistic user interface. You launch such programs directly from the Windows command prompt and control them exclusively by using the keyboard. The only interface between the user and the program is a character-oriented monitor output and the keyboard.

The following sections concentrate exclusively on designing console applications. As mentioned before, console applications let you focus on the fundamentals. The following step-by-step instructions show you how to set up a console application, after you have started Visual Studio:

1. From the File menu, select the New command, and then click Project. Visual Studio will display the dialog box shown in Figure 1-2.
2. Under Installed Templates, open the branch Visual Basic, and then select Windows.
3. In the center pane, select Console Application, as shown in Figure 1-2.
4. Enter the name of your new project. If you would like to create an explicit solution directory, be sure to select the Create Solution Directory check box.

Note Visual Studio is designed to support anything from tiny sample programs to extremely complex and extensive projects. So extensive in fact, that on the one hand, a single project can lose clarity and transparency, but on the other hand, by dividing a project into parts, you can often make some of those parts available for use in other projects. This is why you would usually not simply create projects, but also a "project folder", called a solution (Visual Basic Express uses slightly different terminology, but in general it

manages solutions, as well). In the simplest case a solution contains only one project—your Windows or console application. A solution directory makes sense, when you expect your solution to contain several projects. In addition to your main application these other projects might include, for example, additional class libraries or completely different project types, such as web services. In this case, the root directory contains only the solution file, which, by the way, has the extension *.sln*.

Tip Because there are quite a lot of templates for different purposes and different programming languages, you can always use the search box in the upper-right corner to find the template you're looking for in no time: simply type in keywords or keyword abbreviations like "visual basic" ("vb" works just as well) or "console" to narrow the list down to those templates that match the keyword.

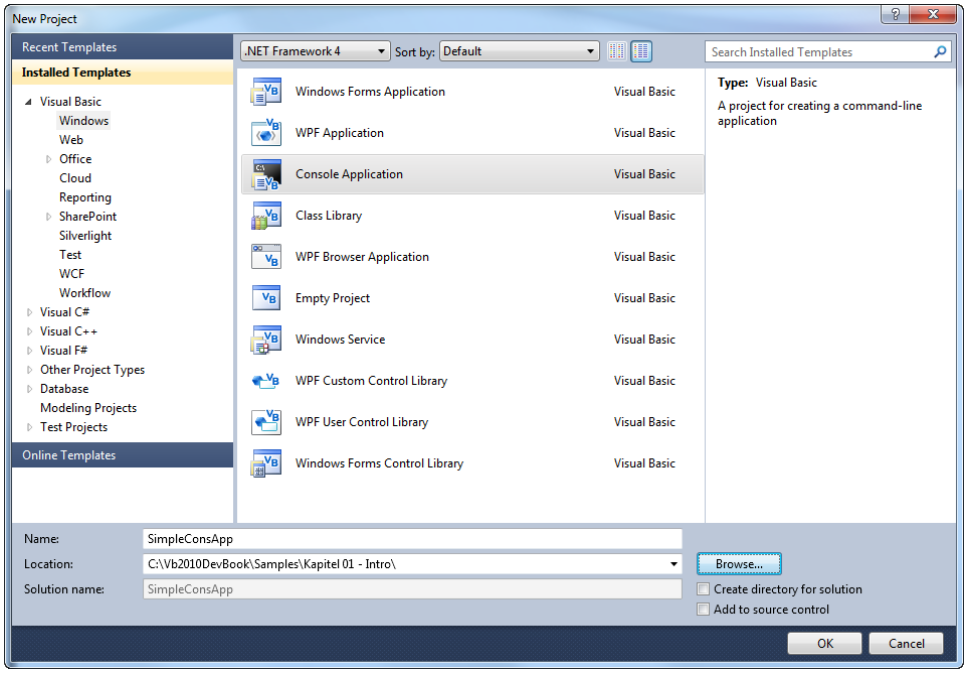

FIGURE 1-2 Use the New Project dialog box to create a new project for a console application.

5. Click the Browse button located in the lower-right portion of the window, specify the *path* to the location where you would like to save your project, and then click OK when you are done.

After you click OK, the Visual Basic Code Editor opens. Enter the following lines between the commands *Sub Main* and *End Sub*.

```
Sub Main()

    Dim dateOfBirth As Date
    Dim age As Integer

    Console.Write("Please enter your date of birth (mm/dd/yyyy): ")
    dateOfBirth = Date.Parse(Console.ReadLine())
    age = (Now.Subtract(dateOfBirth)).Days \ 365
    Console.Write("You are {0} years old", age)
    Console.ReadKey()
End Sub
```

Your results should look similar to Figure 1-3.

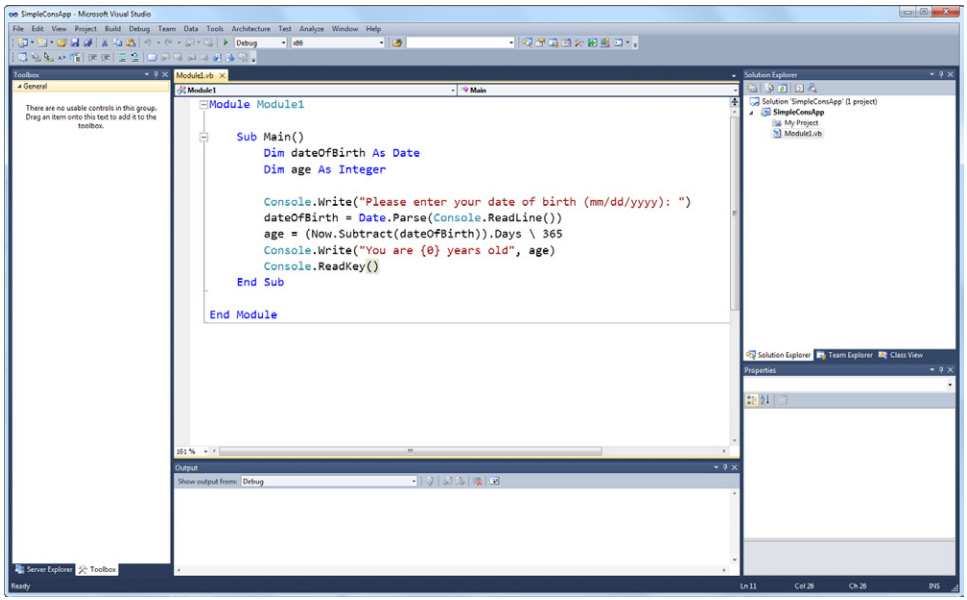

FIGURE 1-3 The Visual Studio 2010 IDE showing a new console application and a few lines of code.

Starting an Application

Start your new application by either pressing F5, clicking the start icon on the Toolbar, or clicking the Start Debugging command on the Debug menu. Upon startup, you are asked to enter your date of birth. You can see immediately how very different console applications are

from the much more familiar Windows applications. You interact with the program solely by using the keyboard and the text-only display on your monitor. Figure 1-4 shows you how this first console application should look.

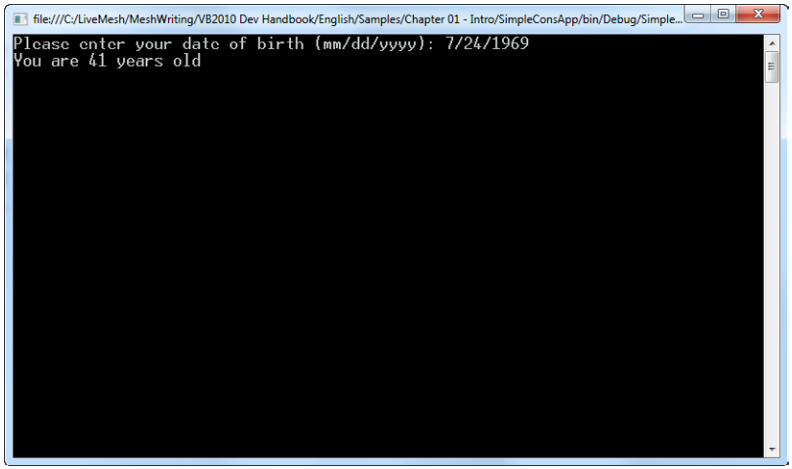

FIGURE 1-4 A typical console application. The user interacts with the application via the keyboard and the text-only output.

Important When you start an application following the steps just described, the speed at which it runs (regardless of whether it's a console application or a Windows Forms application) does not correspond at all to the speed at which it will run outside the Visual Studio user environment. The Visual Studio IDE can help you to check the application for errors and bugs by using the automatically attached debugger. For example, you can insert breakpoints at certain lines (by placing your cursor in a line and pressing F9 to automatically stop your program when it reaches a breakpoint). You can then examine the state of your program as it runs and even step through the program line by line (F11) or procedure by procedure (F10). This all takes time during program execution, whether you are actually using these features or not.

You can circumvent this overhead to see your application run at a more normal speed by instead pressing Ctrl+F5 to start it, or by selecting Start Without Debugging on the Debug menu. Of course, you won't be able to avail yourself of the Debugger functionality when you launch the application this way.

Tip You can also add a toolbar to make the entire debugging functionality more easily available by right-clicking an empty space within the Visual Studio IDE Toolbar to display the context menu and then selecting Debug, which displays the Toolbar, as shown in Figure 1-5.

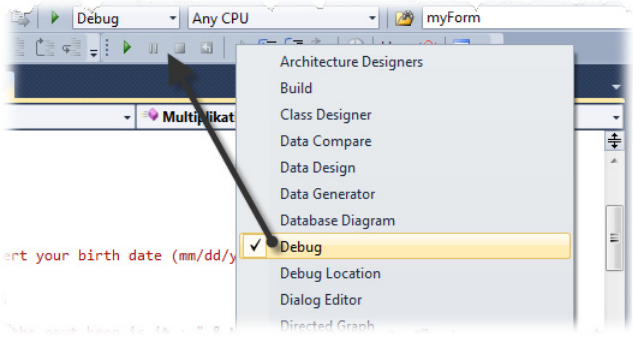

FIGURE 1-5 Open the context menu by right-clicking the empty space in the Toolbar to display additional Toolbars. The Debug Toolbar provides easy access to many debugging functions.

Anatomy of a (Visual Basic) Program

The initial methods to store computer data weren't magnetic; they didn't use diskettes or hard disks. Instead, early computers used punched tape, which was a paper tape with small holes stamped in it in specific patterns that represented the data. The punched tapes were fed through a special reader, and based on the configuration of these holes, the previously punched bits and bytes found their way into the computer. Interestingly, punched-tape devices were first used in the textile industry, not in the shape of an early computer, and not for storing sales volume or customer information; instead, they were used in looms. The information needed to control the loom was stored in the form of small wooden plates, arranged behind one another in a certain order.

This early scheme corresponds to how we define what a computer should do today, and thus to the anatomy of a program:

1. You need something that you process: namely data (or wool, in the textile industry).
2. You need instructions to control how something is processed: the program statements (or the knitting pattern).

Of course, this is an extreme simplification, and it doesn't even begin to suggest the incredible variety and number of the possibilities. This first application is but the tip of the iceberg. To demonstrate this, Figure 1-6 presents another, slightly more polished version of the first example. This version is intended to give you a better understanding of the different aspects of typical Visual Basic program anatomy (even though it's still only a console application).

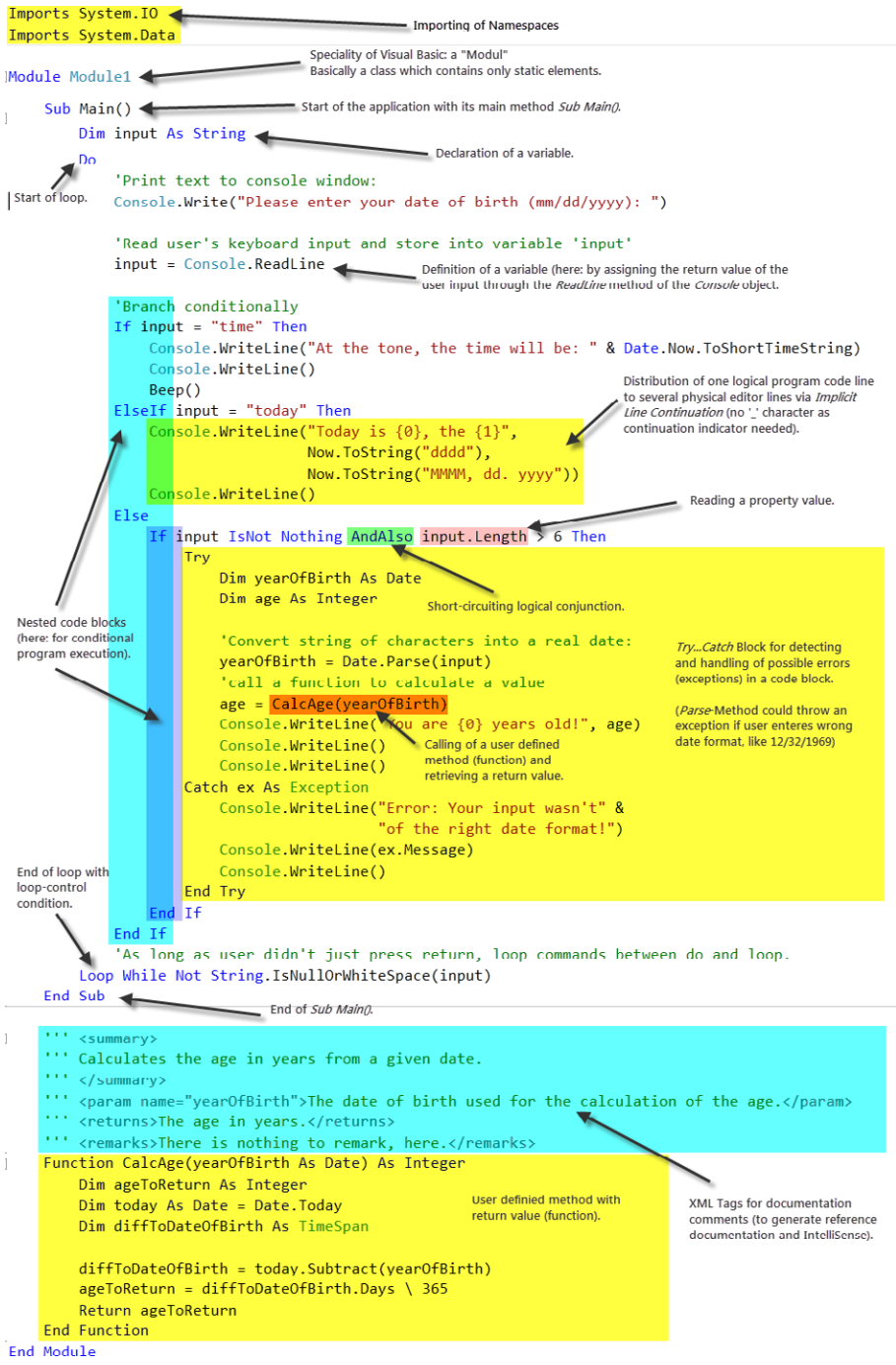

FIGURE 1-6 The anatomy of a small Visual Basic application.

Figure 1-6 also demonstrates that while an application consists of data and program statements, program statements can obviously have a variety of different structures. These structures are easier to understand by looking at how the example program works again (see Figure 1-7), before focusing on its internal functions and the individual components of the BASIC language.

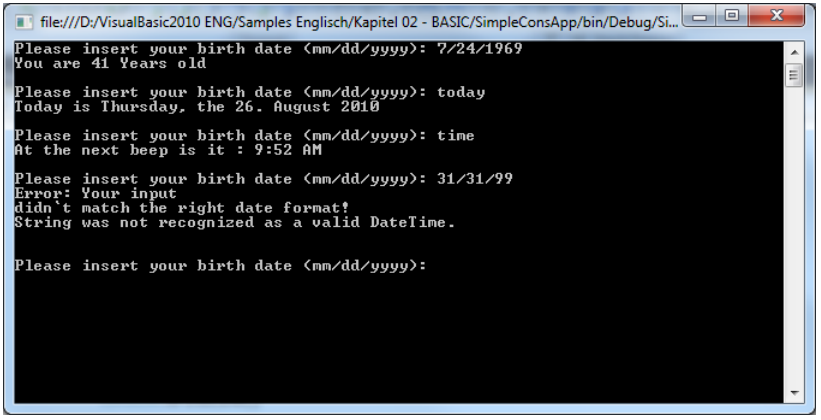

FIGURE 1-7 The Professional Edition of this example has a few more features up its sleeve than the first "lite" version.

To begin, after starting the application, you either enter a birth date or one of the commands **today** or **time**. The program responds as shown in Figure 1-7. If you enter a date using an invalid format, the program recognizes and catches the error and displays a corresponding message.

Internally, this application functions rather simply. The schematic provides an admittedly alternative glimpse into the elements used to build the application.

Starting Up with the *Main* Method

Each console application starts with a certain method, which in Visual Basic (and many other languages) always has the name *Main*, and contains statements that are encapsulated between *Sub* and *End Sub* commands (in Visual Basic only). A *Sub* in Visual Basic defines a method without return values. A console application needs at least this one *Main* method so that Windows knows where your program begins. In Visual Basic, you define methods that return values with the *Function* keyword and designate the return value with the *Return* keyword. The following section explains this in more detail.

The History of Important Programming Languages, and the History of *Sub Main*

In today's business world, when we talk about programming languages, we essentially only encounter two separate concepts: procedurally-structured and object-oriented programming. As the following section shows, procedural means that the code can be reused or used multiple times because it can be called from different points within the program and always returns to the same place from which it was called. By the way, another word for *method* is *procedure*, and this is where reusable code is placed, which is why this form of conceptional structuring is called procedural programming.

The term structured programming comes from certain structural constructs, such as *If … Then … Else* or *Switch* constructs. And object-oriented programming adds another abstraction level on top of that, but this is not of interest to us at the moment. What we are focusing on here is where it all started.

We need to go back to 1958; a time when developers still had to program their computers, on foot, so to speak, in direct machine code. A programmer at IBM named John Backus became so annoyed with this way of developing that he approached his supervisors with a request that he be allowed to develop a utility program to alleviate the problem.

At the time, Backus worked on an IBM System 704 mainframe computer. His utility was designed to help programmers by using commands that were a little closer to human language, and which also permitted the analysis and calculation of mathematical functions directly. The high-falutin' name of this project was *The IBM Mathematical Formula Translation System*. However, for Backus to pronounce this name each time he talked about the project would have probably delayed the completion of the project by another year, so he decided on a shorter nickname made up of a couple of first syllables. Thus, one of the first standard languages was born: ForTran (written here with a capital "T" to clarify the syllable origin).

At that time, all standard languages followed the so-called imperative concept, by which commands are handled one after the other. There were really no structures, not to mention procedures. The few deviations from the normal program execution took place when jump targets were reached (for example, a *GOTO* statement) or when one of the three passable jump targets could be reached with an *IF* statement (depending on whether the expression to be examined was less than, equal to, or greater than 0). With each subsequent development, the programming languages slowly evolved via various Fortran and Algol versions until eventually CPL (*Combined Program Language*) was created. This was a programming language that tried to unite the best aspects of Fortran and Algol (the languages for scientific applications) as well as Cobol (the language for finance-mathematical applications). Unfortunately the result was a monster, which, in spite of possessing the ideal requirements to serve as a platform for a language compiler, was much too large and sluggish.

Martin Richards of Cambridge University took CPL under his wing (some say the abbreviation stands for *Cambridge Program Language*, because it was developed in part at Cambridge[1]). He slimmed it down and thus created the language BCPL, which could only be used correctly on mainframe computers, but which was now suitable as a platform for further compiler development.

Just as an aside, BCPL was the first programming language that marked language structures{

```
      well, {
       let's say:
            { a basic approach }
         }
  }
```

for the first time with curly brackets.

Around 1969, Ken Thompson and Dennis Ritchie of the famous Bell Laboratories set for themselves the goal of getting BCPL to run on mini-computers. At the time, mini-computers were also minimalists with respect to hardware, so this was an ambitious goal. To complicate matters even further, there was only one way to do things: *every* element that wasn't absolutely vital for the language had to be removed. Thompson and Ritchie did exactly that and more; in their eagerness to save space, they didn't even spare the name. Therefore, BCPL became simply "B," which was—as you might begin to suspect—the daddy of C and the grand-daddy of C++ (and eventually C#). As a point of interest, the tutorial about B implementation by B.W. Kernighan and Murray Hill states right away on the first page, directly below Section 2, "General Program Layout":

```
main( ) {
            statements
        }
newfunc(arg1, arg2) {
            statements
        }

fun3(arg) {
             more statements
          }
```

[1] To be accurate, CPL was a common project between Cambridge University and the computer department of London University, which led to assertions that that was the reason to change the original name *Cambridge Programming Language* to *Combined Programming Language,* rather than being a combination of two computer language worlds.

And further:

> *" All B programs consist of one or more "functions", which are similar to the functions and subroutines of a Fortran program, or the procedures of PL/I. main is such a function, and in fact all B programs must have a main. Execution of the program begins at the first statement of main, and usually ends at the last. Main will usually invoke other functions to perform its job, some coming from the same program, and others from libraries."*[2]

So now you know a little more about the evolution of the important programming languages of today as well as the historical significance of the *Main* function, which existed more than 35 years ago in B, and whose name survived until today as a symbol identifier for the entry point to start a program.

Methods with and Without Return Values

The *Main* method enjoys a special status within an application because it needs at least one user-defined (custom) type, within which it can reside, and it defines the starting point of the application solely by its name. A custom type is essentially a class—Visual Basic also has a special kind of class called a *module*. A module is essentially just a class that has methods available to any caller who can access the module. These methods are also called *static* methods.

Methods are the units into which your program is arranged. Just as the operating system calls the *Main* method, you can call methods in your programs. These can be custom methods that you write or methods exposed by objects that the .NET Framework provides.

You can assign general requirements to methods for completing their tasks—but you don't have to. Such general requirements are called *parameters* or *arguments*. Whether a method can accept arguments is determined by its *signature*. The signature specifies which arguments and which types of arguments the method expects as well as the order or sequence in which it expects them.

Referring back to Figure 1-6, you can see several methods and how those methods are defined in the program. For example, *Console.WriteLine* calls a method of the class *Console*. The method displays whatever is passed as an argument in a certain format—also defined by an argument—in the console window.

[2] FTP://cm.bell-labs.com/cm/cs/who/dmr/scbref.pdf

Defining Methods Without Return Values by Using *Sub*

In Visual Basic, you define methods in two ways. One way is to use the keyword *Sub*. These are methods that can take arguments but will not return a result, as shown in the following:

```
Sub MethodName([Parameter1 As Typ1[, Parameter2 As Typ2]])
    [Statements]
    [Exit Sub|Return]
End Sub
```

To end method execution early, you can use either *Exit Sub* or *Return* statements—it doesn't matter which one you use. Exiting early makes sense only when you're checking for a certain condition or state, which you might do by using *If … Then … Else* constructs (you will learn more about them later in this chapter).

Defining Methods with Return Values by Using *Function*

Methods can also return a function result. In Visual Basic, you define such a method with the *Function* keyword, as follows:

```
Function MethodName([Parameter1 As Typ1[, Parameter2 As Typ2]]) as ReturnType
    [Statements]
    [Exit Sub]
    [Statements]
    Return FunctionResult
End Sub
```

The method *CalcAge* in Figure 1-6 is an example of this. This method receives a birth date as an argument and returns an integer to the calling instance. The return value type is specified with the words *As Integer* at the end of the method definition.

For methods that return values (*Functions* in Visual Basic), *Return* has a different meaning than for methods, which don't return values (*Sub* methods). *Return* not only ends execution as it does in a *Sub* method, but it also specifies *which* value to return. The value follows the *Return* keyword.

Declaring Variables

The application example *declares* a variable in the first line of its *Main* method. Variables are places in which a method or program can store data during its lifetime. Thus, the following line determines that the variable *input* can take content of the type *string*:

```
Dim input As String
```

The *String* type restricts the *input* variable to character strings: in other words, any kind of text. The *String* keyword, therefore, represents a type that contains text. If you were to

specify *Integer* as the type, the variable *input* could save numbers between approximately –2 billion and +2 billion, but only integers, no fractions. If you were to specify *Double* as the type, the *input* variable could also hold fractions. However, their range of value becomes smaller as the fractions become more accurate. Fractions are also called *floating point numbers*, because the decimal point can be moved.

Important In .NET (not just in Visual Basic), *type safety* is very important, because using the wrong types is a common source of errors. For example, if you write and sort a date in the Irish date format (dd.mm.yyyy) as a string, then 11.10.2002 is bigger than 10.11.2005, because 11 is greater than 10. But when you look at it as a date, the second date is of course later (greater) than the first, which is the earlier or smaller date value. One way to look at it is in the form of the number of elapsed time units. For dates further in the past, fewer seconds have passed than for later dates. Sorting dates as strings leads to a completely different result than sorting by date.

For a description of the different types you can use in .NET Framework, and therefore also in your Visual Basic programs, see Chapter 6, "The Essential .NET Data Types."

Visual Basic and the *Dim* Keyword: a Brief History

There are historical reasons why variables in Visual Basic are typecast by using the keyword *Dim*. This process is correctly called *Declaring of Variables*. In early versions of BASIC, it was possible to use a variable by simply providing a name, without defining anything about the type of that variable beforehand. The BASIC interpreter created and typed variables only at runtime, according to the context of their assignment. However, this was not the case for so-called arrays (groups of variables), which could be accessed by using one or more index values. For example, you can access individual array elements within loops by using counter variables. Of course, that is still possible, because arrays are still a central part of all .NET languages. The point is that even in early BASIC, you already had to define the dimensions and limits of an array (upper and lower bounds and number of dimensions). You did that by using the *DIM* statement.

Simply declaring a variable doesn't give it a value. However, when a variable type has some kind of basic state, it automatically receives an appropriate default value; for example, all numeric data types have the value 0, *False* is the default value for *Boolean* variables, and *Null (nothing)* for strings. Here, *Null* can actually be a value or state of a variable (and philosophically speaking, the computer therefore creates a contradiction in terms: "the state *is* nothing"), but just accept that for now. (You can find some good background information about *Null* on Wikipedia.)

The .NET Framework—or more precisely, the Common Type System (CTS), which regulates different data types throughout the .NET Framework—differentiates between two data

types: value types and reference types. The essential and highly optimized value types, built directly into .NET Framework, are, for example, *Integer*, *Long*, *Single*, *Double*, *Decimal*, *Date*, and *String*. Because they are so deeply embedded into the .NET Framework and at the same time so highly optimized and not derived from some other type, they are also called *primitive data types*.

Note Technically speaking, this last statement isn't entirely correct. For example, you might have heard someone say correctly that all data types are based on *Object*. Conceptually, however, this statement is inaccurate, because all primitive data types are handled specially by the Common Language Runtime (CLR), even if they are based on *ValueType* and therefore on *Object*.

Many different data types are required to save all the different kinds of data correctly—after all, you wouldn't necessarily put your coats in a drawer while hanging your socks in the closet; that might work, but it wouldn't be very practical. So the .NET Framework has a number of primitive data types that are designed for various tasks, as presented in the following list:

- *Byte*, *Short*, *Integer*, *Long*, *SByte*, *UShort*, *UInteger*, *ULong* These data types save numerical values without decimal places. Using these types, you couldn't save the constant *Pi* with anything close to its actual value; it would end at 3 and drop the post-decimal fractional portion. These so-called integer data types differ only by value range and the memory space they require. For example, *Long* requires 8 bytes, but it covers a wide value range, from about –9 billion to +9 billion. The *Byte* type, on the other hand, needs only one byte, but only covers values between 0 and 255.

Note This value of 255, which appears so arbitrary and uneven to mere mortals, in fact has historic underpinnings. The smallest unit of information a computer can handle is one *Bit*: A stream flows or it doesn't, which corresponds to *True* or *False* or *0* or *1*. However, just 0 and 1 aren't sufficient to represent the world around us.

During the design phase of one of the first IBM mainframe computers, a design engineer named Werner Buchholz placed a few bits (initially 6) behind each other and thus turned them into a *Bite*. However, because the terms Bite and Bit were so often mixed up, *Bite* became *Byte*. But 6 Bits corresponds to a value range of 0–63 ($2^6 = 64$), which even back then often didn't cover a large enough value range. Thus, with the passing of time, 8 Bits became the basic size unit for a *Byte*.

All data storage in computing is based on the byte as a size unit: a *Short* requires 2 bytes, and therefore covers a value range from –32768 to +32767. The *UShort* type (the *U* before the data type name indicates that it is *unsigned)* does not have a sign (– or +). Therefore, the data range changes to 0 to 65,535—negative values are not allowed for integer data types that start with "U." The most commonly used integer data type is called *Integer*: it consists of 4 bytes, which corresponds to 32 bits, and 32 bits can naturally be handled most quickly by a 32-bit processor. Typically, using *Integer* data types is the fastest way to count within your programs.

- *Single* and *Double* These data types require four and eight bytes, respectively, and can represent decimals. They are floating-point types, which means that the accuracy depends on the size of the value range. Rounding errors are very common with these data types. Therefore, you should not use them for financial or mathematical applications, but rather for calculating graphs and the like, where speed is more essential than accuracy.
- *Decimal* This data type is more suitable for high accuracy. Rounding errors, such as when converting from a base-2 to a base-10 system, cannot happen with these because the values are saved differently internally than when using *Single* or *Double*. However, the processor cannot calculate the data type directly, so calculations using this type execute more slowly than those that use *Single* or *Double*. But *Decimal* is still fast enough for financial calculations.
- *Date* This type stores date values (calendar data). Internally, a *Date* is an 8-byte floating-point number, where the pre-decimal places represent the actual date (day, month, and year) and the post-decimal places represent the time. You will find more details about this data type in Chapter 6.
- *Boolean* This one is easy because it corresponds to one bit and saves only the numbers 0 and 1, which you can think of as *False* and *True*.
- *Char* This data type saves a character. For modern programming purposes, characters require 2 bytes, because 256 different characters are no longer enough.
- *String* This type saves character strings (text). Strictly speaking, strings are arrays of *Char* values.

In addition to these primitive value types, the .NET Framework has "normal" value types, which are composed of primitive data types. For example, the type *Size* is made up of two integer values; it provides methods and properties to define the size of objects or perform size calculations. Another example is the *Location* type, which saves coordinates.

You provide a variable with a value by using an assignment statement, which is explained in the next section.

Nullables

You can declare all value types as *nullables*. A nullable is unique because in addition to a value, it can also have *no* stored value, which is normally not possible for a value type.

As an example, if you declare the following *Integer* variable, you can use it directly—remember, that's part of the nature of value types. They don't need to have an initial value assigned to them in Visual Basic:

```
Dim count As Integer ' Count is 0
```

```
count = count + 1    ' Works, count is now 1
```

In the first line of the preceding code, the integer value type cannot support the idea that *count* has not been assigned a value. This is only possible when a variable has been defined as nullable for a specific type, which you do by appending a type literal (the question mark) to the end of the type name (*Integer*, in this example), as demonstrated in the following:

```
Dim nullableCount As Integer?          ' Count is now Nothing
nullablecount = 0                      ' Assign initial value
nullablecount = nullablecount + 1      ' Works, count is now 1
```

An interesting and useful characteristic of the nullable value type is that you can query it to determine if it has a value by using the *HasValue* property, as shown in the following example:

```
Dim nullableCount As Integer?
nullableCount = 0
nullableCount = nullableCount + 1

Dim anotherNullable As Integer?

Console.WriteLine("Does nullableCount have a value: " & nullableCount.HasValue.ToString)
Console.WriteLine("Does anotherNullable have a value: " & anotherNullable.HasValue.ToString)
```

The preceding sample code produces the following output in the console window:

```
Does nullableCount have a value: True
Does anotherNullable have a value: False
```

To delete the value of a nullable, assign *Nothing* to it:

```
nullableCount = Nothing
```

Nullables *don't* cause an error when you try to access their value to perform calculations; the result stays "Nothing, unknown, I don't know what it contains"—in short, *Nothing*. The sample code could be explained as follows:

```
Dim anotherNullable As Integer?         ' Doesn't have a value or value is unknown.
anotherNullable = anotherNullable + 1   ' and that's how it stays.
Console.WriteLine("anotherNullable is: " & anotherNullable.ToString)
```

When you run that code, the output is:

```
anotherNullable is:
```

There is literally no output. When trying to perform an addition, Visual Basic can determine that there is nothing in *anotherNullable*. It's not defined, and Visual Basic interprets this state with "I don't know what's in *anotherNullable*." The response is as follows: when Visual Basic doesn't know what a variable contains, and a value is supposed to be added to it, it also doesn't know what the result is. Therefore, the result is again *Nothing*.

Note Nullables are used mainly in database applications, wherein it's customary to define database tables in which the columns of some rows don't contain values. Therefore, there's a difference between a field in a table that contains the value 0 and one that contains no value. Nullables in .NET accommodate such situations. You'll find more detailed explanations about nullables in Chapter 18, "Advanced Types."

Expressions and Definitions of Variables

For numeric as well as date calculations or string operations, it is important to have an understanding of expressions. The following line (see Figure 1-6) is a *String* expression. The *ReadLine* method returns a string (in this case, whatever text the user has entered via the keyboard) and assigns the result to the variable *input*.

```
input = Console.ReadLine()
```

Another example of an expression in the code in Figure 1-6 is the static *Parse* method of the *Date* data type, as shown here:

```
birthyear = Date.Parse(input)
```

By assigning this expression in this manner, the string, which represents the date, is converted into a true date type. Remember, it's important to use the correct type in the correct place. A computer doesn't sort a string that happens to represent a date value as a true date (refer to the sidebar earlier in this chapter for more information). Therefore, you must instruct the *Date* type to parse the string (to go through it character by character and analyze it), to interpret the characters as a date and then convert them accordingly.

Defining and Declaring Variables at the Same Time

You can declare and assign of variables in one operation. So, instead of writing this

```
Dim christmas10 As Date
christmas10 = #12/24/2010#
```

you can condense it to the following:

```
Dim christmas10 As Date = #12/24/2010#
```

Note Beginning with Visual Basic 2008, the *As <type>* portion of a variable declaration is no longer required within procedures (for example, within methods defined with *Sub* or *Function*) when the declaration and assignment happen at the same time. When the option *Local Type Inference* is set (more about this in the section, "Local Type Inference," later in this chapter), the Visual Basic compiler can recognize or *infer* the type for the declaration from the expression type during the expression assignment. Without Local Type Inference, you need to write an assignment and declare a *String* variable as follows:

```
Dim almostGerman as String = "Bratwurst und Sauerkraut are yummy!"
```

Using Local Type Inference, it's sufficient to write the statements as shown here:

```
Dim almostGerman = "Bratwurst und Sauerkraut are yummy!"
```

The compiler recognizes, that the constant expression "Bratwurst und Sauerkraut are yummy!" is a string, and infers the correct type when assigning a type. It doesn't recognize, though, that the expression is fundamentally false. Bratwurst only works with French fries—NEVER with Sauerkraut.

Complex Expressions

The result, or the return value, of a method can also serve as the operand of an operator, and you can therefore combine the return values of multiple methods into a complex expression. Just to make this a little more clear, first let's review the method from the initial example:

```
Function CalcAge(ByVal birthYear As Date) As Integer

    Dim retAge As Integer
    Dim today As Date = Date.Now
    Dim diffToBirthdate As TimeSpan

    diffForBirthdate = today.Subtract(bithyear)
    retAge = diffToBirthdate.Days \ 365
    Return retAge

End Function
```

This is constructed in such a way that the result is calculated in several intermediate steps. It's possible not only to calculate the value (which is stored in the next-to-last line in *retAge* and returned with *Return* in the last line), but also to return it at the same time. This is how the code would look rewritten as a complex expression:

```
Function CalcAge(ByVal birthyear As Date) As Integer

    Return Date.Now.Subtract(birthYear).Days \ 365

End Function
```

The preceding condensed code and the longer form shown in the prior example provide the exact same result.

It's not too hard to understand numeric expressions. Most of us were probably confronted with them for the first time in grade school. String expressions behave similarly. For example, the following declaration, expression calculation, and the subsequent definition for *dbl* is 55:

```
Dim dbl As Double
dbl = 5
dbl = dbl +5 * 10
```

Note This calculation follows the rules of priority during the evaluation: "Power before parenthesis, before period, before prime."

Using this as a guide, you can see why "Adriana Ardelean" is stored in the *str* variable as the resulting value in the following:

```
Dim str As String
str = "-> Adriana Ardelean <-"
str = str.Substring(3, 16)
```

Therefore, you can also see why the following code always results in a date that represents two days in the future:

```
Dim dayaftertomorrow As Date
dayAfterTomorrow = Date.Now
dayAfterTomorrow = dayAfterTomorrow.AddDays(2)
```

Boolean Expressions

Numeric expressions, expressions that calculate date values, and string expressions are relatively easy to read and understand. They are similar to the typical curve sketching formulas we all learned in ninth or tenth grade.

How would you read the following expression?

```
Dim var = 5 = 5
```

Is this line valid? If yes, which type has *var* been assigned and what value does it carry?

To begin, yes, this line of code is valid. Second, it defines a variable of type *Boolean*. As mentioned earlier, variables of this type don't have a very large number range because they can only return one of two states, *True* or *False*.

Now let's look at the expression from a slightly more analytical standpoint: 5 = 5 is a true assertion. The result of the expression 5 = 5 is therefore *True*. In this case, the operator is the equal operator (not the assignment operator, which defines a variable; it's is the same operator character, but different context and different meaning), which always returns a *Boolean*

result, the variable *var* will also be defined as *Boolean* by Local Type Inference. Try printing the result by using the statement following:

```
Console.WriteLine(var.ToString)
```

This statement prints the value, *True*.

Boolean variables and *Boolean* statements are important because they are used as arguments in statements for conditional program edits or for testing a loop exit criterion. For example, during an *If* query, the block of code between *If* and *End If* is run only when the result of the *Boolean* expression behind *If* is *True*. And if the construct contains an *Else* branch, that branch is run when the result is *False*. The section "*If … Then … Else … ElseIf … End If*," later in this chapter, provides more detail about how this works.

Comparing Objects and Data Types

In the real world, objects are things that you might or might not be able to touch, but you can at least describe them in some way. Take for example a bucket: you can clearly picture it in your mind, but you fail to imagine a repository *per se*, because it's not concrete enough a definition. The only way to think about it is to manifest it into something defined: If your repository becomes a drawer, a bowl, a can, or a bottle, you'd be able to picture it in your mind. But you can't think of just a repository. It's the same in programming, with some slight differences: objects are abstract entities. Apart from threads, which we'll discuss later in the book, they are pretty much the most abstract entities known in programming.

The best way to begin to explain what an object is would be to explain what it is not. For example, an *Integer* variable is not an object. Neither is a date variable nor a *Boolean* variable. A data type called *Point* (which determines a position for drawing) isn't an object, either.

However, the entity on which you draw things, the content of a window or a *PictureBox*, or a printer context are all objects. A brush that you use for drawing is an object. A button is an object. A *TextBox*, into which you can enter text in a Windows or web application, is also an object. A ToolTip is an object; a TCP/IP connection can also be an object (one that controls that connection). Strings are a little more tricky. Is a string an object? In principle, yes; by definition, no.

As you might have already noticed, objects and value data types have something in common. Both save data, and both regulate access to this data. But in general, objects are more complex; they need more memory and have more methods that let you do something with the object; for instance, setting focus to a *TextBox*, establishing a connection, bringing a button to the foreground. They also have more complex properties. Examples of these include specifying or determining the text inside a *TextBox*, setting or querying the buffer size of a TCP/IP connection, toggling a menu and thus setting or determining its "active" state,

specifying or retrieving the background color of a button, and so on. And objects can trigger events (a button was *clicked*, the text in a *TextBox changed*, the data *received* in an open connection, and so forth).

Simple variable types, such as *Integer*, *Double*, *Date*, or *Boolean* can be used directly after declaration. Among other things, the .NET Framework infrastructure—the CLR to be exact—also ensures that the appropriate amount of memory is reserved on the processor stack for these value data types. A variable in your program is therefore connected to a memory address in the processor stack.

Note The processor stack is a special memory range for caching temporary information, which the processor (or a running process with the help of the processor) can access extremely fast.

Essentially, there is no space for more complex objects. There might be room for one, but if you want to program a picture organizer, you will need to have more than one object in memory at the same time. And that's why space must be reserved in memory for these complex objects. The memory to be reserved is called the *Managed Heap* in .NET-speak. This is because you don't need to worry about other data structures infringing upon the memory space for your pictures; it's all managed for you. And you don't need to free up memory space manually when the object is no longer needed—that's managed for you, as well.

Thus, the difference between objects in programming and real life is that in the latter we don't distinguish between small objects which we can handle very, very fast and normal or large size objects. In programming, we do this when we talk about data stores for the different purposes:

The rule of thumb is a follows:

- **Value Types** Can make due with very little memory space, but they are extremely fast
- **Reference Types** Can use a lot of memory space, but they are slower (but still relatively fast)

Deriving from Objects and Abstract Objects

Objects in .NET have another interesting characteristic: they build upon each other. This is no different from objects in daily life. For example, as I mentioned before, a container is an abstract object, which we can classify in general but not in concrete terms. A milk carton is certainly a container, but a container is not necessarily a milk carton. A container could also be a bottle, or a bucket, or a barrel. The intersection of certain properties makes all these objects a container. Therefore, we can say that all objects that contain liquid, such as milk cartons, bottles, barrels, or buckets, are *derived* from container.

Programming works the same way. You can create a template (called a *class*) for an object but only use this template for creating further templates. The differences between these templates then flow into the derivations; by using additional properties and methods, the abstract class *Container* becomes the more specific class *Bottle*, or *MilkCarton*, or *Barrel*. And then you use this new template to bring into being (to *instantiate*) the actual barrel. You are instantiating an object from a class.

Here's another analogy: molds and sand. The mold is a class, the instantiation is the sand, which you take from a sand heap. You only have one mold—one class—but by filling it with sand, you can create countless objects. In .NET-speak: "You are reserving memory on the Managed Heap, so a class can be instantiated into an object."[3] For a more complete discussion of classes and objects, see Chapter 8, "Class Begins."

This is also the reason why objects for new creations (for instantiating) need the keyword *New*. So for example, to add a new button to a Windows Form, the following lines are required:

```
'Instantiate new button object
Dim t As New System.Windows.Forms.Button

'Set Text property
t.Text = "New button"

'Add to form
myForm.Controls.Add(t)

'Focus button
t.Focus()
```

Classes and objects are such an extensive topic that they have their own section in this book. With a basic idea of how this works, however, it will be easier for you to use classes and objects in the many examples that you'll see prior to reaching that part of the book. Most important, you need to be able to distinguish objects (reference types) from primitive data types (value types).

Properties

Objects and value types can have *properties*. Properties describe states that can be determined or changed. In contrast, methods typically perform a task. Even in everyday language, these things can't always be separated very easily. Properties are very similar to methods. Setting a property initially corresponds to calling a method that doesn't return a value, and reading a property corresponds to a function call to a method that returns a result.

[3] Translated into "sandbox speech" this means: "You take sand from the sand heap, so a mold can make a mud pie with it".

For example, the length of a string is a property. Let's start by declaring and defining a *String* variable, as follows:

```
Dim aName as String = "Adriana Ardelean"
```

You can then determine the string's length (the number of characters it contains) by using the appropriate property, as shown here:

```
Dim lengthOfName As Integer = aName.Length
```

Some properties, such as the current date (*Date.Now*) or the length of a string (*stringVar.Length*), can only be read, but not changed. Such properties are called, not surprisingly, *read-only* properties.

Other properties can be both read and written. For example, by using the *Enabled* property of a control, such as a button, you can determine whether a user is able to select it in a window:

```
aButton.Enabled = True      ' now it is usable
aButton.Enabled = False     ' now it isn't any longer.

'Query property and respond:
If aButton.Enabled Then
   ' The code here will be run only when the button is enabled.
End If
```

Type Literal for Determining Constant Types

Type literals force a constant to be a certain type. (Thus, in my opinion, I think they should be called "Type forcing literals for constants.") You have already learned about type literals in several examples. To assign a string to a variable, you define it as having the type *String*, and put whatever you would like to assign to it between quotes, as illustrated here:

```
Dim AText As String = "Put in quotes"
```

You don't do this for numeric variables, however:

```
Dim Pi as Double = 3.1415926
```

To force a numeric constant to be a string so that you can assign it to a *String* variable without having to convert it, you also place it in quotes. Quotes turn a number into a string:

```
Dim PiAsText As String="3.141592657"
```

Be aware that by using this technique, you can't calculate with *PiAsText*. The variable *PiAsText* is a string (like *AText* in the previous code line), which in this case happens to contain a string that we humans can interpret to be a number. From the computer's point of view, though, there is no difference between `"3.1415926"` and `"Hello, nice weather today!"`.

In the .NET versions of Visual Basic (all versions since Visual Basic 2002), type literals aren't just limited to strings; they are also used with other data types. For example, to assign a date constant to a variable of the type *Date*, use the # (number) character, as follows:

```
Dim KlausBirthday As Date=#07/24/1969#
```

It is important to write the date using the United States format: month first, then day, then year, even if you're on a Spanish, Italian, French or German Windows system.

Apart from quotes for strings, this is the only other literal type character that you use to wrap a constant. Other type literals are simply placed behind the constant. In some cases, these can consist of two letters instead of just one.

Table 1-1 shows how to define constants with type literals. If a variable type character exists for a specific type, it is also presented.

TABLE 1-1 Type Literal and Variable Type Characters of Primitive Data Types, as of Visual Basic 2005

Type name	Type character	Type literal	Example
Byte	–	–	`Dim var As Byte = 128`
SByte	–	–	`Dim var As SByte = -5`
Short	–	S	`Dim var As Short = -32700S`
UShort	–	US	`Dim var As UShort = 65000US`
Integer	%	I	`Dim var% = -123I` or `Dim var As Integer = -123I`
UInteger	–	UI	`Dim var As UInteger = 123UI`
Long	&	L	`Dim var& = -123123L` or `Dim var As Long = -123123L`
ULong	–	UL	`Dim var As ULong = 123123UL`
Single	!	F	`Dim var! = 123.4F or Dim var As Single = 123.4F`
Double	#	R	`Dim var# = 123.456789R` or `Dim var As Double = 123.456789R`
Decimal	@	D	`Dim var@ = 123.456789123D` or `Dim var As Decimal = 123.456789123D`
Boolean	–	–	`Dim var As Boolean = True`
Char	–	C	`Dim var As Char = "A"C`
Date	–	#MM/dd/yyyy HH:mm:ss# or #MM/dd/yyyy hh:mm:ss am/pm#	`Dim var As Date = #12/24/2008 04:30:15 PM#`
Object	–	–	In a variable with the type *Object*, any type can be boxed or referenced
String	$	"String"	`Dim var$ = "String"` or `Dim var As String = "String"`

Type Safety

.NET languages follow the rule of type safety. Type safe means that you can't just mix different types randomly during assignments. For example, the following line will not compile and causes the error message shown in Figure 1-8:

```
Dim aDifferentString As String
aDifferentString = 1.23
```

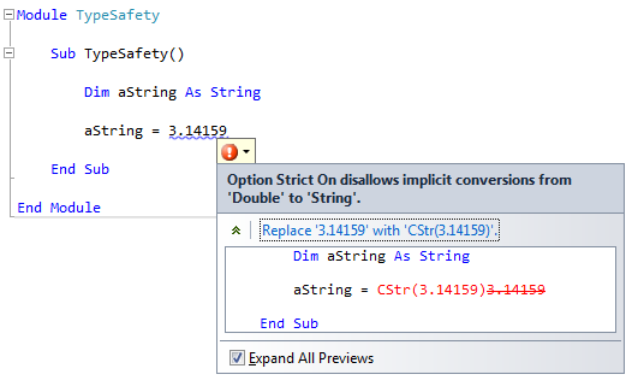

FIGURE 1-8 Type safety in .NET enforces the rule that only equal or safe types can be assigned implicitly to each other.

Note By default, Visual Basic .NET simulates the non-existent type safety used by Visual Basic 6.0 and Visual Basic for Applications (for example, for macro programming in Microsoft Word or Microsoft Excel). You can (and should) change this default behavior for all new projects. To do that, from the Tools menu, select Options | Projects And Solutions, and then select all the check boxes under VB Defaults.

That customizes the settings for all future projects. To customize the settings for the currently open project only, from the Project menu, select Properties, and then click the Compile tab. Set the drop-down lists for all compilation options to On.

To view what the problem is, click the red bar in the squiggly line (see Figure 1-8) to activate autocorrect for intelligent compiling. The problem arises in this example because in .NET, you can only assign the same types implicitly, or types that are different but for which an implicit conversion is definitely safe. And what does "definitely safe" mean? It is definitely safe, for example, to assign an *Integer* type to a *Long* type, as shown in the following:

```
Dim aLong As Long
Dim anInt As Integer = 10000
aLong = anInt
```

The computer would not complain about that assignment in any .NET language, because nothing can go wrong. *Integer* covers a much smaller number range than *Long*; therefore, an implicit conversion is *widening* the type, without possible risk.

Figure 1-9 demonstrates that it's a different story the other way around.

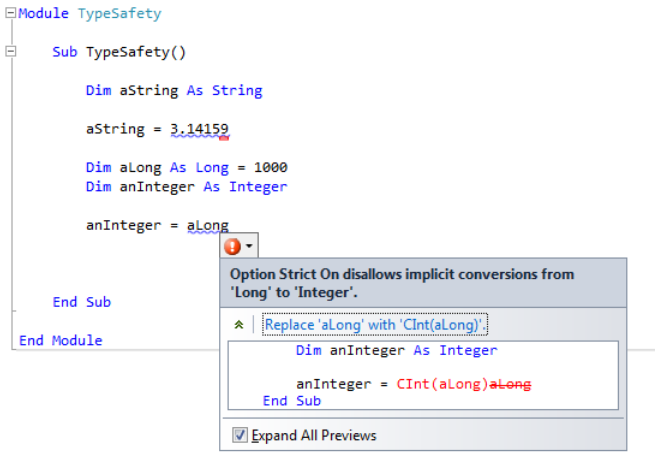

FIGURE 1-9 Whereas *smaller* types can safely be converted into *larger* ones, converting larger to smaller types is not type safe and, therefore, not permitted implicitly.

When converting from *Long* to *Integer*, information can become lost, so .NET doesn't classify these *narrowing* kinds of conversions as type safe. Of course, you *can* perform such a conversion—just not implicitly. You need to instruct .NET explicitly that you are aware of the risk and perform appropriate actions to be able to do the conversion. As you can see in Figure 1-9, autocorrect for intelligent compiling makes a direct suggestion: "*Replace 'aLong' with 'CInt(aLong)'*" The *CInt* (Convert to *Int*eger) command tells the compiler explicitly that you want to make the conversion from *Long* to *Integer*.

By now, you can probably guess the purpose of type literals in Visual Basic, which you saw in the previous section. Type safety also applies to constants. Therefore, there must be a way to force a constant to be a certain type. This is exactly what happens when you use type literals. Does the following work implicitly?

```
Dim aChar As Char
Dim aString As String = "Hello"
aChar = aString
```

No. Because *Char* can contain only one character, the "*ello*" portion of the "Hello" string would be lost during the conversion. A *Char* type can take only a single Unicode character, not an entire *String*. Even this code will not work properly:

```
Dim aChar As Char = "H"
```

String is *String*, and *Char* is *Char*. You define a *String* in quotes, no matter how many characters it has. And even if it could be converted (as in the preceding code line), type safety would be compromised. You need to turn "H" into a *Char* type, which you can do by placing the "c" (for character) type literal behind it, as shown in Figure 1-10.

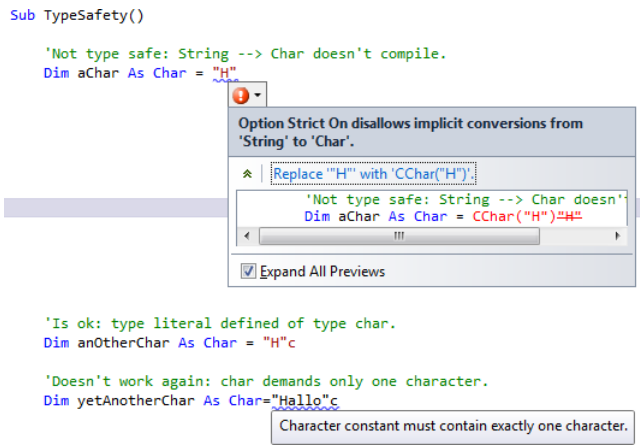

FIGURE 1-10 Use type literals to force constants to have a certain type.

The type safety rule doesn't apply only to *String* and *Char* types; it also applies to the various numeric data types. The following code section shows a few examples that demonstrate when it makes sense to employ type literals:

```
'Error: Implicit doesn't work from Double to Decimal.
Dim decimal1 As Decimal = 1.2312312312312
'Here it is a Decimal, because of the D at the end:
Dim decimal2 As Decimal = 1.2312312312312D

Dim decimal3 As Decimal = 9223372036854775807   ' OK.
' Overflow - without type literal it is implicitly a
' Long value, and in this case it is outside its value range:
Dim decimal4 As Decimal = 9223372036854775808
' With the type literal "D" Decimal is forced as a constant, and it's correct:
Dim decimal5 As Decimal = 92233720368547758O8D  ' No overflow.

'Error: Without type literal it's again implicitly a Long,
'but outside of the Long value range:
Dim ushort1 As ULong = 9223372036854775808

' With the type literal Decimal is forced as a constant, and it's correct:
Dim ushort2 As Decimal = 92233720368547758O8UL  ' No overflow.
```

Local Type Inference

Beginning with Visual Basic 2008, you can assign types to variables, based on their initial assignments. This becomes apparent in the following assignment:

```
Dim blnValue = True
```

When you assign the value *True* to a primitive variable and type safety is defined, then the variable must have the *Boolean* data type. The same logic applies to the following assignment:

```
Dim strText = "A string."
```

In this case, *strText must* be a string—the assignment defines it. It's different for numeric variables. It's important to know that by assigning an integer to a variable, which hasn't yet been assigned a type, the compiler assigns the type *Integer*, and by assigning a floating-point number, the compiler gives the variable the type *Double*. These standard types of constants already existed; in the end, it's the constants that determine, according to their type literals, which type they represent. Here are a few more examples:

```
Dim anInteger = 100 ' Integer, simple number without floating point defines integer constant
Dim aShort = 101S  ' Short, because the type literal S defines a Short constant
Dim aSingle = 101.5F  ' Single, because the type literal F defines a Single constant
```

Important Local type inference works only at the procedure level, not at class level (which is why it's called *local* type inference).

You control whether local inference is in effect with the statement *Option Infer*, which takes either the parameter *Off* or *On*. By default, local type inference is enabled. To turn it off, use the following statement.

```
Option Infer Off
```

You would place the *Option Infer Off* (or *On*) statement directly at the beginning of the code file for the classes or modules of this code file, or globally for the entire project. To do this, in Solution Explorer, right-click the project (not the solution) to open the context menu, and then select Properties (see Figure 1-11).

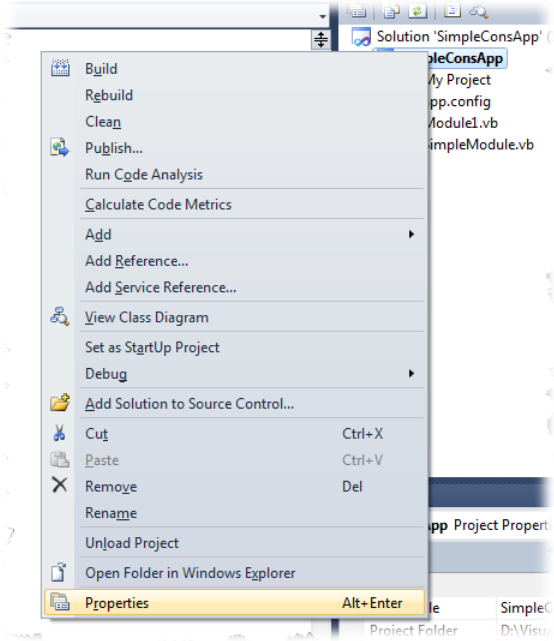

FIGURE 1-11 Opening the context menu of the project in Solution Explorer.

In the dialog box that appears, click the Compile tab, and then set the option for Option Infer, as desired (see Figure 1-12).

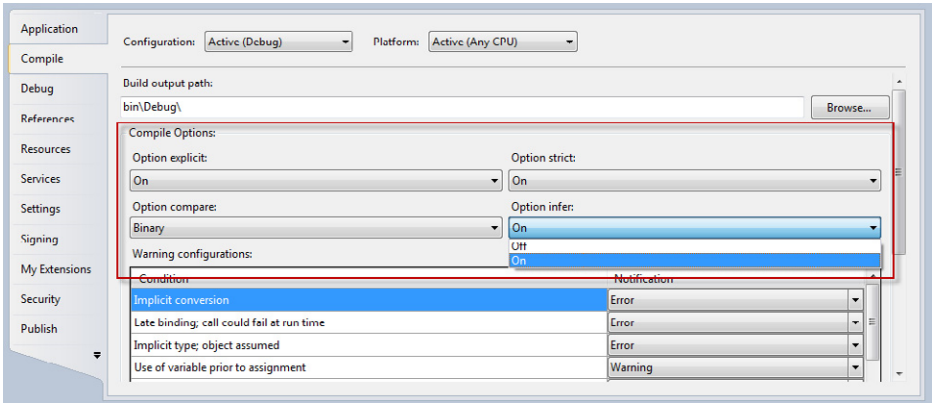

FIGURE 1-12 Setting the local type inference option on the Compile tab.

Arrays and Collections

This chapter has already presented two ways of storing data:

- Value types and primitive value types for saving fast, but small data structures, such as date values, integer values, floating-point values, True/False values, positions and sizes, and characters and strings.
- Reference types for saving larger amounts of data, such as controls and pictures.

You use both types within a program by employing variables. However, in some cases this can pose a problem: what do you do when you need to find another value in a table, based on the value of a variable?

Suppose that you have saved five names in variables. This, in and of itself, is not a problem:

```
Dim Name1 As String = "Lisa Feigenbaum"
Dim Name2 As String = "Sarika Calla"
Dim Name3 As String = "Ramona Leenings"
Dim Name4 As String = "Amanda Silver"
Dim Name5 As String = "Tanja Gelo"
```

Now you would like to access a name, and its number is saved in a variable, as shown in Figure 1-13.

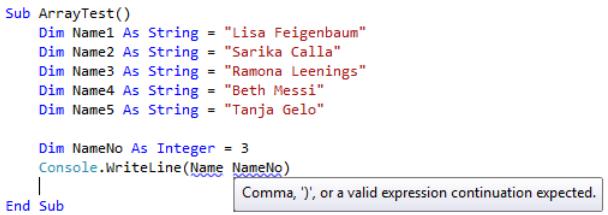

FIGURE 1-13 To find a variable by using another variable, such as in a list, you won't get very far by using normal variable names.

The solution to this problem lies in a concept called *arrays*. An array can save as many values as you determine, under a specified name. Each name is associated with a unique index value. You can then use the index value to identify which value you want to access:

```
Dim Names(0 To 4) As String
Names(0) = "Adriana Ardelean"
Names(1) = "Sarika Calla"
Names(2) = "Ramona Leenings"
Names(3) = "Beth Messi"
Names(4) = "Lisa Feigenbaum"

Dim NameNo As Integer = 3
Console.WriteLine(Names(NameNo))
```

Important In contrast to Visual Basic 6.0 or VBA, arrays in .NET are always 0-based. This means that the first element in an array has an index of 0. When dimensioning an array in Visual Basic, you always specify the upper limit, not the number of elements (as you would in C#, for example). While setting the dimensions, you can also use a short form. So, for example, the first line (highlighted in bold), in the previous code sample can also be written as follows:

```
Dim Names(4) As String
```

Tip You can also define arrays with array initializers. Using the following syntax, you specify the elements directly, which saves typing:

```
Dim Names() As String = {"Adriana Ardelean",
                         "Sarika Calla",
                         "Ramona Leenings",
                         "Beth Messi",
                         "Lisa Feigenbaum"}

Dim NameNo As Integer = 3
Console.WriteLine(Names(NameNo))
```

When you define an array this way, note that you don't specify the upper limit; the compiler derives the upper limit from the number of elements that lie between the curly braces.

If you don't know how many elements the array might have when you first create it, you should use a *List* type instead. With a *List*, you can easily add new elements by using the *Add* method, as shown in the following example:

```
Dim OtherNames As New List(Of String)
OtherNames.Add("Adriana Ardelean")
OtherNames.Add("Sarika Calla")
OtherNames.Add("Ramona Leenings")
OtherNames.Add("Beth Messi")
OtherNames.Add("Lisa Feigenbaum")

'You can also access a dynamic list,
'as in arrays, via the index:
Dim NameNo As Integer = 3
Console.WriteLine(OtherNames(NameNo))
```

For the moment, this is all the information we're going to cover about arrays and lists. You will learn more about them in Chapter 19, "Arrays and Collections," which is dedicated to these concepts, and you will encounter them again in the context of loops and in other places throughout the book.

Executing Program Code Conditionally

The *Boolean* data type is usually required when evaluating decisions, and used to control whether program code is executed or not, depending on its value. Such decision-making statements include *If*, *Case [Is]*, *While*, or *Until*.

The function *IIf* does not control the program execution but should be mentioned here because of its resemblance to the *If* statement. *IIf* returns a function result based on a passed-in *Boolean* value or expression. When the passed-in value or expression is *True*, *IIf* returns one result; if it is *False*, it returns the second.

If ... Then ... Else ... ElseIf ... End If

It is likely that you have employed the *If* statement many, many times, and you probably have its use down pat. But in the interest of being complete, let's examine it a little more closely.

In its simplest form, *If* causes code to run that's placed between *If* and *End If* when the *Boolean* expression behind *If* evaluates to *True*. The following construct demonstrates the concept of comparisons with *Boolean* expressions:

```
locBoolean = True
If locBoolean Then
      'Only runs when locBoolean is True.
End If
```

Some developers might unnecessarily use the following expression:

```
locBoolean = True
If locBoolean = True Then
      'Only runs, if locBoolean is True.
End If
```

In fact, in this identification, `locBoolean = True` is not a characteristic of the *If* statement, but basically just a normal variable assignment. When *locBoolean* has a value of *True,* the entire expression is *True*, as well. The only thing the statement *If* does is examine the *Boolean* value that follows it. It then runs the following statements only if the expression evaluates to *True*. That's why the value doesn't need to be additionally checked by the programmer—it's redundant. The equal sign here is the comparison operator. In other words, if you replace *locBoolean* with its current value, the preceding code looks as follows:

```
If True = True Then
```

You can quite confidently replace this by *If True Then* ... or *If locBoolean Then* ... (if it is true, then ...).

However, it's certainly confusing in BASIC (not only in Visual Basic), that assignment operators and comparison operators use the same character (the equal sign). For example, the following expression is a valid statement:

```
Dim locBoolean = "Klaus" = "Uwe"
```

The first equal sign functions as an assignment operator; the second is a *Boolean* comparison operator. The comparison operator has a higher priority than the assignment operator, so in this example *locBoolean* takes the value *False*, because the string "Klaus" does not equal "Uwe." Otherwise, this example would produce a type conversion error.

However, other languages aren't that much better. For example, C++ uses a single equal sign (=) for assignment, and a double equal sign (==) for comparison, which is not intuitive at all. As many as 5–8 percent of all errors in C++ programs can be traced back to this confusion, which is a much higher number than the errors caused by the incorrect use of the equals character in Visual Basic.

The *If* code block can be followed by an *Else* code block, which runs when the *Boolean* expression behind the *If* evaluates to *False*. In addition, you can use an *ElseIf* code block to insert further evaluations into the *If* construct. The code block behind the last *Else* code block, if present, runs only when none of the conditions of the individual *If* or *ElseIf* sections returned *True*, as shown in the example that follows:

```
locString1 = "Santa Claus, you think he's in town already?"
locString2 = "Santa Klaus*"
locBoolean = (locString1 Like locString2)   ' returns False; checks for similarity (see
➥below)

If locBoolean Then
    'Boxing is possible, too:
    If locString2 = "Santa Klaus" Then
        Console.WriteLine("Name found!")
    Else
        Console.WriteLine("No name found!")
    End If
ElseIf Now = #12:00:00 PM# Then
    Console.WriteLine("Noon!")
ElseIf Now = #12:00:00 AM# Then
    Console.WriteLine("Still up so late?")
Else
    Console.WriteLine("It is any other time or locString1 was not like locString1...")
End If
```

The Logical Operators *And*, *Or*, *Xor*, and *Not*

Visual Basic has a variety of logical operators, which can be applied to numerical as well as *Boolean* date types. The latter are mainly of interest because they can be used to formulate almost natural language conditions for program sequencing, for example in *If* constructs.

If you act on the assumption that the two numeric values 0 and 1, represented by the *Boolean* type, correspond to *True* and *False*, it's possible to formulate connections such as, "If statement1 *and* statement2 are true, then..."

The most important logical operators are:

- *And* Executes a logical AND operation. Both statements must be true to make the expression true, as illustrated in the following table:

Statement 1	Statement 2	Result
False (0)	False (0)	False (0)
True (1)	False (0)	False (0)
False (0)	True (1)	False (0)
True (1)	True (1)	True (1)

- *Or* Executes a logical Or operation. When at least one of the statements is true, the expression is true:

Statement 1	Statement 2	Result
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
True (1)	True (1)	True (1)

- *Xor* Executes a logical exclusive Or operation. Only when exactly *one* of the two statements is true is the statement true; otherwise, it's false:

Statement 1	Statement 2	Result
False (0)	False (0)	False (0)
True (1)	False (0)	True (1)
False (0)	True (1)	True (1)
True (1)	True (1)	False (0)

- *Not* Negates the statement. If it is true, it becomes false, and if it is false, it becomes true:

Statement	Result
False (0)	True (1)
True (1)	False (1)

The following example illustrates how to use some of the logical operators. This code checks whether the character a user enters when selecting an option in a console application is within a certain range:

```
    Sub ConditionCheck()

        'Output options and read characters from the keyboard.
        Console.Write("Which function would you like to execute (1-9, 0 or 'end' to
➥end): ")
        Dim input = Console.ReadLine

        'When entering "0" or "end",
        If input = "0" Or input.ToUpper = "END" Then
            'End method.
            Exit Sub
        End If

        'When the pressed character (string length=1) is greater than or equal to "1"
        'and lesser or equal to "9"...
        If input.Length = 1 And input >= "1" And input <= "9" Then
            '...then it was a valid selection, ...
            Console.WriteLine("This function is possible!")
            '...otherwise...
        Else
            '...not.
            Console.WriteLine("You have made the wrong choice.")
        End If
    End Sub
```

Note Logical operators can be applied to *Boolean* values and other numeric data types. By doing this, the bits that compose the values internally are linked. For example, the following code results in 5:

```
13 And 7
```

This is because the following operation is executed in binary:

```
    1101 (13)
And 0111 (07)
-------------
    0101 (05)
```

Each bit of the initial value is linked with each bit of the second value using *And*—and it then returns the result.

Comparison Operators That Return *Boolean* Results

Visual Basic understands the following comparison operators, which compare two expressions and return a *Boolean* result:

- *Expression1* = *Expression2* Checks for equality; returns *True* when both expressions are the same.
- *Expression1* > *Expression2* Returns *True* when *Expression1* is greater than *Expression2*.

- *Expression1 < Expression2* Returns *True* when *Expression1* is less than *Expression2*.
- *Expression1 >= Expression2* Returns *True* when *Expression1* is greater than or equal to *Expression2*.
- *Expression1 <= Expression2* Returns *True* when *Expression1* is less than or equal to *Expression2*.
- *Expression1 <> Expression2* Checks for inquality; returns *True* when *Expression1* is not the same as *Expression2*.
- *Expression1 Is [Expression2|Nothing]* Checks for equality of an object reference (only applicable to reference types); returns *True* when *Expression1* points to the same data memory range as *Expression2*. When *Expression1* is not assigned to a range of memory (defined object variable, but not an instantiated object), the comparison with *Is* uses *Nothing* to return the *Boolean* value of *True*.
- *Expression1 IsNot [Expression2|Nothing]* Check for the non-equality of an object reference (only applicable to reference types); returns *True* when *Expression1* points to a different data memory range than *Expression2*. When *Expression1* points to a valid memory range with instance data, the comparison with *IsNot* uses *Nothing* to return the *Boolean* value of *True*.
- *String1 Like String2* Checks for similarity between two strings; a sample comparison can make the comparison more flexible. If both strings are equal, according to certain rules *True* is returned, otherwise *False*.

You can find more details in MSDN under the item, Like Operator.

The following lines of code demonstrate the use of comparison operators:

```
Dim locString1 As String = "Uwe"
Dim locString2 As String = "Klaus"

locBoolean = (locString1 = locString2)      ' Returns False.
locBoolean = (locString1 > locString2)      ' Returns True.
locBoolean = (locString1 < locString2)      ' Returns False.
locBoolean = (locString1 >= locString2)     ' Returns True.
locBoolean = (locString1 <= locString2)     ' Returns False.
locBoolean = (locString1 <> locString2)     ' Returns True.
locBoolean = (locString1 Is locString2)     ' Returns False.

locString2 = "Uwe"
String.Internal(locString2)                   ' Now returns True, because both
locBoolean = (locString1 Is locString2)       ' String objects point to a range. (see
➥chapter 7)

locString1 = "Santa Claus, you think he's in town already?"
locString2 = "Santa Klaus*"
locBoolean = (locString1 Like locString2)   ' Returns True.
```

Short Circuit Evaluations with *OrElse* and *AndAlso*

Take a look at the following code block:

```
'Short circuit evaluation speeds up the process.
If locChar < "0" OrElse locChar > "9" Then
    locIllegalChar = True
    Exit For
End If
```

Note the keyword *OrElse*. Another keyword that you can use that functions in a similar manner to *OrElse* is *AndAlso*. Both correspond to the commands *Or* and *And*, and they also serve to logically link and evaluate *Boolean* expressions—they just operate differently (and sometimes faster). Let's look at an example from our daily life to clarify the concept.

Suppose that you're wondering whether you should bring an umbrella along for your daily walk; it might rain, *or else* at least it looks pretty dark outside. But you would no longer need to consider how the sky looks if just before you leave, you discover that it actually *is* raining. If it's raining right now, there is no need for you to check the second criterion—the umbrella must come along; otherwise, you're going to get wet. That's exactly what *OrElse* does (or *AndAlso* respectively). This approach is called *Short Circuit Evaluation*.

Especially with objects or when calling methods, short circuit evaluation can help make your programs safer, as the following example shows:

```
Private Sub btnAndAlsoDemo_Click(ByVal sender As System.Object, ByVal e As System.
EventArgs) _
Handles btnAndAlsoDemo.Click
    Dim aString As String = "Klaus is the word that starts the sentence"
    If aString IsNot Nothing AndAlso aString.Substring(0, 5).ToUpper = "KLAUS" Then
        MessageBox.Show("The string begins with Klaus!")
    End If

    If aString IsNot Nothing And aString.Substring(0, 5).ToUpper = "KLAUS" Then
        MessageBox.Show("The string begins with Klaus!")
    End If
End Sub
```

As expected, this program code displays the message text twice, because *aString* has content in both cases, and in both cases, the string starts with "Klaus" (after all, it's the same string). Now replace the bold-highlighted line in preceding example with the line that follows:

```
    Dim aString As String = Nothing
```

Figure 1-14 shows the results when you run the program one more time.

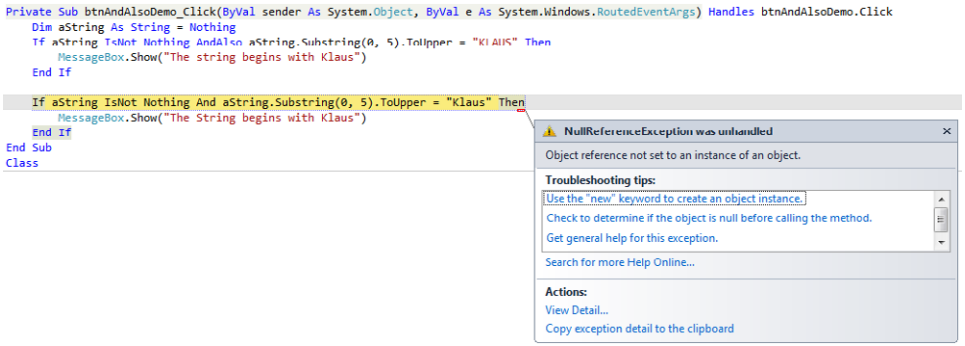

FIGURE 1-14 *AndAlso* helps you with combined queries for *Nothing* and the use of instance methods.

Here, it becomes apparent how *AndAlso* works. The first query works because the second part, which is linked with *AndAlso,* doesn't even execute the code `aString.Substring(0, 5).ToUpper = "KLAUS"` any more. That's because the object *aString* was *Nothing*. When using *AndAlso*, if the first test evaluates to false, the rest of the checks are no longer of interest, and they are therefore ignored.

In the second version that uses *And*, the second part executes—even though it makes no sense to execute the code because the first expression is false. But because *aString* now has the value *Nothing,* you can't use its instance functions (*SubString*, *ToUpper*), so the program aborts with a *NullReferenceException*.

Select ... Case ... End Select

As shown in the previous section, you can use *ElseIf* for option analysis when you want to evaluate several *Boolean* expressions and you need to respond by running the corresponding program code. You can achieve the same result in a much more elegant fashion by using a *Select* construct. *Select* prepares an expression for comparison with a *Boolean* result; the actual comparison takes place by using one or several different *Case* statements, each of which must be followed by a corresponding comparison argument of the same type (or implicitly convertible). If none of the conditions following the *Case* statements apply, an optional *Case Else* at the end can execute statements that you want to run when none of the *Case* statements prove true. The *End Select* command completes the construct. When a *Case* expression evaluates to *True*, *Select* runs the code for that *Case*, but doesn't perform any further evaluations.

When evaluating conditions, *Case* by default checks only for equality; however, by adding the keyword *Is*, you can also use other comparison operators. The following example shows how to apply them:

```
Dim locString1 as String = "Miriam"

Select Case locString1
   Case "Miriam"
      Console.WriteLine("Hit!")
   Case Is > "N"
      Console.WriteLine("Name comes after 'M' in the alphabet")


   Case Is < "M"
      Console.WriteLine("Name comes before 'M' in the alphabet")

   Case Else
      Console.WriteLine("Name starts with 'M'")

   'Case Like "Miri"
      'This doesn't work!!!

End Select
```

However, comparison operations and conditional execution happen in one go here. This means that the following construct won't work:

```
'It doesn't work this way!!!
Select Case locBoolean

   Case
Console.WriteLine("Was true!")

End Select
```

The compiler has every right to complain.

The *If* Operator and *IIf* Function

If exists not only as a structure statement (*If* … *Then* … *Else*), but also as a function—and in two variations. The *IIf* function (mentioned briefly earlier, and yes, it's spelled with a double "I") takes three parameters, and returns either the first or the second expression as a result, depending on the result of the first *Boolean* expression:

```
Dim c As Integer
'Returns 10
c = CInt(IIf(True, 10, 20))
'Returns 20
c = CInt(IIf(False, 10, 20))
```

IIf has one big disadvantage: when using the *IIf* function, the result needs to be cast to the correct type, because the *IIf* function only returns a base *Object* type, as shown here:

```
Dim c As Integer
'Returns 10
c = CInt(IIf(True, 10, 20))
```

Beginning with Visual Basic 2008, this has become easier because the keyword *If* (with one "I") has been extended, so you can use it the same way as the *IIf* function:

```
Dim c As Integer
'Returns 20
c = If(False, 10, 20)
```

For even fewer keystrokes, combine the *If* operator with local type inference, as follows:

```
'Returns 20
Dim c = If(False, 10, 20)
```

However, be careful; mixing various return types can cause the compiler to stumble, as shown in Figure 1-15.

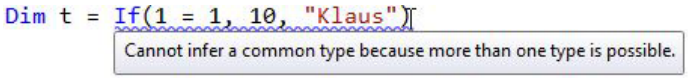

FIGURE 1-15 Using *If* instead of *IIf* only works if the compiler has the chance to clearly determine the corresponding types.

Loops

Loops let you to run the commands and the statements they contain over and over again, until a certain loop end state (called an *exit condition*) is achieved. Referring back to Figure 1-6, you can see an example of a loop in a *Do ... Loop* construct, which allows users to repeatedly enter birth dates or commands until they type a whitespace (a space or a tab or a return) at the command prompt. When that happens, the variable *input* stays empty, and the loop reaches its exit condition, which was defined after the keyword *Loop*.

In addition to *Do ... Loop*, Visual Basic recognizes *For ... Next* loops, which repeat all internal statements until a counter variable has exceeded or dropped below a specified value. Also, in Visual Basic, you can use *For Each ... Next* loops, which repeat their internal statements as often as there are elements in an object list, as well as *While ... End While* loops, which repeat their internal statements as long as a certain exit condition is still valid (defined right after the *While* keyword). When there is no exit condition for a loop, it is called an *infinite loop*—and the content of the loop will run until you can no longer pay your electric bill (which, in a manner of speaking, constitutes an exit condition.)

For … Next Loops

For … Next loops let you repeat the statements within the code block for a predefined number of times. The syntax for such a construction is as follows (elements in square brackets are optional, which means that they don't *need* to appear in each *For … Next* loop construct):

```
For counterVariable [As dataType] = Start To End [Step by step]
    [Statements]
    [Exit For]
    [Continue For]
Next [counterVariable]
```

The *counterVariable* of the *For* statement is a central part of the entire loop construct; it is therefore not optional—you must define it. *DataType* determines which numeric type *counterVariable* should be (for example, *Integer* or *Long*), but you can omit that when your project has the local type inference turned on, or if you already defined the variable beforehand. The latter is important when you want the variable to be valid after the loop exits and you want to use the value.

Important Theoretically, you can also use floating point variable types, such as *Double* and *Single*, but you should try to avoid those. Because of rounding inaccuracies, which are normal during the conversion from the internal binary system to the decimal system, you can't specify the query for marginal values precisely. Therefore, a loop could potentially be repeated one too many or too few times, even though you don't expect it, or a targeted check for a certain value could fail because of rounding errors. For example, even though the value 63.0000001 is close to 63, it's still not 63. Checking the variable to see if it's equal to 63 by using the *If* command would fail.

If you can't do without floating-point numbers in loops, consider using the more accurate but much slower *Decimal* type, or test for certain values in ranges rather than total equality, such as:

```
If doubleVar >= 10.3 And doubleVar < 10.4 Then
    [Statements]
End If
```

The Start and *End* are required and define the initial value for *counterVariable*. In other words, they specify where the counting starts as well as the limiting value, which determines how many times the loop will execute. By default, the loop counter is incremented by using a step size of one for every loop iteration; however, by using the *StepSize* parameter you can define the increment.

Another way to exit a *For … Next* loop is with the *Exit For* command. Using *Exit For* makes sense only if you place it inside an *If* query within a loop. *Exit For* exits the loop and continues running the program by running statements that follow the *Next* keyword, which marks the end of the loop. The keyword *Next* at the end of the loop is required; it delimits the bottom of the *For* loop.

Tip Use a *For* … *Next* loop construct when you already know how many loop iterations to expect. For example, a *For* … *Next* loop is perfectly suited for an array with a certain number of elements that to process:

```
Dim Names() As String = {"Lisa Feigenbaum",
                         "Sarika Calla",
                         "Ramona Leenings",
                         "Adriana Ardelean"}

'Run through For loop and return:
For index As Integer = 0 To Names.Length - 1
    Console.WriteLine(Names(index))
Next
```

For cases in which you don't know beforehand how many iterations will be necessary, it is better to use a *Do* … *Loop*, which is described in more detail later in this chapter.

The previous code example displays the following output:

```
Lisa Feigenbaum
Sarika Calla
Ramona Leenings
Adriana Ardelean
```

Note that the elements of an array or a list are always 0-based. That's why the value 1 is subtracted from the length of the array; that's the upper limit for running through the array (Length returns the number of array elements).

You can "nest" *For* … *Next* loops, placing an inner loop inside an outer loop; however, they can't overlap. You can also explicitly tie a *Next* statement to a particular *For* statement by adding the counter variable name after the *Next* statement. When a *Next* statement of an outer nesting level has been placed before the *Next* statement of an inner level loop, the compiler complains, and you will see a corresponding message in the error list. On the one hand, the compiler can recognize the overlap error only when you specify the corresponding counter variable in each *Next* statement; on the other hand, it assumes the correct counter variable, when you don't name one explicitly. Therefore, the latter (omitting the counter variable after the *Next* statement) is often the better solution.

Note The *stepSize* value can be positive or negative for integers. If the step size is negative, you need to adjust the loop's start and end values accordingly—in this case, the start value must then be greater than the end value so that several loop iterations can take place. If you don't specify a value for *stepSize*, the default is 1, as mentioned earlier. Also important: if *start* is greater than *end*, the step size is not automatically set to –1; unless you want 1 as the step size, you must explicitly specify the step size so that several loop iterations can take place.

Important *start*, *end* and *stepSize* can be either constant values or calculated and complex expressions, as presented here:

```
For dayCounter= 1 To (Now.Date - New Date(Now.Year, Now.Month, 1)).TotalDays
    (Statements)
Next
```

This example states, as applicable to all expressions: they are evaluated only once, then the loop begins to run. It is important to be aware of that when considering performance issues (constructs, as in the example, don't slow the program down, because they are evaluated only once) and program flow. You cannot change the limit value that controls the end of the loop while the loop is running. Also important: you shouldn't manipulate the counter variable in code within the loop, because that can lead to unforeseen results or errors.

Nesting *For* Loops

For loops can be nested within each other, which means that within a *For* loop, you can place other *For* loops. Remember that the total number of iterations of the innermost loop will be the inner loop count multiplied by the outer loop count when all loops use constant values. For example:

```
For count1 As Integer = 1 To 10
    For count2 As Integer = 1 To 10
        For count3 As Integer = 1 To 10
            'The statements are run through a total of 10*10*10=1000 times.
        Next count3
    Next
Next count1
```

Tip You don't need to specify the *counterVariable* after a *Next* statement. However, it sometimes makes your program easier to read, especially for deeply nested code.

For … Each Loops

A *For … Each* loop is similar to a *For … Next* loop, but it repeats the inner statements for each element of a specified list or array. In general, you use *For … Each* as follows:

```
For Each element [As DataType] In list
    [Statements]
    [Exit For]
    [Statements]
    [Continue For]
Next [element]
```

The variable *element* represents an item in the list. The *For … Each* loop thus runs through the list elements—one list element is assigned to the *element* variable for each loop iteration. The inner statements in the loop are run repeatedly, once for each element contained in the list or array *list*.

Optionally, you can use *Exit … For* to prematurely end the loop iterations. This is why it makes sense to put *Exit … For* into its own *If* block.

The *Next* statement is required; it ends the definition of the *For* loop. However, you don't need to specify *element* after *Next*.

Note Chapter 19 contains detailed information about lists as well as more in-depth coverage of the functionality of *For … Each* loops.

The following example shows how to use a *For … Each* loop to run through the *String* array created in the previous example. It displays the contents of all the array elements in the console window, as shown in the following:

```
Dim Names() As String = {" Adriana Ardelean",
                         "Sarika Calla",
                         "Ramona Leenings",
                         "Beth Messi",
                         " Lisa Feigenbaum "}

'Iteration using ForEach loop and return:
For Each name In Names
    Console.WriteLine(name)
Next
```

This code displays the following output in the console window:

```
Adriana Ardelean
Sarika Calla
Ramona Leenings
Beth Messi
Lisa Feigenbaum
```

Tip Sometimes you need to know when the last element in a list has been reached. In such cases, don't use *For … Each*; use *For … Next* instead:

```
'Last element is treated differently:
For index As Integer = 0 To Names.Length - 1
    If index < Names.Length - 1 Then
        Console.WriteLine(Names(index))
    Else
        Console.WriteLine("and last but not least: " & Names(index))
    End If
Next
```

Of course, it's possible to use an additional counter variable in the *For … Each* loop, but that's pretty much redundant here, because you can just use *For … Next*.

This code displays the following output in the console window:

```
Adriana Ardelean
Sarika Calla
Ramona Leenings
Beth Messi
and last but not least: Lisa Feigenbaum
```

Do … Loop and *While … End While* Loops

The *Do … Loop* and *While … End* loops repeat statements inside the program, either as long as a condition is *True,* or until the condition becomes *True*. Together with *Do*, the keyword *Until* indicates that a condition must become *True*. To run a loop as long as the condition is *True,* use *While*.

The *While* and *Until* keywords are placed either after the loop start (*Do*) or after the loop end (*Loop*).

Note that a *Do While … Loop* provides the same result as a *While … End While* loop. For historic reasons you can use a different syntax to get to the same result. You can create a loop construct with the appropriate exit conditions using the following code variations:

```
Do {While|Until} condition
    [statements]
    [Exit Do]
    [statements]
Loop
```

Or:

```
Do
    [statements]
    [Exit Do]
    [statements]
Loop {While|Until} condition
```

Or:

```
While condition
    [statements]
    [Exit While]
    [statements]
End While
```

When using a *While ... End While* or *Do ... Loop, Boolean* variables or expressions serve as exit conditions. The following lines of code show a few examples for using these loop types:

```
Dim Names() As String = {"Adriana Ardelean",
                         "Sarika Calla",
                         "Ramona Leenings",
                         "Beth Messi",
                         "Lisa Feigenbaum"}

'Return the names in ascending order ...
Dim index As Integer = 0
Do While index < Names.Length
    Console.WriteLine(Names(index))
    index += 1
Loop

Console.WriteLine("-------------------------")
Console.WriteLine("      and backwards:")
Console.WriteLine("-------------------------")

'... and in descending order.
index = Names.Length - 1
While index > -1
    Console.WriteLine(Names(index))
    index -= 1
End While
```

These lines of code display the following on the monitor:

```
Adriana Ardelean
Sarika Calla
Ramona Leenings
Beth Messi
Lisa Feigenbaum
-------------------------
      and backwards:
-----------------------
Lisa Feigenbaum
Beth Messi
Ramona Leenings
Sarika Calla
Adriana Ardelean
```

Exit—Leaving Loops Prematurely

The *Exit* statement causes a program to exit the loop and continue execution after the end of the loop. For a *For* loop, program execution continues after the *Next* statement; for a *Do* loop, it continues after the *Loop* command; and for a *While* loop, the program continues after *End While*. Sometimes, you want to end a loop early, such as when you discover a condition that makes continuation unnecessary, impossible, or undesirable; for instance, an inaccurate

value, or a call for exiting. As an example, when you catch an exception in a *Try … Catch … Finally* statement (see the section "Error Handling in Code," later in this chapter), you can, for example, use *Exit For* at the end of the *Finally* block.

You can insert the desired number of *Exit* statements at any point within a loop. *Exit* is often used after the evaluation of a condition; for example, in an *If … Then … Else* structure.

Continue—Repeating Loops Prematurely

You can also run a loop early, which means that in *For … Next* loops, you can, in certain cases, treat *Next* preferentially. You can do this by using *Continue*. Of course, you can use *Continue* in all other loop types, as well.

Simplified Access to Object Properties and Methods Using *With … End With*

By using *With … End With*, you can include a number of elements (properties, methods) of an object within a code block repeatedly without to the need to name the object name each time anew. When the fully-qualified name for the corresponding object variable is very long, *With … End With* can be used not only to save keystrokes, but also to enhance performance. At the same time, you lower the risk of misspelling one of its elements.

For example, if you'd like to use several different properties of a single object, write the statements for property assignments into a *With … End With* structure. When you do this, you no longer need to point to the object in each property assignment. A single reference to the object, by which you place a period just before the property name, is enough (see Figure 1-16).

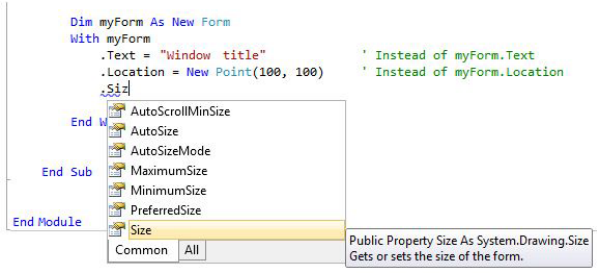

FIGURE 1-16 In a *With … End With* block, use the period to access the list of methods and properties of the object, which was specified behind *With*.

Note Remember that *With … End With* builds a structure, as well. Variables that are defined between *With* and *End With* only exist in this scope. The following section addresses this topic further.

The Scope of Variables

Declaring variables at procedure level (for instance, within the *Sub* and *Function* methods, or *Property* procedures, which are covered in Chapter 9, "First Class Programming") has an effect on their scope. While in VBA and Visual Basic 6.0, for example, variables within procedures are still valid throughout that procedure from the moment they are declared, in .NET versions of Visual Basic, they are valid from the moment they are declared only within the code block in which they are defined. A "code block" in this context basically means a construct that encapsulates code in some way; for example, *If ... Then ... Else* blocks, *For ... Next* or *Do ... Loop* loops, or even *With* blocks, among others. The Visual Studio IDE makes it relatively easy to see where blocks apply: each structure statement that causes the editor to indent code placed between the start and end commands for the block automatically limits the scope of the variables defined in that block. Figure 1-17 shows an example of this.

FIGURE 1-17 Variables are scoped to the structure block in which they are declared.

Figure 1-17 illustrates how the second time the program tries to access *fiveFound*, after completing the loop structure range (and thus also leaving the scope of the variable), an error occurs.

To make *fiveFound* accessible in both the *For ... Next* and the *If* structure block, you need to move the *fiveFound* definition to the procedure level, making the changes highlighted in the following code:

```
Sub Main()

    Dim fiveFound As Boolean

    For counter As Integer = 0 To 10
        'Dim fiveFound As Boolean
        If counter = 5 Then
            fiveFound = True
        End If
    Next
```

```
    If fiveFound Then
        Debug.Print("The number 5 was part of the list of numbers!")
    End If
End Sub
```

This variable scoping rule means that you can declare variables with the same name multiple times, as long as they are in different structural code blocks, as the following example shows:

```
Sub Main()
    For counter As Integer = 0 To 10
        Dim fiveFound As Boolean
        If counter = 5 Then
            fiveFound = True
        End If
    Next
    For counter As Integer = 0 To 10
        Dim fiveFound As Boolean
        If counter = 5 Then
            fiveFound = True
        End If
    Next
End Sub
```

The preceding code declares the variable *fiveFound* twice within a single procedure, but no error occurs, because each declaration is placed in a different scope.

Note, however, that you can't declare variables with the same name in nested structures; that won't work because variables in a parent structure are always accessible from a child structure. Figure 1-18 shows how the corresponding error message would look.

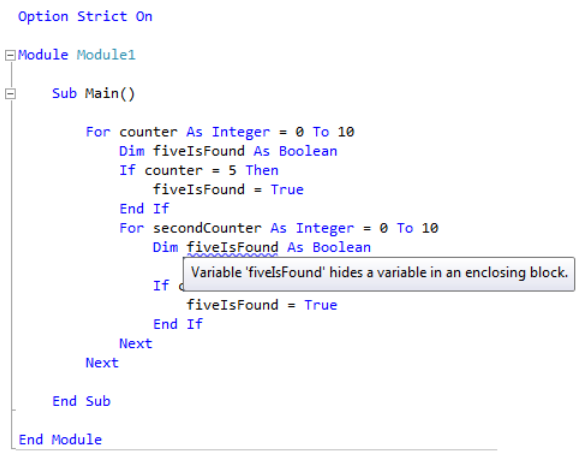

FIGURE 1-18 Variables of a parent scope must not hide the variables in a child scope.

The += and –= Operators and Their Relatives

Visual Basic provides abbreviation operators for numeric calculations, such as the += and –= operators, which add or subtract and assign values in one operation, and for string concatenations, the &= operator, which appends and assigns a string to another string in one operation. The following example (from the next section) shows how you would use +=:

```
Do
    If (helpValue And 1) = 1 Then
        resultValue += value1
    End If
    value1 = value1 << 1
    helpValue = helpValue >> 1
Loop Until helpValue = 0
```

There are other abbreviation operators in Visual Basic that work in much the same way. Table 1-2 shows these operators, along with a brief description:

TABLE 1-2 Operator Abbreviations in Visual Basic

Operation	Abbreviation	Description
var = var + 1	*var += 1*	Increase the variable content by one
var = var—1	*var –= 1*	Decrease the variable content by one
*var = var * 2*	*var *= 2*	Multiply the variable content by two (double it)
var = var / 2	*var /= 2*	Divide the variable content by two (floating-point division)
var = var \ 2	*var \= 2*	Divide the variable content by two (integer division)
var = var ^ 3	*var ^= 3*	Raise the variable content to the power of three
varString = VarString & "Ramona"	*varString &= "Ramona"*	Add the string "Ramona" to the content of the string *varString*

Note Using abbreviations simply saves keystrokes, which means it's less work, but doesn't cause the code to run any faster.

It doesn't matter whether you write the following:

```
intvar = intvar + 1
```

Or, whether you use this, instead:

```
intvar += 1
```

The compiler generates the same code with either syntax.

The Bit Shift Operators << and >>

In addition to the previously mentioned operators, Visual Basic 2003 already introduced bit shift operators. These operators shift the individual bits of integer values to the left or to the right. This book doesn't discuss the functionality of the binary system in depth, but briefly, shifting the bits of an integer value to the left doubles the value; shifting it to the right divides it by two (without a remainder).

For example, a binary 101 (decimal 5) becomes binary 10 (decimal 2) when shifted one bit to the right. For right shifts, the rightmost value simply falls off. Similarly, binary 101 becomes binary 1010 (decimal 10) with a shift to the left; left shifts add zeroes at the right.

The code that follows demonstrates a multiplication algorithm at bit level. Older developers might remember their times with the Commodore 64. Back then, such algorithms were used in Assembler every day.

```
Private Sub MultiplicationWithBitShift
    Dim value1,value2, resultValue, helpValue As Integer
    value1 = 10
    value2 = 5
    resultValue = 0
    helpValue = value2

        'This algorithm works the same as multiplying
        'the old fashioned way in the decimal system:
        '
        '(10)   (5)
        '1010 * 101 =
        '------------
        '      1010 +
        '     0000  +
        '    1010
        '------------
        '    110010 = 50

        'The "5" is shifted to the right bitwise,
        'to test its outer right bit. If it is set,
        'the value 10 is first added, and then its
        'complete "bit content" is moved one place to the left;
        'when it is not set, nothing happens.
        'This process repeats until all
        'bits of "5" are used up, i.e. the variable helpValue,
        'which processes this value, has become 0.
        'For a multiplication, as many additions are necessary
        'as there are bits set in the second value.
```

```
    Do
        If (helpValue And 1) = 1 Then
            resultValue += wert1
        End If
        value1 = value1 << 1
        helpValue = helpValue >> 1
    Loop Until helpValue = 0
    Console.WriteLine("The result is:" & resultValue)
End Sub
```

Error Handling in Code

To be honest, error handling in the old Visual Basic versions, which still function in .NET today, has always been anathema to me. If an error occurred after a very long routine in the finished program, it was difficult and required a lot of effort to locate the exact position of the error. The system variable *Erl* made it possible to show the line in which the error occurred in the error handler, but to achieve this, you had to number the code lines manually—a procedure that seemed rather antediluvian, even back then.

Today, discovering the exact location of an error is much, much easier, even though Visual Basic still allows the original error-handling syntax. Here's a look at the before and after versions. The following example shows a small VBA routine, which reads a file into a *String* variable, or at least it tries to do that. Watch for comments in the list, which mark the program execution in case of an error with numbers.

Note The following example is Visual Basic 6.0/VBA Code.

```
Private Sub Command1_Click()

    Dim fileNotFoundFlag As Boolean
    On Local Error GoTo 1000

    Dim ff As Integer
    Dim myFileContent As String
    Dim lineMemory As String
    ff = FreeFile
    '1: The error occurs here
    Open "C:\ATextFile.TXT" For Input As #ff
    '3: then go here
    If fileNotFound Then
        '4: to show this message.
        MsgBox ("The file does not exist")
```

```
    Else
        'This block is only run,
        'when everything was ok.
        Do While Not EOF(ff)
            Line Input #ff, lineMemory
            myFileContent = myFileContent & lineMemory
        Loop
        Close ff
        Debug.Print myFileContent
    End If
    'And that's also important so that the
    'program doesn't run into the error routine.
    Exit Sub

    '2: Then the program jumps here
1000 If Err.Number = 53 Then
        fileNotFound = True
        Resume Next
    End If

End Sub
```

Amazing, isn't it? Just to catch a simple error, the program must jump back and forth like mad all over the code. And this example catches only one single error!

What's even more uncomfortable for me is that the *On Error GoTo* statement is still possible in all VB.NET derivations—even though it's now completely unnecessary—as you will see in a moment. Reading a text file functions a little differently here in the following example, because *Open* and *Close* don't exist in this Visual Basic 6.0 format any longer. But that's not what this section is about.

Important The only difference, at least concerning the error handling, is that the line numbers must have colons, like other jump labels. Translating the program above into VB.NET might yield the following possible result (unfortunately):

```
Private Sub btnFileReadDotNet_Click(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles btnFileReadDotNet.Click
    'IMPORTANT:
    'To access the IO objects, "Imports System.IO" must be
    'placed at the beginning of the code file!

    Dim FileNotFoundFlag As Boolean
    'Same rubbish here!
    On Error GoTo 1000
```

```
    Dim myFileContent As String
    '1: If an error occurs here
    Dim fileStreamReader As New StreamReader("C:\ATextFile.txt")
    '3: to land here again with Resume Next
    If FileNotFoundFlag Then
        '4: and to catch the error
        MessageBox.Show("File was not found!")
    Else
        'When no error occured, this block is run
        myFileContent = fileStreamReader.ReadToEnd()
        fileStreamReader.Close()
        'And the file content is displayed.
        Debug.Print(myFileContent)
    End If
    Exit Sub
    '2: program execution continues here
1000: If Err.Number = 53 Then
        FileNotFoundFlag = True
        Resume Next
    End If
End Sub
```

Elegant Error Handling with *Try/Catch/Finally*

VB.NET offers a much more elegant way of implementing error handling. In contrast to *On Error GoTo*, with the *Try/Catch/Finally* structure you can handle errors at the spot where they actually occur. Take a look at the following example, which is an adaptation of the previous example, this time using *Try/Catch*:

```
Private Sub btnFileReadDotNet_Click(ByVal sender As System.Object, ByVal e As System.
➥EventArgs) _
Handles btnFileReadDotNet.Click
    'IMPORTANT:
    'To access the IO objects, "Imports System.IO"
    'must be placed at the beginning of the code file!

    Dim myFileContent As String
    Dim fileStreamReader As StreamReader

    Try
        'Try the following commands.
        fileStreamReader = New StreamReader("C:\myTextFile.txt")
        myFileContent = fileStreamReader.ReadToEnd()
        fileStreamReader.Close()
        Debug.Print(myFileContent)
```

```
    Catch ex As FileNotFoundException
        'Here only FileNotFoundExceptions are caught
        MessageBox.Show("File not found!" & vbNewLine & _
            vbNewLine & "The exception text was:" & vbNewLine & ex.Message, _
            "Exception", MessageBoxButtons.OK, MessageBoxIcon.Error)
    Catch ex As Exception
        'Here all other exceptions, which
        'have not been handled yet, are caught
            'Here all other exceptions, which
            'have not been handled yet, are caught
            MessageBox.Show("An exception occured while processing the file!" & _
               vbNewLine & "The exception had the type:" & ex.GetType.ToString & _
               vbNewLine & vbNewLine & "The exception text was:" & vbNewLine & _
               ex.Message, "Exception", MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
End Sub
```

What happens here? All the statements placed between the *Try* statement and the first *Catch* statement occur in a kind of "trial mode." If an error occurs during execution of any of these wrapped statements, the program automatically jumps to the *Catch* block that most closely matches the error that occurred.

Catching Multiple Exception Types

If you enter the command *Try* in the Visual Studio Code Editor and then press Return, the Editor automatically inserts the following block:

```
Try

Catch ex As Exception

End Try
```

Exception is the name of the class that is highest in the Exception inheritance hierarchy, so it can catch all exceptions—without exception (sorry... I had to do it). In the preceding code, any exception that occurs is instantiated in the variable *ex*. But sometimes you want to handle different errors differently, so it makes sense (as seen in the sample code) to differentiate between the many different exception types. Perhaps your program needs to react differently to an exception triggered by a non-existent file than to one in which the file does exist, but is currently in use.

You can try this out by using the sample code. When you run it, you see an exception called *FileNotFoundException*. A *Catch* block using this class name for the exception type (*Catch ex As FileNotFoundException*) catches only exceptions of that particular type. In the preceding example, that exception produces the message box shown in Figure 1-19.

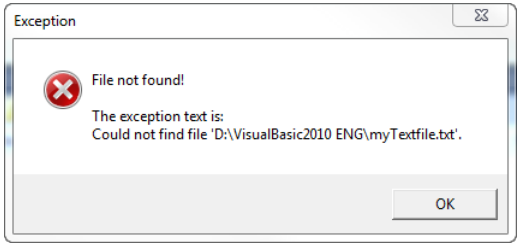

FIGURE 1-19 The sample code returns this message when the file didn't exist; in other words, when a *FileNotFoundException* occurred.

If you now create a text file on drive C under this name, and then open that file in Word, for example, the line again produces an exception, as illustrated here:

```
fileStreamReader = New StreamReader("C:\myTextFile.txt")
```

This time, however, it's not a *FileNotFoundException*, it's an *IOException*, which produces the error output shown in Figure 1-20.

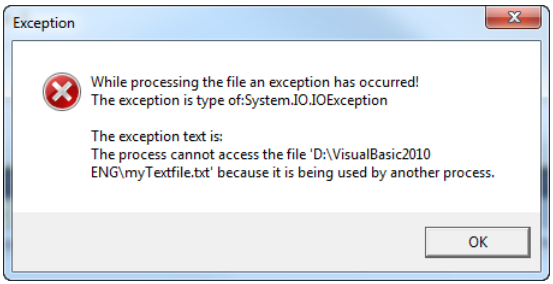

FIGURE 1-20 Now this file exists, but it's already open in Word.

This exception is caught with *Catch ex as Exception*, even though it has the type *IOException* because, *IOException* is derived from *Exception*. In object-oriented programming, you create extended classes by inheriting from existing classes, and from those classes you can derive others that are even further specialized, and so on. It's the same for exception classes. *Exception* is the base class. The exception class *SystemException* is derived from *Exception*, and *IOException* is derived from *SystemException*. Because the example doesn't explicitly handle *IOException* separately, the runtime chooses the *Catch* block that represents at least one of the base classes of the exception class that occurred. Using *Exception* is the catch-all method, because *all* other exception classes are based on it. The exception *FileNotFoundException* is handled separately, but for all other exceptions the *Catch ex as Exception* block executes.

Important You can implement as many *Catch* branches into a *Try-Catch* block as you like, and therefore, target and catch all conceivable exception types. However, be sure that you put the more specialized exception types at the top, and those they are based on further below. Otherwise, you'll get into trouble because the runtime executes the *Catch* block that first matches a base exception class, which can occur before the *Catch* block with the specialized exception. Figure 1-21 illustrates the potential problem:

```
Try
    fileStreamReader = New StreamReader("D:\VisualBasic2010 ENG\myTextfile.txt")
    myFileContent = fileStreamReader.ReadToEnd()
    fileStreamReader.Close()
    Debug.Print(myFileContent)
    Console.Write(myFileContent)
Catch ex As Exception
    MessageBox.Show("While processing the file an exception has occurred!" & vbNewLine & _
                    "The exception is type of:" & ex.GetType.ToString & vbNewLine & _
                    vbNewLine & "The exception text is:" & vbNewLine & ex.Message, _
                    "Exception", MessageBoxButtons.OK, MessageBoxIcon.Error)
Catch ex As FileNotFoundException
 'Catch' block never reached, because 'System.IO.FileNotFoundException' inherits from 'System.Exception'.
                    "Exception", MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try
```

FIGURE 1-21 Place *Catch* blocks with specialized exception classes before *Catch* blocks that handle base exception classes; otherwise, the computer complains.

Using the *Finally* Block

Program code placed in a *Finally* block *will always be executed*. Even when an error occurs and is caught with *Catch*, or even if that *Catch* block contains a *Return* statement, the *Finally* block will still execute.

Normally, *Return* is the last command in a procedure, no matter where *Return* was placed. In certain cases, however, it doesn't make much sense, especially when working with error handling, and therefore, *Finally* is the golden exception here.

Suppose that you are reading from a file, and to do so, you have opened the file. When opening the file, there was no error, but there was an error while reading it. Maybe you have read past the file end, and now lines which you are trying to process are empty (therefore, *Nothing*). You have handled this case (catching a *Null* reference) with the corresponding *Catch* block, and because there is nothing else to do, you want to leave your read routine directly from the *Catch* block, with *Return*, after an error message has been displayed. The problem is that the file is still open, and you should close it. With *Finally*, it is possible to implement such a process elegantly.

The following example simulates such a process. It reads the file c:\aTextFile.txt line by line, converting the individual lines into uppercase letters before combining them into a text block. However, because it tries to read too many lines, the *ToUpper* string method, which converts the line into uppercase letters, stops working at the point where the *ReadLine* function finds no more content, and therefore, returns *Nothing*:

```
Private Sub btnTryCatchFinally_Click(ByVal sender As System.Object,
   ByVal e As System.EventArgs) Handles btnTryCatchFinally.Click
    'IMPORTANT:
    'To access the IO objects, "Imports System.IO"
    'must be placed at the beginning of the code file!

    Dim meinDateiInhalt As String
    Dim fileStreamReader As StreamReader
    Try
        'Try the following commands.
        fileStreamReader = New StreamReader("C:\aTextFile.txt")
        Dim locLine As String
        myFileContent = ""
        'Now we read line by line, but too many lines,
        'and therefore shoot past the end of the file:
        Try
            ' If your file "C:\aTextFile" doesn't contain
            ' exactly 1001 lines, it goes bust somewhere in here:
            For lineCounter As Integer = 0 To 1000
                locLine = fileStreamReader.ReadLine()
                'locLine is now Nothing, and the conversion into
                'capitals can no longer work.
                locLine = locLine.ToUpper
                myFileContent &= locLine
            Next
        Catch ex As NullReferenceException
            MessageBox.Show("The line could not be converted, " &
               "because it was empty!")
            ' Return? But the file is still open!!!
            Return
        Finally
            ' No matter! Even with Return in the Catch block
            ' Finally is still executed!
            fileStreamReader.Close()
        End Try
        'But this line runs only when the Try succeeds:
        Debug.Print(myFileContent)
    Catch ex As FileNotFoundException
        'Only FileNotFoundExceptions are caught here
        MessageBox.Show("File not found!" & vbNewLine &
            vbNewLine & "The exception text was:" & vbNewLine & ex.Message,
            "Exception", MessageBoxButtons.OK, MessageBoxIcon.Error)
    Catch ex As Exception
        'Here all other exceptions, which
        'have not been handled yet, are caught
        MessageBox.Show("An exception occured while processing the file!" &
           vbNewLine & "The exception had the type:" & ex.GetType.ToString &
           vbNewLine & vbNewLine & "The exception text was:" & vbNewLine &
           ex.Message, "Exception", MessageBoxButtons.OK, MessageBoxIcon.Error)
    End Try
End Sub
```

Tip You can retrace this process quite well by setting a breakpoint (press F9) in the line that contains the *Return*, statement, and then running the program. When you step through the program by pressing F11, you will notice that even after the *Return* statement, the code in the *Finally* block is still executed.

Chapter 6

The Essential .NET Data Types

In this chapter:

In Chapter 1, "Beginners All-Purpose Symbolic Instruction Code," you learned about variables, including what the different types of variables are, and you saw a few examples of declaring and using variables of various data types. However, there's still a lot to learn about the base data types, which are part of the Microsoft .NET Framework. These are the data types that you'll be using over and over as a developer.

The base data types are mainly *primitive* data types, such as *Integer*, *Double*, *Date*, *String*, and so on, with which you're already familiar. They are an integral part of the C# and Microsoft Visual Basic.NET languages; you can recognize them easily because the Microsoft Visual Studio code editor colors them blue as soon as you declare them.

Base data types include all the types that are part of any .NET programming language, and they expose the following characteristics:

- **They can be used as variable values.** You can set the value of each base data type directly. For example, specifying a value of *123.324D* identifies a variable of type *Decimal* with a specific magnitude.
- **They can be used as constant values.** It is possible to declare a base data type as a constant. When a certain expression is exclusively defined as a constant (such as the expression *123.32D*2+100.23D*), it can be evaluated during compilation.
- **They are recognized by the processor.** Many operations and functions of certain base data types can be delegated by the .NET Framework directly to the processor for execution. This means that no program logic is necessary to calculate an arithmetic expression (for example, a floating-point division). The processor can do this by itself, so such operations are therefore very fast. Most of the operations of the data types *Byte*, *Short*, *Integer*, *Long*, *Single*, *Double*, and *Boolean* fall into this category.

Numeric Data Types

For processing numbers, Visual Basic provides the data types *Byte*, *Short*, *Integer*, *Long*, *Single*, *Double*, and *Decimal*. The data types *SByte*, *UShort*, *Uinteger*, and *ULong* were introduced with Visual Basic 2005. They differ in the range, the precision, or *scale*, of the values that they can represent (for instance, the number of decimal points), and their memory requirements.

Note The nullable data types have been around since Visual Basic 2005, and we discussed them briefly in the introduction to Chapter 1. Nullable data types behave similarly to their regular data type counterparts, but they can also reflect a non-defined state, namely *null*[1] (or *Nothing* in Visual Basic). With the release of Visual Basic 2008, these data types are also part of the Visual Basic language syntax, and they are defined by the question mark symbol as type literal. Here's how you declare an *Integer* data type as nullable:

```
Dim t As Integer?
```

You'll see more about nullable data types in Chapter 18, "Advanced Types."

Defining and Declaring Numeric Data Types

All numeric data types (as with all value types) are declared *without* the keyword *New*. Constant values can be directly assigned to numeric types in the program's code. There are certain keywords that define the type of a constant value. A variable of the type *Double* can, for example, be declared with the following statement:

```
Dim aDouble As Double
```

You can then use *aDouble* immediately in the code. Numeric variables can be assigned values, which are strings made up of digits followed (if necessary) by a type literal. In the following example, the type literal is the *D* following the actual value:

```
aDouble = 123.3D
```

Just as with other base data types, declaration and assignment can take place in a single statement. Therefore, you can replace the two preceding statements with the following single statement:

```
Dim aDouble As Double = 123.3D
```

If you use local type inference (refer to Chapter 1 for more information), you don't even need to specify the type or procedure level (for example, in a *Sub*, a *Function* or a *Property*, but not for module or class variables); you can let the compiler infer the correct type from the type of the expression or the constant from which the variable is assigned:

[1] Not to be confused with the value *0*! Null denotes the absence of a value.

```
Dim aDouble = 123.3D
```

Note Local type inference must be switched on through the property settings of your project. Alternatively, you can place *Option Infer On* at the top of the code file in which you want to use local type inference.

The example applies to all other numeric data types equally—the type literal can of course differ from type to type.

TABLE 6-1 Type Literals and Variable Type Declaration Characters of the Base Data Types in Visual Basic 2010

Type name	Type declaration character	Type literal	Example
Byte	–	–	`Dim var As Byte = 128`
SByte	–	–	`Dim var As SByte = -5`
Short	–	S	`Dim var As Short = -32700S`
UShort	–	US	`Dim var As UShort = 65000US`
Integer	%	I	`Dim var% = -123I` or `Dim var As Integer = -123I`
UInteger	–	UI	`Dim var As UInteger = 123UI`
Long	&	L	`Dim var& = -123123L` or `Dim var As Long = -123123L`
ULong	–	UL	`Dim var As ULong = 123123UL`
Single	!	F	`Dim var! = 123.4F` or `Dim var As Single = 123.4F`
Double	#	R	`Dim var# = 123.456789R` or `Dim var As Double = 123.456789R`
Decimal	@	D	`Dim var@ = 123.456789123D` or `Dim var As Decimal = 123.456789123D`
Boolean	–	–	`Dim var As Boolean = True`
Char	–	C	`Dim var As Char = "A"c`
Date	–	#MM/dd/yyyy HH:mm:ss# or #MM/dd/yyyy hh:mm:ss am/pm#	`Dim var As Date = #12/24/2008 04:30:15 PM#`
Object	–	–	In a variable of the type *Object*, any type can be boxed or referenced by it
String	$	"String"	`Dim var$ = "String"` or `Dim var As String = "String"`

Delegating Numeric Calculations to the Processor

The example that follows shows how to leave some mathematical operations to the processor. To do this, you need to know that, due to its computational accuracy, the *Decimal* type is calculated not by the floating-point unit of the processor, but by the corresponding program code of the Base Class Library (unlike *Double* or *Single*).

Note This example goes a little deeper into the system. You will learn how to view the Assembler and machine language representation—the concrete compilation of your program, the way the processor sees it. Although this is not a requirement for developing applications by any means, it's definitely an interesting experiment, and it will help you understand how the processor processes data.

Companion Content Open the solution Primitives01.sln, which you can find in the \VB 2010 Developer Handbook\Chapter 06\Primitives01 folder.

Before you run the following sample code, press F9 to set a breakpoint in the highlighted line.

```
Public Class Primitives
    Public Shared Sub main()
        Dim locDouble1, locDouble2 As Double
        Dim locDec1, locDec2 As Decimal

        locDouble1 = 123.434D
        locDouble2 = 321.121D
        locDouble2 += 1
        locDouble1 += locDouble2
        Console.WriteLine("Result of the Double calculation: {0}", locDouble1)

        locDec1 = 123.434D
        locDec2 = 321.121D
        locDec2 += 1
        locDec1 += locDec2
        Console.WriteLine("Result of the Decimal calculation: {0}", locDec1)

    End Sub
End Class
```

When you start the program, it will stop at the line with the breakpoint. On the Debug/Window menu, select Disassembly. This window will display what the Just-in-Time (JIT) compiler has done with the program, which is first compiled in the IML, as shown in the code that follows.

Note The Disassembly window can display only assembly code, which is not very well optimized. No setting can change that—you will always see the debug code, not the optimized code. Later, in the optimized code, the processor registers are used as carriers for local variables, whenever possible, which drastically increases the execution speed of your applications.

```
       Dim locDouble1, locDouble2 As Double
       Dim locDec1, locDec2 As Decimal

        locDouble1 = 123.434D
00000055  movsd       xmm0,mmword ptr [000002E8h]
0000005d  movsd       mmword ptr [rsp+50h],xmm0
        locDouble2 = 321.121D
00000063  movsd       xmm0,mmword ptr [000002F0h]
0000006b  movsd       mmword ptr [rsp+58h],xmm0
        locDouble2 += 1
00000071  movsd       xmm0,mmword ptr [000002F8h]
00000079  addsd       xmm0,mmword ptr [rsp+58h]
0000007f  movsd       mmword ptr [rsp+58h],xmm0
```

The numbered lines correspond to assembly language statements and show the operations required by the processor to execute the preceding Visual Basic statement. These are the statements that the processor understands, and no matter what language you're using to write your applications, at the end of the day, your code must be translated into a series of assembly statements. That's what compilers do for you. The listings in this section demonstrate (if nothing else) what a "high-level" language is all about.

Unlike what you might have expected, no special methods of the *Double* structure were called. Instead, the addition happens via the floating-point functionality of the processor itself (*addsd*,[2] marked in bold). It's quite different further down in the disassembly, where the same operations are carried out by using the *Decimal* data type:

```
locDec2 += 1
0000017e  mov         rcx,129F1180h
00000188  mov         rcx,qword ptr [rcx]
0000018b  add         rcx,8
0000018f  mov         rax,qword ptr [rcx]
00000192  mov         qword ptr [rsp+000000A8h],rax
0000019a  mov         rax,qword ptr [rcx+8]
0000019e  mov         qword ptr [rsp+000000B0h],rax
000001a6  lea         rcx,[rsp+000000A8h]
000001ae  mov         rax,qword ptr [rcx]
000001b1  mov         qword ptr [rsp+000000E0h],rax
000001b9  mov         rax,qword ptr [rcx+8]
000001bd  mov         qword ptr [rsp+000000E8h],rax
000001c5  lea         rcx,[rsp+40h]
000001ca  mov         rax,qword ptr [rcx]
000001cd  mov         qword ptr [rsp+000000D0h],rax
```

[2] Scalar Double-Precision Floating-Point Add

```
000001d5  mov         rax,qword ptr [rcx+8]
000001d9  mov         qword ptr [rsp+000000D8h],rax
000001e1  lea         r8,[rsp+000000E0h]
000001e9  lea         rdx,[rsp+000000D0h]
000001f1  lea         rcx,[rsp+000000B8h]
000001f9  call        FFFFFFFFEF381460
// Here the addition routine of …
000001fe  mov         qword ptr [rsp+00000128h],rax
00000206  lea         rcx,[rsp+000000B8h]
0000020e  mov         rax,qword ptr [rcx]
00000211  mov         qword ptr [rsp+40h],rax
00000216  mov         rax,qword ptr [rcx+8]
0000021a  mov         qword ptr [rsp+48h],rax
        locDec1 += locDec2
0000021f  lea         rcx,[rsp+40h]
00000224  mov         rax,qword ptr [rcx]
00000227  mov         qword ptr [rsp+00000110h],rax
0000022f  mov         rax,qword ptr [rcx+8]
00000233  mov         qword ptr [rsp+00000118h],rax
0000023b  lea         rcx,[rsp+30h]
00000240  mov         rax,qword ptr [rcx]
00000243  mov         qword ptr [rsp+00000100h],rax
0000024b  mov         rax,qword ptr [rcx+8]
0000024f  mov         qword ptr [rsp+00000108h],rax
00000257  lea         r8,[rsp+00000110h]
0000025f  lea         rdx,[rsp+00000100h]
00000267  lea         rcx,[rsp+000000F0h]
0000026f  call        FFFFFFFFEF381460
// … Decimal is called. Here also.
           .
           .
           .
```

The preceding code demonstrates that the addition requires many more preparations. This is because the values to be added must first be copied to the stack. The actual addition isn't performed by the processor itself, but by the corresponding routines of the Base Class Library (BCL), which is called by using the *Call* statement, shown in the disassembly (highlighted in bold).

Tip This example illustrates why the performance of the *Decimal* data type is only about one tenth of the performance of the *Double* data type. Therefore, you should use *Decimal* only when you need to perform exact financial calculations and can't tolerate any rounding errors. (For more information, read the section, "Numeric Data Types at a Glance," later in this chapter.)

A Note About Common Language Specification Compliance

Some of the data types introduced in Visual Basic 2005 don't conform to the Common Language Specification (CLS), or they simply aren't CLS-compliant. Some generic data types and some primitive data types that save integer values without prefixes (as well as the base data type *SByte*) belong to this group. Methods that accept types that are not CLS-compliant as arguments (or return values) should not be provided in components intended for use by other .NET programming languages. You must not assume that these types are "accessible" in all .NET languages. Visual Basic does not automatically check for CLS compliance. If you have components checked for compliance by the Visual Basic compiler, you can use the *CLSCompliant* attribute at class level, as follows:

```
<CLSCompliant(True)> _
    Public Class AClass

    Private myMember As UShort

    Public Property NotCLSCompliant() As UShort
        Get
            Return myMember
        End Get
        Set(ByVal value As UShort)
            myMember = value
        End Set
    End Property
End Class
```

Checking a single method for CLS compliance works the same way:

```
    <CLSCompliant(True)> _
    Public Shared Function ANonCLSComplianceMethod() As UShort
        Dim locTest As ClassLibrary1.AClass
        locTest = New ClassLibrary1.AClass
    End Function
```

Note Contrary to common belief, it's not true that a method, an assembly, or even your entire application is non–CLS-compliant if you are using just a single non-compliant type. Your assembly becomes non–CLS-compliant only when you expose variables of non–CLS-compliant types by making them *Public*.

Numeric Data Types at a Glance

The following short sections describe the use of numeric data types and the range of values that you can represent with each numeric type.

Byte

.NET data type: *System.Byte*

Represents: Integer values (numbers without decimal points) in the specified range

Range: 0 to 255

Type literal: Not available

Memory requirements: 1 byte

Declaration and example assignment:

```
Dim aByte As Byte
aByte = 123
```

Description: This data type stores only unsigned positive numbers in the specified numeric range.

CLS-compliant: Yes

Conversion of other numeric types: *CByte(objVar)* or *Convert.ToByte(objVar)*

```
aByte = CByte(123.45D)
aByte = Convert.ToByte(123.45D)
```

SByte

.NET data type: *System.SByte*

Represents: Integer values (numbers without decimal points) in the specified range

Range: –128 to 127

Type literal: Not available

Memory requirements: 1 byte

Declaration and example assignment:

```
Dim aByte As SByte
aByte = 123
```

Description: This data type saves negative and positive numbers in the specified numeric range.

CLS-compliant: No

Conversion of other numeric types: *CSByte(objVar)* or *Convert.ToSByte(objVar)*

```
aByte = CSByte(123.45D)
aByte = Convert.ToSByte(123.45D)
```

Short

.NET data type: *System.Int16*

Represents: Integer values (numbers without decimal points) in the specified range

Range: –32,768 to 32,767

Type literal: S

Memory requirements: 2 bytes

Declaration and example assignment:

```
Dim aShort As Short
aShort = 123S
```

Description: This data type stores signed numbers (both negative and positive) in the specified range. Conversion to the *Byte* data type can cause an *OutOfRangeException*, due to the larger scope of *Short*.

CLS-compliant: Yes

Conversion of other numeric types: *CShort(objVar)* or *Convert.ToInt16(objVar)*

```
'Decimal points are truncated
aShort = CShort(123.45D)
aShort = Convert.ToInt16(123.45D)
```

UShort

.NET data type: *System.UInt16*

Represents: Positive integer values (numbers without decimal points) in the specified range

Range: 0 to 65,535

Type literal: US

Memory requirements: 2 bytes

Declaration and example assignment:

```
Dim aUShort As UShort
aUShort = 123US
```

Description: This data type stores unsigned numbers (positive only) in the specified numeric range. Conversion to the *Byte* or *Short* data types can cause an *OutOfRangeException*, due to the (partially) larger scope of *Byte* or *Short*.

CLS-compliant: No

Conversion of other numeric types: *CUShort(objVar)* or *Convert.ToUInt16(objVar)*

```
'Decimal points are truncated
aUShort = CUShort(123.45D)
aUShort = Convert.ToUInt16(123.45D)
```

Integer

.NET data type: *System.Int32*

Represents: Integer values (numbers without decimal points) in the specified range

Range: –2,147,483,648 to 2,147,483,647

Type literal: I

Memory requirements: 4 bytes

Declaration and example assignment:

```
Dim anInteger As Integer
Dim anDifferentInteger%     ' also declared a integer
anInteger = 123I
```

Description: This data type stores signed numbers (both negative and positive) in the specified range. Conversion to the *Byte*, *Short*, and *UShort* data types can cause an *OutOfRangeException*, due to the larger scope of *Integer*. By appending the "%" (percent) character to a variable, the *Integer* type for the variable can be forced. However, in the interest of better programming style, you should avoid this technique.

CLS-compliant: Yes

Conversion of other numeric types: *CInt(objVar)* or *Convert.ToInt32(objVar)*

```
anInteger = CInt(123.45D)
anInteger = Convert.ToInt32(123.45D)
```

UInteger

.NET data type: System.UInt32

Represents: Positive integer values (numbers without decimal points) in the specified range

Range: 0 to 4,294,967,295

Type literal: UI

Memory requirements: 4 bytes

Declaration and example assignment:

```
Dim aUInteger As UInteger
aUInteger = 123UI
```

Description: This data type stores unsigned numbers (positive only) in the specified range. Conversion to the data types *Byte*, *Short*, *Ushort*, and *Integer* can cause an *OutOfRangeException*, due to the (partially) larger scope of *UInteger*.

CLS-compliant: No

Conversion of other numeric types: *CUInt(objVar)* or *Convert.ToUInt32(objVar)*

```
aUInteger = CUInt(123.45D)
aUInteger = Convert.ToUInt32(123.45D)
```

Long

.NET data type: *System.Int64*

Represents: Integer values (numbers without decimal points) in the specified range.

Range: –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Type literal: L

Memory requirements: 8 bytes

Declaration and example assignment:

```
Dim aLong As Long
Dim aDifferentLong& ' also defined as long
aLong = 123L
```

Description: This data type stores signed numbers (both negative and positive) in the specified range. Conversion to all other integer data types can cause an *OutOfRangeException*, due to the larger scope of *Long*. You can force a variable to a *Long* by appending the "&" (ampersand) character to a variable. However, in the interest of better programming style, you should avoid this technique.

CLS-compliant: Yes

Conversion of other numeric types: *CLng(objVar)* or *Convert.ToInt64(objVar)*

```
aLong = CLng(123.45D)
aLong = Convert.ToInt64(123.45D)
```

ULong

.NET data type: *System.UInt64*

Represents: Positive integer values (numbers without decimal points) in the specified range

Range: 0 to 18.446.744.073.709.551.615

Type literal: UL

Memory requirements: 8 bytes

Declaration and example assignment:

```
Dim aULong As ULong
aULong = 123L
```

Description: This data type stores unsigned numbers (positive only) in the specified numeric range. Conversion to all other integer data types can cause an *OutOfRangeException*, due to the larger scope of *ULong*.

CLS-compliant: No

Conversion of other numeric types: *CULng(objVar)* or *Convert.ToUInt64(objVar)*

```
aULong = CULng(123.45D)
aULong = Convert.ToUInt64(123.45D)
```

Single

.NET data type: *System.Single*

Represents: Floating-point values (numbers with decimal points whose scale becomes smaller with the increasing value) in the specified range

Note Scale in this context refers to the number of decimal points of a floating-point number.

Range: –3.4028235*1038 to –1.401298*10–45 for negative values; 1.401298*10–45 to 3.4028235*1038 for positive values

Type literal: F

Memory requirements: 4 bytes

Declaration and example assignment:

```
Dim aSingle As Single
Dim aDifferentSingle! ' also defined as Single
aSingle = 123.0F
```

Description: This data type stores signed numbers (both negative and positive) in the specified range. By appending the "!" (exclamation) character to a variable, you can foce the variable to the *Single* type. However, in the interest of better programming style, you should avoid this technique.

CLS-compliant: Yes

Conversion of other numeric types: *CSng(objVar)* or *Convert.ToSingle(objVar)*

```
aSingle = CSng(123.45D)
aSingle = Convert.ToSingle(123.45D)
```

Double

.NET data type: *System.Double*

Represents: Floating-point values (numbers with decimal points whose scale becomes smaller with the increasing value) in the specified range

Range: –1.79769313486231570*10308 to –4.94065645841246544*10–324 for negative values; 4.94065645841246544*10–324 to 1.79769313486231570308 for positive values

Type literal: R

Memory requirements: 8 bytes

Declaration and example assignment:

```
Dim aDouble As Double
Dim aDifferentDouble# ' also defined as Double
aDouble = 123.0R
```

Description: This data type stores numbers (both negative and positive) in the specified range. By appending the "#" (hash) character to a variable, you can force it to the *Double* type. However, in the interest of better programming style, you should avoid this technique.

CLS-compliant: Yes

Conversion of other numeric types: *CDbl(objVar)* or *Convert.ToDouble(objVar)*

```
aDouble = CDbl(123.45D)
aDouble = Convert.ToDouble(123.45D)
```

Decimal

.NET data type: *System.Decimal*

Represents: Floating-point values (numbers with decimal points whose scale becomes smaller with the increasing value) in the specified range

Range: Depends on the number of used decimal places. If no decimal places are used (called a scale of 0) the max/min values are between ±79,228,162,514,264,337,593,543,950,335. When using a maximal scale (28 places behind the period; only values between –1 and 1 can be stored at this scale) the max/min values are between ±0.9999999999999999999999999999.

Type literal: D

Memory requirements: 16 bytes

Declaration and example assignment:

```
Dim aDecimal As Decimal
Dim aDifferentDouble@ ' also defined as Decimal
aDecimal = 123.23D
```

Description: This data type stores signed numbers (both negative and positive) in the specified range. By appending the "@" (ampersand) character to a variable you can force the *Decimal* type. However, in the interest of better programming style, you should avoid this technique.

Important For very high values, you must attach the type literal to a literal constant to avoid an *Overflow* error message.

Note No arithmetic functions are delegated to the processor for the data type *Decimal*. Therefore, this data type is processed much more slowly than the floating-point data types *Single* and *Double*. At the same time, however, there will be no rounding errors due to the internal display of values in the binary system. You will learn more about this in the following section.

CLS-compliant: Yes

Conversion of other numeric types: *CDec(objVar)* or *Convert.ToDecimal(objVar)*

```
aDecimal = CDec(123.45F)
aDecimal = Convert.ToDecimal(123.45F)
```

The Numeric Data Types at a Glance

Table 6-2 presents a list of the numeric data types, along with a brief description.

TABLE 6-2 The Numeric Base Data Types in .NET 2.0/3.5/4.0 and Visual Basic 2010

Type name	.NET type name	Task	Scope
Byte	*System.Byte*	Stores unsigned integer values with a width of 8 bits (1 byte)	–0 to 255
SByte	*System.SByte*	Stores signed integer values with a width of 8 bits (1 byte)	–127 to 128
Short	*System.Int16*	Stores signed integer values with a width of 16 bits (2 bytes)	–32,768 to 32,767
UShort	*System.UInt16*	Stores unsigned integer values with a width of 16 bits (2 bytes)	0 to 65,535
Integer	*System.Int32*	Stores signed integer values with a width of 32 bits (4 bytes) **Note:** On 32-bit systems, this integer data type is processed most quickly	–2,147,483,648 to 2,147,483,647
UInteger	*System.UInt32*	Stores unsigned integer values with a width of 32 bit (4 bytes)	0 to 4,294,967,295
Long	*System.Int64*	Stores signed integer values with a width of 64 bits (8 bytes)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ULong	*System.UInt64*	Stores unsigned integer values with a width of 64 bits (8 bytes)	0 to 18,446,744,073,709,551,615
Single	*System.Single*	Stores floating-point numbers with single precision; requires 4 bytes for display	–3.4028235E+38 to –1.401298E-45 for negative values; 1.401298E-45 to 3.4028235E+38 for positive values
Double	*System.Double*	Stores floating-point numbers with double precision; requires 8 bytes for display. **Note:** This is the fastest data type for floating-point number calculations because it is delegated directly to the math unit of the processor for calculation.	1.79769313486231570E+308 to -4.94065645841246544E-324 for negative values; 4.94065645841246544E-324 to 1.79769313486231570E+308 for positive values
Decimal	*System.Decimal*	Stores floating-point numbers in binary-coded decimal format. **Note:** This is the lowest data type for floating-point number calculations, but its special form of representing values excludes typical computer rounding errors.	0 to ±79,228,162,514.264,337.593,543,950,335 (±7.9...E+28) without decimal character; 0 to ±7.92281625142643375935439503 35 with 28 places to the right of the decimal character; smallest number not equal 0 (zero) is ±0.0000000000000000000000000001 (±1E-28)

Avoiding Single and Double Rounding Errors

It's not unusual that some numeric systems are unable to display exact values for fractions. However, programmers repeatedly believe that they have found an error in a programming language or claim that the computer can't calculate correctly. You have already experienced rounding and conversion errors from one numeric system into another in your daily life with the base-10 system. For example, dividing the number 1 by 3 results in a number with infinite decimal points (0.333333333333…). Representing the fraction one-third using a base-3 system requires considerably fewer numbers; it's simply 0.1.

It doesn't really matter how many numbers you use to display a fraction, but as long as you use a finite number of digits in numeric systems to display a fraction, there will be situations in which rounding errors are unavoidable.

For example, 3*1/3 in a base-3 system leads to the following calculation:

```
 0.1
+0.1
+0.1
=========================================
+1.0
```

In the decimal system, this also corresponds to 1.0. But performing this same addition in the decimal system is imprecise, because even if you use 60 decimal places to represent the number, you never reach the value 1 in the addition, as shown here:

```
 0,333333333333333333333333333333333333333333333333333333333333
+0.333333333333333333333333333333333333333333333333333333333333
+0.333333333333333333333333333333333333333333333333333333333333
================================================================
 0.999999999999999999999999999999999999999999999999999999999999
```

The total value in the preceding calculation is very close to 1—but it's not quite 1. If you have multiple intermediate results during the course of a calculation, such representation errors can quickly lead to bigger mistakes that will become relevant at some point.

The computer has the same problem with certain numbers when it calculates in the binary system. Even though it can display the number 69.82 in the decimal system correctly, it runs into problems with the binary system. Converting 69 works without issues, but it becomes difficult with 0.82.

Once you know that decimal places are represented by negative powers of the base number, you can try to approximate the fractional part (0.82) by using the following calculations:

```
0.5             1*2^-1  intermediate result: 0.5
0.25            1*2^-2  intermediate result: 0.75
0.125           1*2^-3  intermediate result: 0.8125
0.0625          0*2^-4  intermediate result: 0.8125
0.03125         0*2^-5  intermediate result: 0.8125
```

```
0.015625         0*2^-6  intermediate result: 0.8125
0.0078125        0*2^-7  intermediate result: 0.8125
0.00390625       1*2^-8  intermediate result: 0.81640625
0.001953125      1*2^-9  intermediate result: 0.818359375
0.0009765625     1*2^-10 intermediate result: 0.8193359375
0.00048828125    1*2^-11 intermediate result: 0.81982421875
```

At this point, the computer has generated the binary digits 0.11100001111, but we have not reached the desired goal. The truth is that you can play this game for all eternity, but you will never be able to represent the number 0.82 in the decimal system with a finite number of digits in the binary system.

Companion Content What's the impact to programming in Visual Basic? Take a look at the sample program in the \VB 2010 Developer Handbook\Chapter 06\Primitives02 folder.

```
Public Class Primitives

    Public Shared Sub main()

        Dim locDouble1, locDouble2 As Double
        Dim locDec1, locDec2 As Decimal

        locDouble1 = 69.82
        locDouble2 = 69.2
        locDouble2 += 0.62

        Console.WriteLine("The statement locDouble1=locDouble2 is {0}",
           locDouble1 = locDouble2)
        Console.WriteLine("but locDouble1 is {0} and locDouble2 is {1}", _
                          locDouble1, locDouble2)

        locDec1 = 69.82D
        locDec2 = 69.2D
        locDec2 += 0.62D
        Console.WriteLine("The statement locDec1=locDec2 is {0}", locDec1 = locDec2)

        Console.WriteLine()
        Console.WriteLine("Press key to exit!")
        Console.ReadKey()

     …

    End Sub

End Class
```

At first glance, you'd think that both *WriteLine* methods return the same text. You don't need to use a calculator to see that the value (and thus the first variable) within the program represents the addition of the second and third value; therefore, both variable values should

be the same. Unfortunately that's not the case. Although the second part of the program achieves the correct result using the *Decimal* data type, the *Double* type fails in the first part of the program.

The second *WriteLine* method is even more confusing, because both variables appear to contain the same value to the last decimal place. Here's the output from the preceding code:

```
The statement locDouble1=locDouble2 is False
but locDouble1 is 69.82 and locDouble2 is 69.82
The statement locDec1=locDec2 is True

Press key to exit!
```

So what happened here? During the conversion from the internal binary number system to the decimal number system, a rounding error takes place that conceals the true result. Based on this experiment, the following remarks can be made. If at all possible, try to avoid using fractioned *Double* or *Single* values as counters or conditions within a loop; otherwise, you run the risk that your program becomes bogged down in endless loops as a result of the inaccuracies just mentioned. Therefore, follow these rules:

- Use *Single* and *Double* data types only where the umpteenth number behind the comma is not important. For example, when calculating graphics, where rounding errors are irrelevant due to a smaller screen resolution, you should always choose the faster processor-calculated data types, *Single* and *Double*, over the manually calculated Decimal data type.
- When working with finance applications you should *always* use the *Decimal* data type. It's the only data type that ensures that numeric calculations that cannot be represented exactly will not result in major errors.
- If possible, never use the *Decimal* data type in loops, and do not use it as a counter variable. The type is not directly supported by the processor, so it degrades your program's performance. Try to get by with one of the many integer variable types.
- When you need to compare *Double* and *Single* variables to one another, you should query their deltas rather than comparing the values directly, as in the following code:

```
If Math.Abs(locDouble1 – locDouble2) < 0.0001 then
   'Values are nearly the same, i.e. the same.
End If
```

Methods Common to all Numeric Types

All numeric data types have methods that are used the same way for all types. They convert a string into the corresponding numeric value or a numeric value into a string. Other methods serve to determine the largest or smallest value a data type can represent.

Converting Strings into Values and Avoiding Culture-Dependant Errors

The static functions *Parse* and *TryParse* are available to all numeric data types to convert a string into a value. For example, to convert the numeric string "123" into the integer value 123, you can write:

```
Dim locInteger As Integer
locInteger = Integer.Parse("123")
```

Important Beginning with Visual Basic 2005, you can no longer write the following:

```
locInteger = locInteger.Parse("123")   'Should not be done like this any longer!
```

Because *Parse* is a static function, you should no longer call it via an object variable—use only the corresponding class name. A program that addresses the static function via an object variable can still be compiled, but the Code Editor will display a warning.

You can also try to convert the string into a numeric value, as shown here:

```
Dim locInteger As Integer
If Integer.TryParse("123", locInteger) Then
    'Conversion successful
Else
    'Conversion not successful
End If
```

If the conversion is successful, the converted number is displayed in the output variable—*locInteger* in this example. The .NET Framework equivalent of *Integer* also permits conversions via this code:

```
locInteger = System.Int32.Parse("123") 'This would work, too.
```

And of course, it's also possible to make the conversion via the *Convert* class in .NET Framework style by using the following:

```
locInteger = Convert.ToInt32("123")     'And this would work.
```

Finally, there's an old-fashioned way in Visual Basic:

```
locInteger = CInt("123")                'Last option.
```

But watch out: you might find differences when running programs on a non–English-language system because of the default cultural setting. For example, if you run the following program on a German system, it will not act the way you might expect:

```
Dim locString As String = "123.23"
Dim locdouble As Double = Double.Parse(locString)
Console.WriteLine(locdouble.ToString)
```

You might expect the string to be converted correctly into the value 123.23. Instead the program returns the following:

```
12323
```

This is definitely not the expected result. However, if you run the program on an English system, the result will be correct, as expected:

```
123.23
```

Well, maybe not quite. Germans are used to separating the decimal places from integer places by a comma. English speaking countries use a period, and the preceding output uses the correct English formatting. What's the impact of this behavior on your programs? To begin, you should avoid saving numeric constants as text in the program code itself if you want to convert them to a numeric type later on (as shown in the example). When you define numeric data types within your programs, make those definitions directly in code. Do not use strings (text in quotes) and the corresponding conversion functions. You have probably already noticed that numeric strings placed in code (without quotes) for assigning a value must always adhere to the English formatting.

Note To be accurate, the behavior that was just described isn't caused by an English-language system; it is because that operating system has default cultural settings. Of course, a German or other non–English-language system can also be configured to yield the same result.

As long as you don't need to exchange files with information saved as text across cultural borders for which your program has to generate values, you have nothing to worry about: if your application is run on an English-language system, your numbers are written into the file with a period as separator; in the German-speaking areas, a comma is used. Because cultural settings are taken into account when reading a file, your application should be able to generate the correct values back from the text file.

It becomes a bit more problematic when the files containing the text are exchanged across cultural borders. This can happen pretty easily; for example, you might access a database server in a company with a .NET Windows Forms application from a German Windows 7 system, because many IT departments exclusively run English-language versions on their servers for a variety of reasons. Therefore, a platform in the United States would export the file with a period separator, and in Germany, the *Parse* function would recognize the period as a thousands-separator, and thus erroneously treat the fractional digits as significant integer digits. In this case, you need to ensure that any export of a text file is culturally neutral, which you can achieve as follows:

You can use the *Parse* function and the *ToString* function of all numeric types to control the conversion by a *format provider*. For numeric types, .NET offers many different format providers: some help you control the format depending on the application type (financial, scientific, and so on), others control it depending on the culture, namely the classes *NumberFormatInfo* and *CultureInfo*. You can pass either to the *ToString* or the *Parse* function (assuming they have been properly initialized).

Important You should always use the following procedure for applications that will be used internationally to avoid type conversion errors from the start:

```
Dim locString As String = "123.23"
Dim locdouble As Double

locdouble = Double.Parse(locString, CultureInfo.InvariantCulture)
Console.WriteLine(locdouble.ToString(CultureInfo.InvariantCulture))
Console.ReadLine()
```

Note To be able to access classes and functions that control globalization, you must bind the namespace *System.Globalization* with the *Imports* statement at the beginning of the program, as follows:

```
Imports System.Globalization
```

The static property *InvariantCulture* returns an instance of a *CultureInfo* class that represents the current system locale.

Note Should the conversion fail simply because the string doesn't contain a convertible format, and thus can't be converted into a value, .NET Framework generates an exception (see Figure 6-1). The exception can either be caught with *Try … Catch*, or alternatively, you can use the static method *TryParse*, which never causes an exception during conversion attempts.

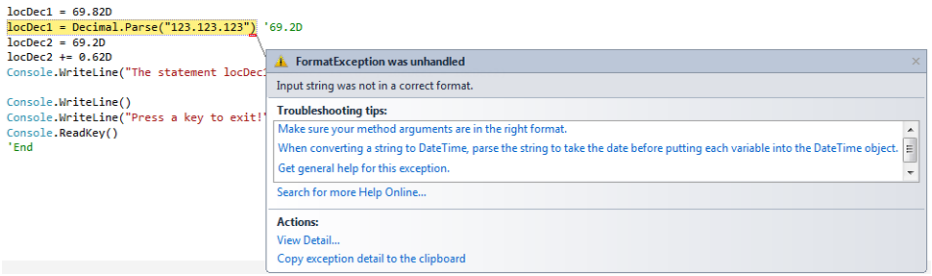

FIGURE 6-1 If a string that represents a numeric value cannot be converted due to its format, the .NET Framework generates an exception.

Performance and Rounding Issues

If you are using type-safe programming in Visual Basic .NET (which you should always do by using *Option Strict On* in the project properties), it is customary to convert a floating-point number into a value of the type *Integer* by using the conversion operator *CInt*. But a lot of programmers don't know that the Visual Basic compiler behaves completely differently than the casting operator in C#. *CInt* in Visual Basic uses commercial rounding, so the compiler turns

```
Dim anInt = CInt(123.54R)
```

into:

```
Dim anInt = CInt(Math.Round(123.54R))
```

It is *not possible* to implement a simple *CInt* (as used by the Visual Basic compiler itself, and as is the default in C#) in Visual Basic itself. When converting a floating-point in C#, the decimal places after the integer are simply truncated—they are not rounded. To simulate this, you need to use the following construct:

```
Dim anInt = CInt(Math.Truncate(123.54R))
```

The problem is that the compiler generates the following completely redundant code from it:

```
Dim anInt = CInt(Math.Round(Math.Truncate(123.54R)))
```

When it comes to processing graphics, for example, this means a huge performance compromise, of course, because two functions are called from the Math Library. C# is noticeably faster because it provides a *CInt* directly.

Tip If you do run into performance problems due to this behavior, create a C# assembly that provides a function for converting *Double*, *Single*, or *Decimal* directly to *Integer* values. It's quite probable that this function will be inlined (the JIT compiler transfers the code of the C# function to the assembly without generating a function call jump), and therefore, you can achieve almost the same performance as in C#.

Determining the Minimum and Maximum Values of a Numeric Type

The numeric data types recognize two specific static properties that you can use to determine the largest and smallest representable value. These properties are called *MinValue* and *MaxValue*—and just like any static function, you can call them through the type name as shown in the following example:

```
Dim locDecimal As System.Decimal

Console.WriteLine(Integer.MaxValue)
Console.WriteLine(Double.MinValue)
Console.WriteLine(locDecimal.MaxValue) ' Compiler gives a warning – use type name instead.
```

Special Functions for all Floating-Point Types

Floating-point types have certain special properties that simplify processing of abnormal results during calculations (such as the *Infinity* and the *NaN* properties). To check for non-numeric results in your calculations, use the following members of the floating-point data types.

Infinity

When a floating-point value type is divided by 0, the .NET Framework does not generate an exception; instead, the result is *infinity*. Both *Single* and *Double* can represent this result, as shown in the following example:

```
Dim locdouble As Double
locdouble = 20
locdouble /= 0
Console.WriteLine(locdouble)
Console.WriteLine("The statement locDouble is +infinite is {0}.",
   locdouble = Double.PositiveInfinity)
' I suggest using the IsPositiveInfinity method and replacing the last statement
' with an If statement:
If locdouble.IsPositiveInfinity Then
…
Else
…
End If
```

When you run this example, no exception occurs, but the program displays the following on the screen:

```
+infinite
The statement locDouble is +infinite is True.
```

Instead of performing the comparison to infinity by using the comparison operator, you can also use the static function *IsInfinity*, as follows:

```
Console.WriteLine("The statemdnt locDouble is +infinity is {0}.", locdouble.
IsInfinity(locdouble))
```

You can also use the *IsPositiveInfinity* and *IsNegativeInfinity* methods to determine whether a value is infinitely large or infinitely small (a very large negative value).

To assign the value *infinite* to a variable, use the static functions *PositiveInfinity* and *NegativeInfinity*, which return appropriate constants.

Not a Number: *NaN*

The base floating-point types cover another special case: the division of 0 by 0, which is not mathematically defined and *does not return a valid number*:

```
'Special case: 0/0 is not mathematically defined and returns "Not a Number"
aDouble = 0
aDouble = aDouble / 0
If Double.IsNaN(aDouble) Then
    Debug.Print("aDouble is not a number!")
End If
```

If you run this code, the output window will display the result of the *If* query.

Important You can test these special cases only via properties that static functions directly "append" to the type. With the floating-point type constant *NaN*, you can assign the value "not a valid number" to a variable, but you can't use the constant to test for the not-a-number state, as shown in the example that follows.

```
Dim aDouble As Double

'Special case: 0/0 is not mathematically defined and returns "Not a Number"
aDouble = 0
aDouble = aDouble / 0

'The text should be returned as expected,
'but isn't!
If aDouble = Double.NaN Then
    Debug.Print("Test 1:aDouble is not a number!")
End If

'Now the test can be performed!
If Double.IsNaN(aDouble) Then
    Debug.Print("Test 2:aDouble is not a number!")
End If
```

The preceding example displays only the second message in the output window.

Note Both features, *NaN* and *Infinity*, let your programs behave totally differently in contrast to Visual Basic for Applications (or the old Visual Basic 6.0 for that matter). So be careful if you migrate applications, or even only methods, from Visual Basic 6.0 to VB.NET: some algorithms in VBA/Visual Basic 6.0 *expect* errors at certain points to occur, but since a division by zero doesn't necessarily lead to an error in VB.NET (this happens only with integer data types), those algorithms might return incorrect results in certain cases.

Converting with *TryParse*

All numeric data types expose the static method *TryParse*, which attempts to convert a string into a value. Unlike *Parse*, the *TryParse* method doesn't generate an exception when the conversion fails. Instead, you pass a variable name as a reference argument to the method, and the method returns a result, indicating whether the conversion was successful (*True*) or not (*False*), as shown here:

```
Dim locdouble As Double
Dim locString As String = "Onehundredandtwentythree"

'locdouble = Double.Parse(locString) ' Exception
'Not working, either, but at least no exception:
Console.WriteLine("Conversion successful? {0}", _
         Double.TryParse(locString, NumberStyles.Any, New CultureInfo("en-En"), locdouble))
```

Special Functions for the *Decimal* Type

The value type *Decimal* also has special methods, many of which aren't of any use in Visual Basic (you can use them, but it doesn't make much sense—they were added for other languages that don't support operator overloading). Take, for example, the static *Add* function, which adds two numbers of type *Decimal* and returns a *Decimal*. You can use the + operator of Visual Basic instead, which can also add two numbers of the type *Decimal*—and does so in much more easily readable code. Therefore, it makes sense to use the functions presented in Table 6-3.

TABLE 6-3 The Most Important Functions of the *Decimal* Type

Function name	Task
Remainder(Dec1, Dec2)	Determines the remainder of the division of both decimal *Decimal* values.
Round(Dec, Integer)	Rounds a *Decimal* value to the specified number of decimal places.
Truncate(Dec)	Returns the integer part of the specified *Decimal* value.
Floor(Dec)	Rounds the *Decimal* value to the next smaller number.
Negate(Decimal)	Multiplies the *Decimal* value by –1.

The *Char* Data Type

The *Char* data type stores a character in Unicode format (more about this topic later in the chapter) using 16 bits, or 2 bytes. Unlike the *String* type, the *Char* data type is a value type. The following brief overview gives you more details:

.NET data type: *System.Char*

Represents: A single character

Range: 0–65.535, so that Unicode characters can be displayed

Type literal: c

Memory requirements: 2 bytes

Delegation to the processor: Yes

CLS-compliant: Yes

Description: *Char* values are often used in arrays, because in many cases it's more practical to process individual characters than it is to process strings. Like any other data type, you can define *Char* arrays with constants (you'll see an example shortly). The following section on strings contains examples on how to use *Char* arrays instead of *strings* for character-by-character processing.

Even if *Char* is saved internally as an unsigned 16-bit value, and is therefore like a *Short*, you cannot implicitly convert a *Char* into a numeric type. In addition to the possibility described in the Online Help, however, you can use not only the functions *AscW* and *ChrW* to convert a *Char* to a numeric data type, and vice versa, but also the *Convert* class, for example:

```
'Normal declaration and definition
Dim locChar As Char
locChar = "1"c
Dim locInteger As Integer = Convert.ToInt32(locChar)
Console.WriteLine("The value of '{0}' is {1}", locChar, locInteger)
```

When you run this example, it displays the following output:

```
The value of '1' is 49
```

You can also use the functions *Chr* and *Asc*, but they work only for non-Unicode characters (ASCII 0–255). Due to various internal scope checks, they also have an enormous overhead; therefore, they are nowhere near as fast as *AscW*, *ChrW* (which are the fastest, because a direct and internal type conversion of *Char* into *Integer*, and vice versa, takes place) or the *Convert* class (which has the advantage of being easily understood by non-Visual Basic developers as well).

Declaration and Sample Assignment (also as Array):

```
'Normal declaration and definition
Dim locChar As Char
locChar = "K"c

'Define and declare a Char array with constants.
Dim locCharArray() As Char = {"A"c, "B"c, "C"c}

'Convert a Char array into string.
Dim locStringFromCharArray As String = New String(locCharArray)
```

```
'Convert a string into a Char array.
'Of course that's also possible with a string variable.
locCharArray = "This is a string".ToCharArray
```

The *String* Data Type

Strings are used to store text. Unlike in Visual Basic for Applications (or Visual Basic 6.0), the modern versions of Visual Basic offer an object-oriented programming (OOP) approach for working with strings, which makes string processing much easier. Your programs will be easier to read if you follow the OOP concepts.

You have already encountered strings several times in the previous chapters. Nevertheless, it's worth taking another look behind the scenes. In conjunction with the Regular Expression (*Regex*) class, the .NET Framework offers the best possible string support.

In contrast to other base types, strings are reference types. However, you don't have to define a new *String* instance with the keyword *New*. The compiler intervenes here because it has to generate special code anyhow.

Note Similar to nullables (see Chapter 18) when boxing, CLR interrupts the default behavior for reference types and changes it. Strictly speaking, since strings are reference types, they need to be instantiated with *New*. The equal operator would also need to assign just one reference. For strings, however, when assigning an instance to an object variable, the content is cloned, which also deviates from the default. You can read more about reference types in Chapter 12, "Typecasting and Boxing Value Types."

The following sections provide an overview of the special characteristic of strings in the BCL. At the end of this section, you'll find an example application that illustrates the most important string manipulation functions.

Strings—Yesterday and Today

Beginning with Visual Studio 2002 and .NET Framework 1.0, Microsoft introduced a completely new way to approach string programming in Visual Basic. This was the result of the new implementation of the data type *String*, which is created by instantiating a class, like other objects.

Almost all commands and functions that were "standalone" in Visual Basic 6.0 and VBA still exist in the .NET Framework versions of Visual Basic. But they are not only superfluous—you can reach the same goal much more elegantly with the existing methods and properties of the *String* object—but they also slow down programs unnecessarily, because internally they call the *String* object functions anyhow.

For (almost) all the old string functions, there is a corresponding class function that you should use as a substitute. The following sections demonstrate the handling of strings with the help of a few short examples.

Declaring and Defining Strings

As with all other base data types, you can declare strings without using the *New* keyword; you can perform assignments directly in the program. For example, you can declare a string with the statement:

```
Dim aString As String
```

You can then immediately use the new string variable. The instance of the *String* object is created at IML level. Strings are defined as a list of characters between quotes, as shown in the following example:

```
aString = "Susan Kallenbach"
```

Just as with other base data types, declaration and assignment can both take place in an assignment. Therefore, you can combine the two single statements shown in the preceding statements into the following shorter, single statement:

```
Dim aString As String = "Susan Kallenbach"
```

Handling Empty and Blank Strings

For decades, Visual Basic developers have become accustomed to using the code pattern shown in the following to check whether a string variable is not defined (*Nothing*) or points to an empty string:

```
Dim myString As String
myString = WinFormsTextBox.Text

If myString Is Nothing Or myString = "" Then
     MessageBox.Show("myString is empty.")
End If
```

Initially, the data type *String* acts like a regular reference type. As soon as it is declared, it becomes *Nothing*, because its content points to *nothing*. But it also has a second state that corresponds to the value "empty," which occurs when it points to a memory area reserved for saving characters, but that doesn't contain any characters. In this case, the *String* variable saves an empty string. Yet a third possibility occurs when the string *does* contain data (at least from the computer's standpoint), but that data does not represent visible content, because it's not printable (or more precisely, it's not visible on the screen or when printed). Such characters are called whitespace characters, which include space characters, tabs, and other control characters.

All these states are checked by a static method of the *String* type, and it simplifies testing for such states:

```
If String.IsNullOrWhiteSpace(myString) Then
    MessageBox.Show("my string is empty.")
End If
```

When you run this code, you always receive the message "my string is empty" when the *myString* variable is empty (points to an empty string), is *Nothing*, or contains only one or more whitespaces.

The method *IsNullOrWhiteSpace* has been available since .NET Framework 4.0. If you need to maintain backward compatibility with earlier versions of the .NET Framework, or if you want to allow whitespaces as valid entries, it is better to use the *IsNullOrEmpty* method, which returns *True* for *Nothing* and empty strings, as illustrated in the following:

```
If String.IsNullOrEmpty(myString) Then
    MessageBox.Show("myString is empty.")
End If
```

Automatic String Construction

Normally, a class constructor creates an instance and a structure to pre-assign certain values to parameters—you'll find a lot more on constructors in this book's OOP section.

Even though you create strings in the same manner as all other base data types, you still have the option of calling a constructor method. However, you don't employ the constructor exclusively for re-instantiating an empty *String* object (the parameterless constructor is not permitted), but you actually emulate, among others, the old *String$* function from Visual Basic 6.0 or VBA.

With their help it was possible to generate a string programmatically and save it in a *String* variable.

Note While many of the old Visual Basic 6.0/VBA commands are still available all of the .NET Framework versions up to 4.0, the *String* function itself no longer exists in the .NET versions of Visual Basic—possibly due to the type identifier of the same name.

To use the *String* constructor as a *String$* function substitute, do the following:

```
Dim locString As String = New String("A"c, 40)
```

The type literal "c" indicates that you need to pass a value of the type *Char* in the constructor. Unfortunately this limits the repetition function to one character, which wasn't the case with *String$*. Fortunately, it's no problem to implement a *Repeat* function, which resolves this issue:

```
Public Function Repeat(ByVal s As String, ByVal repeatitions As Integer) As String
    Dim sBuilder As New StringBuilder

    For count As Integer = 1 To repeatitions
        sBuilder.Append(s)
    Next

    Return sBuilder.ToString
End Function
```

Note This construct mainly serves as an example, and you should only construct strings in this manner if there are very few characters. You should use the larger text segments of the *StringBuilder* class for performance reasons. The *StringBuilder* class is described in the section, "*StringBuilder* vs. *String*: When Performance Matters," later in this chapter. Why this is the case is explained in the section, "No Strings Attached, or Are There? Strings are Immutable!"

Apart from using a constructor to generate strings by repeating the same character, you can also use one to create a string from a *Char* array or a portion of a *Char* array, as shown in the following example:

```
Dim locCharArray() As Char = {"K"c, "."c, " "c, "L"c, "ö"c, "f"c, "f"c, "e"c, "l"c, "m"c,
➥"a"c, "n"c, "n"c}
Dim locString As String = New String(locCharArray)
Console.WriteLine(locString)
locString = New String(locCharArray, 3, 6)
Console.WriteLine(locString)
```

When you run this program, the console window displays the following output:

```
K. Löffelmann
Löffel
```

Assigning Special Characters to a String

To include quotes in the string itself, use repeated double quotes. For example, to define the string "Adriana said, "it's only 1pm, I'm going to sleep in a little longer!" in a program, you would write the assignment as follows:

```
locString = "Adriana said, ""it's only 1pm, I'm going to sleep in a little longer!""."
```

To include other special characters, use the existing constants in the Visual Basic vocabulary. For example, to build a paragraph into a string, you need to insert the ASCII codes for *line-feed* and *carriage return* into the string. You achieve this by using the constants shown in the following example:

```
locOtherString = "Adriana said ""I'm going to sleep in a little longer!""" & vbCr & vbLf & _
                 "She fell asleep again immediately."
```

You could use *vbNewLine* or the shorthand *vbCrLf* character constants instead. For example, you can save keystrokes with the following version, which delivers the same result:

```
locOtherString = "Adriana said ""I'm going to sleep in a little longer!""" & vbNewLine & _
                 "She fell asleep again immediately."
```

Table 6-4 presents the special character constants that Visual Basic offers.

TABLE 6-4 The Constants That Correspond to the Most Common Special Characters

Constant	ASCII	Description
vbCrLf or vbNewLine	13; 10	Carriage return/line feed character
vbCr	13	Carriage return character
vbLf	10	Line feed character
vbNullChar	0	Character with a value of 0
vbNullString		String with a value of 0. Doesn't corresponds to a string with a length of 0 (""); this constant is meant for calling external procedures (COM Interop).
vbTab	9	Tab character
vbBack	8	Backspace character
vbFormFeed	12	Not used in Microsoft Windows
vbVerticalTab	11	Control characters for the vertical tab which isn't used in Microsoft Windows

Memory Requirements for Strings

Each character in a string requires two bytes of memory. Even though strings are returned as letters, each character is represented in memory by a numeric value. The values from 1 to 255 correspond to the *American Standard Code for Information Interchange*—ASCII for short—which standardizes only values up to 127 for each character set. Special characters are defined in the 128–255 range, and those characters depend on the specific character set used. Generally the codes for the special characters of the European countries, such as "öäüÖÄÜâéè", have the same code for each font (the exception proves the rule, as usual). Values above 255 represent special characters that are used, for example, for the Cyrillic, Arabic, or Asian characters. This coding convention, called Unicode, permits a considerably larger total number of characters. The .NET Framework generally saves strings in *Unicode* format.

No Strings Attached, or Are There? Strings are Immutable!

Generally, strings are reference types, but they are strictly static in memory and are therefore immutable. In practice, that means that there is no restriction as to how you handle strings: even though you might think you have changed a string, you have actually created a new one that contains the changes. You need to know that when it comes to applications that perform many string operations, you should use the *StringBuilder* class instead, which is noticeably more powerful, even though it doesn't offer the flexibility of the string manipulation methods or the advantages of the base data types. (The section, "When Speed Matters," discusses this topic in more detail.)

You can read more about reference and value types in Chapter 8, "Class Begins."

Much more important is the impact of the immutability of strings in your programs: even though strings are considered reference types, they behave like value types, because they cannot be changed. When two string variables point to the same memory area and you change the content of a string, it appears as though you are changing the value of the original variable, but in reality, such operations create a completely new *String* object in memory and change the existing variable to point to it. This way, you never end up in the situation you know from reference types, in which changing the object content via the object variable never causes another string variable, which points to the same memory area, to return the edited string. This explains why strings are reference types, but at the same time, they "look and feel" like value types.

You will see more about this in the following section along with some practical examples.

Memory Optimization for Strings

For saving strings the .NET Framework uses an *internal pool* that to avoids redundancies in string storage. If you define two strings within your program using the same constant, the Visual Basic compiler recognizes that they're the same and places only one copy of the string in memory. At the same time, it allows both object variables to point to the same memory area, as the following example proves:

```
Dim locString As String
Dim locOtherString As String

locString = "Adriana" & " Ardelean"
locOtherString = "Adriana Ardelean"
Console.WriteLine(locString Is locOtherString)
```

When you start this program, it returns *True*—meaning that both strings point to the same memory area. This condition exists only as long as the compiler can recognize the equality of the strings, and to do this, their values must be specified in a single statement. For example, in the following code, the compiler can no longer recognize the equality of the strings, so the result would be *False*:

```
Dim locString As String
Dim locOtherString As String

locString = "Adriana"
locString &= " Ardelean"
locOtherString = "Adriana Ardelean"
Console.WriteLine(locString Is locOtherString)
```

It's obvious that a behavior to avoid redundancies at runtime would take too much time and can't be used in a sensible way. If there are a lot of strings, the BCL would waste too much time searching for strings that already existing. However, you do have the option to specifically add a string created at runtime to a pool. If you add several identical strings to the internal pool, they are not assigned redundantly—several identical strings then share the same memory. Of course, this only makes sense when you can predict that there will be many conformant strings within a program. The example that follows shows how you explicitly add a string with the static function *Intern* to the internal pool:

```
Dim locString As String
Dim locOtherString As String

locString = "Adriana"
locString &= " Ardelean"
locString = String.Intern(locString)
locOtherString = String.Intern("Adriana Ardelean")
Console.WriteLine(locString Is locOtherString)
```

When you start this program, the output is again *True*.

Determining String Length

VBA/Visual Basic 6.0 compatible command: Len

.NET versions of Visual Basic: *strVar.Length*

Description: With this command you determine the length of a string in characters, and not in bytes.

Example: The following example accepts a string from a user and returns the string's characters in inverse order:

```
Dim locString As String
Console.Write("Enter a text: ")
locString = Console.ReadLine()
For count As Integer = locString.Length - 1 To 0 Step -1
   Console.Write(locString.Substring(count, 1))
Next
```

You can find the same example with Visual Basic 6.0 compatibility commands in the next section.

Retrieving Parts of a String

VBA/Visual Basic 6.0 compatible command(s): *Left*, *Right*, *Mid*

.NET versions of Visual Basic: *strVar.SubString*

Description: Use this command to retrieve a certain part of a string as another string.

Note Why the good-old *Left* and *Right* methods of the class *String* were omitted is a question only the programmer can answer. Maybe they were simply forgotten, or the programmer knew only C and couldn't imagine a world as simple as BASIC.

Example: The following example reads a string from the keyboard and then returns its characters in inverse order. You can find the same example in the previous section with the functions of the *String* object.

```
Dim locString As String
Console.Write("Enter a text: ")
locString = Console.ReadLine()
For count As Integer = Len(locString) To 1 Step -1
    Console.Write(Mid(locString, count, 1))
Next
```

Padding Strings

VBA/Visual Basic 6.0 compatible command(s): *RSet*, *LSet*

.NET versions of Visual Basic: *strVar.PadLeft*; *strVar.PadRight*

Description: With these commands, you can increase the length of a string to a certain number of characters; the string is padded with blank characters at the beginning or the end.

Example: The following example demonstrates the use of the *PadLeft* and the *PadRight* method:

```
Dim locString As String = "This string is so long"
Dim locString2 As String = "Not this one"
Dim locString3 As String = "This"
Len(locString)
locString2 = locString2.PadLeft(locString.Length)
locString3 = locString3.PadRight(locString.Length)

Console.WriteLine(locString + ":")
Console.WriteLine(locString2 + ":")
Console.WriteLine(locString3 + ":")
```

When you run this program the following output is generated:

```
This string is so long:
         Not this one:
This                  :
```

Note The strings are so perfectly aligned only because the Console window uses a monospaced font by default. You can't align strings in a typical window that uses a proportional font with the *PadLeft* and *PadRight* methods.

Find and Replace Operations

VBA/Visual Basic 6.0 compatible command(s): *InStr*, *InStrRev*, *Replace*

.NET versions of Visual Basic: *strVar.IndexOf*; *strVar.IndexOfAny*; *strVar.Replace*; *strVar.Remove*

Description: With the Visual Basic 6.0 compatible command *InStr*, you can search for the occurrence of a character or a string within a string. *InStrRev* does the same, but it starts the search from the end of the string. *Replace* lets you to replace a substring within a string with another string.

Using the *IndexOf* method of the *String* class, you can search for the occurrence of a character or a string within the current string. Furthermore, the *IndexOfAny* method lets you find the occurrences of a group of characters passed as a *Char* array within the string. The *Replace* method replaces individual characters or strings with others in the current string, and the *Remove* method removes a specific substring from the string.

Example: The following examples demonstrate how to use the *Find* and *Replace* methods of the *String* class:

Companion Content Open the corresponding solution (.sln) for this example, which you can find in the \VB 2010 Developer Handbook\Chapter 06\Strings – Find and Replace folder.

```
Imports System.Globalization

Module Strings
    Sub Main()
        Dim locString As String = _
            "Common wisdoms:" + vbNewLine + _
            "* If you would shout for 8 years, 7 months, and 6 days," + vbNewLine + _
            "  you would have produced enough energy to heat a cup of coffee." + _
➥vbNewLine + _
            "* If you hit your head against the wall, you use up 150 calories." +
```

```
➥vbNewLine + _
            "* Elephants are the only animals who can't jump." + vbNewLine + _
            "* A cockroach can live for 9 days without a head before it dies of
➥hunger." + vbNewLine + _
            "* Gold and other metals originate solely from" + vbNewLine + _
            "  supernovae." + vbNewLine + _
            "* The Moon consists of debris from a collision of a" + vbNewLine + _
            "  planet the size of Mars with the Earth." + vbNewLine + _
            "* New York is called the ""Big Pineapple"", because ""Big Pineapple"" in
➥the language of" + vbNewLine + _
            "  Jazz musicians meant ""hitting the jackpot"". To have a career in New
➥York" + vbNewLine + _
            "  meant their jackpot." + vbNewLine + _
            "* The expression ""08/15"" for something unoriginal was originally " +
➥vbNewLine + _
            "  the label of the machine gun LMG 08/15;" + vbNewLine + _
            "  It become the metaphor for unimaginative, military drills." +
➥vbNewLine + _
            "* 311 New Yorkers are being bit by rats per year in average." +
➥vbNewLine + _
            "  But 1511 New Yorkers have been bit by other New Yorkers at the same
➥time."
        'Replace number combination with letters
        locString = locString.Replace("08/15", "Zero-eight-fifteen")

        'Count punctuation
        Dim locPosition, locCount As Integer

        Do
            locPosition = locString.IndexOfAny(New Char() {"."c, ","c, ":"c, "?"c},
➥locPosition)
            If locPosition = -1 Then
                Exit Do
            Else
                locCount += 1
            End If
            locPosition += 1
        Loop

        Console.WriteLine("The following text...")
        Console.WriteLine(New String("="c, 79))
        Console.WriteLine(locString)
        Console.WriteLine(New String("="c, 79))
        Console.WriteLine("...has {0} punctuation.", locCount)
        Console.WriteLine()
        Console.WriteLine("And after replacing 'Big Pineapple' with 'Big Apple' it
➥looks as follows:")
        Console.WriteLine(New String("="c, 79))
```

```
        'Another substitution
        locString = locString.Replace("Big Pineapple", "Big Apple")
        Console.WriteLine(locString)
        Console.ReadLine()

    End Sub

End Module
```

The example displays the following output on the screen:

```
The following text...
================================================================================
Common wisdoms:
* If you would shout for 8 years, 7 months, and 6 days,
  you would have produced enough energy to heat a cup of coffee.
* If you hit your head against the wall, you use up 150 calories.
* Elephants are the only animals who can't jump.
* A cockroach can live for 9 days without a head before it dies of hunger.
* Gold and other metals originate solely from
  Supernovae.
* The Moon consists of debris from a collision of a
  planet the size of Mars with the Earth.
* New York is called the "Big Pineapple", because "Big Pineapple" in the language of
  Jazz musicians meant "hitting the jackpot". To have a carreer in New York
  meant their jackpot.
* The expression "08/15" for something unoriginal was originally
  the label of the machine gun LMG 08/15;
  It become the metaphor for unimaginative, military drills.
* 311 New Yorkers are being bit by rats per year in average.
  But 1511 New Yorkers have been bit by other New Yorkers at the same time.
================================================================================
...has 23 punctuation marks.

And if you replace 'Big Pineapple' with 'Big Apple' it looks as follows:
================================================================================
Common wisdoms:
* If you would shout for 8 years, 7 months, and 6 days,
  you would have produced enough energy to heat a cup of coffee.
* If you hit your head against the wall, you use up 150 calories.
* Elephants are the only animals who can't jump.
* A cockroach can live for 9 days without a head before it dies of hunger.
* Gold and other metals originate solely from
  Supernovae..
* The Moon consists of debris from a collision of a
  planet the size of Mars with the Earth.
* New York is called the "Big Apple", because "Big Apple" in the language of
  Jazz musicians meant "hitting the jackpot". To have a carreer in New York
  meant their jackpot.
```

```
* The expression "08/15" for something unoriginal was originally
  the label of the machine gun LMG 08/15;
  It become the metaphor for unimaginative, military drills.
* 311 New Yorkers are being bit by rats per year in average.
  But 1511 New Yorkers have been bit by other New Yorkers at the same time.
```

Tip The example presented in the section "Splitting Strings," contains a custom function called *ReplaceEx*, which you can use to search for several characters, replacing found occurrences with a specified character.

Trimming Strings

VBA/Visual Basic 6.0 compatible command(s): *Trim*, *RTrim*, *LTrim*

.NET versions of Visual Basic: *strVar.Trim*, *strVar.TrimEnd*, *strVar.TrimStart*

Description: These methods remove characters from both the beginning and the end of a string (*Trim*) or at either end of a string (*TrimStart* and *TrimEnd*). For these methods, the object methods of the strings are preferable to the compatibility functions, because for the former you can also specify which characters should be trimmed, as the following example demonstrates. In contrast, the Visual Basic 6.0 compatibility functions only allow space characters to be trimmed.

Example: The following example generates a *String* array whose individual elements have unwanted characters at the beginning and the end (not just space characters), which are removed by using the *Trim* function.

Note This example also shows that strings act differently than common objects, even though they are considered reference types, because they are immutable. If you assign two object variables to an object and change the content of a variable, the object variable will also represent the changed object content. Even if a string seems to be changed, in fact, it's not. Rather, another completely new instance is created, which reflects the changes. The original string is discarded (you can read more about this topic in the section, "No Strings Attached, or Are There? Strings are Immutable!," earlier in this chapter).

```
Dim locStringArray() As String = { _
   "  - Here the actual text starts!", _
   "This text ends with strange characters! .-     ", _
   "  - Here both sides are problematic -  "}
```

```
For Each locString As String In locStringArray
   locString = locString.Trim(New Char() {" "c, "."c, "-"c})
   Console.WriteLine("Clean and neat: " + locString)
Next

'Important: String is a reference type, but nothing has changed for the array.
'That's because strings aren't changed directly, but are always created anew and thus
changed.
For Each locString As String In locStringArray
   Console.WriteLine("Still messy: " + locString)
Next
```

If you run this program, the following output is generated:

```
Clean and neat: Here the actual text starts!
Clean and neat: This text ends with strange characters!
Clean and neat: Here both sides are problematic
Still messy:   - Here the actual text starts!
Still messy: This text ends in strange characters! .-
Still messy:   - Here both sides are problematic –
```

Splitting Strings

VBA/Visual Basic 6.0 compatible command(s): *Split*

.NET versions of Visual Basic: *strVar.Split*

Description: The .NET *Split* method of the *String* class is superior to the compatibility function in that it permits you to specify several separator characters in a *Char* array. This makes your programs more flexible when it comes to analyzing and rebuilding text.

Example: The following example separates the individual terms or sections of a string into partial strings separated by different separator characters. These separated strings are later presented as elements of a *String* array and are further prepared with additional functions.

> **Companion Content** Open the corresponding solution (.sln) for this example, which you can find in the \VB 2010 Developer Handbook\Chapter 06\String – Split folder.

```
Module Strings
    Sub Main()
        Dim locString As String = _
           "Individual, elements; separated by, different - characters."
        Console.WriteLine("From the line:")
        Console.WriteLine(locString)
        Console.WriteLine()
        Console.WriteLine("Becomes a string array with the following elements:")
        Dim locStringArray As String()
        locStringArray = locString.Split(New Char() {","c, ";"c, "-"c, "."c})
```

```
        For Each locStr As String In locStringArray
            Console.WriteLine(ReplaceEx(locStr, New Char() {","c, ";"c, "-"c, "."c}, _
                   Convert.ToChar(vbNullChar)).Trim)
        Next
        Console.ReadLine()
    End Sub

    Public Function ReplaceEx(ByVal str As String, ByVal SearchChars As Char(), _
                        ByVal ReplaceChar As Char) As String
        Dim locPos As Integer
        Do
            locPos = str.IndexOfAny(SearchChars)
            If locPos = -1 Then Exit Do
            If AscW(ReplaceChar) = 0 Then
                str = str.Remove(locPos, 1)
            Else
                str = str.Remove(locPos, 1).Insert(locPos, ReplaceChar.ToString)
            End If
        Loop
        Return str
    End Function
End Module
```

When you run this program, it generates the following output:

```
The line:
Individual, elements; separated, by, different- characters.

Becomes a string array with the following elements:
Individual
elements
separated
by
different
characters
```

Iterating through Strings

The following code segment uses a further variation for iterating through the individual characters in a string. The *Chars* property of a *String* object is an array of *Char* values, which represent the individual characters in the string. Because the *String* object also offers the function *GetEnumerator*, you have the following option for iterating via a string:

```
For Each locChar As Char In "This is a string"
   'Do something.
Next
```

Thanks to the power of the *String* class, the functions *Find* and *Replace* are surprisingly easy to implement. In the following code, assume that the *fneFiles* collection contains a list of file names:

```
Private Sub btnCheckFound_Click(ByVal sender As System.Object, ByVal e As System.
➥EventArgs) _
                Handles btnCheckFound.Click
    For Each locFEI As FilenameEnumeratorItem In fneFiles.Items
        Dim locFilename As String = locFEI.Filename.Name
        If locFilename.IndexOf(txtSearch.Text) > -1 Then
            locFEI.Checked = True
        Else
            locFEI.Checked = False
        End If
    Next
End Sub

Private Sub btnReplaceChecked_Click(ByVal sender As System.Object,ByVal e As System.
➥EventArgs) _
    Handles btnReplaceChecked.Click
    For Each locFEI As FilenameEnumeratorItem In fneFiles.Items
        Dim locFilename As String = locFEI.SubItems(1).Text
        If locFEI.Checked Then
            If locFilename.IndexOf(txtSearch.Text) > -1 Then
                locFilename = locFilename.Replace(txtSearch.Text, txtReplace.Text)
                locFEI.SubItems(1).Text = locFilename
            End If
        End If
    Next
End Sub
```

StringBuilder vs. *String*: When Performance Matters

As you saw earlier in this chapter, .NET provides you with extremely powerful tools for processing strings. If you read the section on memory management, you probably noticed that string processing speed in certain scenarios leaves much to be desired. The reason is simple: strings are immutable. If you are working with algorithms that build strings character-by-character, then for every character you add to the string, a completely new string must be created—and that takes time.

The *StringBuilder* class represents an alternative for such operations. In no way does it provide the functionality of the *String* class in terms of methods, but it does offer an essential advantage: it is managed dynamically, and therefore, it is disproportionately faster. When it comes to building strings (by appending, inserting, or deleting characters), you should use a *StringBuilder* instance—especially when you're dealing with large amounts of data.

Using the *StringBuilder* class is quite simple. First, you need to declare the *System.Text* namespace, which you can bind into your class or module file with an *Imports* statement, as follows:

```
Imports System.Text
```

Next, you simply declare a variable with the type *StringBuilder*, and then initialize it with one of the following constructor calls:

```
'Declaration without parameters:
Dim locSB As New StringBuilder
'Declaration with capacity reservation
locSB = New StringBuilder(1000)
'Declaration from an existing string
locSB = New StringBuilder("Created from a new string")
'Declaration from string with the specification of a capacity to be reserved
locSB = New StringBuilder("Created from string with capacity for more", 1000)
```

Note that you can specify an initial capacity when defining a *StringBuilder* object. This way, the space that your *StringBuilder* object will eventually require is reserved immediately—no additional memory needs to be requested at runtime, which improves performance.

To add characters to the string, use the *Append* method. Using *Insert*, you can place characters anywhere into the string stored in the *StringBuilder* object. *Replace* lets you replace one string with another, whereas *Remove* deletes a specified number of characters from a specific character position onward. Here's an example:

```
locSB.Append(" - and this gets appended to the string")
locSB.Insert(20, ">>this ends up somewhere in the middle<<")
locSB.Replace("String", "StringBuilder")
locSB.Remove(0, 4)
```

When you have finished building the string, you can convert it into a "real" string by using the *ToString* function:

```
'StringBuilder has finished building the string
'Convert to string
Dim locString As String = locSB.ToString
Console.WriteLine(locString)
```

When you execute the preceding statements, the console window displays the following text:

```
StringBuilder Create>>this ends up somewhere in the middle<<d from string with capacity for
more - and that is added to the StringBuilder
```

Performance Comparison: *String* vs. *StringBuilder*

> **Companion Content** Open the solution (.sln) for this example, which you can find in the \VB 2010 Developer Handbook\Chapter 06\StringVsStringBuilder folder.

This section's project demonstrates the efficiency of the *StringBuilder* class. The program creates a number of *String* elements, each consisting of a fixed number of random characters. When you start the program, enter the values of the following parameters as prompted:

```
Enter the string length of an element: 100
Enter a number of elements to be generated: 100000

Generating 100000 string elements with the String class...
Duration: 2294 milliseconds

Generating 100000 string elements with the StringBuilder class...
Duration: 1111 milliseconds
```

The preceding code demonstrated that using the *StringBuilder* class with an element length of *100* characters already doubles the speed. Restart the program. To see a really impressive increase in speed, enter an element length of *1000* and specify the value of *10000* for the number of elements to be generated, as shown in the following example:

```
Enter the string length of an element: 1000
Enter a number of elements to be generated: 10000

Generating 10000 string elements with the String class...
Duration: 6983 milliseconds

Generating 10000 string elements with the StringBuilder class...
Duration: 1091 milliseconds
```

With these parameters, the *StringBuilder* is approximately six times faster than the *String* class. The lengthier the generated strings, the more sense it makes to use a *StringBuilder* object.

The program relies on the *StopWatch* class for timing the operations. With this class, you can measure time durations with extreme precision—and it's simple to use. The following code sample, which reuses some code from earlier in the chapter, shows its use:

```
Imports System.Text

Module StringsVsStringBuilder

    Sub Main()
        Dim locAmountElements As Integer
        Dim locAmountCharsPerElement As Integer
        Dim locVBStringElements As VBStringElements
        Dim locVBStringBuilderElements As VBStringBuilderElements

        'StringBuilderExamples()
        'Return

        Console.Write("Enter the string length of an element: ")
        locAmountCharsPerElement = Integer.Parse(Console.ReadLine)
        Console.Write("Enter a number of elements to be generated: ")
```

```
        locAmountElements = Integer.Parse(Console.ReadLine)
        Console.WriteLine()
        Console.WriteLine("Generating " & locAmountElements &
                    " string elements with the String class...")
        Dim locTimeGauge = Stopwatch.StartNew
        locVBStringElements = New VBStringElements(locAmountElements,
➥locAmountCharsPerElement)
        locTimeGauge.Stop()
        Console.WriteLine("Duration: " & locTimeGauge.ElapsedMilliseconds.ToString())
        locTimeGauge.Reset()
        Console.WriteLine("Generating " & locAmountElements &
                    " string elements with the StringBuilder class...")
        locTimeGauge.Start()
        locVBStringBuilderElements =
                    New VBStringBuilderElements(locAmountElements,
➥locAmountCharsPerElement)
        locTimeGauge.Stop()
        Console.WriteLine("Duration: " & locTimeGauge.ElapsedMilliseconds.ToString())
        locTimeGauge.Reset()
        Console.WriteLine()
        Console.ReadLine()
    End Sub

    Sub StringBuilderExamples()

        'Declaration without parameters:
        Dim locSB As New StringBuilder
        'Declaration with capacity reservation
        locSB = New StringBuilder(1000)
        'Declaration from an existing string
        locSB = New StringBuilder("Created from a new string")
        'Declaration from string with the specification of a capacity to be reserved
        locSB = New StringBuilder("Created from string with capacity for more", 1000)

        locSB.Append(" - and this gets appended to the string")
        locSB.Insert(20, ">>this ends up somewhere in the middle<<")
        locSB.Replace("String", "StringBuilder")
        locSB.Remove(0, 4)

        'StringBuilder has finished building the string
        'Convert to string umwandeln
        Dim locString As String = locSB.ToString
        Console.WriteLine(locString)
        Console.ReadLine()
    End Sub

End Module

Public Class VBStringElements

    Private myStrElements() As String
```

```
    Sub New(ByVal AmountOfElements As Integer, ByVal AmountChars As Integer)

        ReDim myStrElements(AmountOfElements - 1)
        Dim locRandom As New Random(DateTime.Now.Millisecond)
        Dim locString As String

        For locOutCount As Integer = 0 To AmountOfElements - 1
            locString = ""
            For locInCount As Integer = 0 To AmountChars - 1
                Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)
                If locIntTemp > 26 Then
                    locIntTemp += 97 - 26
                Else
                    locIntTemp += 65
                End If
                locString += Convert.ToChar(locIntTemp).ToString
            Next
            myStrElements(locOutCount) = locString
        Next
    End Sub

End Class

Public Class VBStringBuilderElements

    Private myStrElements() As String

    Sub New(ByVal AmountOfElements As Integer, ByVal AmountChars As Integer)

        ReDim myStrElements(AmountOfElements - 1)
        Dim locRandom As New Random(DateTime.Now.Millisecond)
        Dim locStringBuilder As StringBuilder

        For locOutCount As Integer = 0 To AmountOfElements - 1
            locStringBuilder = New StringBuilder(AmountChars)
            For locInCount As Integer = 0 To AmountChars - 1
                Dim locIntTemp As Integer = Convert.ToInt32(locRandom.NextDouble * 52)
                If locIntTemp > 26 Then
                    locIntTemp += 97 - 26
                Else
                    locIntTemp += 65
                End If
                locStringBuilder.Append(Convert.ToChar(locIntTemp))
            Next
            myStrElements(locOutCount) = locStringBuilder.ToString
        Next
    End Sub

End Class
```

The *Boolean* Data Type

The data type *Boolean* saves binary states, which means it doesn't save much. Its value can be *False* or *True*—it can't save anything else. This data type is most frequently used when running conditional program code (you saw the basics of conditional code in Chapter 1).

.NET data type: *System.Boolean*

Represents: One of two states: *True* or *False*.

Type literal: Not available

Memory requirements: 2 bytes

Note: To define a *Boolean* variable, use the keywords *True* and *False* directly and without quotes in the program text, such as in the following example:

```
Dim locBoolean As Boolean
locBoolean = True 'Expression is true.
locBoolean = False 'Expression is false.
```

Converting to and from Numeric Data Types

You can convert a *Boolean* type to a numeric data type.

Important Visual Basic deviates from the .NET Framework in its internal representation of the primitive *Boolean* data type. For example, when you convert a *Boolean* data type into an *Integer* data type using Visual Basic commands, the value *True* is converted to –1. But when using .NET Framework conversions, such as the *Convert* class, *True* is converted to *+1*.

The following example shows how this works:

```
Dim locInt As Integer = CInt(locBoolean)    ' locInt is -1
locInt = Convert.ToInt32(locBoolean)        ' locInt is now +1!!!
Dim locLong As Long = CLng(locBoolean)      ' locLong is -1
locLong = Convert.ToInt64(locBoolean)       ' locLong is +1
```

When converting it back, the behavior of the *Convert* class of .NET Framework and the conversion statements of Visual Basic are identicial. Only the numeric value 0 returns the *Boolean* result of *False*, all other values result in *True*, as the following example shows.

Companion Content Open the corresponding solution (.sln) for this example, which you can find in the \VB 2010 Developer Handbook\Chapter 06\Primitives03 folder.

```
locBoolean = CBool(-1)                  ' locBoolean is True.
locBoolean = CBool(0)                   ' locBoolean is False.
locBoolean = CBool(1)                   ' locBoolean is True.
locBoolean = Convert.ToBoolean(-1)      ' locBoolean is True.
locBoolean = Convert.ToBoolean(+1)      ' locBoolean is True.
locBoolean = CBool(100)                 ' locBoolean is True.
locBoolean = Convert.ToBoolean(100)     ' locBoolean is True.
```

Converting to and from Strings

When you convert a *Boolean* data type into a string—for example, to save its status in a file—the respective value is converted to one of two strings represented by the static read-only properties *TrueString* and *FalseString* of the *Boolean* structure. In the current version of .NET Framework (4.0, as of this writing), they result in "True" and "False," regardless of the computer's cultural settings. So, no matter if your programs run on a United States platform or, for example, on a German platform, it always becomes either "True" or "False", never "Wahr" or "Falsch". When converting a *String* into *Boolean*, these strings return the values *True* and *False*, respectively.

The *Date* Data Type

The *Date* data type stores and manipulates date values. It helps you to calculate time differences, parse date values from strings, and convert date values into formatted strings.

.NET data type: *System.DateTime*

Represents: Dates and time from 1.1.0001 (0 hours) to 31.12.9999 (23:59:59 hours) with a resolution of 100 nanoseconds (this unit is called a *tick*)

Type literal: Enclosed in pound-signs, *always* United States culture format (#MM/dd/yyyy HH:mm:ss#)

Memory requirements: 8 bytes

Note *Date* also is a base data type, for which you can define a date variable directly by using literals in the program code. The same rules apply for numeric values: The United States date/time format is important. If you are not familiar with this format, here's a brief explanation.

For the United States, you write date values in the format month/day/year, separated by a slash. This syntax can easily cause confusion if you are not familiar with it. For example, the date 12/17/2011 is clearly a United States date, because there is no month 17. However, the date 12/06/2011 could be interpreted as either June 12th or December 6th. There is a similar

issue with the time of day. In the United States, you might find a 24-hour display for a bus schedule or in the military, but otherwise, the postfixes "AM" (for "ante meridian"—before noon) and "PM" (for "post meridian"—after noon) defines which 3 o'clock is intended. The formatting scheme becomes even more problematic with 12:00 (there is no 0:00!). Maybe you've had your own experience when trying to program a video recorder that recorded not your desired TV program, but another that was broadcast 12 hours later (or earlier). 12:00 AM corresponds to midnight or the 0:00 hour on the 24-hour clock; 12:00 PM corresponds to noon.

Value assignments to a *Date* data type in program code occur by surrounding the date/time string characters with "#" (hash) characters. The following example shows how this works:

```
Dim Midnight As Date = #12:00:00 AM#
Dim Noon As Date = #12:00:00 PM#
Dim NewYearsEve As System.DateTime = #12/31/2010#
Dim TimeForChampagne As System.DateTime = #12/31/2010 11:58:00 PM#
Dim TimeForAspirin As System.DateTime = #1/1/2011 11:58:00 AM#
```

The editor helps you to find the correct format by translating the 24-Hour format into the 12-hour format, automatically. For example, it converts the expression *#0:00#* to *#12:00:00 AM#* automatically.

It also adds missing entries for minutes and seconds if you inadvertently enter a value that contains only the hours portion. You can enter times in the 24-hour format; the editor will automatically convert them to the 12-hour format.

TimeSpan: Manipulating Time and Date Differences

What's unique about the *Date* data type is that it supports calculations to determine time differences, representing a length of time with *TimeSpan* objects. These objects represent time intervals, not time values. Unlike the *Date* type, *TimeSpan* is not a .NET base data type.

The *TimeSpan* data type is quite easy to use. You can subtract one data value from another to determine the time span between the two dates, or add a time span to a date, or subtract it from a date (see the following example) to calculate the date after so many months, days or hours.

> **Companion Content** Open the corresponding solution (.sln) for this example, which you can find in the \VB 2010 Developer Handbook\Chapter 06\DateTime folder.

```
Dim locDate1 As Date = #3:15:00 PM#
Dim locDate2 As Date = #4:23:32 PM#
Dim locTimeSpan As TimeSpan = locDate2.Subtract(locDate1)
Console.WriteLine("The time span between {0} and {1} is", _
                  locDate1.ToString("HH:mm:ss"), _
                  locDate2.ToString("HH:mm:ss"))
```

```
Console.WriteLine("{0} second(n) or", locTimeSpan.TotalSeconds)
Console.WriteLine("{0} minute(n) and {1} second(n) or", _
                  Math.Floor(locTimeSpan.TotalMinutes), _
                  locTimeSpan.Seconds)
Console.WriteLine("{0} hour(s), {1} minute(s) and {2} second(s) or", _
                  Math.Floor(locTimeSpan.TotalHours), _
                  locTimeSpan.Minutes, locTimeSpan.Seconds)
Console.WriteLine("{0} Ticks", _
                  locTimeSpan.Ticks)
```

A Library with Useful Functions for Date Manipulation

In the same example, you will find a class file called *DateCalcHelper.vb* that contains a static class of the same name. This class provides some useful functions that simplify the calculation of certain relative points in time, and shows how to perform calculations with date values.

Thanks to the XML comments in the example, the class is self-explanatory. When you develop you own programs that make intensive use of relative point-in-time calculations, just add this code file to your project (or the assembly of your project).

The following code shows the function names along with their explanations (in bold):

```
Public NotInheritable Class DateCalcHelper

    ''' <summary>
    ''' Calculates the date which corresponds to the 1st of the month,
    ''' which results from the specified date.
    ''' </summary>
    ''' <param name="CurrentDate">Date, whose month the calculation is based on.
➥</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function FirstDayOfMonth(ByVal CurrentDate As Date) As Date
        Return New Date(CurrentDate.Year, CurrentDate.Month, 1)
    End Function

    ''' <summary>
    ''' Calculates the date which corresponds to the last day of the month,
    ''' which results from the specified date.
    ''' </summary>
    ''' <param name="CurrentDate">Date, whose month the calculation is based on.
➥</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function LastDayOfMonth(ByVal CurrentDate As Date) As Date
        Return New Date(CurrentDate.Year, CurrentDate.Month, 1).AddMonths(1).
➥AddDays(-1)
    End Function
```

```
''' <summary>
''' Calculates the date which corresponds to the first of the year,
''' which results from the specified date.
''' </summary>
''' <param name="CurrentDate">Date, whose year the calculation is based on.
''' </param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function FirstOfYear(ByVal CurrentDate As Date) As Date
    Return New Date(CurrentDate.Year, 1, 1)
End Function

''' <summary>
''' Calculates the date which corresponds to the first Monday of the first week of
''' the month, which results from the specified date.
''' </summary>
''' <param name="CurrentDate">Date, whose week the calculation is based on.
''' </param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function MondayOfFirstWeekOfMonth(ByVal CurrentDate As Date) As Date
    Dim locDate As Date = FirstDayOfMonth(CurrentDate)
    If Weekday(locDate) = DayOfWeek.Monday Then
        Return locDate
    End If
    Return locDate.AddDays(6 - Weekday(CurrentDate))
End Function

''' <summary>
''' Calculates the date which corresponds to the Monday of the week,
''' which results from the specified date.
''' </summary>
''' <param name="CurrentDate">Date, whose week the calculation is based on.
''' </param>
''' <returns></returns>
''' <remarks></remarks>
Public Shared Function MondayOfWeek(ByVal CurrentDate As Date) As Date
    If Weekday(CurrentDate) = DayOfWeek.Monday Then
        Return CurrentDate
    Else
        Return CurrentDate.AddDays(-Weekday(CurrentDate) + 1)
    End If
End Function

''' <summary>
''' Calculates the date which corresponds to the first Monday
''' of the second week of the month,
''' which results from the specified date.
''' </summary>
''' <param name="CurrentDate">Date, whose week the calculation is based on.
''' </param>
```

```
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function MondayOfSecondWeekOfMonth(ByVal currentDate As Date) As
➥Date
        Return MondayOfFirstWeekOfMonth(currentDate).AddDays(7)
    End Function

    ''' <summary>
    ''' Calculates the date which corresponds to Monday of the last week of the month,
    ''' which results from the specified date.
    ''' </summary>
    ''' <param name="CurrentDate">Date, whose week the calculation is based on.
    ''' </param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function MondayOfLastWeekOfMonth(ByVal CurrentDate As Date) As Date
        Dim locDate As Date = FirstDayOfMonth(CurrentDate).AddDays(-1)
        If Weekday(locDate) = DayOfWeek.Monday Then
            Return locDate
        End If
        Return locDate.AddDays(-Weekday(CurrentDate) + 1)
    End Function

    ''' <summary>
    ''' Results in the date of the next work day.
    ''' </summary>
    ''' <param name="CurrentDate">Date the calculation is based on</param>
    ''' <param name="WorkOnSaturdays">True, if Saturday is a work day.</param>
    ''' <param name="WorkOnSundays">True, if Sunday is a work day.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
    Public Shared Function NextWorkday(ByVal CurrentDate As Date,
             ByVal WorkOnSaturdays As Boolean, _
             ByVal WorkOnSundays As Boolean) As Date
        CurrentDate = CurrentDate.AddDays(1)
        If Weekday(CurrentDate) = DayOfWeek.Saturday And Not WorkOnSaturdays Then
            CurrentDate = CurrentDate.AddDays(1)
        End If
        If Weekday(CurrentDate) = DayOfWeek.Sunday And Not WorkOnSundays Then
            CurrentDate = CurrentDate.AddDays(1)
        End If
        Return CurrentDate
    End Function

    ''' <summary>
    ''' Results in the date of the previous work day.
    ''' </summary>
    ''' <param name="CurrentDate">Date the calculation is based on</param>
    ''' <param name="WorkOnSaturdays">True, if Saturday is a work day.</param>
    ''' <param name="WorkOnSundays">True, if Sunday is a work day.</param>
    ''' <returns></returns>
    ''' <remarks></remarks>
```

```
    Public Shared Function PreviousWorkday(ByVal CurrentDate As Date,
            ByVal WorkOnSaturdays As Boolean, _
            ByVal WorkOnSundays As Boolean) As Date
        CurrentDate = CurrentDate.AddDays(-1)
        If Weekday(CurrentDate) = DayOfWeek.Sunday And Not WorkOnSundays Then
            CurrentDate = CurrentDate.AddDays(-1)
        End If
        If Weekday(CurrentDate) = DayOfWeek.Saturday And Not WorkOnSaturdays Then
            CurrentDate = CurrentDate.AddDays(-1)
        End If
        Return CurrentDate
    End Function
End Class
```

Converting Strings to Date Values

Just like the base numeric data types, you can also convert strings that represent date values into a *Date* data type. The *Date* data type provides two functions, *Parse* and *ParseExact*, that analyze a string and build the actual date value from it.

Conversions with *Parse*

When using *Parse*, the parser uses every trick in the book to convert a date, a time, or a combination of both into a time value, as shown in the following example:

```
Dim locToParse As Date
locToParse = Date.Parse("13.12.10") ' OK, basic European setting is processed.
locToParse = Date.Parse("6/7/10")   ' OK, but European date is used in spite of "/".
locToParse = Date.Parse("13/12/10") ' OK, as above.
locToParse = Date.Parse("06.07")    ' OK, is extended by the year.
locToParse = Date.Parse("06,07,10") ' OK, comma is acceptable.
locToParse = Date.Parse("06,07")    ' OK, comma is acceptable; year is added.
'locToParse = Date.Parse("06072010") ' --> Exception: was not recognized as a valid date!
'locToParse = Date.Parse("060705")   ' --> Exception: was not recognized as a valid date!
locToParse = Date.Parse("6,7,4")    ' OK, comma is acceptable; leading zeros are added.

locToParse = Date.Parse("14:00")    ' OK, 24-hour display is acceptable.
locToParse = Date.Parse("PM 11:00") ' OK, PM may be in front of...
locToParse = Date.Parse("11:00 PM") ' ...and behind the time specification.
'locToParse = Date.Parse("12,00 PM") ' --> Exception: was not recognized as a valid date!

'Both date/time combinations work:
locToParse = Date.Parse("6.7.10 13:12")
locToParse = Date.Parse("6,7,10 11:13 PM")
```

As you can see here, a format entry that is very common in European locales is not recognized: when the individual value groups of the date are written sequentially but without a separating character. However, there is a solution to this problem as well.

Note The date 6/7/10 represents a July date, for example, in the Irish locale, and a June date in the United States locale.

Conversion with *ParseExact*

If, in spite of all its flexibility, *Parse* fails to recognize a valid date/time format, you can still set a recognition pattern for the entry by using the method *ParseExact* for string conversions.

Note You should also use *ParseExact* if you don't want to allow as much flexibility as *Parse* permits.

This is especially true when you need to differentiate between time and date values. For a field in which the users of your program must enter a time, your program knows, for example, that the value 23:12 refers to the time 23:12:00, and not to the date 23.12.2000. *Parse* wouldn't work here, because it can't recognize the context.

Apart from the string to be analyzed, *ParseExact* requires at least two additional parameters: a string that contains the specific recognition pattern, and a *format provider* that provides further formatting requirements. There are several different format providers that you can use—but you can access them only after inserting the following line, which imports the required namespace at the beginning of your module or class file:

```
Imports System.Globalization
```

The simplest version that will recognize a time entry as the time of day, if it has been entered in the above format, it would look like this:

```
locToParse = Date.ParseExact("12,00", "HH,mm", CultureInfo.CurrentCulture)
```

The string "HH" specifies that hours are expressed in the 24-hour format. If you use the lower-case pattern "hh" instead, the parser will recognize only the 12-hour format. Next, the input contains a comma, which becomes the separator character, and finally the format specifies that minutes come last, using the string "mm."

In practice, it's rare that users follow specific requirements; therefore, your program should ideally recognize several different versions of time entries. With the *ParseExact* function, you can specify a range of possible formats for the parser to perform the conversion. All you need to do is define a *String* array containing the permitted formats, and then pass it to the *ParseExact* method along with the string to be parsed. If you decide to use this method, however, you also need to specify a parameter that regulates the parsing flexibility (for example, if the input strings that will be analyzed are allowed to contain whitespace, which then will be ignored). The entry is regulated by a parameter of the type *DateTimeStyles* which allows the settings listed in Table 6-5.

TABLE 6-5 The Extended Settings That Can Be Used with *Parse*

Member name	Description	Value
AdjustToUniversal	Date and time must be converted to Universal Time or Greenwich Mean Time (GMT)	16
AllowInnerWhite	Additional whitespaces within the string are ignored during parsing, unless the *DateTimeFormatInfo* format patterns contain spaces	4
AllowLeadingWhite	Leading whitespaces are ignored during parsing, unless the *DateTimeFormatInfo* format patterns contain spaces	1
AllowTrailingWhite	Trailing whitespaces are ignored during parsing, unless the *DateTimeFormatInfo* format patterns contain spaces	2
AllowWhiteSpaces	Additional whitespaces, which are located at any position within the string, are ignored during parsing, unless the *DateTimeFormatInfo* format patterns contain spaces. This value is equivalent to the combination of *AllowLeadingWhite*, *AllowTrailingWhite*, and *AllowInnerWhite*	7
NoCurrentDateDefault	Date and time are inseparately combined in the *Date* data type. Even if only a time is assigned, the *Date* value will always show a valid date. This setting specifies that the *DateTime.Parse* method and the *DateTime.ParseExact* method use a date according to the Gregorian calendar with year = 1, month = 1, and day = 1, when the string only contains the time, but not the date. If this value isn't specified, the current date is used.	8
None	Specifies that the default formatting options must be used; for instance, the default format for *DateTime.Parse*, and *DateTime.ParseExact*.	0

The following lines of code show how to use *ParseExact* to convert strings into date values with specific requirements for date formats:

```
Imports System.Globalization

Module Module1

    Sub Main()

        Dim locToParseExact As Date
        Dim locTimePattern As String() = {"H,m", "H.m", "ddMMyy", "MM\/dd\/yy"}

        'Works: it's in the time pattern.
        locToParseExact = Date.ParseExact("12,00", _
                          locTimePattern, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.AllowWhiteSpaces)
```

```
        'Works: it's in the time pattern, and whitespaces are permitted.
        locToParseExact = Date.ParseExact(" 12 , 00 ", _
                          locTimePattern, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.AllowWhiteSpaces)

        'Doesn't work: it's in the time pattern, but whitespaces are not permitted.
        'locToParseExact = Date.ParseExact(" 12 , 00 ", _
        '                  locTimePattern, _
        '                  CultureInfo.CurrentCulture, _
        '                  DateTimeStyles.None)

        'Works: it's in the time pattern.
        locToParseExact = Date.ParseExact("1,2", _
                          locTimePattern, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.None)
        'Works: it's in the time pattern.
        'But the date corresponds to 1.1.0001 and is therefore
        'not displayed as a Tooltip, contrary to all the other
        'examples shown here.
        locToParseExact = Date.ParseExact("12.2", _
                          locTimePattern, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.NoCurrentDateDefault)

        'Works: it's not in the time pattern, because seconds are used
        'locToParseExact = Date.ParseExact("12,2,00", _
        '                  locTimePattern, _
        '                  CultureInfo.CurrentCulture, _
        '                  DateTimeStyles.NoCurrentDateDefault)

        'Doesn't work: the colon is not in the time pattern.
        'locToParseExact = Date.ParseExact("1:20", _
        '                  locTimePattern, _
        '                  CultureInfo.CurrentCulture, _
        '                  DateTimeStyles.None)

        'Now it works, because it's used as date in the time pattern.
        '(third element in the string array)
        locToParseExact = Date.ParseExact("241205", _
                          locTimePattern, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.AllowWhiteSpaces)

        'Works: US format is used,
        'as defined by the slashes and the group order.
        '(fourth element in the string array).
        locToParseExact = Date.ParseExact("12/24/05", _
                          locTimePattern, _
                          CultureInfo.CurrentCulture, _
                          DateTimeStyles.AllowWhiteSpaces)
    End Sub

End Module
```

When you define slashes as group separators, remember to always put a backslash in front so the separators aren't processed as control characters.

.NET Equivalents of Base Data Types

There is a .NET equivalent for each base data type in Visual Basic, as shown in Table 6-6.

TABLE 6-6 The Base Visual Basic Data Types and Their .NET Equivalents

Base data type in Visual Basic	.NET data type equivalent
Byte	*System.Byte*
SByte	*System.SByte*
Short	*System.Int16*
UShort	*System.UInt16*
Integer	*System.Int32*
UInteger	*System.UInt32*
Long	*System.Int64*
ULong	*System.UInt64*
Single	*System.Single*
Double	*System.Double*
Decimal	*System.Decimal*
Boolean	*System.Boolean*
Date	*System.DateTime*
Char	*System.Char*
String	*System.String*

Table 6-6 illustrates that it doesn't matter at all whether you declare a 32-bit integer with

```
Dim loc32BitInteger as Integer
```

or with

```
Dim loc32BitInteger as System.Int32
```

The object variable *loc32BitInteger* ends up with the exact same type in both cases. Take a look at the IML-generated code that follows:

```
Public Shared Sub main()

    Dim locDate As Date = #12/14/2003#
    Dim locDate2 As DateTime = #12/14/2003 12:13:22 PM#
    If locDate > locDate2 Then
        Console.WriteLine("locDate is larger than locDate2")
```

```
    Else
        Console.WriteLine("locDate2 is larger than locDate")
    End If
```

The generated lines of code verify that this is true:

```
.method public static void  main() cil managed
{
  // Code size       75 (0x4b)
  .maxstack  2
  .locals init ([0] valuetype [mscorlib]System.DateTime locDate,
           [1] valuetype [mscorlib]System.DateTime locDate2,
           [2] bool VB$CG$t_bool$S0)
  IL_0000:  nop
  IL_0001:  ldc.i8     0x8c58fec59f98000
  IL_000a:  newobj     instance void [mscorlib]System.DateTime::.ctor(int64)
  IL_000f:  nop
  IL_0010:  stloc.0
  IL_0011:  ldc.i8     0x8c59052cd35dd00
  IL_001a:  newobj     instance void [mscorlib]System.DateTime::.ctor(int64)
  IL_001f:  nop
  IL_0020:  stloc.1
  IL_0021:  ldloc.0
  IL_0022:  ldloc.1
.
.
.
```

As the code highlighted in bold shows, both local variables have been declared as *System.DateTime* type. Notice also that *Date* variables are represented internally as *Long* values).

The GUID Data Type

GUID is the abbreviation for *Globally Unique Identifier*. The term "Global" refers to the fact that even though two GUID generators aren't aware of each other's existence and are spatially separated from each other, they are highly unlikely to produce two identical identifiers.

GUIDs are 16-byte (128-bit) values, usually represented in the format {XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}, where each "X"represents a hexadecimal number between 0 and F.

In .NET, the name of the data type (GUID) corresponds to its abbreviation.

Note Don't use the GUID constructor to create a new GUID. Because a GUID is a value type, doing so wouldn't make any sense anyway. If you use the default constructor (the constructor with no parameters) with a value type, the value type is created on the Managed Heap but is then immediately discarded. Instead, use the static function *NewGuid* to create a new GUID, as shown in the following example:

```
Dim t As Guid
t = Guid.NewGuid
```

GUIDs are generally used as primary keys in databases, because it is extremely improbable that two computer systems generate identical GUIDs. Database tables with foreign key identifiers based on GUIDs are particularly useful for synchronizing databases that cannot be constantly connected for technical reasons.

The *Guid* data type has a constructor with parameters that let you recreate an existing GUID via a string. Using it this way makes sense, for example, when a component (class, structure) that you have developed must always return the same unique identifier.

The following automatically implemented property of a class could therefore implement a *UniqueID* property:

```
Public Class AClass

    'The property always returns the same GUID:
    Property UniqueID As Guid = New Guid("{46826D55-6FDD-44FA-BADE-515E04770816}")
End Class
```

You can read more about classes and properties in Part II, "Object-Oriented Programming."

Tip When you want only a new GUID string, you don't need to write an application that uses *NewGuid* to create one. Instead, you can use a feature already in Visual Studio. On the Tools menu, click Create GUID. The Create GUID dialog box appears. In the dialog box, select Registry Format, and then click Copy (see Figure 6-2). Go back to the Code Editor and paste the clipboard content as a parameter for the *Guid* constructor, as shown in the preceding example.

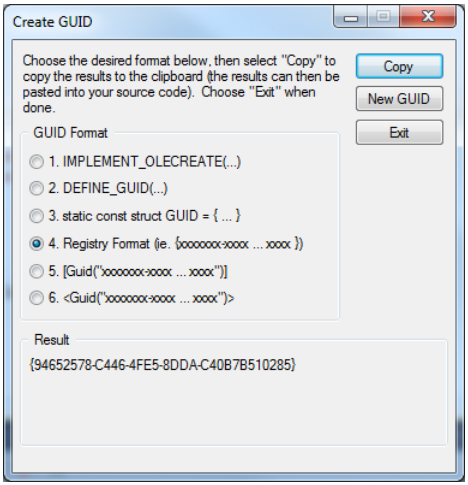

FIGURE 6-2 Using the Create GUID dialog box in the Visual Studio IDE, you can easily create a GUID constant.

Constants and Read-Only Fields (Read-Only Members)

Apart from the variable types discussed in this chapter, there are two *value storage types* in Visual Basic, which at first glance seem rather similar, *constants* and *read-only* fields. These two value storage types are defined only once during your program's lifetime, but they behave very differently under certain circumstances—and their contents are saved in completely different manners.

Before discussing the differences, let's examine what they have in common. Read-only fields and constants are used in similar contexts: namely, when a value must be used at different places within a program. For example, you would probably use a constant to return the name of your program or the expiration date of a demo version. When queried at different places within your program the constant must always have the same value. To avoid having to change the value in the source code in many different places (should you need to make any changes later on) you define this value as a constant or read-only field centrally, and then substitute the field or constant name for the value where you would use the value in the code.

Of course, you need to ensure that this value cannot be overwritten. Therefore, neither constants nor read-only fields can be changed.

Constants

You define constants with the *Const* keyword at the module level. Module level means that you can define a constant in a class, in a module, or in a structure. The syntax for defining a constant is just like a variable, but you need to add the keyword *Const*, as shown in the following example:

```
Public Const APPLICATIONNAME As String = "Type demo"
```

Or:

```
Private Const EXPIRATIONDATE as Date=#12/31/2010#
```

You can also define other constants this way—you just need to specify the appropriate type with the *As* clause and a value.

But watch out!

You can only define actual constants as constants. Even if, for example, the return value of a method, such as *Date.MaxValue* actually has the *characteristics* of a constant, you can't assign it to a constant. For example, the following statement will cause an error:

```
Private Const EXPIRATIONDATE as Date=Date.MaxValue
```

In this case, you should use a read-only field, as explained in the following section.

There's another important issue—the use of public constants across several assemblies.

Important You might think that a constant could therefore be "misused" as a field variable, which is read-only. However, that can backfire, when public constants are accessed in other assemblies.

When you give a constant a value, a concrete value is *never* created and saved at runtime. Therefore, constants don't need any memory space at runtime—neither on the Managed Heap, nor on the stack, nor within any processor registers. Instead, the actual value is hidden in the metafiles of the assembly in which the constant is defined. Unfortunately, that can lead to problems if different versions are used.

Because they are saved only in the metadata of the assembly, constants can be evaluated only at compilation time, not at design time. Therefore, for example, if you define a public constant in assembly A and access it from assembly B, you basically only need assembly A during the *creation* of assembly B. To put it simply, this is because the compiler only looks in assembly A, *while* it is creating assembly B, and transfers the values of the constants to assembly B where necessary. If assembly A is exclusively used to call constants, you could dispose of it afterward. And that's the problem. Because assembly A isn't actually accessed at runtime but only at compilation time, it's not enough to exchange assembly A, if the value of the constant defined there changes. Since assembly B accessed assembly A only at compilation time, it is not affected by the change.

For this reason you need to be careful with public constants and *always recreate an application with all connected assemblies when a scenario such as the one described above takes place*. To avoid this kind of behavior from the start, an alternative is to use read-only properties. The topic of properties is discussed in detail in Chapter 9, "First Class Programming."

Read-Only Fields

Read-only fields can also define constant values. These are also defined exclusively at module level (class, structure, module), and differ from a typical variable declaration by the keyword *ReadOnly*:

```
Private ReadOnly THEDATE As Date=#07/24/1969#
Friend ReadOnly MAXDATE As Date=Date.MaxValue
```

Or:

```
Public ReadOnly PERMITTED_CITIES As New List(Of String) From {"Lippstadt", "Las Vegas",
                                                              "Kempen", "Los Angeles"}
```

These examples show that unlike regular constants, read-only fields are executable statements.

Important An assignment to a read-only field might occur only in the constructor of the class, the structure, or the module, as you can see in the following code segment.

Therefore, the following construction is permitted:

```
Public Class Test
    Friend ReadOnly MAXDATE As Date
    Public ReadOnly PERMITTED_CITIES As List(Of String)

    Sub New()
        'Permitted only once!
        MAXDATE = Date.MaxValue
        PERMITTED_CITIES = New List(Of String) From {"Lippstadt", "Las Vegas",
➥"Kempen", "Los Angeles"}
    End Sub
.
.
.
```

If you attempt to assign or re-assign a value to a read-only field from within a method, as in the following code segment, Visual Basic generates a design-time warning. The following code will cause the error *"A read-only variable cannot be the target of an assignment."*

```
Sub NewMethod()
    MAXDATE = Date.MaxValue
    PERMITTED_CITIES = New List(Of String) From {"Munich", "Paris", "London", "Seattle"}
```

Important With objects that can be defined as read-only fields, as we have seen in the code sample, the definition as *ReadOnly* only prohibits a completely new assignment. However, since in this case the variable is a pointer to an object, nothing stands in the way of manipulating the object content. Therefore, the following code is valid—although whether it's practical is a different question:

```
        'But this works:
        PERMITTED_CITIES.Clear()
        PERMITTED_CITIES.AddRange(New List(Of String) From {"Munich", "Paris",
            "London", "Seattle"})
    End Sub

End Class
```

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Klaus Löffelmann

Klaus Löffelmann is a Microsoft MVP for Visual Basic .NET, and has been a professional software developer for over 20 years. He has written several books about Visual Basic and is the owner and founder of ActiveDevelop in Lippstadt, Germany, a company specializing in software development, localization, technical literature, and training/coaching with Microsoft technologies.

Sarika Calla Purohit

Sarika Calla Purohit is a Software Design Engineer Test Lead on the Visual Studio Languages team at Microsoft. She has been a member of the Visual Studio team for over eight years and has contributed to Visual Basic .Net since version 1.1. Most recently, her team was responsible for testing the Visual Basic IDE in Visual Studio 2010.

What do you think of this book?

We want to hear from you!

To participate in a brief online survey, please visit:

microsoft.com/learning/booksurvey

Tell us how well this book meets your needs—what works effectively, and what we can do better. Your feedback will help us continually improve our books and learning resources for you.

Thank you in advance for your input!

Microsoft® Press

Stay in touch!

To subscribe to the *Microsoft Press® Book Connection Newsletter*—for news on upcoming books, events, and special offers—please visit:

microsoft.com/learning/books/newsletter