

*Rich Client Applications with C# or VB.NET*

Covers  
.NET 1.1 &  
Visual Studio .NET 2003



*Programming*

# .NET Windows Applications

**O'REILLY®**

*Jesse Liberty & Dan Hurwitz*

# Programming .NET Windows Applications



Microsoft's .NET Windows Forms provide a better way to develop standalone and networked Windows applications for PCs and other devices. This new technology gives you more power and simpler deployment, using a streamlined programming model that deals automatically with many tedious details that once plagued developers.

As with most things .NET, the only hitch is the learning curve. With this tutorial, you will learn all aspects of using .NET Windows Forms class libraries and the associated programming tools in Visual Studio .NET to build applications for the Windows desktop platforms. You'll learn how to design applications that either function alone on a PC or work in combination with your web-based application server to take advantage of the richer Windows interface and higher level of security.

You will discover how your new Windows applications can sidestep problems that used to arise from the use of DLLs (known collectively as "DLL hell"), and how .NET Windows Forms can be used as an alternative to ASP.NET and browser-based approaches for building rich web application clients.

Jesse Liberty, the author of O'Reilly's *Programming C#* and *Learning Visual Basic .NET*, is well-known for his clear and concise style, prompting one reviewer to say, "It's as if he knows exactly what questions I'm going to ask ahead of time." Jesse also coauthored *Programming ASP.NET* with contract programmer **Dan Hurwitz**, and the two have teamed up again to bring you this comprehensive tutorial—without a doubt, the best source available for learning how to program with .NET Windows Forms.

Twitter: @oreillymedia  
facebook.com/oreilly

US \$49.95

CAN \$77.95

ISBN: 978-0-596-00321-0



9

O'REILLY®  
oreilly.com

---

# **Programming .NET Windows Applications**

## Other Microsoft .NET resources from O'Reilly

---

<b>Related titles</b>	Programming C#	ADO.NET in a Nutshell
	C# in a Nutshell	.NET Windows Forms in a Nutshell
	Programming Visual Basic .NET	.NET Framework Essentials
	Programming ASP.NET	Mastering Visual Studio .NET
	ASP.NET in a Nutshell	

---

### **.NET Books Resource Center**

*dotnet.oreilly.com* is a complete catalog of O'Reilly's books on .NET and related technologies, including sample chapters and code examples.



*ONDotnet.com* provides independent coverage of fundamental, interoperable, and emerging Microsoft .NET programming and web services technologies.

---

### **Conferences**

O'Reilly & Associates bring diverse innovators together to nurture the ideas that spark revolutionary industries. We specialize in documenting the latest tools and systems, translating the innovator's knowledge into useful skills for those in the trenches. Visit *conferences.oreilly.com* for our upcoming events.



Safari Bookshelf (*safari.oreilly.com*) is the premier online reference library for programmers and IT professionals. Conduct searches across more than 1,000 books. Subscribers can zero in on answers to time-critical questions in a matter of seconds. Read the books on your Bookshelf from cover to cover or simply flip to the page you need. Try it today with a free trial.

---

# Programming .NET Windows Applications

*Jesse Liberty and Dan Hurwitz*

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

## **Programming .NET Windows Applications**

by Jesse Liberty and Dan Hurwitz

Copyright © 2004 O'Reilly & Associates, Inc. All rights reserved.  
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*safari.oreilly.com*). For more information, contact our corporate/institutional sales department: (800) 998-9938 or *corporate@oreilly.com*.

**Editors:** Tatiana Apandi Diaz and Val Quercia

**Production Editor:** Mary Brady

**Cover Designer:** Ellie Volckhausen

**Interior Designer:** David Futato

### **Printing History:**

October 2003: First Edition.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly & Associates, Inc. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly & Associates, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps. The association between the image of a darter and the topic of .NET Windows applications is a trademark of O'Reilly & Associates, Inc.

IntelliSense, JScript, Microsoft, Visual Basic, Visual C++, Visual Studio, Windows, and Windows NT, Visual C#, and Visual J# are registered trademarks of Microsoft Corporation.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 0-596-00321-8

[M]

---

# Table of Contents

<b>Preface</b> .....	<b>ix</b>
<b>1. Windows Forms and the .NET Framework</b> .....	<b>1</b>
The .NET Framework	1
Windows Forms	3
<b>2. Getting Started</b> .....	<b>5</b>
System Requirements	5
Hello World	7
<b>3. Visual Studio .NET</b> .....	<b>31</b>
Overview	31
Start Page	32
Projects and Solutions	34
The Integrated Development Environment (IDE)	36
Building and Running	70
<b>4. Events</b> .....	<b>71</b>
Publish and Subscribe	71
Performance	90
Some Examples	90
<b>5. Windows Forms</b> .....	<b>117</b>
Web Applications Versus Windows Applications	118
The Forms Namespace	120
Form Properties	124
Forms Inheritance	126
User Interface Design	144

<b>6. Dialog Boxes</b>	<b>168</b>
Modal Versus Modeless	168
Form Properties	169
DialogResult	174
Termination Buttons	179
Apply Button	181
CommonDialog Classes	189
<b>7. Controls: The Base Class</b>	<b>219</b>
Control Class	219
<b>8. Mouse Interaction</b>	<b>272</b>
SystemInformation Properties	272
Mouse Events	276
<b>9. Text and Fonts</b>	<b>305</b>
Text	305
Fonts	307
<b>10. Drawing and GDI+</b>	<b>342</b>
The Drawing Namespace	342
The Analog Clock Project	361
<b>11. Labels and Buttons</b>	<b>423</b>
Label	423
Button Classes	444
<b>12. Text Controls</b>	<b>474</b>
Text	474
Editable Text Controls: TextBoxBase	475
RichTextBox	503
<b>13. Other Basic Controls</b>	<b>529</b>
Containers	529
Tabbed Pages	540
PictureBox	553
ScrollBar	560
TrackBar	573
Up-Down Controls	578
ProgressBar	592

<b>14. TreeView and ListView</b> .....	<b>597</b>
Class Hierarchy	597
Splitter	598
TreeView	604
ListView	635
<b>15. List Controls</b> .....	<b>681</b>
Class Hierarchy	681
ListControls	681
<b>16. Date and Time Controls</b> .....	<b>738</b>
Class Hierarchy	738
Date and Time Values	738
DateTimePicker	748
MonthCalendar	759
Timer Component	776
<b>17. Custom Controls</b> .....	<b>791</b>
Specializing an Existing Control	792
Creating a User Control	797
Creating Custom Controls from Scratch	819
<b>18. Menus and Bars</b> .....	<b>836</b>
Creating Your First Menu	836
The MainMenu Object	837
Toolbars	880
Writing It by Hand	886
Status Bars	900
<b>19. ADO.NET</b> .....	<b>910</b>
Bug Database: A Windows Application	911
The ADO.NET Object Model	914
Getting Started with ADO.NET	919
Managed Providers	936
Binding Data	940
Data Reader	942
Creating a DataGrid	946

<b>20. Updating ADO.NET .....</b>	<b>985</b>
Updating with SQL .....	985
Updating Data with Transactions .....	993
Updating Data Using DataSets .....	1025
Multiuser Updates .....	1049
Command Builder .....	1070
<b>21. Exceptions and Debugging .....</b>	<b>1080</b>
Bugs Versus Exceptions .....	1080
Exceptions .....	1081
Throwing and Catching Exceptions .....	1082
Bugs .....	1084
Debugging in Visual Studio .NET .....	1084
Assert Yourself .....	1104
<b>22. Configuration and Deployment .....</b>	<b>1109</b>
Class Hierarchy .....	1109
Configuration .....	1110
Assemblies .....	1133
Build Configurations .....	1165
Deployment .....	1168
<b>Appendix: Characters and Keys .....</b>	<b>1191</b>
<b>Index .....</b>	<b>1203</b>

---

# Preface

Windows Forms represents the third generation of Windows development. When Microsoft first released Windows in 1985, programmers built applications using the Windows API, typically in C. Many of us learned how to build these applications from Charles Petzold, and this is a good place to thank him for his seminal book on Windows programming.

By 1992, many programmers were building Windows applications in C++ using the Microsoft Foundation Classes (MFC). Mike Blaszcack wrote a killer book on this topic, and it remains a classic. In essence, the MFC represented an object-oriented wrapper on the more procedural API.



In the 1990s, the alternative to building C++/MFC applications was using VB and its Rapid Application Development environment.

Microsoft first announced the third generation of Windows development, Windows Forms, and the .NET platform in July 2000. In short, C# (and Visual Basic .NET) and Windows Forms replace C++ and the MFC as well as classic VB. This book aims to provide a complete tutorial to this new way of creating Windows applications.



On a personal note, having spent nine years building MFC applications in C++ (and having earned much of my livelihood writing books about C++) you might expect me to have a certain resistance to the new paradigm. About an hour after writing my first C#/Windows Forms application, I said to my dog, “I’ll never go back, and you can’t make me.” The improvements were so significant, and the increase in productivity so unmistakable, that there was no doubt in my mind that Windows Forms would totally replace C++/MFC in my development of Windows applications.

# About This Book

This book will teach you all you need to know to use Windows Forms effectively. We assume you have some background with either C# or Visual Basic .NET (VB.NET), or sufficient programming experience to pick up what you need to know from the examples shown.

Windows Forms is not difficult. All of its concepts are straightforward, and the Visual Studio .NET environment makes building powerful applications much simpler than writing code by hand. The only difficulty of Windows Forms is that many pieces must be woven together to build a robust, scalable, and efficient application.

You will find two authors' names on this book. Each chapter was written initially by one or the other author, but all chapters were then edited by both authors. Jesse Liberty then extensively edited and rewrote every chapter to give the book a more unified voice. The chapters were subsequently edited by the O'Reilly editors and then again by the authors. The bottom line is that although two authors wrote this book, it should read as if it were written by a single author.

## How the Book Is Organized

Chapter 1, *Windows Forms and the .NET Framework*, is an introduction to Windows Forms and the .NET Framework, and is compatible with .NET 1.1 and Visual Studio 2003.

Chapter 2, *Getting Started*, covers system requirements and walks you through the creation of several simple “Hello World” applications, using both a text editor and Visual Studio .NET.

Chapter 3, *Visual Studio .NET*, gives a thorough review of the Integrated Development Environment (IDE) that is provided by Microsoft for developing .NET applications.

Chapter 4, *Events*, covers the use of events in .NET Forms applications, and includes extensive examples involving keyboard events and text box validation.

Chapter 5, *Windows Forms*, covers topics common to all .NET Forms applications, including the Form class and the Control class, as well as a discussion of forms inheritance and user interface design.

Chapter 6, *Dialog Boxes*, describes the different types of dialog boxes, including those you can create from scratch and those provided as part of the CommonDialog classes.

Chapter 7, *Controls: The Base Class*, covers the features common to all controls in .NET Forms, including such things as parent/child relationships, ambient properties, size and location, anchoring and docking, and keyboard interaction. It also describes image lists.

Chapter 8, *Mouse Interaction*, covers the use of the mouse with .NET Windows applications, including mouse events and properties.

Chapter 9, *Text and Fonts*, discusses the use of the written word as part of Windows applications, including the Font class and techniques for drawing and measuring text strings.

Chapter 10, *Drawing and GDI+*, covers the Drawing namespace, which provides support for rendering graphics as part of a .NET application. It also includes a sample project, which creates a wicked cool analog clock on your screen.

Chapter 11, *Labels and Buttons*, begins the detailed coverage of the native controls available to the .NET developer. This chapter covers labels, link labels, buttons, checkboxes, and radio buttons.

Chapter 12, *Text Controls*, continues the discussion of native controls, with descriptions of the editable text controls, including the text box and rich text box.

Chapter 13, *Other Basic Controls* covers the rest of the native basic controls, including containers such as the panel and the group box, tabbed pages, the picture box, scrollbars and trackbars, up-down controls (sometimes known as spinners), and the progress bar.

Chapter 14, *TreeView and ListView*, describes the controls necessary to create hierarchical user interfaces as typified by Windows Explorer. A clone of Windows Explorer is developed as an exercise.

Chapter 15, *List Controls*, describes native controls used for presenting lists, including the listbox, the checked listbox, and the combo box.

Chapter 16, *Date and Time Controls*, starts with the techniques that deal with date and time values in .NET, including the DateTime and TimeSpan structures. It then describes the DateTimePicker and MonthCalendar controls and the Timer component.

Chapter 17, *Custom Controls*, describes how you can create your own controls to use when the native controls don't do what your application needs. These custom controls can extend or combine existing controls or can be built entirely from scratch.

Chapter 18, *Menus and Bars*, describes the provisions for creating menus, toolbars, and status bars in .NET Forms applications.

Chapter 19, *ADO.NET*, covers the .NET database technology and how to use databases in your applications.

Chapter 20, *Updating ADO.NET*, describes how to update the data in your database, including the use of transactions and multiuser updates.

Chapter 21, *Exceptions and Debugging*, describes error handling and debugging in the .NET Framework, including the debugger included as part of Visual Studio .NET.

Chapter 22, *Configuration and Deployment*, describes how to configure and deploy .NET Windows applications. It also includes a description of .NET assemblies.

The *Appendix* lists several tables of data useful to .NET programmers, including the ASCII character set, members of the `KeyCode` enumeration for mapping keyboard keys, and standard and system color names.

## Who This Book Is for

This book was written for programmers and web developers who want to build desktop applications using Microsoft's powerful new .NET platform. Many readers will have experience with the Microsoft Foundation Classes or writing to the Windows API, but they may find that while the Windows Forms applications accomplish the same tasks, the approach is often quite different.

It might be helpful to first read a primer on C# or VB.NET (see Jesse Liberty's *Programming C#* (O'Reilly) or *Programming Visual Basic .NET* (O'Reilly)), but this is not required. Experienced VB, Java, or C++ developers may decide that they can pick up what they need to know about the languages just by working through the exercises in this book.

## Conventions Used in This Book

The following font conventions are used in this book:

*Italic* is used for:

- Pathnames, filenames, and program names.
- Internet addresses, such as domain names and URLs.
- New terms where they are defined.

Constant Width is used for:

- Command lines and options that should be typed verbatim.

*Constant-Width Italic* is used for replaceable items, such as variables or optional elements, within syntax lines or code.

**Constant-Width Bold** is used for emphasis within program code.

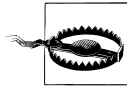
**C#** Indicates C# code.

**VB** Indicates VB.NET code.

Pay special attention to notes set apart from the text with the following icons:



This is a tip. It contains useful supplementary information about the topic at hand.



This is a warning. It helps you solve and avoid annoying problems.

## Version Support

All code in this book was tested both with Version 1.0 and 1.1 of the .NET Framework and Visual Studio .NET.

## Support: A Note From Jesse Liberty

As part of my responsibilities as an author, I provide ongoing support for my books through my web site. You can also obtain the source code for all examples in *Programming .NET Windows Applications* at my site, <http://www.LibertyAssociates.com>.

From my web site, you can access a dedicated book-support discussion forum with a section set aside for questions about *Programming .NET Windows Applications*. Before you post a question, however, please check my web site to see if there is a Frequently Asked Questions list or an errata file. If you check these files and still have a question, then please post to the discussion center.

The most effective way to get help on the discussion forum is to ask a very precise question or to create a very small program that illustrates your area of concern or confusion. You may also want to check the various newsgroups and discussion centers on the Internet. Microsoft offers a wide array of newsgroups, and Development (<http://discuss.develop.com>) has a wonderful .NET email discussion list.

We have tested and verified the information in this book to the best of our ability, but you may find that features have changed (or even that we have made mistakes!). Please let us know about any errors you find, as well as your suggestions for future editions, by posting to my discussion forum.

## We'd Like to Hear from You

If you would like to provide feedback or suggestions to the editors, please write to:

O'Reilly & Associates, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472  
(800) 998-9938 (in the United States or Canada)  
(707) 829-0515 (international/local)  
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

*<http://www.oreilly.com/catalog/pnetwinaps>*

To comment or ask technical questions about this book, send email to:

*[bookquestions@oreilly.com](mailto:bookquestions@oreilly.com)*

You can also send messages electronically. To be put on the mailing list or request a catalog, send email to:

*[info@oreilly.com](mailto:info@oreilly.com)*

For more information about this book and others, as well as additional technical articles and discussion on Windows Forms and the .NET Framework, see the O'Reilly & Associates web site:

*<http://www.oreilly.com>*

and the O'Reilly .NET DevCenter:

*<http://www.oreilynet.com/dotnet>*

## Acknowledgments

From Jesse Liberty:

John Osborn signed me to O'Reilly, and has nurtured my work and created a special niche for my books, for which I will be forever be in his debt. Valerie Quercia continues to be a phenomenal editor who adds tremendous value to my books.

This book would not be nearly as complete were it not for the extraordinary skills of Dan Hurwitz. He literally made this project possible, and I am grateful to him for both his ongoing contributions and his friendship.

Seth Weiss provides perspective and support, and Mike Kralej is like a tiny thruster rocket on the side of a lumbering ship—providing intermittent abrupt nudges in the right direction.

Stacey, Robin, and Rachel offer the love and support that make writing possible and worthwhile.

This book is dedicated to my mom, Edythe Levine, who has set a very high standard for courage and responsibility.

From Dan Hurwitz:

First and foremost I would like to thank my wife Jennifer, for her love, tolerance, and unwavering support. It sounds like a cliché, but without her help, it would not have been possible for me to put in the tremendous amount of work required to write this book. I would also like to thank my father, brothers Marvin and David, and good

friends Joe, Tom, Peter, David, Grover, and Ann, who, along with Jennifer and many others, have helped me get through a very difficult period of my life. Finally, I would like to thank Jesse, my very good friend and coauthor, for providing the opportunity, again, for us to work together.

From both authors:

We would like to thank Ian Griffiths for his extraordinary technical editing of this manuscript, his expertise, and his advice. In addition, we'd like to thank Tatiana Diaz, who stitched together our otherwise disparate pieces into a single coherent work. Ian, Tatiana, Weimeng Lee, and others at O'Reilly, contributed to making this book far better than it otherwise would have been.



# **Windows Forms and the .NET Framework**

.NET is a new development framework that provides a fresh application programming interface to the services and APIs of classic Windows operating systems and brings together several disparate technologies that emerged from Microsoft during the late 1990s. These new technologies include COM+ component services, a commitment to XML and object-oriented design, and a clean interface to the Internet.

To lay the foundation for a full understanding of Windows Forms, this chapter begins with an introduction to the .NET platform and a focus on the .NET Framework.

## **The .NET Framework**

Microsoft .NET supports a Common Type Specification (CTS) that lets you choose the syntax with which you are most comfortable. You can write classes in C# and derive from them in VB.NET. You can throw an exception in VB.NET and catch it in a C# class. Suddenly the choice of language is a personal preference rather than a limiting factor in your application's development.

The .NET Framework sits on top of the operating system, which can be any modern flavor of Windows,\* and consists of multiple components. Currently, the .NET Framework contains:

- An expanding list of official languages (e.g., C#, VB.NET, and JScript .NET)
- The Common Language Runtime (CLR), an object-oriented platform for Windows and web development that all these languages share
- A number of related class libraries, collectively known as the Framework Class Library (FCL).

\* Because of the Common Language Runtime architecture, in theory the operating system can be any OS, including Unix.

Figure 1-1 more fully breaks down the .NET Framework into its system architectural components.

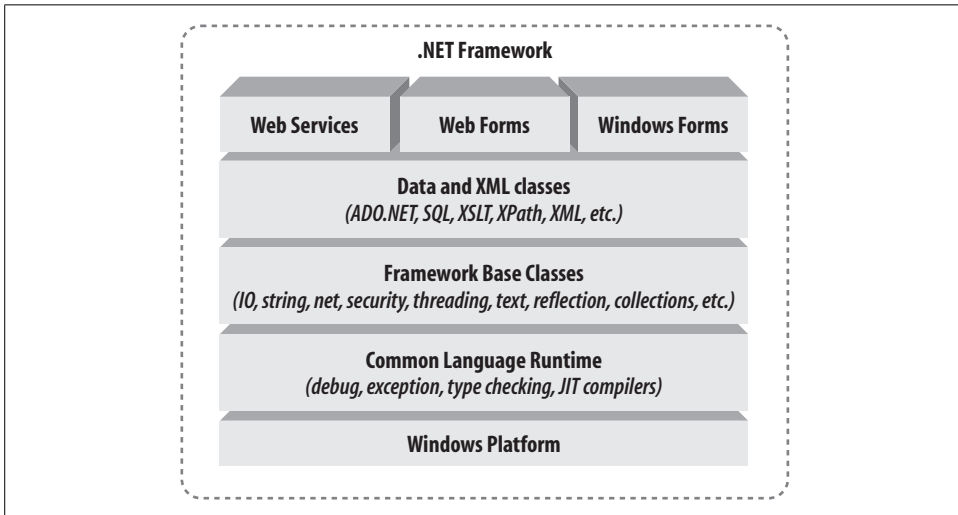


Figure 1-1. .NET Framework architecture

The CLR executes your program; it activates objects, performs security checks on your code, lays your objects out in memory, executes them, and handles garbage collection.

In Figure 1-1, the layer on top of the CLR is a set of framework base classes, followed by an additional layer of data and XML classes, plus another layer of classes intended for applications based on Windows Forms, Web Forms, or web services. Collectively, these classes are known as the Framework Class Library (FCL). With more than 5,000 classes, the FCL facilitates rapid development of applications for either the desktop or the Web.

The set of framework base classes support rudimentary input and output, string manipulation, security management, network communication, thread management, text manipulation, reflection, and collections functionality.

Above the base class level are classes that support data management and XML manipulation. The data classes support persistent management of data that is maintained on backend databases. These classes together are referred to as ADO.NET. Some classes are optimized for Microsoft SQL Server relational database, and some are generic classes that interact with OLE DB-compliant databases. The .NET Framework also supports classes that let you manipulate XML data and perform XML searching and translations. The data handling aspects of the .NET Framework are covered in Chapter 19.

Going beyond the framework base classes and the data and XML classes (and to some extent, building on their technology) are yet another tier of classes geared toward three different technologies:

#### *Windows Forms*

Allows the development of Windows desktop applications with rich and flexible user interfaces. These desktop applications can interact with other computers on the local network or over the Internet through the use of web services.

#### *Web Forms*

Allows the development of robust, scalable web pages and web sites.

#### *Web services*

Allows the development of applications that provide method calls over the Internet.

To learn more about Web Forms and web services, please see *Programming ASP.NET*, Second Edition, by Jesse Liberty and Dan Hurwitz (O'Reilly).

## **Windows Forms**

Windows Forms is the name Microsoft gave to its desktop development technology. Using Windows Forms, it is easier than ever to create applications that are dynamic and data-driven, and that scale well. Used in conjunction with Visual Studio .NET, Windows Forms technology allows you to apply Rapid Application Development (RAD) techniques to building Windows applications. Simply drag and drop controls onto your form, double-click on a control, and write the code to respond to the associated event. In short, the RAD techniques previously available only to VB.NET programmers is now fully realized for all .NET languages.

## **Languages: C# and VB.NET**

You can program Windows Forms in any language that supports the .NET Common Language Specification (CLS). The examples in this book will be given in C# and VB.NET. We believe that C# and VB.NET are very similar, and if you know one you will have no problem with examples shown in the other. That said, we offer the examples in both languages to simplify the process of learning the technology.

## **Visual Studio .NET**

Since all Windows Forms source files are plain text, you can develop all your applications by using your favorite text editor (e.g., Notepad). In fact, many examples in this book are presented just that way. However, Visual Studio .NET offers many advantages and productivity gains. These include the items listed next.

- Visual development of Windows Forms
- Drag-and-drop Windows controls
- IntelliSense and automatic code completion
- Integrated debugging
- Automated build and compile
- Integration with the Visual SourceSafe source control program
- Fully integrated and dynamic help

# Getting Started

The start of any journey is often the hardest part, especially if the goal is unclear or seems daunting. So too with learning a new computer technology. One way to alleviate this difficulty is to present, right up front, a clear idea of what is needed to start the journey and examples that demonstrate the possibilities that lie at the end of the road.

The previous chapter introduced the .NET Framework and overall architecture. In later chapters, you will learn how to create Windows applications using .NET and the Windows Forms technology.

This chapter will cover what software you need on your computer to develop applications using the .NET Framework. Then it will show you what a Windows Forms application looks like. It will do this using the traditional route of a simple program to say “Hello World.” In this case, there will actually be three successive Hello World programs, each showing progressively more capability. Each of the three programs will be developed twice—once in Notepad and again in Visual Studio .NET—to show the advantages of a good development environment.

## System Requirements

This being the new millennium, you need a lot of horsepower to develop and run any modern Windows application, no less so for .NET. Fortunately, memory and disk space are modestly priced commodities these days.

Microsoft officially recommends a 600 MHz Pentium III–class processor or better for developing .NET applications, and RAM ranging from 96 to 256 MB, depending on the operating system. The application will run fine, if slowly, on a 300 MHz machine with 512 MB of RAM. However, as with money and brains, you can never have too much memory, and we recommend the biggest, fastest machine you can afford with at least 512 MB of RAM if at all possible.



Visual Studio .NET is a program that benefits from a lot of screen real estate, so a large, high-resolution monitor makes the development experience much more productive. You should consider a screen resolution of  $1024 \times 768$  to be the minimum. Both authors of this book use high-speed Pentium machines with 512 MB of RAM and two large monitors running at  $1280 \times 1024$ , powered by an Appian (<http://www.apian.com>) dual-headed graphics adapter.

To develop .NET applications, the minimum you will need to install is a supported version of Windows (NT 4 Workstation or Server, 2000 Professional or Server, XP Professional, or .NET Server) and the .NET Software Development Kit (SDK) (downloadable from Microsoft). This software will provide all necessary documentation, compilers and tools, the .NET Framework, and the CLR. You will have to write all your code in a text editor, such as Notepad, or a third party tool.

To be most productive with .NET, we recommend you purchase Visual Studio .NET. Visual Studio .NET includes the SDK and documentation, along with an integrated editor, debugger and other useful tools. Some examples in this book will be developed using only a text editor, but most will be developed in Visual Studio .NET. You can save money by buying the C#- or VB.NET-only version

To run an application developed by .NET on a client machine, i.e., a machine without an installed development environment, the .NET Framework Redistributable Package must be downloaded from Microsoft and installed on each client machine. This is possible on all the versions of Windows, mentioned earlier, plus Windows 98 and Windows Me. Deployment is covered in Chapter 22.

If you plan on doing any development that uses the Internet, such as ASP.NET projects, Internet deployment of Windows desktop applications, or the creation or consumption of web services, use an Internet connection for your final testing. For all these activities except the consumption of web services, you also need to install Internet Information Services (IIS) on your development machine. After IIS is installed, you will need to reinstall your .NET product. Bummer, eh? The best solution is to install IIS *first*, and *then* the .NET product.



Actually, it is possible to configure IIS after installing .NET by running the `aspnet_regiis.exe` command-line utility. From a command prompt enter `aspnet_regiis -i`.

This utility can also enable different web applications to run with different versions of the CLR on the same machine.

IIS is not installed by default with any of these operating systems but can be added easily after the OS is installed, if necessary. To add IIS to Windows 2000 or XP, go to the Control Panel, choose Add/Remove Programs, and then Add/Remove Windows Components. Select and install IIS. You will probably need to provide a Windows installation CD as part of the process. To add IIS to NT, install the Windows NT4

Option pack, downloadable from Microsoft over the Internet, and install Internet Information Server 4.0. Don't forget to reinstall any .NET development products after installing IIS.



If you are installing IIS on a system using either the FAT16 or FAT32 filesystems, then manually configure the FrontPage 2000 Server Extensions. To do this, go to Control Panel, then Administrative Tools, and then Computer Management. Open the Computer Management dialog box and drill down to Internet Information Services (IIS). Right-click on Default web site or web sites (depending on the operating system), and select Configure Server Extensions. Follow the wizard. If the Configure Server Extensions menu item is missing, then the server extensions are already installed.

If you are planning any development that uses database access, you need to install a database. ADO.NET, the database-enabling technology within the .NET Framework, works with any OLE DB-compliant database, although it works best (of course) with Microsoft SQL Server. If you don't have Microsoft SQL Server, Microsoft Access, or another ODBC compliant database installed on your development machine, install the Microsoft SQL Server Desktop Engine (MSDE). This can either be done directly when the .NET product is installed, or the MSDE installation files can be copied to the machine as part of the .NET setup, and then the MSDE installed later.



Some examples in this book assume that you have installed either SQL Server or MSDE.

## Hello World

A long-standing tradition among programmers is to begin study of any new language by writing a program that prints “Hello World” to the screen. In deference to tradition, the first windows applications you create will do just that.

In this section, you will create three progressively more interesting versions of the venerable Hello World program. These versions will demonstrate some of the fundamental features of a Windows application. The first version will be a console application that writes a line of text to the system console (also known as a Command Prompt Window. Some old-timers still call it a DOS box, which is technically no longer accurate.). The next version will be a true Windows application, even if it is somewhat limited. The final version will add a button to demonstrate event handling. (Chapter 4 will cover events in detail.)

## Using a Text Editor

The tool you are most likely to use when developing Windows applications is Visual Studio .NET. You may use any editor you like, however. All source code and

configuration files for all .NET applications (Windows and web) are flat ASCII text files—easily created, read, and modified using any text editor, ranging from Notepad or WordPad (included with Windows) to powerful third-party code editors and development environments.



Both Visual Studio .NET and the C# command-line compilers support different language encodings. In Visual Studio .NET, encoding is accessed under File → Advanced Save Options. The C# command-line compiler has a /codepage option to specify the codepage. The VB.NET command-line compiler does not support alternative encodings. In any case, the default code page is UTF8, which is a superset of flat ASCII.

Using Visual Studio .NET has several advantages. The code editor provides indentation and color coding of your source code, the IntelliSense feature helps you choose and enter the right commands and attributes, and the integrated debugger helps you find and fix errors in your code.

The disadvantage of using Visual Studio .NET, however, is that it automatically generates copious amounts of boilerplate code and default object names. As a beginner, you may be better off doing more of the work yourself, giving up the support of the IDE in exchange for the opportunity to see how things really work.

You enhance the clarity, readability, and maintainability of your program by using your own names for namespaces, classes, methods, and functions, rather than using the default names provided by Visual Studio .NET.

Each of the three versions of Hello World mentioned above will first be developed by using a simple text editor to create the source code. The same three versions will then be created using Visual Studio .NET.

All code examples will be presented in both VB.NET and C#, unless both language versions are nearly identical.

## Hello World as a console application

The first version of the Hello World program created here will be a *console application*. A console application has no user interface (UI) other than a command prompt. It has no windows, buttons, menus, listboxes, or other graphical elements. All it can do is execute program code, accept input, and display text.

For most purposes, the input console is the keyboard and the output console is the command prompt window. The Console class of the .NET Framework encapsulates both the input and output console, and provides properties and methods for communicating with the console. Text written to (or read from) the console can also be directed to or from other devices or files using streams. The Hello World program shown here will just send some characters to the screen.

If a console application EXE file is double clicked in Windows Explorer, the console application will open its own command prompt window, execute, and close the window. For a quickly running program like Hello World, it all happens so fast that you barely see the screen flicker.

To execute a console application and actually see the results, open a command prompt and run the program by typing the executable name from the command line.

## Using the Command Line in .NET

For any of the tools or utilities provided as part of the .NET SDK to run from a command line, the Path property of the operating system environment for that command window must include the correct location of the tool or utility executable. The easiest way to ensure that the Path is set correctly is to not open a normal Command window (Start → Programs → Accessories → Command Prompt), but instead to open a special command prompt window provided as part of .NET. This command window has the Path correctly set to include the locations of all the .NET tools and utilities.

Click on the Start button, and then Programs → Microsoft Visual Studio .NET 2003 → Visual Studio .NET Tools → Visual Studio .NET 2003 Command Prompt. You will probably want to copy this shortcut to someplace more accessible, such as the Windows desktop or a quick launch toolbar.

To execute a console application, either navigate to the directory where the console application lives (using the `cd` command) and then type the name of the program, or enter the full path to the program as part of the name.

The code listings shown in Example 2-1 and Example 2-2 are the Hello World console applications in C# and VB.NET, respectively. These programs use the `WriteLine` method to output a line of text to the system console, which is your computer screen.



This book is not a primer on C#, VB.NET, or the .NET framework. We assume you are familiar with this material, and we will not explain the language fundamentals. For a full exploration of VB.NET, see Jesse Liberty's *Programming Visual Basic .NET*, and for C#, see his book *Programming C#* (both from O'Reilly).

A significant theme of this book is that the choice between C# and VB.NET is purely syntactic: you can express almost any programming idea in either language. Write in whichever language is more comfortable for you. The transition from VB.NET or VBScript to VB.NET may be slightly easier than to C#, but much of the Microsoft and third-party documentation is in C#.

This book shows most examples in both languages, with a slight preference for C# because it is a bit more terse. In any case, you will notice that in most cases, the differences between the languages are small and easily understood.

Open a text editor, such as Notepad, and enter the code shown in Example 2-1 or Example 2-2. Save the file to the name shown in the caption for each code listing.

*Example 2-1. Hello World console application in C# (HelloWorld-console.cs)*

```
C# namespace ProgrammingWinForms
{
    public class HelloWorld
    {
        static void Main()
        {
            System.Console.WriteLine("Hello World");
        }
    }
}
```

*Example 2-2. Hello World console application in VB.NET (HelloWorld-console.vb)*

```
VB namespace ProgrammingWinForms
    public class HelloWorld
        shared sub Main()
            System.Console.WriteLine("Hello World")
        end sub
    end class
end namespace
```



The principal differences between C# and VB.NET are that C# is case sensitive, statements in C# are terminated with a semicolon, and namespaces, classes, and methods in C# are contained within curly braces, whereas VB.NET uses the keyword end.

Although VB.NET is not case sensitive, Visual Studio .NET does impose its own casing rules on VB.NET source code (something that is lacking in C#). Since many of the examples in this book were created outside Visual Studio .NET, the VB.NET code in those examples does not necessarily follow the standard casing. It still compiles fine.

## Compiling the program

To convert your source code to an executable program, it must be compiled. When working outside Visual Studio .NET, this is done using a command-line compiler. The SDK provides compilers for each supported language. This book uses both the C# and the VB.NET compilers.

Open the Visual Studio .NET command prompt, as discussed in the earlier sidebar “Using the Command Line in .NET.” This will ensure that the proper path is set. Navigate to the directory where the source file is saved, using the cd command.

To compile the C# version of the Hello World program (the code shown in Example 2-1), use the following command (assuming you have saved the source file with the name *HelloWorld-console.cs*, as shown in the caption of Example 2-1):

**C#** `csc HelloWorld-console.cs`

To compile the VB.NET version, use the following command (again assuming the source file was saved with the name *HelloWorld-console.vb* as shown in the caption in Example 2-2):

**VB** `vbc HelloWorld-console.vb`

In either case, the source file will be processed by the compiler and an EXE file will be created in the current directory. The name of the EXE file will be the same as the source code, without the extension (*HelloWorld-console*) followed by the extension *.exe*. Thus, *HelloWorld-console.exe*.

You can execute the program by typing its name on the command line. Figure 2-1 shows the results of opening a Visual Studio .NET command prompt window, navigating to the proper directory, compiling, and running Hello World for the console in C#.

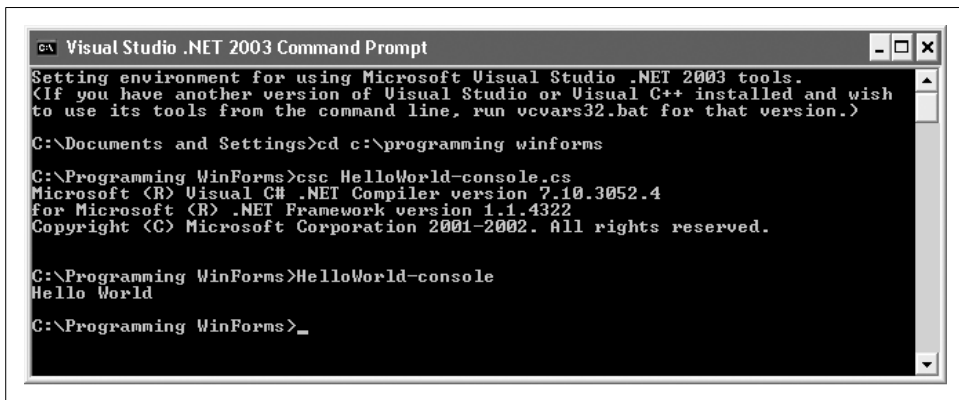


Figure 2-1. Compiling and running HelloWorld-console in C#

This was the simplest kind of compilation. Often, however, you will want the output name to be different from the input name, and for all but the simplest programs, there may be other files that must be referenced as part of the compilation. You control these aspects of command-line compiling with command-line switches. A command-line switch begins with a forward slash. To see all the available options available to the compiler, look in the SDK documentation or enter the appropriate commands:

**C#** `csc /?`

**VB** `vbc /?`

Compile the programs again, this time explicitly specifying the name of the output file and the type of executable (console application). To do so, use the appropriate command line:

```
C# csc /out:csHelloWorld-console.exe /target:exe HelloWorld-console.cs
```

```
VB vbc /out:vbHelloWorld-console.exe /target:exe HelloWorld-console.vb
```

The `/out` parameter specifies the output filename. If no `/out` parameter is specified, the output file will take its name from the source file that contains the Main procedure (in the case of EXE outputs) or the first source file specified (for non-EXE outputs). If there is no path information as part of the `/out` parameter, then the output file will be created in the current directory. You can qualify the filename with a path, either absolute or relative, to specify the output file location.

The `/target` parameter specifies the type of executable that will be created. You may also use `/t` as a shortcut form of `/target`. Table 2-1 lists four legal values for the `/target` parameter.

Table 2-1. Legal values of the `/target` parameter

Value	Short form	Description
<code>/target:exe</code>	<code>/t:exe</code>	Generates a console application with an extension of <code>.exe</code> . This is the default if no target option is specified. A Main procedure is required in at least one source file.
<code>/target:library</code>	<code>/t:library</code>	Generates a dynamic-link library (DLL). No Main procedure is required in any source file. If no <code>/out</code> parameter is specified, the output file will have an extension of <code>.dll</code> .
<code>/target:module</code>	<code>/t:module</code>	Generates a module that can be added to an assembly. If no out parameter is specified, the output file will have an extension of <code>.netmodule</code> . This option is available only via the command line; it is not available from within Visual Studio .NET.
<code>/target:winexe</code>	<code>/t:winexe</code>	Generates an executable Windows program, with an extension of <code>.exe</code> . A Main procedure is required in at least one source file.

One of the most commonly used compiler options (in VB.NET compilations, especially) is `/reference` (the short form is `/r`). This parameter specifies a file that contains an assembly manifest. The manifest exposes the assembly metadata. This allows other parts of the project to learn about and use any types (classes, member variables, methods, etc.) contained in the referenced file(s). If these types have public accessibility, then they will be available to the project being compiled.

Typically, the referenced files are DLLs that contain the .NET Framework class libraries, although you may also reference class libraries developed by yourself or others.



C# does not need the references in the command-line compile because a file called *csc.rsp* contains “default” references for the C# compiler. There is no equivalent in VB.NET, so the references must be included in the command line.

You can reference multiple files either by using multiple `/reference` parameters or by using a single parameter with a comma-separated list of filenames. Be certain not to include any spaces between the filenames if referencing multiple files with a single `/r`. The following two commands are equivalent:

**VB**

```
vbc /out:bin\vbStockTickerCodeBehind.dll /t:library /r:system.dll,system.web.dll,system.web.services.dll,system.data.dll,system.xml.dll StockTickerCodebehind.vb

vbc /out:bin\vbStockTickerCodeBehind.dll /t:library
/r:system.dll /r:system.web.dll /r:system.web.services.dll
/r:system.data.dll /r:system.xml.dll StockTickerCodebehind.vb
```

In these command-line compilations, the VB.NET compiler is executed to compile a source code file named *StockTickerCodebehind.vb*. The output file, *vbStockTickerCodeBehind.dll*, is a library file located in the *bin* subdirectory under the current directory. Five other DLL's are referenced: *system.dll*, *system.web.dll*, *system.web.services.dll*, *system.data.dll*, and *system.xml.dll*.

Sometimes you need to reference an assembly that is not located in either the CLR's system directory or the current directory of the command prompt window. In this case, use the `/lib` (with C#) or `/libpath` (with VB.NET) option to specify a directory to search in. You can search multiple directories by passing in a comma-separated list of directories. The compiler will first search the current directory of the command window, then the CLR system directory, and finally the directories specified in the `/lib` or `/libpath` options.

The `/bugreport` option aids in debugging compile problems. This option opens a text file and causes the compiler to put into it a copy of all the source files used in the compilation (you will probably want to condense and isolate the problem area), all the version information and compiler options, and the compiler output, if any. It will also prompt you for a description of the problem and how you think it should be fixed. These descriptions will accept carriage returns, so you can write a multiline description. The `/bugreport` option takes a fully qualified filename as its value.

The `@` option allows you to specify a file, called a *response file*, which contains compiler options and source code files, just as if they had been entered manually from the command line. Use multiple `@` options to specify multiple response files. Response files can have multiple lines, but each compiler option must be on a single line with no line break. The `#` symbol can be used in response files to comment lines.

When compiling EXE files, the source code must have at least one `Main()` method as an entry point for the program. This entry point must be static in C# or Shared in VB.NET. It can return either void (a sub in VB.NET) or an integer. If there are multiple `Main()` methods in the application, use the `/main` option to specify the *class* that contains the `Main()` method you will use as the entry point.

You can tell the compiler to search for source code files either in the current directory or in a specified directory. To do so, include an optional path name (absolute or relative) or a wildcard as part of the input file. For example, the following command line compiles a Windows application called *HelloWorld.exe*, using all the C# files in the current directory beginning with the characters `HelloWorld`:

```
C# csc /out:HelloWorld.exe /t:winexe HelloWorld*.cs
```

The following command line will search for all similarly named source files in the `c:\projects` directory:

```
C# csc /out:HelloWorld.exe /t:winexe c:\projects\HelloWorld*.cs
```

You can also search for source files in the current or specified directory, plus all of their subdirectories, using the `/recurse` option. For example, the following command line will use all C# source code files in the current directory and all its subdirectories:

```
C# csc /out:HelloWorld.exe /t:winexe /recurse:*.cs
```

This command line will search for all the C# source code files in the *deploy* subdirectory under the current directory, plus all subdirectories under *deploy*:

```
C# csc /out:HelloWorld.exe /t:winexe /recurse:deploy\*.cs
```

## Hello World as a Windows application

The next version of Hello World you create will be a very simple Windows application. Example 2-3 shows the code for this program in C# and Example 2-4 shows it in VB.NET. In both versions, a Windows Form is created with the text on the title-bar set to Hello World.

*Example 2-3. Hello World Windows application in C# (HelloWorld-win.cs)*

```
C# using System.Windows.Forms;

namespace ProgrammingWinForms
{
    public class HelloWorld : System.Windows.Forms.Form
    {
        public HelloWorld()
        {
            Text = "Hello World";
        }

        static void Main()
        {
```

*Example 2-3. Hello World Windows application in C# (HelloWorld-win.cs) (continued)*

```
C#      Application.Run(new HelloWorld());
        }
    }
}
```

*Example 2-4. Hello World Windows application in VB.NET (HelloWorld-win.vb)*

```
VB      imports System.Windows.Forms

        namespace ProgrammingWinForms
        public class HelloWorld : inherits System.Windows.Forms.Form
        public sub New()
            Text = "Hello World"
        end sub

        shared sub Main()
            Application.Run(new HelloWorld())
        end sub
        end class
    end namespace
```

The first line of Example 2-3 and Example 2-4 imports the `System.Windows.Forms` namespace:

```
C#      using System.Windows.Forms;
```

```
VB      imports System.Windows.Forms
```

This example lets you refer to objects in this namespace without the full qualification. When you declare the form, you are then free to refer to the base class as either `System.Windows.Forms.Form` or simply as `Form`.

The third line declares the `Form` class `HelloWorld`:

```
C#      public class HelloWorld : System.Windows.Forms.Form
```

```
VB      public class HelloWorld : inherits System.Windows.Forms.Form
```

The VB.NET version can be written equivalently as:

```
VB      public class HelloWorld
        inherits System.Windows.Forms.Form
```

Notice that the latter version is on two lines, and that it has neither a colon nor the VB.NET line-continuation character.

However you mark the derivation, the fact that your new class derives from `Windows.Forms.Form` makes it a Windows Form application.

The next several lines contain the constructor for the HelloWorld class. In this example, you will set the window caption from within the form's constructor by assigning a string to the form's Text property:

```
C# public HelloWorld()  
{  
    Text = "Hello World";  
}
```

```
VB public sub New()  
    Text = "Hello World"  
end sub
```

Once again, the Main() method is the entry point to the program:

```
C# static void Main()  
{  
    Application.Run(new HelloWorld());  
}
```

```
VB shared sub Main()  
    Application.Run(new HelloWorld())  
end sub
```

The Application class is contained within the System.Windows.Forms namespace. Launch a Windows application by calling the static Run method of the Application class.

As always, the source code must be compiled to create an executable program. The command line for compiling the program is:

```
C# csc /out:csHelloWorld-win.exe /t:winexe HelloWorld-win.cs
```

```
VB vbc /out:vbHelloWorld-win.exe /t:winexe /r:system.dll,system.windows.forms.dll  
HelloWorld-win.vb
```

In the C# compilation, the output file is called *csHelloWorld-win.exe*, and in the VB.NET compilation it is *vbHelloWorld-win.exe*. Both files are located in the current directory. In both cases, the target output type is a Windows application. The VB.NET command line also includes references to several .NET class libraries.

When either output EXE file is executed, the results will look like that shown in Figure 2-2. Notice that the title of the form was set to Hello World. The form has all the functionality one would expect of a rudimentary Windows application: the window can be moved or resized using standard Windows techniques; clicking on the icon in the upper-left corner of the window drops down the standard window menu; and the minimize, maximize, and close window buttons are present and functional in the upper-righthand corner. Not bad for a very small amount of code.

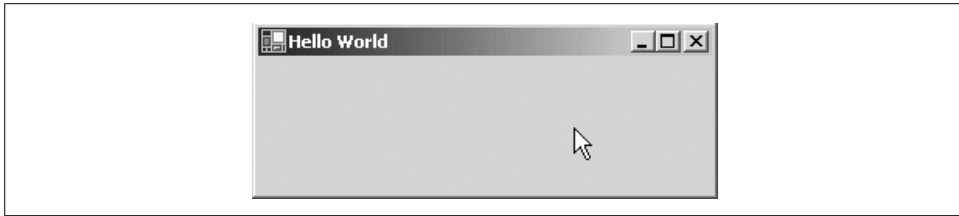


Figure 2-2. Hello World as a Windows application

Notice the correlation, if only by convention, between the namespace referenced in the source code and the files referenced in the compile command. Namespaces are referenced in the source code with using statements in C# and Imports statements in VB.NET. These namespaces are themselves contained within assembly files, most typically DLLs. Table 2-2 shows the correspondence between some of the commonly used namespaces and the assemblies in which they are contained.

Table 2-2. Correspondence of source code and compiler references

Source-code reference	Compiler reference	Comment
-	<i>system.dll</i>	Supplies fundamental classes and base classes. Not necessary in the source code because it is referenced by default.
System.Windows.Forms	<i>system.windows.forms.dll</i>	Contains classes necessary to instantiate form objects.
System.Collections	-	Provides classes and interfaces used by various collections, including Arrays and ArrayLists. Not necessary in the compiler reference because it is included in <i>mscorlib.dll</i> , which is referenced by default.
System.Drawing	<i>system.drawing.dll</i>	Supplies basic drawing capabilities, including Font and Pen classes, and Color, Point, and Rectangle structures.

## Hello World Windows application with a button

The final step in the evolution of this Hello World application will be the addition of a control that generates an event in response to a user action. For this example, you will add a button control that raises the click event when a user clicks on the button. An event handler will handle this click event. Chapter 4 discusses events in detail.

The code in Example 2-5 adds a button control and the click-event handler to the previous example in C#. The additional lines of code are shown in boldface. Example 2-6 shows the equivalent example in VB.NET.

Example 2-5. Hello World Windows application with button control in C#  
(HelloWorld-win-button.cs)

```
C# using System;
using System.Drawing;
using System.Windows.Forms;
```

*Example 2-5. Hello World Windows application with button control in C# (HelloWorld-win-button.cs) (continued)*

**C#**

```
namespace ProgrammingWinForms
{
    public class HelloWorld : System.Windows.Forms.Form
    {
        private Button btn;

        public HelloWorld()
        {
            Text = "Hello World";
            btn = new Button();
            btn.Location = new Point(50,50);
            btn.Text = "Goodbye";
            btn.Click += new System.EventHandler(btn_Click);

            Controls.Add(btn);
        }

        static void Main()
        {
            Application.Run(new HelloWorld());
        }

        private void btn_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
    }
}
```

*Example 2-6. Hello World Windows application with button control in VB.NET (HelloWorld-win-button.vb)*

**VB**

```
imports System
imports System.Drawing
imports System.Windows.Forms

namespace ProgrammingWinForms
    public class HelloWorld : inherits System.Windows.Forms.Form

        Private WithEvents btn as Button

        public sub New()
            Text = "Hello World"
            btn = new Button()
            btn.Location = new Point(50,50)
            btn.Text = "Goodbye"

            Controls.Add(btn)
        end sub
```

*Example 2-6. Hello World Windows application with button control in VB.NET (HelloWorld-win-button.vb) (continued)*

```
VB      public shared sub Main()  
        Application.Run(new HelloWorld())  
      end sub  
  
      private sub btn_Click(ByVal sender as object, _  
        ByVal e as EventArgs) _  
        Handles btn.Click  
        Application.Exit()  
      end sub  
  
    end class  
end namespace
```

The C# code from Example 2-5 is compiled with the following command line:

```
C#      csc /out:HelloWorld-Win-Button.exe /t:winexe HelloWorld-Win-Button.cs
```

The VB.NET code from Example 2-6 is compiled with this command line:

```
VB      vbc /out:vbHelloWorld-win-Button.exe /t:winexe /r:system.dll,system.windows.forms.  
dll,system.drawing.dll HelloWorld-win-Button.vb
```

As above, the VB.NET compiler does not reference any assemblies by default, so they must be explicitly included in the command line.

When the code from Example 2-5 or Example 2-6 is compiled and run, the results look like Figure 2-3.



*Figure 2-3. Hello World with a button control*

In the code that created this application, a private member variable was declared to represent the button:

```
C#      private Button btn;
```

```
VB      Private WithEvents btn as Button
```

The WithEvents keyword in the VB.NET code is required for event handling and will be explained in Chapter 4.

Inside the HelloWorld() constructor, the button variable btn is instantiated as a new instance of the Button class and the Location property is specified as a Point:

```
C# btn = new Button();  
    btn.Location = new Point(50,50);
```

```
VB btn = new Button()  
    btn.Location = new Point(50,50)
```

In the C# version, the event handler for the Click event is added.

```
C# btn.Click += new System.EventHandler(btn_Click);
```

In the VB.NET version, the event handler is hooked up by the combination of the WithEvents keyword in the btn declaration and the Handles keyword in the event-handler method declaration, as you will see momentarily.

Finally in the constructor, the button is added to the Controls collection on the form:

```
C# Controls.Add(btn);
```

```
VB Controls.Add(btn)
```

The btn\_Click method responds to the button Click event:

```
C# private void btn_Click(object sender, EventArgs e)  
{  
    Application.Exit();  
}
```

```
VB private sub btn_Click(ByVal sender as object, _  
    ByVal e as EventArgs) _  
    Handles btn.Click  
    Application.Exit()  
end sub
```



Unlike in VB6, the name of the event handler is insignificant. The Handles keyword determines the events handled by each event-handler method.

Chapter 4 will describe the event handler methods in detail. For now, suffice it to say that when the button is clicked, the Exit() method of the Application class is called, which closes the application.

## Using Visual Studio .NET

Now that you have created the three Hello World programs using a text editor, you will make the same three programs using Visual Studio .NET. This chapter offers a

whirlwind tour of the IDE to show how easy it is to create applications. The next chapter covers Visual Studio .NET in greater detail.

### Hello World as a console application

Open Visual Studio .NET. You will see a Start page with a list of your previous projects, if any, an Open Project button, and a New Project button. Click on the New Project button.

You will be presented with the New Project dialog box. You will see a list of Project Types in the left pane and a list of Templates in the right pane.

In the left pane, click on either Visual Basic Projects or Visual C# projects, depending on which language you wish to use.

In the right pane, click on Console Application.

The name will default to *ConsoleApplication1* and the Location will be the default project directory for your system.



You can change the default Location by clicking Tools → Options. In the tree control on the left, click on Environment → Projects and Solutions. You will see an edit field on the right labeled Visual Studio projects location, along with a Browse button. Either type or browse to the new default directory.

Change the name of the project to *csHelloWorld-Console* or *vbHelloWorld-Console*, depending on which language you are using. The dialog box will look like Figure 2-4.

As indicated by the label below the Location edit field, Visual Studio .NET will create a project in a subdirectory with the same name as the project, located under the default location.

Click OK to create the new project.

Visual Studio .NET will cook for a few moments, and then present a code-editing screen, along with menus and toolbars along the top and information windows along the right edge. If you are using C#, it will look something like Figure 2-5.

If you are using VB.NET, it will look like Figure 2-6.

The next chapter will cover Visual Studio .NET in detail. For now, focus on the code windows.

If you are using C#, notice the commented lines inside the Main() method. Place your mouse cursor at the end of the last commented line and press Enter. This will put the cursor on the next line, properly indented and ready to enter code.

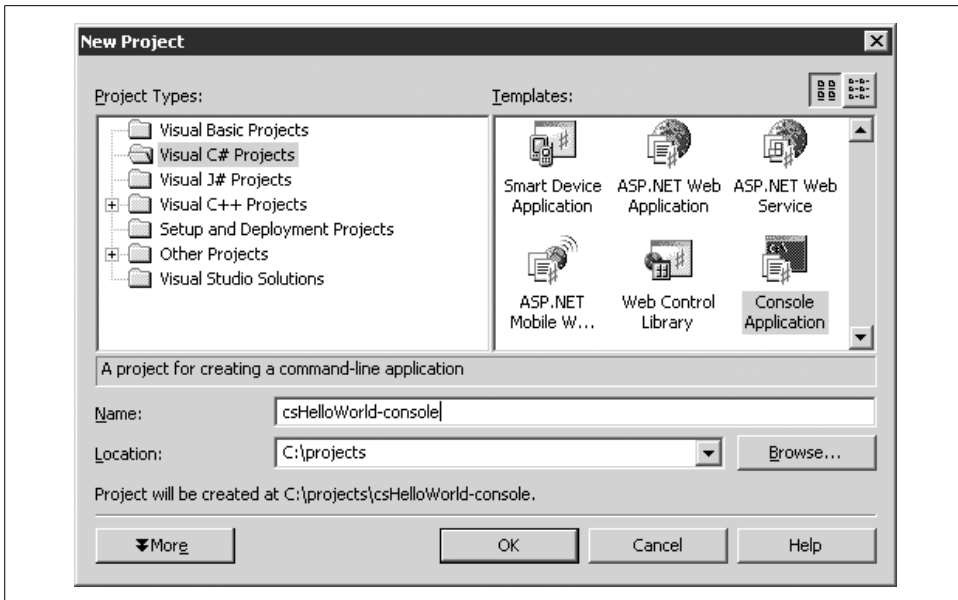


Figure 2-4. New Project dialog box

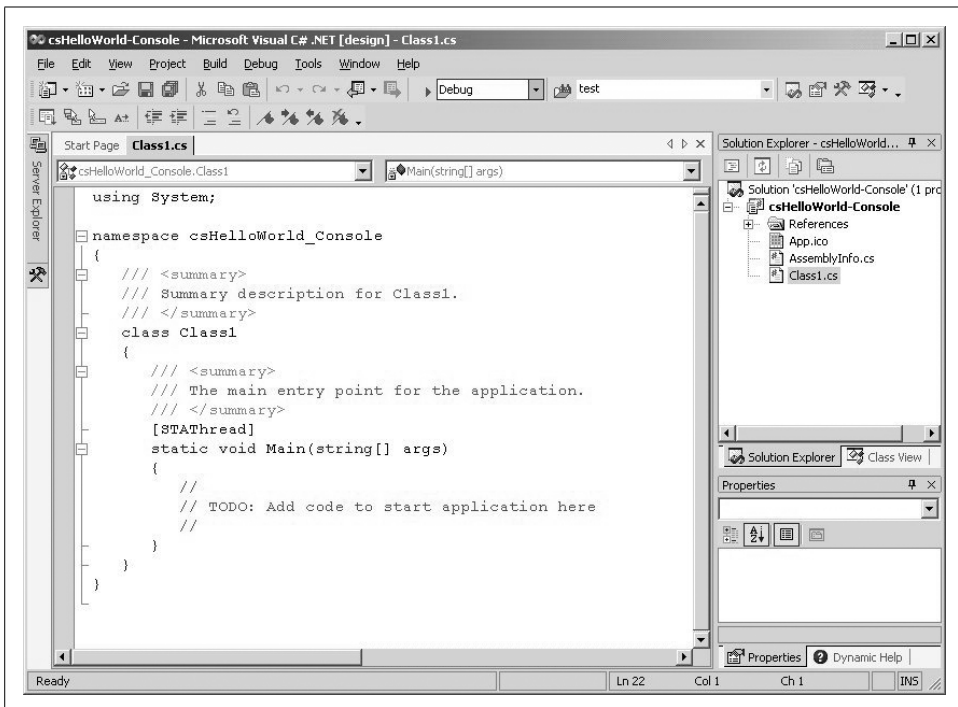


Figure 2-5. C# Console application code-editing screen in Visual Studio .NET

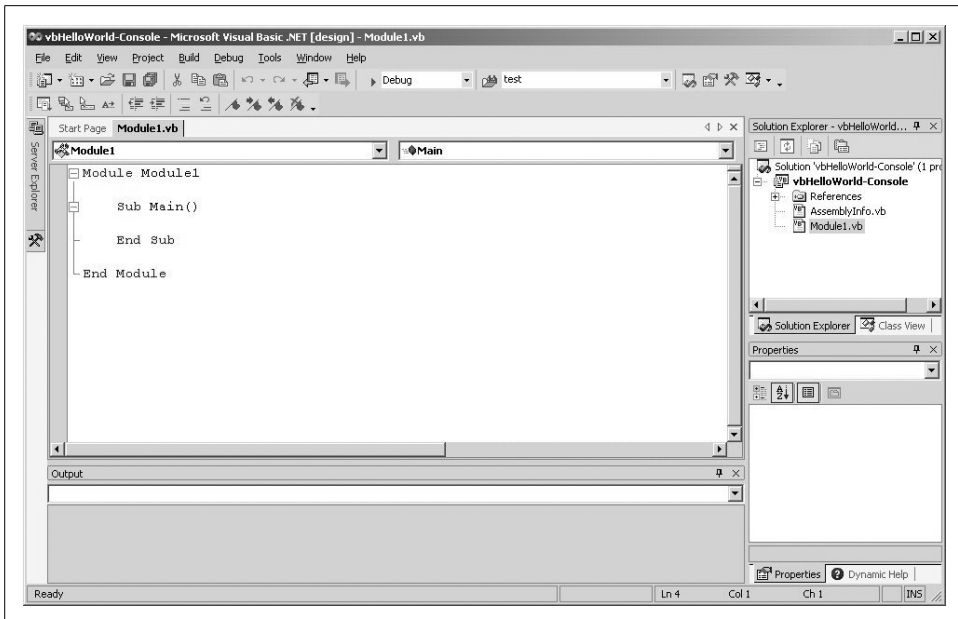


Figure 2-6. VB.NET Console application code-editing screen in Visual Studio .NET

If you are using VB.NET, put your cursor inside the Main() method and tab to get the proper indentation.

Type in the appropriate line of code:

**C#**      `Console.WriteLine("Hello World");`

**VB**      `Console.WriteLine("Hello World")`

As soon as you type the period after the word Console, IntelliSense will display a list of all the possible methods and properties available to the Console class. (Remember that C# is case sensitive.)

You can use the arrow key or the mouse to select one of the methods or properties. Alternatively, just start typing. As you do, the first available selection starting with that character will be highlighted. Successive characters will refine the selection. When the desired method or property is highlighted, press Tab or any other key on the keyboard.

If you press Tab, the selection will be entered in the line of code. If you press any other key, the selection will be entered in the line of code, and that key character will also be entered. When you get to the point of entering arguments for the method, a tool tip will pop up showing all the different valid signatures. The next chapter will explore the IntelliSense feature in more detail.



The behavior of C# and VB.NET differ here. In C#, pressing Enter will insert the selected item without adding a new line, while in VB.NET, Enter will add a new line. Tab works the same in either language, inserting the selection with no additional characters or new lines.

Normally you would press F5 to start a program. However, if you do this for a console application, it will go by too fast to see.

Press Ctrl-F5 to run the program without debugging. A console window, similar to a command prompt window, will appear with the output of your program. It will look something like Figure 2-7.

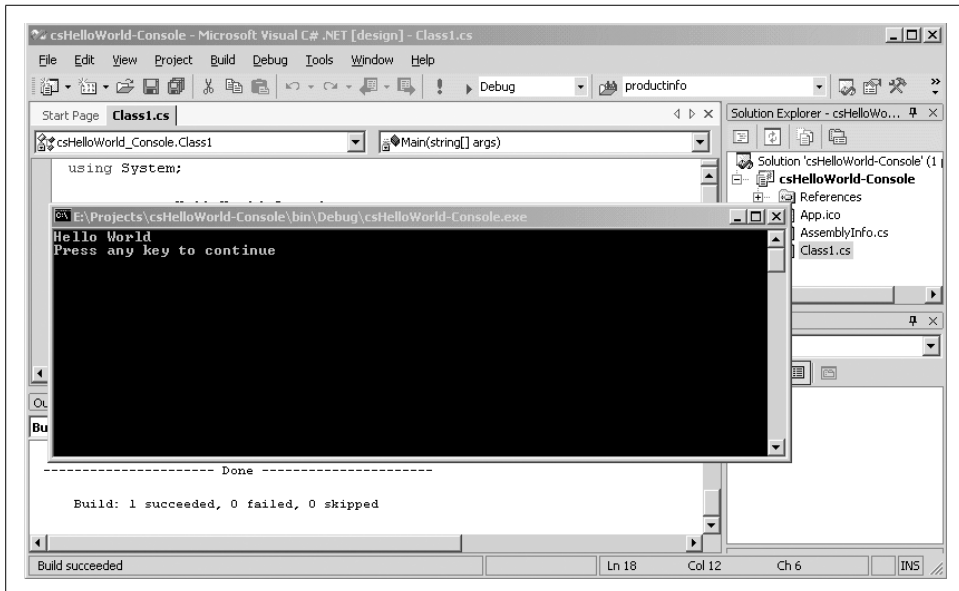


Figure 2-7. Console application output

As you may recall, when you created the console application using a text editor, the relevant line of code had the class System prepended to it, as in:

```
System.Console.WriteLine("Hello World");
```

That is not necessary here because Visual Studio .NET automatically included a reference to the System namespace. In C#, this is immediately apparent from the first line in the code editor:

**C#**      `using System;`

In VB.NET, it is less obvious, but several namespaces are imported by default, rather than with explicit Imports statements. You can see them by right-clicking on the

solution in Solution Explorer and selecting Properties, to display the Property Pages for the project (not to be confused with the Properties window). Under Common Properties, click on Imports to see the namespaces imported by default.

Unlike C#, Visual Studio .NET:

- Automatically provides the boilerplate code to create the skeleton of a program
- Automatically provides default namespace references
- Automatically provides default assembly references
- Provides IntelliSense to minimize typing and coding errors
- Automatically compiles the program when you run the application

## Hello World as a Windows application

As you did with the text-editor versions of Hello World, now create a new version of the Hello World program as a Windows application—this time using Visual Studio .NET.

Open Visual Studio .NET and click on the New Project button on the Start page. In the left side of the New Project dialog box, select either Visual Basic Projects or Visual C# Projects, depending on the language you want to use.

In the right side of the dialog box, select Windows Application. The default name of the project will be *WindowsApplication1*. Change this name to either *csHelloWorld-Win* or *vbHelloWorld-Win*, depending on which language you are using. The New Project dialog should look like Figure 2-8.

The project will be created in a subdirectory with the same name as the project, located under the default location, as indicated by the label under the Location edit field.

After clicking OK on the dialog box, you will be presented with the Visual Studio .NET design page, similar to Figure 2-9.

Figure 2-9 is similar to the console application screen shown in Figure 2-5, except the main design view contains a visual representation of a Windows Form, rather than a code-editing window, and the Properties window along the lower-right side of the screen now shows properties for the *Form1.cs* file, which is currently highlighted in the Solution Explorer.

Click on the form on the design surface. The Properties window will display the properties of the current control, which in this case is the form. Slide down the Properties window until you see the Text property. It currently has the value *Form1*. Change the value to *Hello World*. You will see the titlebar of the form change to say *Hello World*.

To differentiate it even more from the console version, add a label to the form. Click on the View menu item, then Toolbox. The Toolbox will appear on the screen. Click

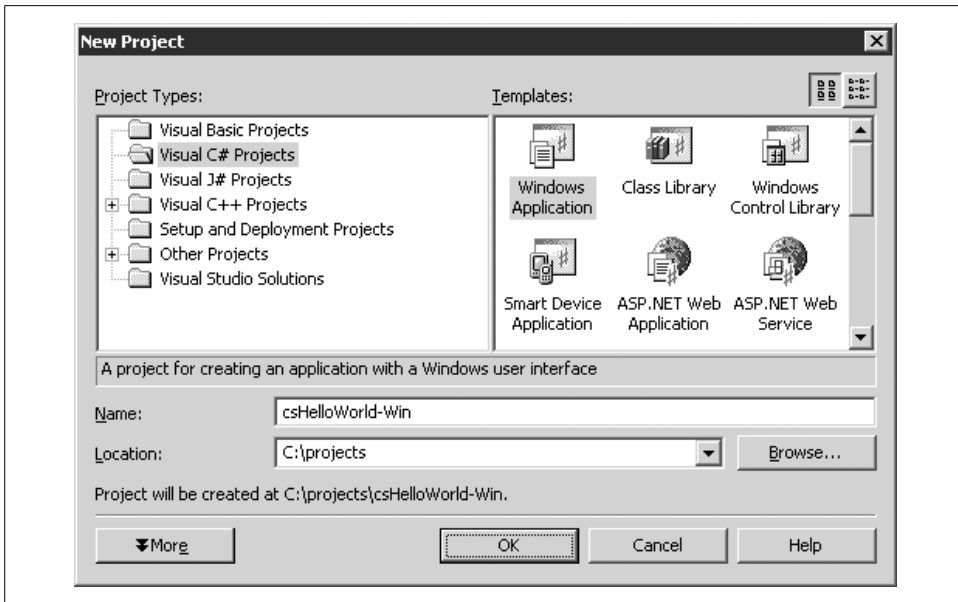


Figure 2-8. New Project Dialog for Hello World Windows application

on the Label control and drag it onto the form. Grab the label control (by clicking on it and dragging) and move it to a suitable location. While the label control is selected, look at the Properties window. It will show the properties for the label. Change the Text property to Visual Studio .NET Version. If necessary, resize the label by clicking on one of the resizing handles and dragging it to enlarge the label until the text no longer wraps. Visual Studio .NET should look something like Figure 2-10.

Run the program by pressing F5 or clicking on the Start icon (▶) on the toolbar. When you do, the window shown in Figure 2-11 will open.

As with the manually coded version, this is a full-fledged Windows application, which can be moved and resized, opens a fully functional window menu once you click the icon in the upper-left corner, and minimize, maximize and close window buttons in the upper-right corner.

### Hello World Windows application with a button

The final step in the evolution of this Hello World program is the addition of a button that can respond to a user action. As with the hand-coded version, the button will raise a click event that the program will handle. However, as you will see, Visual Studio .NET will write most of the code for you.

Open the Toolbox once again. You can open it by either hovering the mouse cursor over Toolbox tab on the left edge of the design surface or clicking on View → Toolbox from the menu.

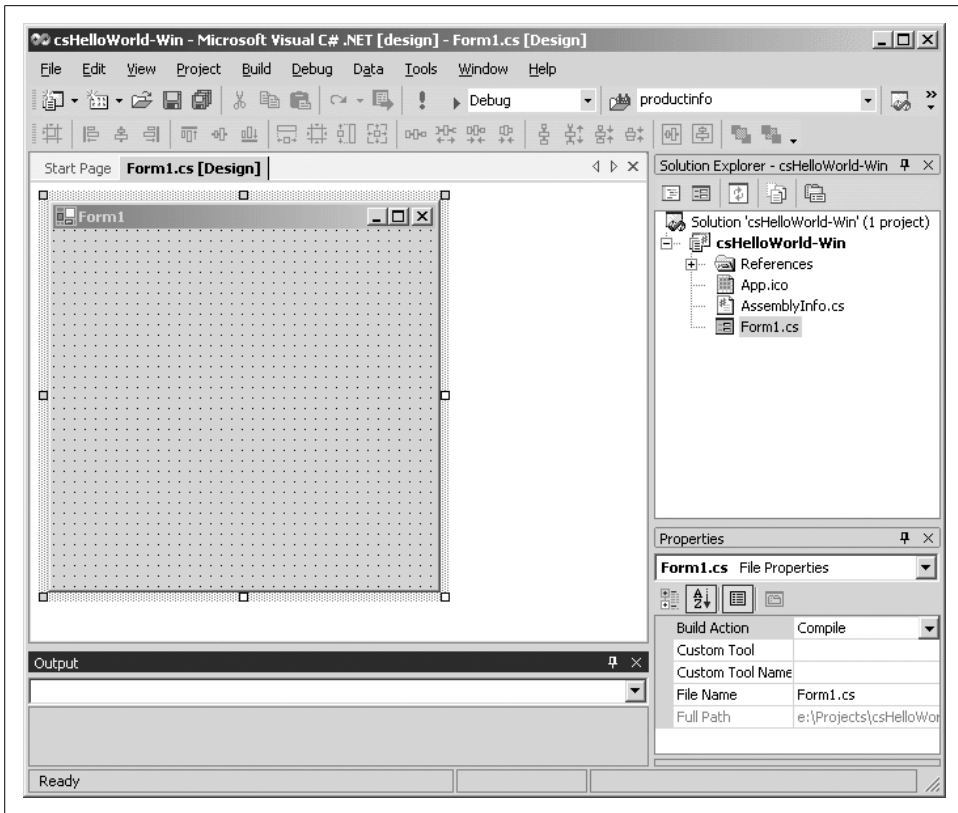


Figure 2-9. Design page for Hello World Windows application



By default, the Toolbox will auto-hide, disappearing from view when the cursor is not over it. It will pop out when the cursor is placed on the Toolbox tab. You can turn this feature off by clicking on the pushpin icon at the top of the Toolbox. The pushpin will be vertical when Auto-Hide is off and sideways when it is on.

Click on the Button control and drag it to a suitable location on the form, or double-click it in the Toolbox to add it to the form and then drag it into position.

While the button is highlighted, go to the Properties window and change the text property to Goodbye. The text written on the button will change accordingly.

Now create and hook up the default event handler by double-clicking the button.

A code window will open up with an event handler method skeleton already created. The cursor will be inside the method, ready to type. Enter the appropriate line of code:

```
C#    Application.Exit();
```

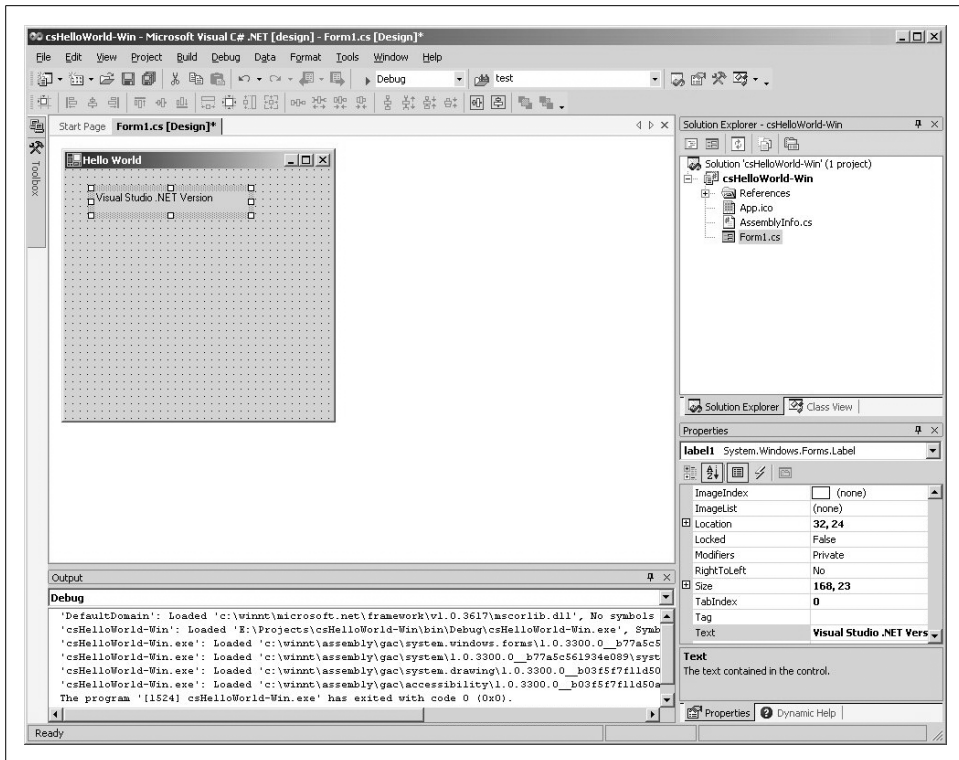


Figure 2-10. Hello World Windows application with label



Figure 2-11. Hello World Windows application

**VB** Application.Exit()

As you saw when entering the code for the Windows version of the console application above, IntelliSense will pop up all the available methods and properties of the Application class as soon as you type the period.

The screen should look like Figure 2-12 if you are using C# or Figure 2-13 if you are using VB.NET.

Run the program by pressing F5 or clicking on the Start icon (▶) on the toolbar. When you do, the window shown in Figure 2-14 will open.

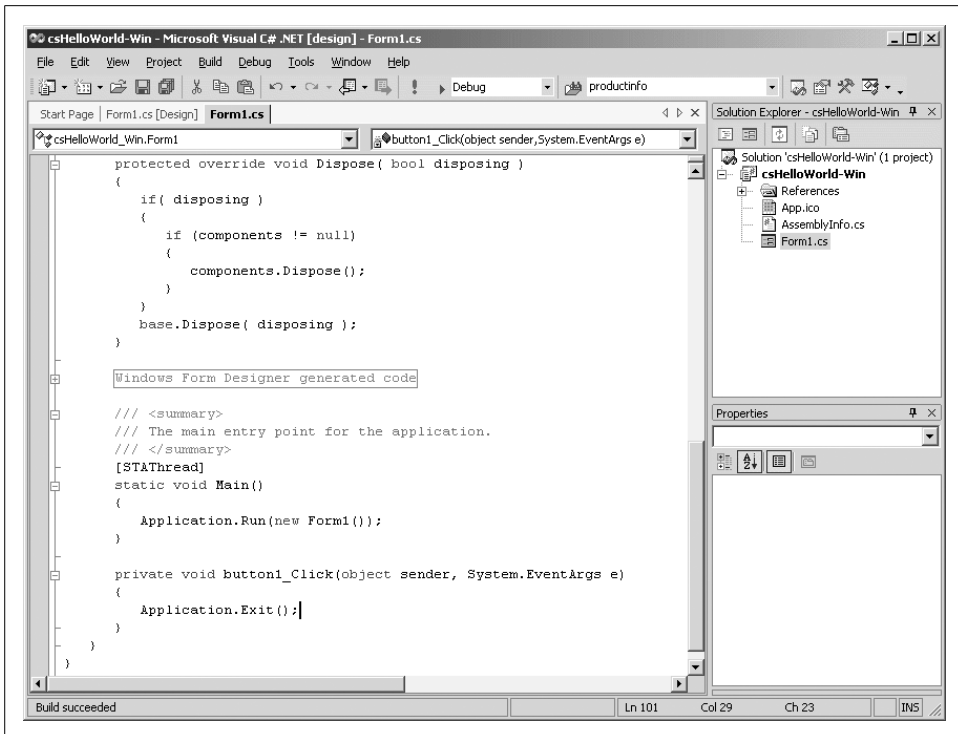


Figure 2-12. Hello World button event handler in C#

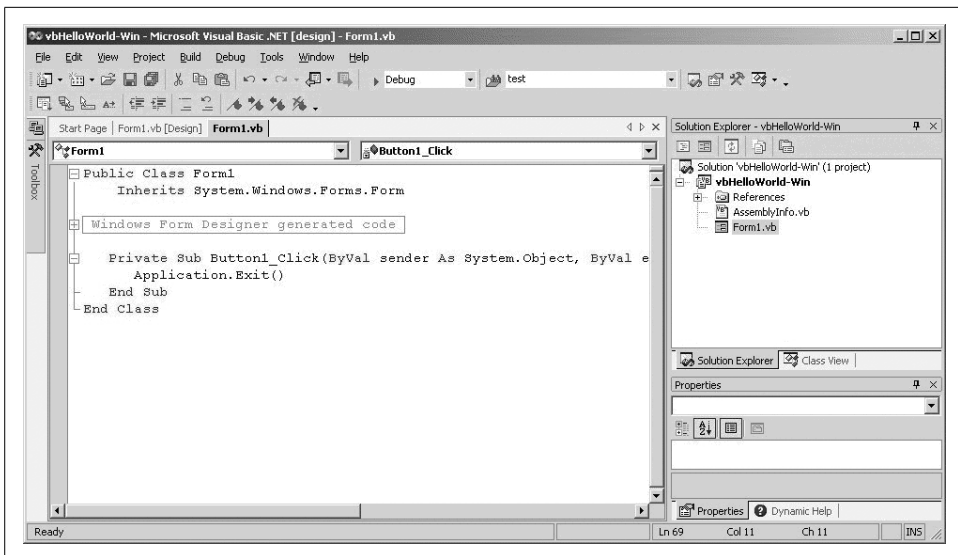
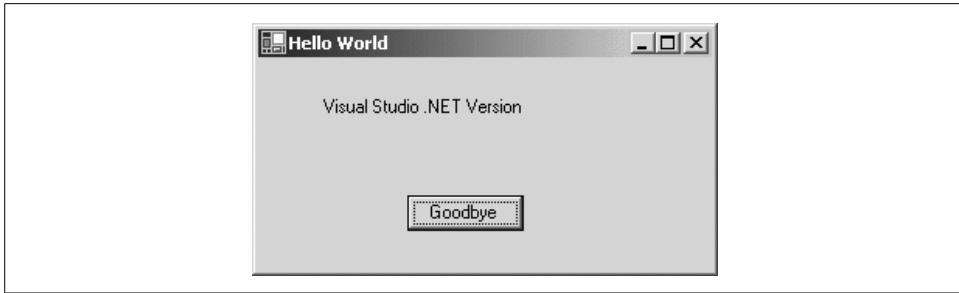


Figure 2-13. Hello World button event handler in VB.NET



*Figure 2-14. Hello World Windows application with button*

Clicking on the Goodbye button will raise the click event, which will be handled by the `Button1_Click` event handler method. Visual Studio .NET automatically provides all the code necessary for creating that event handler and hooking it to the event, greatly easing your programming chores.

---

# Visual Studio .NET

## Overview

If your goal is to produce significant, robust, and elegant applications with few bugs in a minimum amount of time, then a modern integrated development environment (IDE) such as Microsoft Visual Studio .NET is an invaluable tool. Visual Studio .NET offers many advantages to the .NET developer:

- A modern interface using a tabbed document metaphor for code and layout screens, and dockable toolbars and informational windows.
- Convenient access to multiple design and code windows.
- What You See Is What You Get (WYSIWYG) visual design of Windows and Web Forms.
- Code completion that allows you to enter code with fewer errors and less typing.
- IntelliSense pop-up help on every method and function call as you type, providing and types of all parameters and the return type.
- Dynamic, context sensitive help that lets you view topics and samples relevant to the code you are writing at the moment. You can also search the complete SDK library from within the IDE.
- Syntax errors are flagged immediately, allowing you to fix problems as they are entered.
- A Start Page that provides easy access to new and existing projects.
- .NET languages that use the same code editor, shortening the learning curve. Each language can have specialized aspects, but all benefit from shared features such as incremental search, code outlining, collapsing text, line numbering, and color coded keywords.
- An HTML editor that provides Design and HTML views that update each other in real time.
- A Solution Explorer that displays all the files comprising your solution (which is a collection of projects) in a hierarchical, outline.

- A Server Explorer that allows you to log on to servers to which you have network access, access the data and services on those servers, and perform a variety of other chores.
- An integrated Debugger that allows you to step through code, observe program run-time behavior, and set breakpoints, even across multiple languages and processes.
- Customization that allows you to set user preferences for IDE appearance and behavior.
- Integrated build and compile support.
- Integrated support for source control software.
- A built-in task list.

On the negative side, Visual Studio .NET can be a black box and thus inscrutable. It is sometimes difficult to know how Visual Studio .NET accomplishes its legerdemain. While Visual Studio .NET can save you a lot of grunt typing, the automatically generated code can obscure what is really necessary to create good working programs. The proliferation of mysteriously named files across your filesystem can be disconcerting when all you want is a simple housekeeping chore, like renaming a minor part of the project. Worst of all, it occasionally decides to reformat all your carefully constructed code, mashing indents and line breaks like a malevolent typist drunk on too much coffee.

Visual Studio .NET is a large and complex program in its own right, so it is impossible to explore all the possible nooks and crannies in this book. This chapter will lay the foundation for understanding and using Visual Studio .NET and point out traps along the way.



For a thorough coverage of Visual Studio .NET, please see *Mastering Visual Studio .NET*, by Jon Flanders, Ian Griffiths, and Chris Sells (O'Reilly).

## Start Page

The Start Page is what you will see first when you open Visual Studio .NET (unless you configure it otherwise). A typical Start Page is shown in Figure 3-1.

Along the top of the application window is a typical set of menus and buttons. These menus and buttons are context sensitive and will change as the current window changes.

You will see three tabs: Projects, Online Resources, and My Profile. The Projects tab shows the list of existing projects and lets you open a new project. The Online Resources shows a series of links that include:

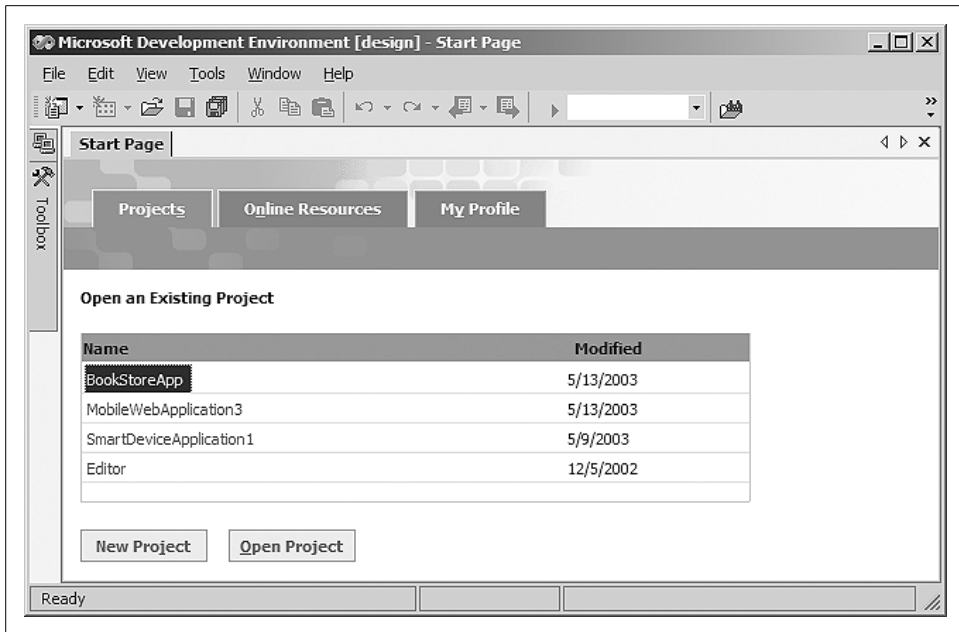


Figure 3-1. Visual Studio .NET Start Page

### *Get Started*

The default selection, provides a means of finding sample code.

### *What's New*

Links to new developments in the .NET world, training and events, and tips.

### *Online Community*

More links to the .NET community, including web sites, newsgroups, tech support resources, code examples, and component sources.

### *Headlines*

Links to news stories about .NET and specific topics such as XML web services.

### *Search Online*

A form for searching the MSDN online library.

### *Downloads*

Links to free and subscriber downloads, including sample applications.

### *XML Web Services*

Forms to search for or register web services.

### *Web Hosting*

Links to hosting providers.

The My Profile tab allows configuration of high-level Visual Studio .NET settings.

# Projects and Solutions

A typical .NET application is comprised of many items: source files, assembly information files, references, icons, and other files and folders. Visual Studio .NET organizes these items into a container called a *project*. One or more projects are contained within a *solution*. When you create a new project, Visual Studio .NET automatically creates the containing solution.

## Solutions

Solutions typically contain one or more project. They may contain other independent items as well. These independent *solution items* are not specific to any particular project, but apply, or *scope*, to the entire solution. The solution items are not an integral part of the application, in that they can be removed without changing the compiled output. You can manage them with source control.

It is also possible to have a solution that does not contain any projects—just solution or miscellaneous files that can be edited using Visual Studio .NET.

Miscellaneous files are independent of the solution or project, but they may be useful. They are not included in a build or compile, but will display in the Solution Explorer (described below) and may be edited there. Typical miscellaneous files include project notes, database schemas, or sample code files.

Solutions are defined within a file named for the solution and have the extension *.sln*. The *.sln* file contains a list of the projects that comprise the solution, the location of any solution-scoped items, and solution-scoped build configurations. Visual Studio .NET also creates a *.suo* file with the same name as the *.sln* file (e.g., *mySolution.sln* and *mySolution.suo*). The *.suo* file contains data used to customize the IDE on a per-user and per-solution basis.

You can open a solution by double-clicking the *.sln* file in Windows Explorer. If the *.sln* file is missing, then recreate that solution from scratch by adding projects into the solution. On the other hand, if the *.suo* file is missing, it will be recreated automatically the next time the solution is opened.

## Projects

A project contains source files and other content. Typically, the build process results in the contents of a project being compiled into an assembly—e.g., an executable file (EXE) or a dynamic link library (DLL).

The data describing the project is contained in a project file named after the project name with a language-specific extension. For VB.NET and C#, the extensions are *.vbproj* and *.csproj*, respectively. The project file contains version information, build settings, references to other assemblies (typically members of the CLR, but also custom developed and third-party components), and source files to include as part of the project.

## Templates

When you create a new project by clicking the New Project button on the Start Page (shown in Figure 3-1), you get the New Project dialog box, shown in Figure 3-2.

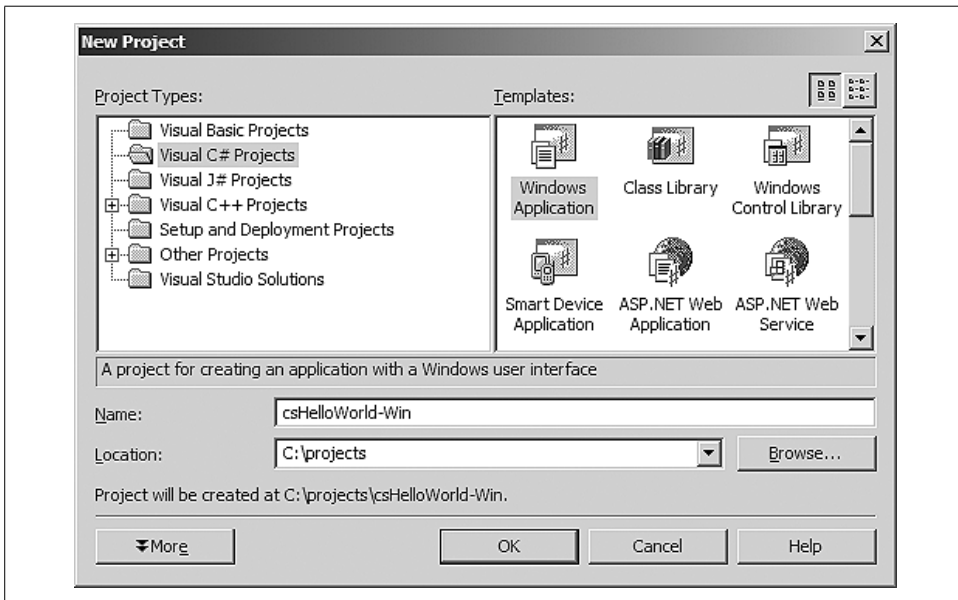


Figure 3-2. New Project dialog box

Select the Project Type and the Template. You will find several templates for each project type. For example, the templates for Visual C# Projects, shown in Figure 3-2, are different from the templates available to Setup and Deployment Projects. By selecting a Visual Studio Solutions project type, you can create an empty solution that is ready to receive whatever items you wish to add.

The template controls what items will be automatically created and included in the project, as well as default project settings. For example, if the project is a C# Windows application, such as the Hello World programs created in Chapter 2, then language-specific *.csproj*, *.csprojusers*, and *.cs* files will be created as part of the project. If the project were a VB.NET project, then the corresponding *.vbproj*, *.vbprojusers*, and *.vb* files would be created instead. If a different template were selected, then an entirely different set of files would be created.

## Project names

Project names may consist of any standard ASCII characters, except for those shown in Table 3-1.

Table 3-1. Forbidden project name characters

Project name	Ascii character
Pound	#
Percent	%
Ampersand	&
Asterisk	*
Vertical bar	
Backslash	\
Colon	:
Double quotation mark	"
Less than	<
Greater than	>
Question mark	?
Forward slash	/
Leading or trailing spaces	
Windows or DOS keywords, such as "nul", "aux", "con", "com1", and "lpt1"	

## The Integrated Development Environment (IDE)

The Visual Studio .NET Integrated Development Environment (IDE) consists of windows for visual design of forms; code-editing windows, menus and toolbars providing access to commands and features; toolboxes containing controls for use on the forms; and windows providing properties and information about forms, controls, projects and the solution.

### Layout

Visual Studio .NET is a Multiple Document Interface (MDI) application. It consists of a single parent window, which contains multiple other windows. All menus, toolbars, design and editing windows, and miscellaneous other windows are associated with the single parent window.

Figure 3-3 shows a typical layout of the IDE. This section will cover the overall layout and many of the features that make working with the IDE so productive.

The Visual Studio .NET window has a titlebar across the top, with menus below. Under the menus are toolbars with buttons that duplicate many common menu commands. Nearly everything that can be done through menus can also be done with context sensitive pop-up menus, as described below. You can customize the menu and toolbars easily by clicking on Tools → Customize.

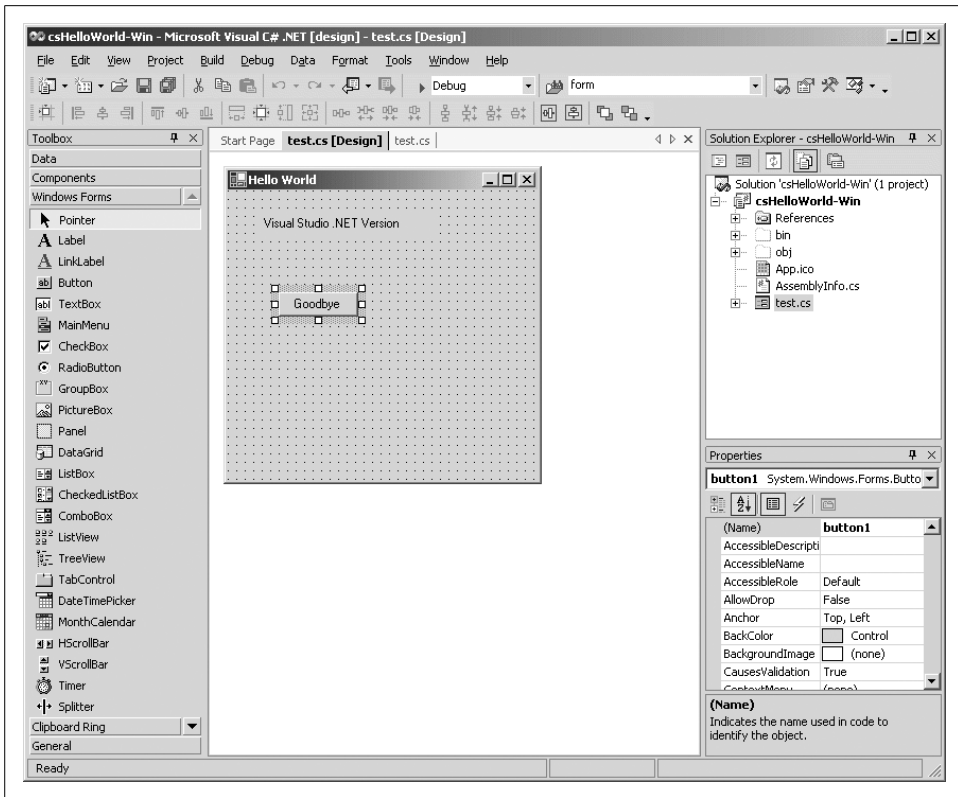


Figure 3-3. Typical IDE layout

The toolbars are docked along the top of the window by default. As with many Windows applications, they can be undocked and moved to other locations, either free-floating or docked along other window edges. Move the toolbars by grabbing them with the mouse and dragging them where you want.

Figure 3-3 shows a design view of a Windows Form, with the design window occupying the main area in the center of the screen. This position allows you to create a visual design by dragging and dropping components from the Toolbox along the left side of the screen.

Along the right side of the screen are two windows, both of which will be covered in more detail below. The upper window is the Solution Explorer. Below it is the Properties window. Many other, similar windows, are available to you, as described later.

All of these windows, plus the Toolbox, are resizable and dockable. You can resize them by placing the mouse cursor over the edge you wish to move. The cursor will change to a double arrow resizing cursor, at which point you can drag the window edge one way or the other.

Right-clicking on the titlebar of a dockable window pops up a menu with four mutually exclusive check items:

#### *Dockable*

The window can be dragged and docked along any side of the Visual Studio .NET window.

#### *Hide*

The window disappears. To see the window again—i.e., to unhide it—use the View main menu item.

#### *Floating*

The window will not dock when dragged against the edge of the Visual Studio .NET window. The floating window can be placed anywhere on the desktop, even outside the Visual Studio .NET window.

You can also double-click on either the titlebar or the tab to dock and undock the window. Double-clicking on the title while docked undocks the entire group. Double-clicking on the tab just undocks the one window, leaving the rest of the group docked.

#### *Auto Hide*

The window will disappear, indicated only by a tab, when the cursor is not over the window. It will reappear when the cursor is over the tab. A pushpin in the upper-right corner of the window will point down when Auto Hide is turned off and point sideways when it is turned on.

In the upper-right corner of the window are two icons:

#### *Pushpin*

This icon toggles the AutoHide property of the window.

When the pushpin points down, the window is pinned in place; AutoHide is turned off. Moving the cursor off the window will not affect its visibility.

When the pushpin points sideways, AutoHide is turned on. Moving the cursor off the window hides the window. To see the window again, click on the tab, which is now visible along the edge where the window had been docked.

#### *X*

The standard close window icon.

The main design window uses a tabbed metaphor—i.e., the tabs along the top edge of that window indicate there are other windows below it. (You can change to an MDI style, if you prefer, in Tools → Options.) Clicking on the tab labeled *test.cs* in Figure 3-3, for example, will bring up the screen shown in Figure 3-4, which contains a code window.

When you switch from a design window to a code window, the menu items, toolbars, and Toolbox change in a context-sensitive manner.

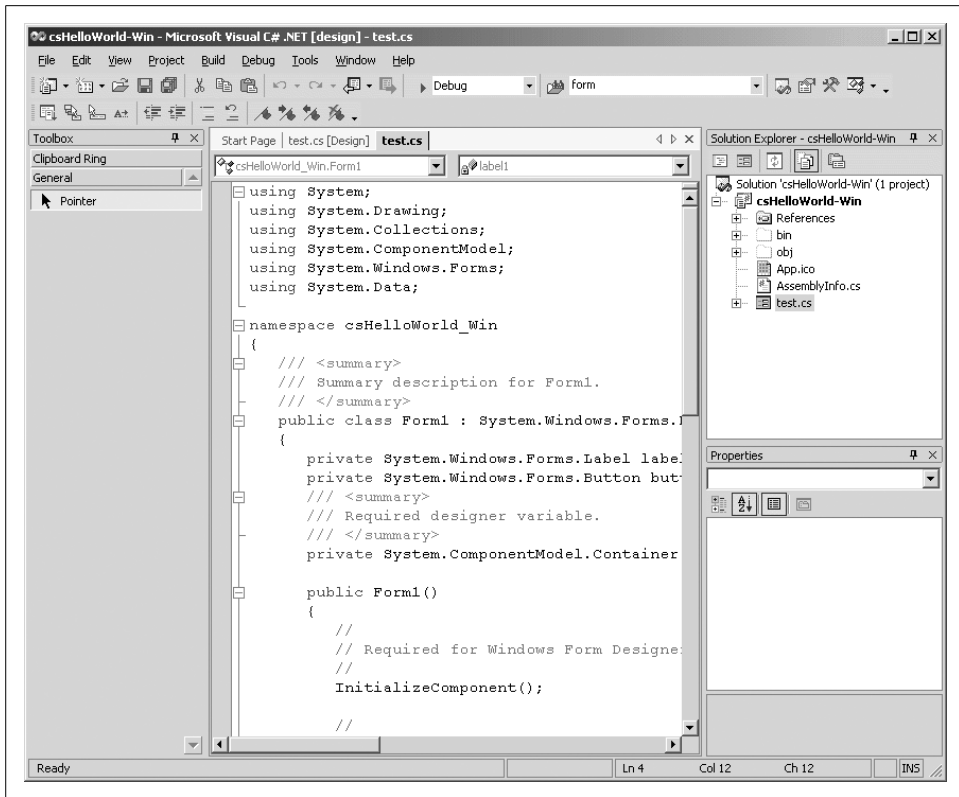


Figure 3-4. Code window in IDE

The code window has drop-down lists at the top of the screen for navigating around the application. The left drop-down contains a list of all the classes in the code and the right drop-down has a list of all objects in the current class. In VB.NET you can also use these drop-downs to select event sources (from the lefthand drop-down) and add event handlers (from the righthand drop-down). This also works in the HTML editor.

Along the bottom edge of the IDE window is a status bar, which shows such information as the current cursor position (when a code window is visible), the status of the Insert key, and any pending shortcut key combinations.

## Menus and Toolbars

The menus provide access to many of Visual Studio .NET's commands and capabilities. The most commonly used menu commands are duplicated with toolbar buttons for ease of use.

The menus and toolbars are context sensitive—i.e., the available selection depends on what part of the IDE is currently selected and what activities are expected or allowed. For example, if the current active window is a code-editing window, the top-level menu commands are:

- File
- Edit
- View
- Project
- Build
- Debug
- Tools
- Window
- Help

If the current window is a design window, then the Data and Format menu commands also become available.

The following sections will describe some of the menu items and their submenus, focusing on the aspects that are interesting and different from common Windows commands.

## File menu

The File menu provides access to a number of file, project, and solution-related commands. Many of these commands are content sensitive. Below are descriptions of those commands that are not self-explanatory.

**New...** As in most Windows applications, the New menu item creates new items to be worked on by the application. In Visual Studio .NET, the New menu item has three submenu items to handle the different possibilities:

### *Project...(Ctrl+Shift+N)*

The Project command brings up the New Project dialog, which is context sensitive. If no project is currently open, as is sometimes the case when Visual Studio .NET is just opened, you will see the dialog box shown in Figure 3-2.

If there is already a project open, then you will get the New Project dialog box shown in Figure 3-5. This dialog box adds radio buttons to give you the choice of adding the new project to the solution or closing the existing solution and creating a new one to hold the new project.

### *File...(Ctrl+ N)*

The File command brings up a New File dialog box, as shown in Figure 3-6. It offers three different categories of files and many different types of files (templates) within each category. Files created this way are located by default in the

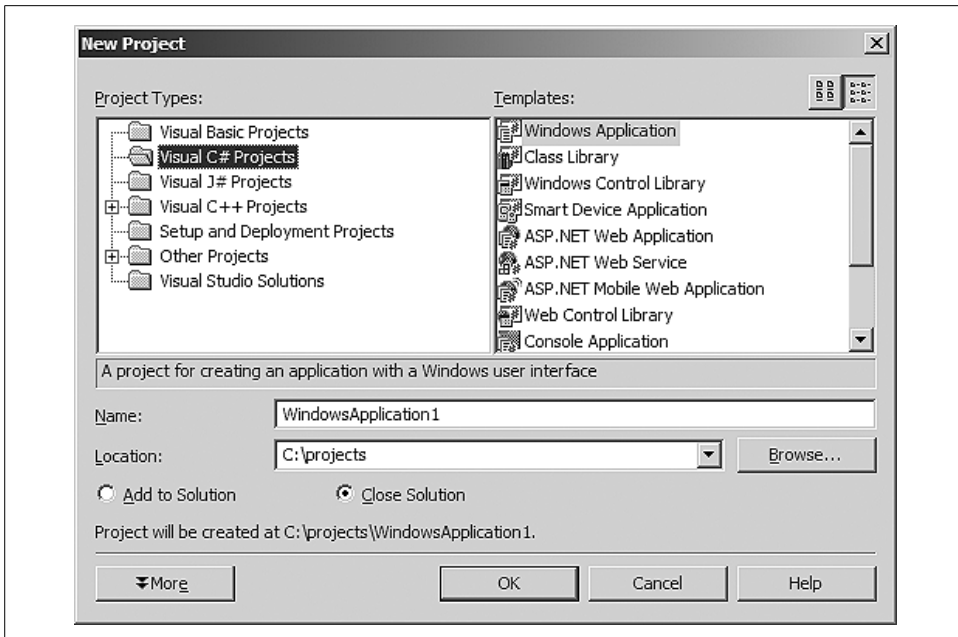


Figure 3-5. New Project dialog box from menu

project directory (although you can browse for a different location). They are displayed in the Solution Explorer if the Show All button is toggled, but are not actually part of the solution unless explicitly added using one of the Add menu items described below. In other words, they are the miscellaneous files described above in the section on Solutions.

#### *Blank Solution...*

The Blank Solution command also brings up a New Project dialog similar to that shown in Figure 3-5, with the Add to Solution radio button grayed out, the default Project Type set to Visual Studio Solutions, and the Template set to Blank Solution. When a blank solution is created, it contains no items. Add items by using one of the Add menu items described below.

The New command has an equivalent button in the Standard Toolbar that exposes the New Project and Blank Solution commands.

**Open...** The Open menu item opens pre-existing items. It has four submenu items:

#### *Project...(Ctrl+Shift+O)*

Opens a previously existing project. The currently opened solution is closed before the new project is opened.

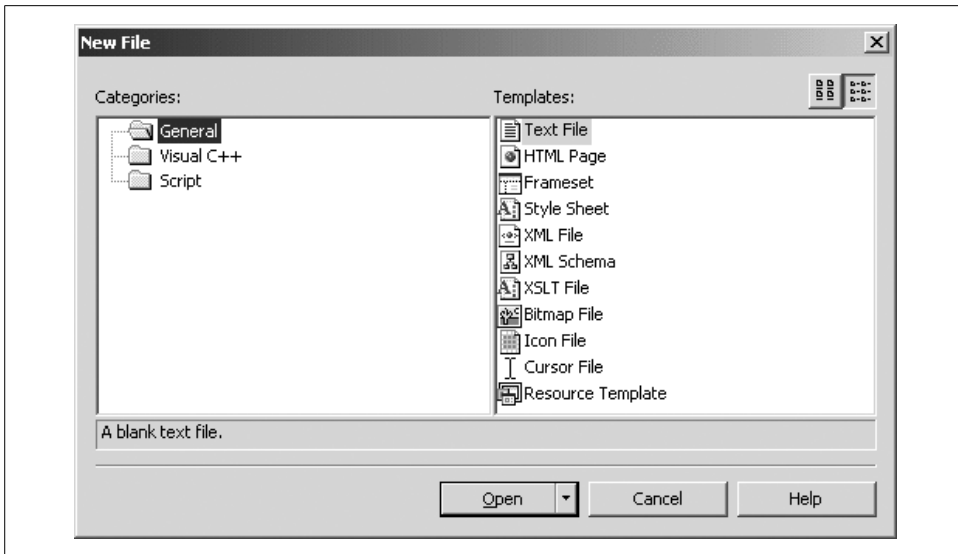


Figure 3-6. New File dialog box

#### *Project From Web...*

An Open Project From Web dialog box is presented, and it accepts a URL pointing to the project to open. As with Open Project, the currently opened solution is closed before the new project is opened.

#### *File...(Ctrl+O)*

Presents a standard Open File dialog box, allowing you to browse to and open any file accessible on your network. Opened files are visible and editable in Visual Studio .NET, but are not part of the project. To make a file part of the project, use one of the Add menu commands described below. The Open File command has an equivalent button on the Standard Toolbar.

#### *File From Web...*

An Open File From Web dialog box is presented and accepts a URL pointing to the file to open. As with Open File, the selected file is not made part of the project.

**Add New Item...(Ctrl+Shift+A).** Add New Item lets you add a new item to the current project. It presents the Add New Item dialog box shown in Figure 3-7. Expanding the nodes in the Categories pane on the left side of the dialog box narrows the list of Templates shown on the right side.

Use this menu item if you want to add new files to your project, including new source code files. For source code, you would typically add a new Class file, which automatically would have the language-specific filename extension.

This command has an equivalent button in the Standard Toolbar. It is also accessible from the context menu in the Solution Explorer.

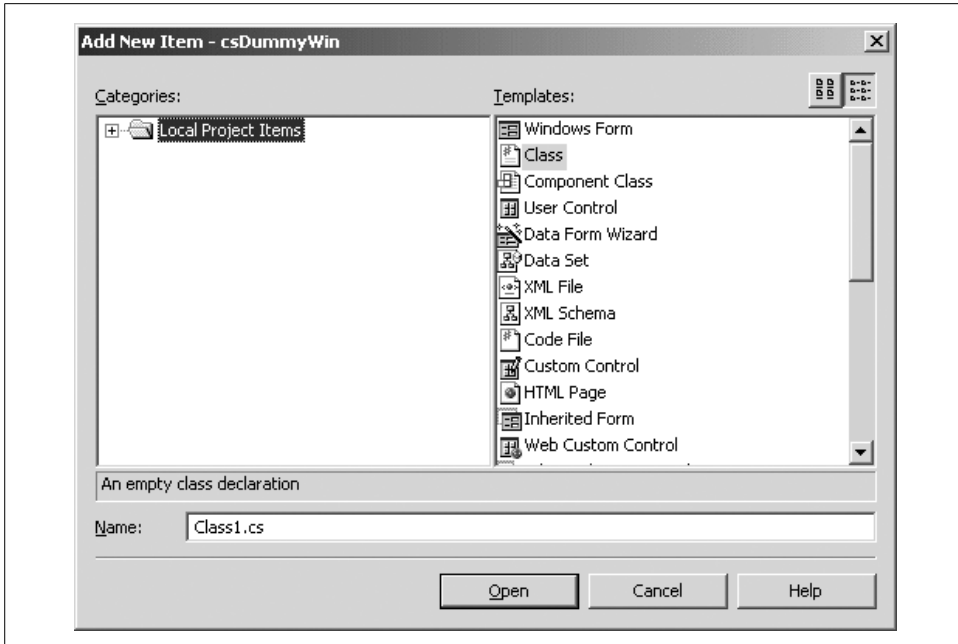


Figure 3-7. Add New Item dialog box

**Add Existing Item...(Shift+Alt+A).** Add Existing Item is very similar to the Add New Item menu item just described, except that it adds already existing items to the current project. If the item added resides outside the project directory, a copy is made and placed in the project directory.

This menu option is also available from the Solution Explorer context menus.

**Add Project.** Add Project has three submenus. The first two, New Project and Existing Project, let you add either a new or pre-existing project to the solution. The third, Existing Project From Web, presents a dialog box that accepts the URL of the project to be added.

**Open Solution.** Clicking on this menu item brings up the Open Solution dialog box, which allows you to browse for the solution to open. The currently open solution will be closed before the new solution is opened.

**Close Solution.** This menu item is only available if there a solution is currently open. If this menu item is selected, the currently open solution will be closed.

**Advanced Save Options...** Advanced Save Options is a context-sensitive submenu that is only visible when editing in a code window. It presents a dialog box that lets you set the encoding option and line ending character(s) for the file.

**Source Control.** The Source Control submenu item allows you to interact with your source control program.

## Edit menu

The Edit menu is fairly standard, containing the typical editing and searching commands. It also has some very interesting capabilities.

**Cycle Clipboard Ring (Ctrl+Shift+V).** The Clipboard Ring is like copy and paste on steroids. Copy different selections to the Windows clipboard, using the Edit → Cut (Ctrl-X) or Edit → Copy (Ctrl-C) commands. Then use Ctrl+Shift+V to cycle through all the selections, allowing you to paste the correct one when it comes around. You can also see the whole clipboard ring in the Toolbox—it's one of the panes that is visible when you're editing a text file.

This submenu item is context sensitive and is visible only when editing a code window.

**Find and Replace → Find in Files (Ctrl+Shift+F).** Find in Files is a very powerful search utility that finds text strings anywhere in a directory or in subdirectories (subfolders). It presents the dialog box shown in Figure 3-8. Checkboxes present several self-explanatory options, including the ability to search using either wildcards or regular expressions.

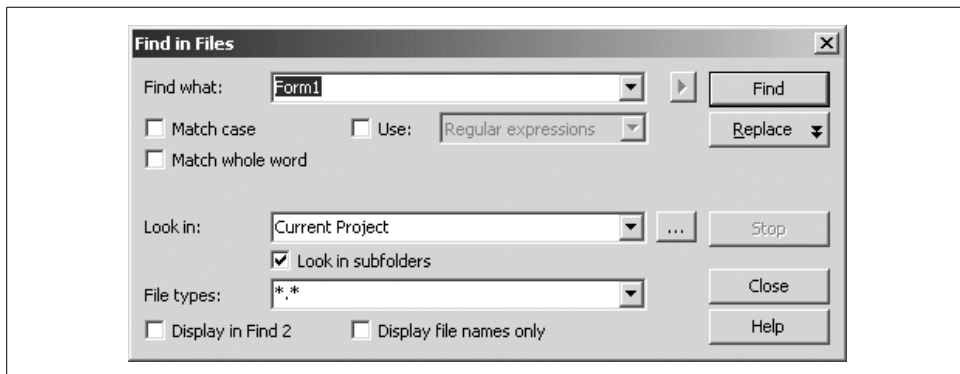


Figure 3-8. Find in Files dialog box

## Regular Expressions

Regular expressions are a language unto themselves, expressly designed for incredibly powerful and sophisticated searches. A full explanation of regular expressions is beyond the scope of this book. For a complete discussion of regular expressions, see the SDK documentation or *Mastering Regular Expressions*, by Jeffrey E. F. Friedl (O'Reilly).

If you click on the Replace button in the Find in Files dialog box, you will get the Replace in Files dialog box shown in Figure 3-9 and described next.

**Find and Replace → Replace in Files (Ctrl+Shift+H).** Replace in Files is identical to the Find in Files command, just described, except that it also allows you to replace the target text string with a replacement text string.

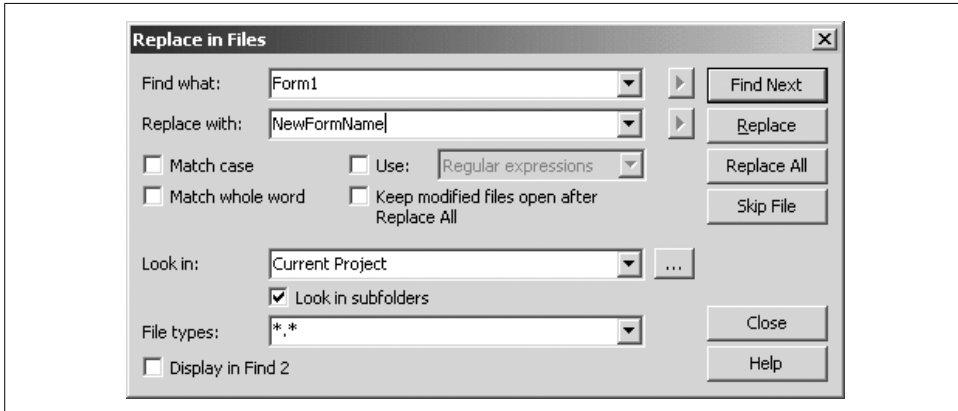


Figure 3-9. Replace in Files dialog box

This command is extremely useful for renaming forms, classes, namespaces, and projects. Renaming objects is a very common requirement, and it is wise—you don't want to be saddled with the default names assigned by Visual Studio .NET.

Renaming should not be difficult, but it can be. Object names are spread throughout a project, often hidden in obscure locations such as solution or project files, and throughout source code files. Although all of these files are text files that can be searched and edited, the task can be tedious and error-prone. The Replace in Files command makes it simple, thorough, and reasonably safe.

**Find and Replace → Find Symbol (Alt+F12).** Clicking on this submenu item brings up the Find Symbol dialog box shown in Figure 3-10. This allows you to search for symbols such as namespaces, classes, and interfaces, and their members such as properties, methods, events, and variables.

The search results will be displayed in a window labeled Find Symbol Results. From there, you can move to each location in the code by double-clicking on each result.

**Go To...** This submenu item brings up the Go To Line dialog box, which allows you to enter a line number and immediately go to that line. It is context sensitive and visible only when editing a text window.

**Insert File As Text...** This submenu item allows you to insert the contents of any file into your source code as though you had typed it in. It is context sensitive and visible only when editing a text window.

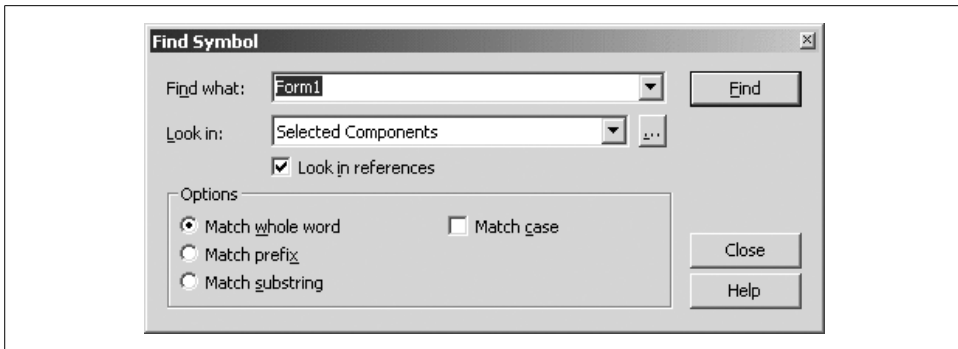


Figure 3-10. Find Symbol dialog box

A standard file browsing dialog box is presented to search for the file that will be inserted. The default file extension will correspond to the project language, but you can search for any file with any extension.

**Advanced.** The Advanced submenu item is context sensitive and visible only when you edit a code window. It has many submenu items, including commands for:

- Creating or removing tabs in a selection (converting spaces to tabs and vice versa)
- Forcing selected text to uppercase or lowercase
- Deleting horizontal whitespace
- Viewing whitespace (make tabs and space characters visible on the screen)
- Toggling word wrap
- Commenting and uncommenting blocks of text
- Increasing and decreasing line indenting
- Incremental search (described below)

**Incremental search (Ctrl+I).** Incremental search lets you search an editing window by entering the search string character by character. As each character is entered, the cursor moves to the first occurrence of matching text.

To use incremental search in a window, select the menu item or press Ctrl+I. The cursor icon will change to a binocular with an arrow indicating the direction of search. Begin typing the text string you want to search for.

The case sensitivity of an incremental search will come from the previous Find, Replace, Find in Files, or Replace in Files search (described above).

The search will proceed downward and left to right from the current location. To search backward, use Ctrl+Shift+I.

The key combinations listed in Table 3-2 apply to incremental searching:


Table 3-2. Incremental searching


Key combination	Description
Esc	Stop the search
Backspace	Remove a character from the search text
Ctrl+Shift+I	Change the direction of the search
Ctrl+I	Move to the next occurrence in the file for the current search text

**Bookmarks.** Bookmarks are useful for marking spots in your code and easily navigating from marked spot to marked spot. Table 3-3 lists five bookmark commands, along with their shortcut key combinations.

This menu item appears only when a code window is the current window.

Table 3-3. Bookmark commands

Command	Key Combination	Description
Toggle Bookmark	Ctrl+K, Ctrl+K	Place or remove a bookmark at the current line. When a bookmark is set, a blue rectangular icon will appear in the column along the left edge of the code window.
Next Bookmark	Ctrl+K, Ctrl+N	Move to the next bookmark.
Previous Bookmark	Ctrl+K, Ctrl+P	Move to the previous bookmark.
Clear Bookmark	Ctrl+K, Ctrl+L	Clear all the bookmarks.
Add Task List Shortcut	Ctrl+K, Ctrl+H	Add an entry to the Task List (described below under the View menu item) for the current line. When a task list entry is set, a curved arrow icon (  ) will appear in the column along the left edge of the code window.

**Outlining.** Visual Studio .NET allows you to *outline*, or collapse and expand sections of your code, to make it easier to view the overall structure. When a section is collapsed, it appears with a plus sign in a box along the left edge of the code window (  ). Clicking on the plus sign expands the region.

You can nest the outlined regions so that one section can contain one or more other collapsed sections. Several commands that facilitate outlining are shown in Table 3-4.

Table 3-4. Outlining commands

Command	Key combination	Description
Hide Selection	Ctrl+M, Ctrl+H	Collapses currently selected text. In C# only, this command is visible only when automatic outlining is turned off or the Stop Outlining command is selected.
Toggle Outlining Expansion	Ctrl+M, Ctrl+M	Reverses the current outlining state of the innermost section in which the cursor lies.
Toggle All Outlining	Ctrl+M, Ctrl+L	Sets all sections to the same outlining state. If some sections are expanded and some collapsed, then all become collapsed.

Table 3-4. Outlining commands (continued)

Command	Key combination	Description
Stop Outlining	Ctrl+M, Ctrl+P	Expands all sections. Removes the outlining symbols from view.
Stop Hiding Current	Ctrl+M, Ctrl+U	Removes outlining information for the currently selected section. In C# only, this command is visible only when automatic outlining is turned off or the Stop Outlining command is selected.
Collapse to Definitions	Ctrl+M, Ctrl+O	Automatically creates sections for each procedure in the code window and collapses them all.
Start Automatic Outlining	N.A.	Restarts automatic outlining after it is stopped.
Collapse Block	N.A.	In C++ only. Similar to Collapse to Definitions, except it applies only to the region of code containing the cursor.
Collapse All In	N.A.	In C++ only. Same as Collapse Block, except it recursively collapses all logical structures in a function in a single step.

The default behavior of Outlining can be set using the Tools → Options menu item. Go to Text Editor, then indicate the specific language for which you wish to set the options. The outlining options can be set for VB.NET under Basic → VB Specific, for C# under C# → Formatting, and for C++ under C/C++ → Formatting.

**IntelliSense.** Microsoft IntelliSense technology makes programmers' lives much easier. It has real-time, context-sensitive help available that appears right under your cursor. Code completion automatically completes your thoughts for you, drastically reducing your need to type. Drop-down-lists provide all methods and properties possible in the current context, and are available at a keystroke or mouseclick.

What's not to love? IntelliSense makes up for a lot of Visual Studio .NET's more, shall we say, exasperating traits.

The default IntelliSense features can be configured by going to Tools → Options, and then the language-specific pages under Text Editor.

Most IntelliSense features appear as you type inside a code window, or allow the mouse to hover over a portion of the code. In addition, the Edit → IntelliSense menu item offers the commands shown in Table 3-5.

Table 3-5. IntelliSense commands

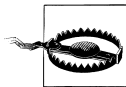
Command	Key combination	Description
List Members	Ctrl+J	Displays a list of all possible members available for the current context. Key-strokes incrementally search the list. Press any key to insert the highlighted selection into your code; that key becomes the next character after the inserted name. Use the Tab key to select without entering any additional characters.  This command can also be accessed by right-clicking and selecting List Member from the context-sensitive menu.

Table 3-5. IntelliSense commands (continued)

Command	Key combination	Description
Parameter Info	Ctrl+Shift+Space	Displays a list of number, names, and types of parameters required for a method, sub, function, or attribute.
Quick Info	Ctrl+K, Ctrl+I	Displays the complete declaration for any identifier, e.g., variable name or class name, in your code. It is also enabled by hovering the mouse cursor over any identifier.
Complete Word	Alt+Right Arrow or Ctrl+Space	Automatically completes the typing of any identifier once you type in enough characters to uniquely identify it. This only works if the identifier is entered in a valid location in the code.

The member list presents itself when you type the dot following any class or member name.

Every member of the class is listed, and each member's type is indicated by an icon. You can find icons for methods, fields, properties, events and so forth. In addition, each icon may have a second icon overlaid to indicate the accessibility of the member: public, private, protected, and so on. If there is no accessibility icon, then the member is public.



If the member list does not appear, you should ensure that you have added all the necessary using (or imports) statements. Also remember that IntelliSense is case-sensitive in C#. Also, sometimes C# needs a rebuild before it will reflect the most recent changes.

Table 3-6 lists all the different icons used in the member lists and other windows throughout the IDE. Table 3-7 lists the accessibility icons.

Table 3-6. Object icons

Icon	Member type
	Class
	Constant
	Delegate
	Enum
	Enum item
	Event
	Exception
	Global
	Interface
	Intrinsic
	Macro
	Map

Table 3-6. Object icons (continued)


















Icon	Member type
	Map item
	Method or function
	Module
	Namespace
	Operator
	Property
	Structure
	Template
	TypeDef
	Union
	Unknown or error
	Variable or field

Table 3-7. Object accessibility icons

Icon	Accessibility
	Shortcut
	Friend
	Internal
	Private
	Protected

## View menu

The View menu is a context-sensitive menu that provides access to the myriad windows available in the Visual Studio .NET IDE. You will probably keep many of these windows open all the time; you will use others rarely, if at all.

The View menu is context sensitive. For example, if your form has no controls on it, the Tab Order submenu will be grayed out.

When the application is running, a number of other windows become visible or available. These windows are accessed via the Debug → Windows menu item, not from the View menu item.

Visual Studio .NET can store several different window layouts. In particular, it remembers a completely different set of open windows during debug sessions than it does during normal editing. These layouts are stored per-user and not per-project or per-solution.

This section covers the areas that may not be self-explanatory.

**Open/Open With...** This menu item lets you open the current item—i.e., the item currently selected in the Solution Explorer (described below)—in the program of your choice. Open uses the default editor, and Open With allows you to pick from a list of programs. You can add other programs to the list.

The Open With command also lets you open an item with the editor of your choice in Visual Studio .NET. For example, you can open a file in the binary viewer when you might normally get the resource viewer. Perhaps most usefully, you can also specify the default editor for an item. For example, you can make a Windows Form open in code view rather than design view by default.

**Solution Explorer (Ctrl+Alt+L).** Projects and solutions are managed using the Solution Explorer, which presents the solution and projects, as well as all the files, folders, and items contained within them, hierarchically and visibly. The Solution Explorer is typically visible in a window along the upper-right side of the Visual Studio .NET screen, although the Solution Explorer window can be closed or undocked and moved to other locations.

To view the Solution Explorer if it is not already visible, select View → Solution Explorer from the Visual Studio .NET menu. Alternatively, press the Ctrl+Alt+L keys simultaneously. Figure 3-11 shows a typical Solution Explorer.

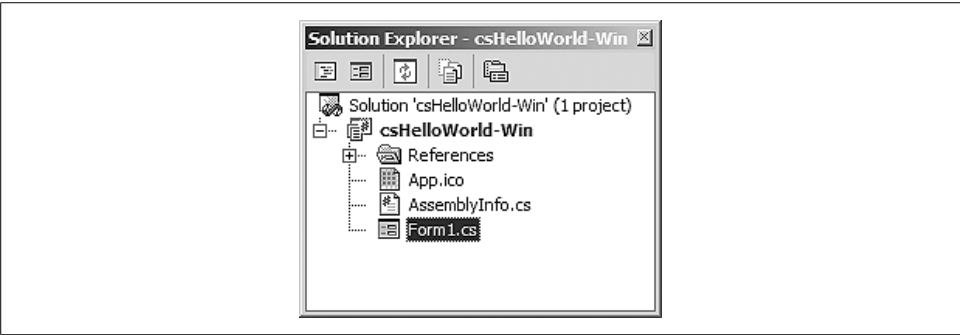


Figure 3-11. Solution Explorer

There are several menu buttons along the top of the Solution Explorer window. These buttons are context sensitive (i.e., they may or may not appear, depending on the currently selected item in the Solution Explorer). Table 3-8 details the purpose of each button.

Table 3-8. Solution Explorer buttons






Button	Name	Shortcut keys	Description
	View Code	F7	Displays code in main window. Only visible for source files.
	View Designer	Shift +F7	Displays visual designer in main window. Only visible for items with visual components.

Table 3-8. Solution Explorer buttons (continued)

Button	Name	Shortcut keys	Description
	Refresh	none	Refreshes the Solution Explorer display.
	Show All Files	none	Toggles display of all files in the Solution Explorer. By default, many files are not shown. If Show All Files is clicked, the solution shown in Figure 3-9 will look like Figure 3-12 after several of the nodes are expanded.
	Properties	Alt+Enter	If the currently highlighted item is a solution or a project, it displays the Properties page for that item. Otherwise, moves the cursor to the Properties window for that item.

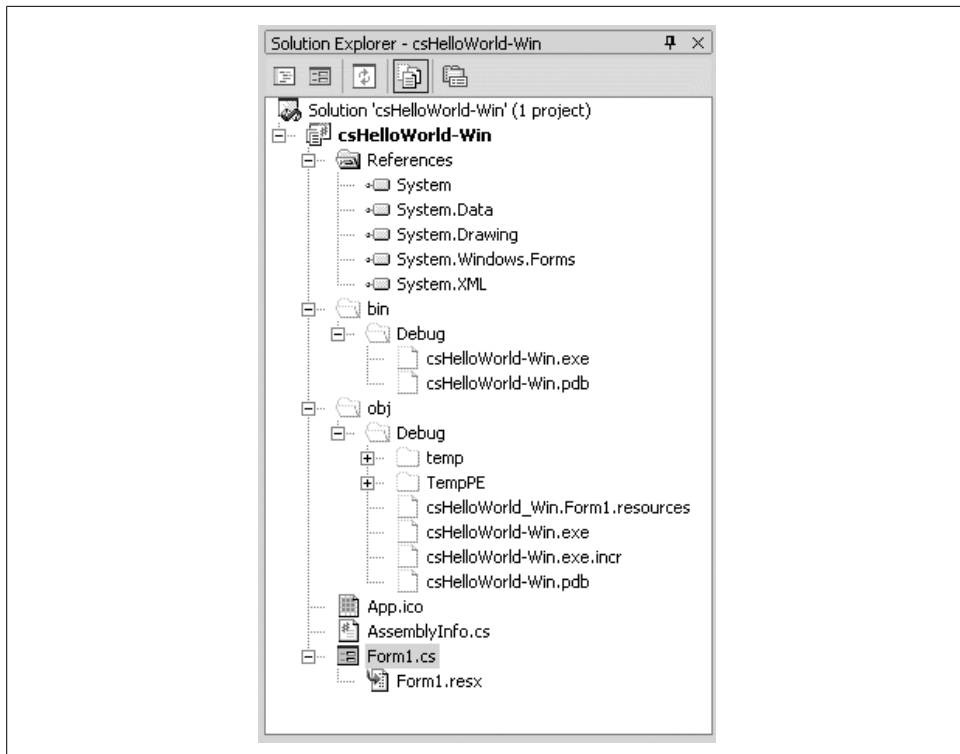


Figure 3-12. Solution Explorer (expanded)

You can also display miscellaneous files in the Solution Explorer. To do so, go to Tools → Options... and then Environment → Documents. Check the checkbox labeled Show Miscellaneous files in Solution Explorer.

Most of the functionality of the Solution Explorer is redundant with the Visual Studio .NET menu items, although it is often easier and more intuitive to perform a given chore in Solution Explorer than in the menus. Right-clicking on any item in the Solution Explorer pops up a context-sensitive menu. Three different pop-up menus

from Solution Explorer are shown in Figure 3-13. From left to right, they are for a solution, a project, and a source-code file.

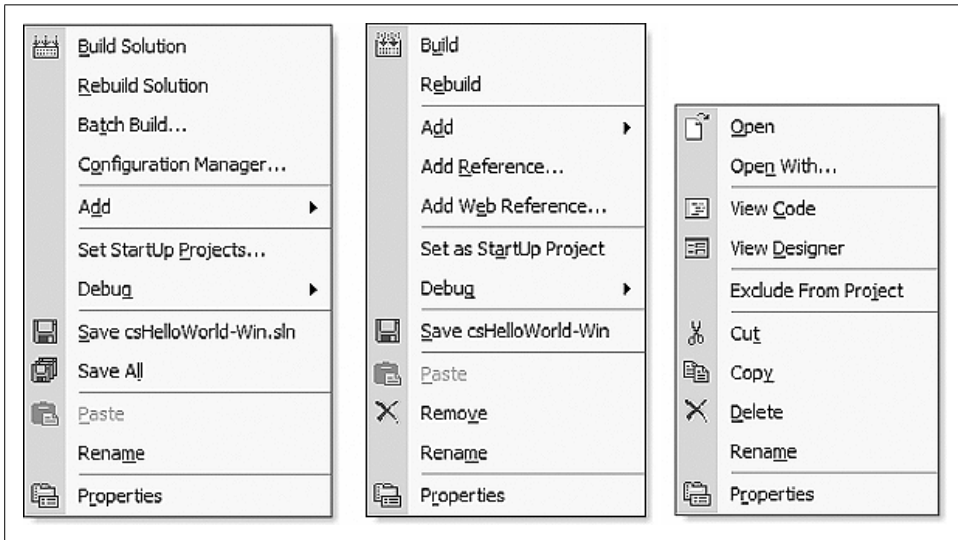


Figure 3-13. Solution Explorer context-sensitive menus

Several points bear mention:

- The Add pop-up menu item for solutions and projects offers submenus that allow new or existing items to be added. This item replicates items contained under the main Project menu.  
Set Startup Projects and Exclude From Project are also replicated under the main Project menu.
- The Build and Rebuild pop-up menu items replicate items contained under the main Build menu.
- The Debug pop-up menu item replicates two items from the main Debug menu.
- If the Properties item is clicked for a source file, the cursor moves to the Properties window. If the Properties item is clicked for a solution or project, the Properties page for that item is opened.

**Properties Windows (F4).** The Properties window displays all the properties for the currently selected item. Some of the properties, such as Font and Location, have sub-properties, indicated by a plus sign next to their entry in the window. The property values on the right side of the window are editable.

One possible source of confusion is that certain items have more than one set of properties. For example, a Form source file can show two different sets of properties,

depending on whether you select the source file in the Solution Explorer or the form as shown in the Design view.

Figure 3-14 shows a typical Properties window with the Font subproperty expanded out.

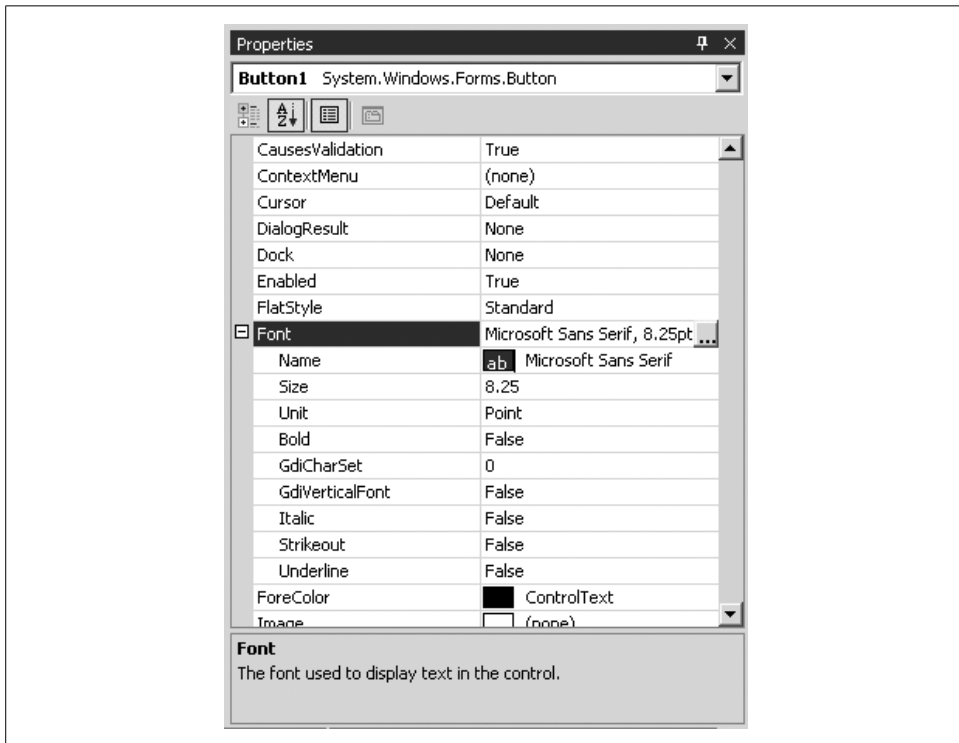


Figure 3-14. Properties window

The name and type of the current object is displayed in the field at the top of the window. In Figure 3-14, it is an object named Button1 of type Button, contained in the System.Windows.Forms namespace.

The Font property has subproperties that may be set either directly in the window or by clicking on the button with three dots on it, which brings up a standard font dialog box. Other properties with subproperties may or may not have a dialog box associated with them, as need be. Other properties, such as the Font Name property, may have drop-downs in the property grid itself.

The property window has several buttons just below the name and type of the object. The first two buttons on the left toggle the list by category or alphabetically. The next button from the left displays properties for an object. The right-most button displays property pages for the object, if there are any.



Some objects have both a Properties window and Properties pages. The Property pages display additional properties from those shown in the Properties window.

If the project is in C#, then an additional lightning bolt button (⚡) is used to create event handlers for an item. Events are covered in Chapter 4.

For some controls, such as TabControl, an additional panel is part of the Properties window with verbs, such as Add Tab and Remove Tab.

The box below the list has a brief description of the selected property.

**Server Explorer (Ctrl+Alt+S).** The Server Explorer allows you to access any server to which you have network access. If you have sufficient permissions, you can log on, access system services, open data connections, access and edit database information, and access message queues and performance counters. You can also drag nodes from the Server Explorer onto Visual Studio .NET projects, creating components that reference the data source.

Figure 3-15 shows a typical Server Explorer. It is a hierarchical view of the available servers. In this figure, only one server is available, ATH13T. The figure shows a drill-down into SQL Server, with the tables in the Northwind database. These tables, and all other objects in this tree view, are directly accessible and editable from the window.

**Class View (Ctrl+Shift+C).** The Class View shows all the classes in the solution hierarchically. A typical Class View, somewhat expanded, is shown in Figure 3-16. The icons used in this window are listed in Table 3-6 and Table 3-7.

As with the Solution Explorer, any item in the class view can be right-clicked, which exposes a pop-up menu with context-sensitive menu items. This provides a convenient way to sort the display of classes in a project or solution, or to add a method, property, or field to a class.

The button on the left above the class list lets you sort the listed classes, either alphabetically, by type, by access, or grouped by type. Clicking on the button itself sorts by the current sort mode, while clicking on the down arrow next to it presents the other sort buttons and changes the sort mode.

The button on the right above the class list allows you to create virtual folders for organizing the listed classes. These folders are saved as part of the solution in the .suo file.

These folders are virtual (i.e., they are illusory). They are used only for viewing the list, and as such they have no effect on the actual items. Items copied to the folder are not physically moved, and if the folders are deleted, the items in them are not lost. If you rename or delete an object from the code that is in a folder, you may need to manually drag the item into the folder again to clear the error node.

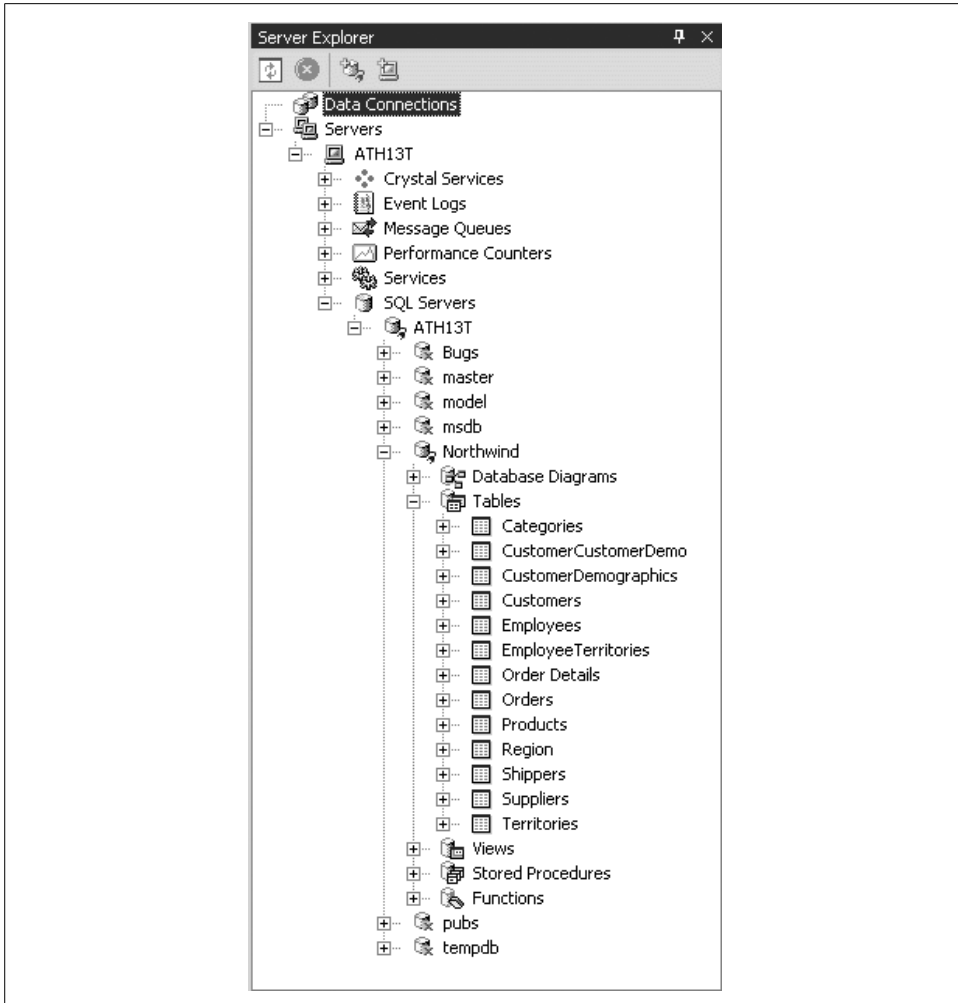


Figure 3-15. Server Explorer

### Object Browser (Ctrl+Alt+J)

The Object Browser is a tool for examining objects such as namespaces, classes, and interfaces, and their members, such as methods, properties, variables, and events. Figure 3-17 shows a typical Object Browser window.

The objects are listed in the pane on the left side of the window, and members of the object, if any, are listed in the right pane. The objects are listed hierarchically, with the ability to drill down through the tree structure. The icons used in this window are listed in Table 3-6 and Table 3-7.

Right-clicking on either an object or a member brings up a context-sensitive pop-up menu with a variety of menu options.

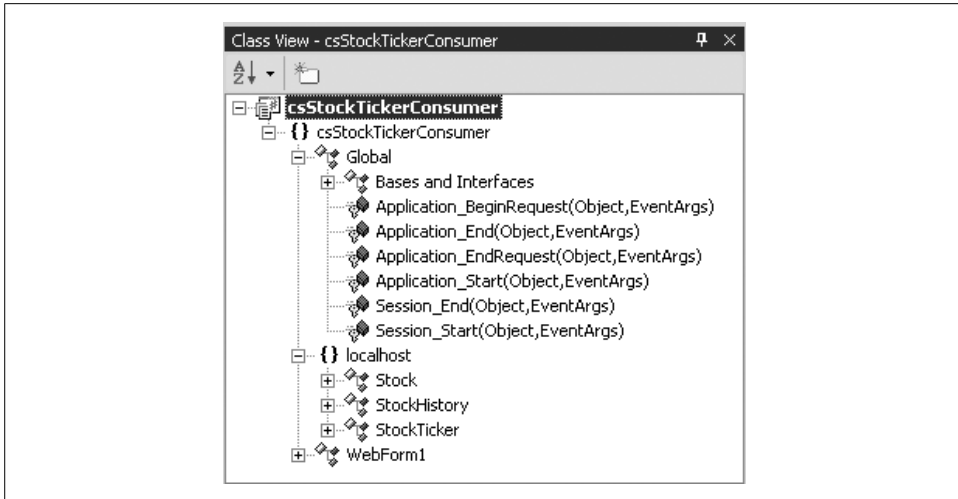


Figure 3-16. Class View

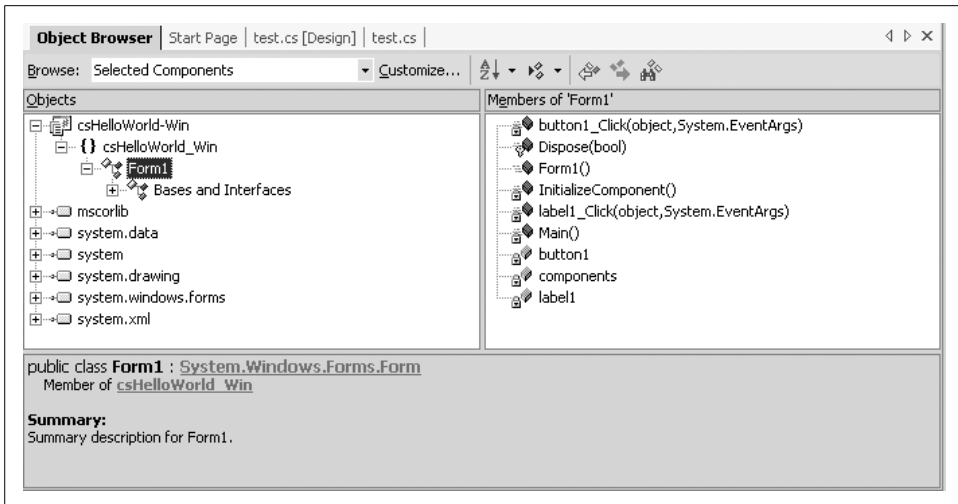


Figure 3-17. Object Browser

**Other Windows.** Several other windows have been relegated to a submenu called Other Windows:

### Macro Explorer (Alt+F8)

Visual Studio .NET offers the ability to automate repetitive chores with macros. A macro is a set of instructions written in VB.NET, either created manually or recorded by the IDE, saved in a file. The Macro Explorer is one of the main tools for viewing, managing, and executing macros. It provides access into the Macro IDE.

Macros are described further in the section below on the Tools → Macro menu command.

#### *Document Outline (Ctrl+Alt+T)*

When you design Web Forms, the Document Outline window is used to provide an outline view of the HTML document.

#### *Task List (Ctrl+Alt+K)*

In large applications, keeping a to-do list can be quite helpful. Visual Studio .NET provides this functionality with the Task List window. You can also provide shortcuts to comments in the Task List along with token strings, such as TODO, HACK, or UNDONE. The compiler also populates the Task List with compile errors.

#### *Command Window (Ctrl+Alt+A)*

The Command window has two modes: Command and Immediate.

Command mode enters commands directly, either bypassing the menu system or executing commands that are not contained in the menu system. (You can add any command to the menu or a toolbar button by using Tools → Customize.)

Immediate mode is used when debugging to evaluate expressions, view and modify variables, and other debugging tasks. The Immediate window and debugging will be covered further in Chapter 21.

For a complete discussion of command window usage, consult the SDK documentation.

#### *Output (Ctrl+Alt+O)*

The Output window displays status messages from the IDE to the developer, including debugger messages, compiler messages, and output from stored procedures.

### **Project menu**

The Project menu provides functionality related to project management. All functionality exposed by the Project menu is available in the Solution Explorer. It is often easier and more intuitive to accomplish your goals in Solution Explorer, but the menus lend themselves to keyboard use.

Each command under this menu pertains to the object currently highlighted in the Solution Explorer.

**Add... Menu Items.** Several menu items allow you to add either an existing or a new item to a project. They are self-explanatory, offering the same functionality as the equivalent items described previously under the File command.

They include:

- Add Windows Form
- Add Inherited Form

- Add User Control
- Add Inherited Control
- Add Component
- Add Class
- Add New Item (*Ctrl+Shift+A*)
- Add Existing Item (*Shift+Alt+A*)

**Exclude From Project.** Exclude From Project removes the file from the project but leaves the file intact on the hard drive. This is in contrast with the Delete popup menu item in the Solution Explorer. That will remove the file from the project *and* delete it from the hard drive (actually into the Recycle Bin). If a resource file is associated with the file, it will also be excluded or deleted, respectively.

The Exclude From Project command is also made available in the Solution Explorer by right-clicking on a file.

**Add Reference...** The Add Reference command is available in the Solution Explorer by right-clicking on a project. In either case, you will get the Add Reference dialog box shown in Figure 3-18. This dialog box allows you to reference assemblies or DLL's external to your application, making the public classes, methods, and members contained in the referenced resource available to your application.

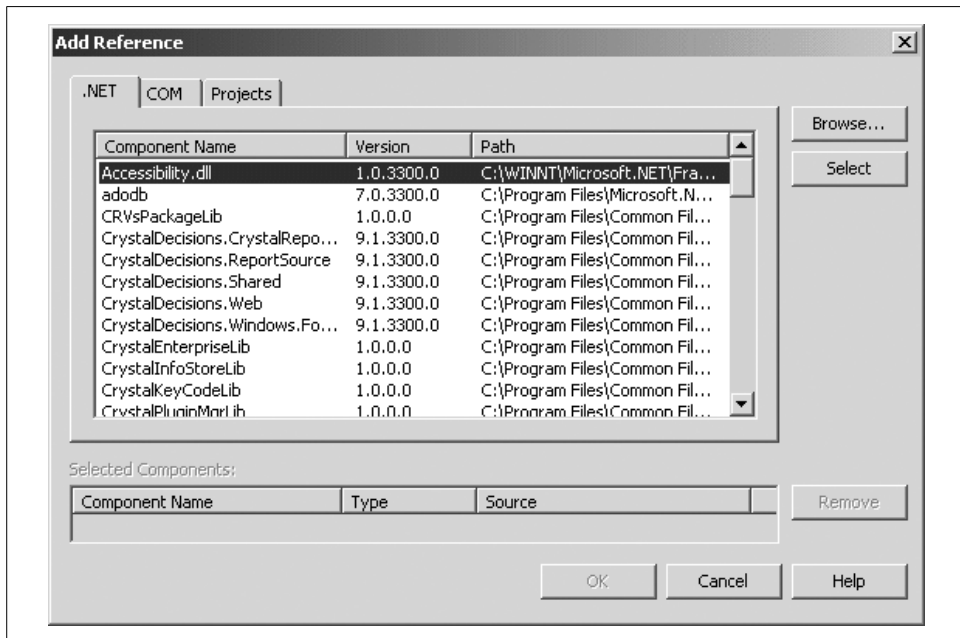


Figure 3-18. Add Reference dialog box

**Add Web Reference...** The Add Web Reference command, also available in the Solution Explorer by right-clicking a project, allows you to add a web reference to your project, thereby becoming a consuming web service application.



Web services are covered in *Programming ASP.NET*, Second Edition, by Jesse Liberty and Dan Hurwitz (O'Reilly).

**Set as StartUp Project.** If there is more than one project in a solution, specify the startup project. This command, also available in the Solution Explorer by right-clicking a project, allows you to make that specification. The project highlighted in Solution Explorer when this command is executed will become the startup project.

**Project Dependencies... / Project Build Order...** These commands, visible only when a solution contains multiple projects, also available in the Solution Explorer by right-clicking a project, presents a dialog box that allows you to control the build order of the projects in a solution. It presents a dialog box with two tabs: one for Dependencies and one for the Build Order.

The Project Dependencies command allows you to specify, for each project in the solution, which projects it depends upon. The dependent projects will be built first.

The Project Build Order command presents a list of all projects in the order in which they will be built.

If you are using Project References (as added with the Add Reference dialog mentioned above), you won't be able to edit either command. Project Dependencies are inferred when there are references between projects in the same solution. Also, you can't change the Build Order in any case—it is always inferred from the dependencies, whether or not those dependencies were automatically inferred.

## Build menu

The Build menu offers menu items for building the current project (highlighted in Solution Explorer) or the solution. It also exposes the Configuration Manager for configuring the build process.

The Build menu will be covered in detail in Chapter 22, which discusses deployment and configuration.

## Debug menu

The Debug menu allows you to start an application with or without debugging, set breakpoints in the code, and control the debugging session.

The Debug menu item will be covered, along with debugging, in Chapter 21.

## Data menu

This context-sensitive menu is visible only in the design mode. It is not available when editing code pages. The commands under it are available only when there are appropriate data controls on the form.

Chapters 19 through 21 cover data controls and data binding.

## Format menu

The Format menu is visible only when in design mode, and the commands under it are available only when one or more controls on the form are selected.

This menu offers the ability to control the size and layout of controls. You can:

- Align controls with a grid or with other controls six different ways
- Change the size of one or more controls to be bigger or smaller (or all be the same)
- Control the spacing horizontally and vertically
- Move controls forward or back in the vertical plane (z-order) of the form
- Lock a control so its size or position cannot be changed

To operate on more than one control, select the controls in one of several ways:

- Hold down Shift or Ctrl while clicking on controls you wish to select.
- Use the mouse to click and drag a selection box around all the controls to be selected. If any part of a control falls within the selection box, then that control will be included.
- To unselect one control, hold down Shift or Ctrl while clicking that control.
- To unselect all the controls, select a different control or press Esc.

When operating on more than one control, the last selected control will be the baseline. In other words, if you are making all the controls the same size, they will all become the same size as the last selected control. Likewise, if aligning a group of controls, they will all align with the last selected control.

As controls are selected, they will display eight resizing handles. These resizing handles will be white for all the selected controls except the baseline, or last control, which will have black handles.

With that in mind, all the commands under the Format menu are fairly self-explanatory.

## Tools menu

The Tools menu presents commands that access a wide range of functionality, ranging from connecting to databases, to accessing external tools, to setting IDE options. Some of the most useful commands are described next.

**Connect to Device...** The Connect to Device command brings up a dialog box that allows you to connect to either a physical mobile device or an emulator.

**Connect to Database...** The Connect To Database command brings up the dialog box that allows you to select a server, log in to that server, and connect to the database on the server. Microsoft SQL Server is the default database (surprise!), but the Provider tab lets you connect to any number of other databases, including any for which there are Oracle, ODBC, or OLE DB providers.

**Connect to Server...** The Connect to Server command brings up the dialog box that lets you specify a server to connect to, either by name or by IP address. It also lets you connect by using a different username and password.

This same dialog box can be exposed by right-clicking on Servers in the Server Explorer and selecting Add Server... from the pop-up menu.

**Add/Remove Toolbox Items...** The Add/Remove Toolbox Items command brings up the Customize Toolbox dialog box shown in Figure 3-19. The dialog box has two tabs: one for adding (legacy) COM components and one for adding .NET CLR-compliant components. All the components available on your machine (including registered COM components and .NET components in specific directories; you can browse for .NET components if they are not listed) are listed in one or the other. In either case, check or uncheck the line in front of the component to include the desired component.



For adding .NET components to the Toolbox, just drag it from Windows Explorer onto the Toolbox.

It is also possible to add other tabbed lists to this dialog box, although the details for doing so are beyond the scope of this book.

You can sort the components listed in the dialog box by clicking on the column head by which you wish to sort.

**Build Comment Web Pages...** This menu command brings up a dialog box that allows you to document your application via HTML pages. These HTML pages automatically display the code structure of your application. Projects are listed as hyperlinks. Clicking on a project brings up a page that shows all the classes as hyperlinks on the left side of the page. Clicking on any class lists all the class members, with descriptions, on the right side of the page.

If your language supports XML code comments (as does C#, but VB.NET does not), then you can add your own comments to your source code, and those comments will display in these web pages.

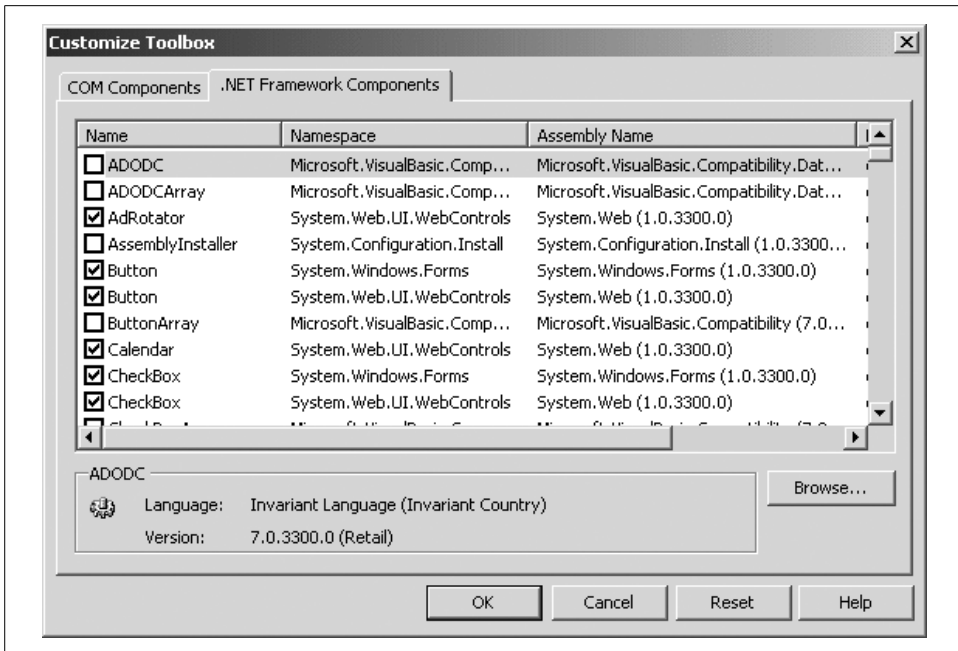
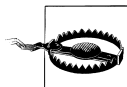


Figure 3-19. *Customize Toolbox dialog box*

Comment web pages are created by default in a subdirectory of the project called *CodeCommentReport*.

**Macros.** Macros are a wonderful feature that allows you to automate tasks in the IDE. Macros can either be coded by hand or recorded as you perform the desired task. If you allow the IDE to record the macro for you, then you can subsequently examine and edit the macro code it creates. This is similar to the macro functionality provided as part of Microsoft Word or Excel.



Be aware that macro recording doesn't work for anything inside a dialog box. For example, if you record the changing of property in a project's Property Pages, the recorded macro will open the Property Pages but won't do anything in there!

You can easily record a temporary macro by using the Macros → Record Temporary-Macro command, or by pressing Ctrl+Shift+R. This temporary macro can then be played back using the Macros → Run TemporaryMacro command, or by pressing Ctrl+Shift+P. It can be saved using the Macros → Save TemporaryMacro command, which will automatically bring up the Macro Explorer, described next.

Macros are managed with a Macro Explorer window, accessed via a submenu of the Macros command, or by pressing Alt+F8, as shown in Figure 3-20 after recording a temporary macro.

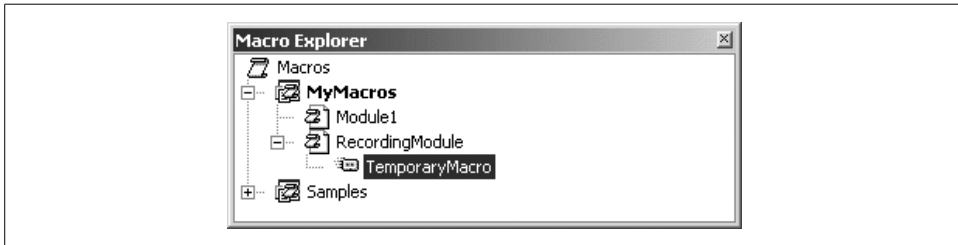


Figure 3-20. Macro Explorer

Right-clicking on a macro in the Macro Explorer pops up a menu with four items:

#### *Run*

Runs the highlighted macro. The macro can also be run by double-clicking on the macro name.

#### *Edit*

Brings up the macro editing IDE, where all macros for the user can be edited. The macro language is VB.NET, irrespective of the language used for the project. The macro editing IDE can also be invoked using the Macros → Macro IDE command, or by pressing Alt+F11.

#### *Rename*

Allows the macro to be renamed.

#### *Delete*

Deletes the macro from the macro file.

All macros are contained in a *macro project* called, by default, MyMacros. This project is comprised of a binary file called *MyMacros.vsmacros* (unless you have elected to convert it to the multiple files format), which is physically located in the *Documents and Settings* directory for each user. You can create a new macro project by using the Macros → New Macro Project command or by right-clicking on the root object in the Macro Explorer and selecting New Macro Project. In either case, you will get the New Macro Project dialog box, which lets you specify the name and location of the new macro project file.

Macro projects contain modules, which are units of code. Each module contains sub-routines, which correspond to the macros. For example, the macro called TemporaryMacro, shown in Figure 3-20 is the TemporaryMacros subroutine contained in the module named RecordingModule, which is part of the MyMacros project.

**External Tools...** Depending on the options selected at the time Visual Studio .NET was installed on your machine, you may have one or more external tools available on the Tools menu. The tools might include Create GUID, ATL/MFC Trace Tool, or Spy++. (Use of these tools is beyond the scope of this book.)

The Tools → External Tools... command allows you to add additional external tools to the Tools menu. When selected, you are presented with the External Tools dialog box. This dialog box has fields for the tool title, the command to execute the tool, any arguments and the initial directory, as well as several checkboxes for different behaviors.

**Customize...** The Customize... command allows you to customize many aspects of the IDE user interface. (The Options... command, described in the following section, lets you set a variety of other program options.) It brings up the Customize dialog box, which has three different tabs, plus one additional button, allowing customization in four different areas.

### Toolbars

This tab, shown in Figure 3-21, presents a checkbox list of all available toolbars, with checkmarks indicating currently visible toolbars. You can control the visibility of specific toolbars by checking or unchecking them in this list, or alternatively, use the View → Toolbars command.

You can also create new toolbars, rename or delete existing toolbars, or reset all the toolbars back to the original installation version on this tab.

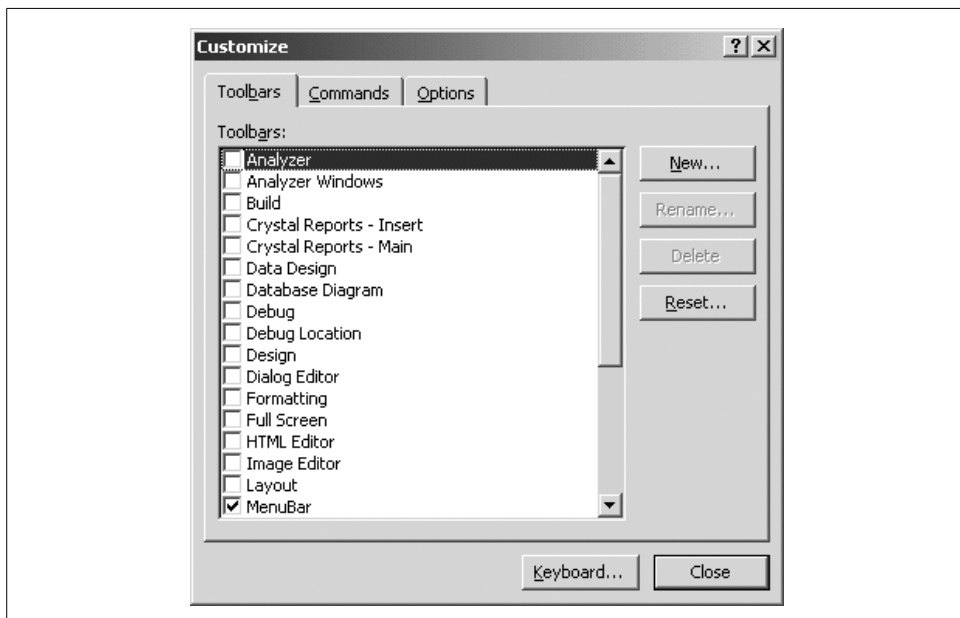


Figure 3-21. Customize dialog (Toolbars tab)

## Commands

The Commands tab, shown in Figure 3-22, allows you to add or remove commands from a toolbar or modify buttons already on the toolbar.

To add a command to a toolbar, select the category and command from the lists in the dialog box, and then use the mouse to drag the command to the desired toolbar.

To remove a command from a toolbar, drag it from the toolbar to anywhere in the IDE while the Customize Commands dialog is showing.

The Modify Selection button is active only when a button on an existing toolbar is selected. It allows you to perform such chores as renaming or deleting the button, changing the image displayed on the button, changing the display style of the button (image only, text only, etc.), and organizing buttons into groups.

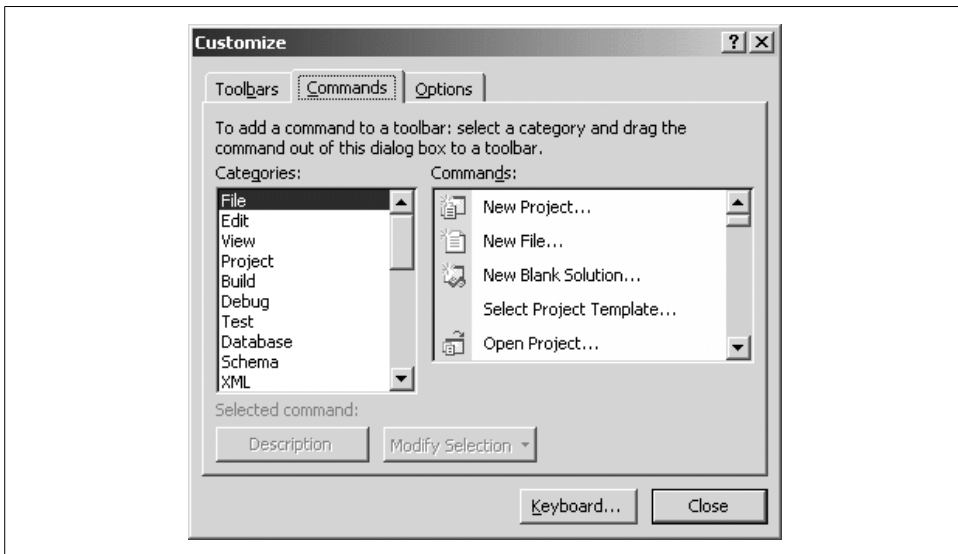


Figure 3-22. Customize dialog (Commands tab)

## Options

The Options tab, shown in Figure 3-23, allows you to change the toolbar's appearance.

The personalized Menus and Toolbars checkboxes are always unavailable and grayed out.

The Other checkboxes allow selection of icon size on buttons, control of tool tips, and the way the menus come in to view (Menu animations).

## Keyboard...

The Keyboard... button brings up the Environment → Keyboard page, shown in Figure 3-24, also accessible under the Tools → Options command described below. This page allows you to define and change keyboard shortcuts for commands.

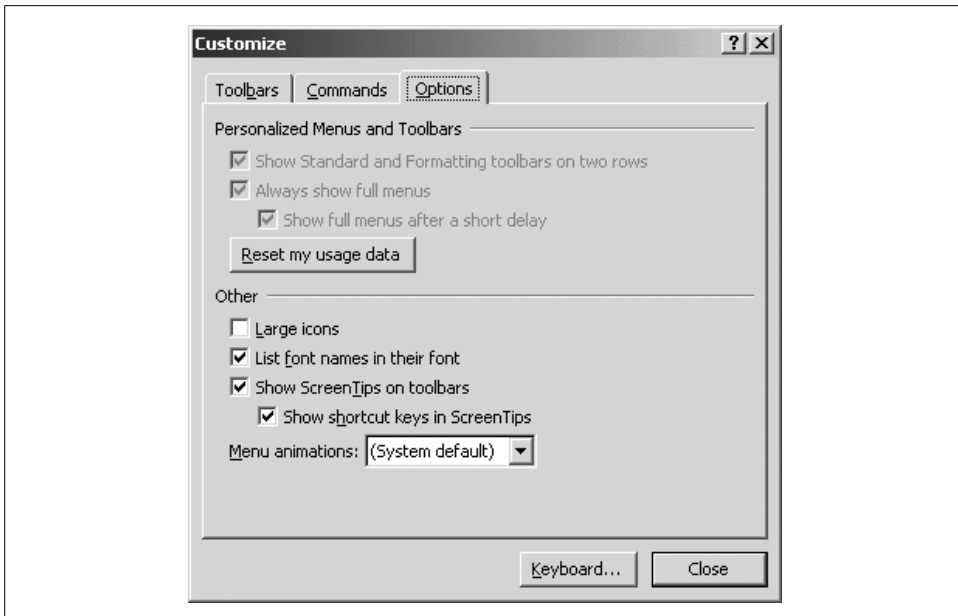


Figure 3-23. Customize dialog (Options tab)

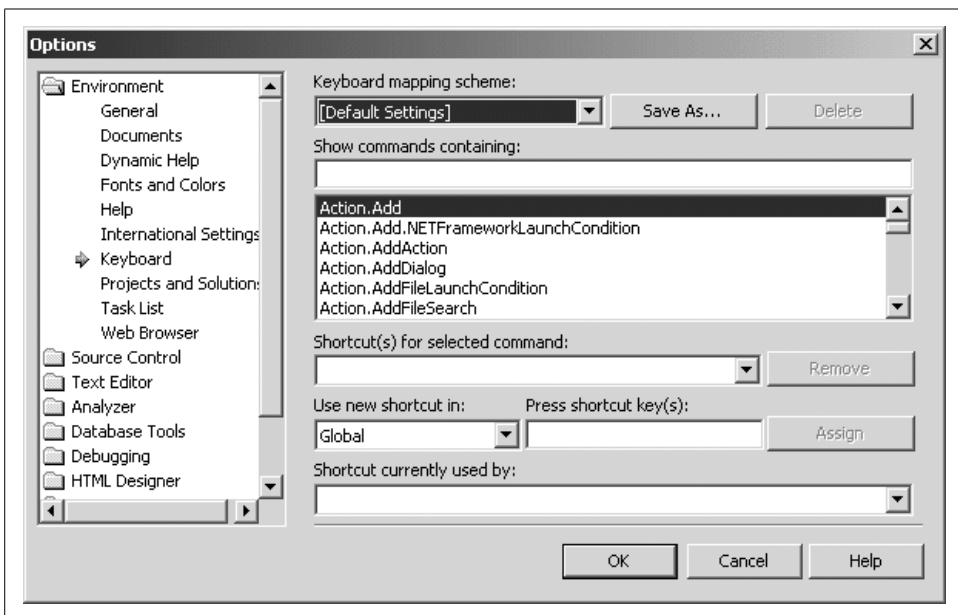


Figure 3-24. Customize dialog (Keyboard button)

**Options...** The Options... command brings up the Options dialog box, shown in Figure 3-24. This dialog box lets you set a wide range of options, ranging from the number of items to display in lists of recently used items, to XML Designer options.

The dialog box displays a hierarchical list of categories on the left side. Selecting any category allows you to drill down through the tree structure. Clicking on a detail item brings up the available properties on the right side of the dialog box.

Most available options are fairly self-explanatory. If you have any questions about specific settings, clicking on the Help button at the bottom of the Options dialog box will bring up context-sensitive help about all the properties relevant to the current detail item.

## Window menu

The Window menu item is a fairly standard Windows application Window command. It displays a list of the currently open windows, allowing you to bring any window to the fore by clicking on it. All the file windows currently displayed in the IDE also have tabs along the top edge of the design window, below the toolbars (unless you selected MDI mode in Tools → Options → Environment → General), and windows can be selected by clicking on a tab.

This is a context-sensitive menu. Table 3-9 lists the menu items available for different circumstances.

*Table 3-9. Window menu item commands*

Current window	Description of available commands
Design	Auto Hide All hides all dockable windows. Clicking on window's pushpin icon turns AutoHide off for that window. New Horizontal/Vertical Tab Group creates another set of windows with its own set of tabs. Close All Documents is self-explanatory. Window list.
Code	Same as for a design window plus the following: New Window creates a new window containing the same file as the current window. Use it to open two windows to the same source file. Split creates a second window in the current window for two different views of the same file. Remove Split removes a split window.
Dockable	This category includes the Solution Explorer, the Properties window, the Class View window, the Toolboxes, etc. These windows are dockable, as indicated by the pushpin icon in the upper-right corner of each. Available menu items are the same as for a design window, with the addition of commands to dock, hide, or float a window.

## Help menu

The Help menu provides access to a number of submenus. Those that are not self-explanatory are described here.

**Dynamic Help (Ctrl+F1).** If you are developing on a machine with enough horsepower, Dynamic Help is wonderful. Otherwise, it is a performance hog. (It can be disabled by unchecking all the checkboxes under Tools → Options → Environment → Dynamic Help) Alternatively, just closing the window is sufficient to prevent the performance hit, and then it is still available when you need it.

That said, using Dynamic Help is very simple. Open a Dynamic Help window by clicking on this menu item or pressing Ctrl+F1. Then wherever the focus is, whether in a design, code, or dockable window, context-sensitive hyperlinks will appear in the Dynamic Help window. Click on any link to bring up the relevant Help topic in a separate window.

**Contents... (Ctrl+Alt+F1)/Index... (Ctrl+Alt+F2)/Search... (Ctrl+Alt+F3).** These three commands provide different views into the SDK help system, allowing you to search by a (pseudo) table of contents, an incremental index, or a search phrase, respectively. The first type of search is an indexed search, while the latter two are full-text searches, so you may get different results by using the different search types using the same phrase.



The Help system exposed by these commands is the same Help system exposed in two other places by the Start button:

- Programs → Microsoft Visual Studio .NET 2003 → Microsoft Visual Studio .NET 2003 Documentation
- Programs → Microsoft .NET Framework SDK v1.1 → Documentation

This Help tool uses a browser-type interface, with Forward and Back navigation and Favorites. The list of topics is displayed in the lefthand pane, and the help topic itself, including hyperlinks, is displayed on the right.


**Index Results... (Shift+Alt+F2).** When searching for Help topics by Index, you will often find many topics for a given index entry. In these cases, the multiple topics are listed in an Index Results window. This window displays automatically if this is the case. This command lets you view the Index Results window if it has been closed.

**Search Results... (Shift+Alt+F3).** The Search Results window is analogous to the Index Results window described previously, except it pertains to searching for Help topics by search phrase.

**Edit Filters...** The SDK Help system is voluminous, with information on the full array of topics that might be found in any .NET installation, as well as a ton of non-.NET stuff as well. The Edit Filters command lets you restrict which Help topics will be searched. For example, if you are working exclusively in C#, you might set the filter to either Visual C# or Visual C# and Related.

**Check for Updates.** This command checks for service releases for your currently installed version of Visual Studio .NET. For this command to work, your machine must be connected to the Internet. If an update is available, you will be prompted to close the IDE before the service release is installed.

## Building and Running

You can run your application at any time by selecting either Start or Start Without Debugging from the Debug menu, or you can accomplish the same results by pressing either F5 or Ctrl+F5, respectively. In addition, you can start the program by clicking the Start icon (  ) on the Standard Toolbar.

The program can be built, i.e., EXE and DLL files generated, by selecting a command under the Build menu. You have the option of building the entire solution or only the currently selected project.

For a full discussion of application deployment, please see Chapter 22.

In the 1950s and 1960s, computer programs allowed for little user interaction. You fed in your data and instructions, and an answer popped out. As computers evolved, simple text-based menus were added. At specified times in the running of the program, the user could make choices and the program would respond accordingly. In the 1980s and 1990s, Graphical User Interfaces (GUIs) were developed, and computer programming was revolutionized.

In a modern Windows program, the user constantly interacts with the system: moving, clicking, and dragging the mouse or entering characters at the keyboard.

In Microsoft Windows the widgets with which the user interacts are called *controls*, and controls are visible on the monitor from the moment a modern program starts. In a Windows application, the user's action completely determines the order of execution of a program. This is called *event-driven* programming.

User actions, such as clicking on a button, generate (or “raise”) *events*. Other events are generated by the system itself. For example, your program might raise an event when a file has been read into memory, your battery's power is running low, or a timer indicates that a specified time interval has passed.

## Publish and Subscribe

In .NET, controls *publish* a set of events to which other classes can *subscribe*. When the publishing class raises an event, all the subscribed classes are notified.



This design is similar to the Publish/Subscribe (Observer) Pattern described in the seminal work *Design Patterns* by Gamma, et al. (Addison Wesley). Gamma describes the intent of this pattern, “Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

With this event mechanism, the control says, “Here are things I can notify you about,” and other classes might sign up, saying, “Yes, let me know when that happens.” For example, a button might notify any number of interested observers when it is clicked. The button is called the *publisher* because the button publishes the Click event, and the other classes are the *subscribers* because they subscribe to the Click event.

## Events and Delegates

Events are implemented with delegates. The publishing class defines a delegate that encapsulates a method that the subscribing classes implement. When the event is raised, the subscribing classes’ methods (the event handlers) are invoked through the delegate.



A *delegate* type defines the signature of methods that can be encapsulated by instances of that delegate type. A delegate can be marked as an event to restrict access to that delegate for use as an event handler.

For more information on the relationship between delegates and events, please see either *Programming VB.NET* or *Programming C#*, by Jesse Liberty (O’Reilly).

When you instantiate a delegate, pass in the name of the method the delegate will encapsulate. Register the event using the += operator (in C#) or the Handles and WithEvents keywords in VB.NET. (VB.NET can alternatively use the AddHandler keyword.) You may register more than one delegate with an event; when the event is raised, each of the delegated methods will be notified.

For example, when declaring an event, the .NET documentation describes the event and delegate used in the Button control’s Click event:

C#

```
public delegate void EventHandler(object sender, EventArgs e);  
public event EventHandler Click;
```

EventHandler is defined to be a delegate for a method that returns void and takes two arguments: one of type Object and the other of type EventArgs. The Click event is implemented with the EventHandler delegate.

## Event Arguments

By convention, event handlers in the .NET Framework are designated in C# to return void, are implemented as a sub in VB.NET, and take two parameters. The first parameter is the “source” of the event: the publishing object. The second parameter is an object derived from EventArgs.

EventArgs is the base class for all event data. Other than its constructor, the EventArgs class inherits all its methods only from Object, though it does add a public

static field *empty*, which represents an event with no state (to allow for the efficient use of events with no state).

If an event has no interesting data to pass, then the passed event argument will be of type *EventArgs*, which has no public properties, being essentially a placeholder. However, if there is interesting data, such as the location of a mouseclick or which key was pressed, then the event argument will be of a type derived from *EventArgs*, and it will have properties for the data being passed.

The general prototype for an event handler is as follows:

```
C# private void Handler (object sender, EventArgs e)
```

```
VB Private Sub Handler (ByVal sender As Object, ByVal e As EventArgs)
```



By convention, the name of the object argument is *sender* and the name of the *EventArgs* argument is *e*.

The *ByVal* keyword in the VB.NET version indicates that the arguments are passed by value, rather than by reference (*ByRef*). If neither *ByVal* nor *ByRef* is included, the default behavior in VB.NET is by value, so the use of the keyword here is redundant. However, using it explicitly serves as a form of documentation, and since Visual Studio .NET explicitly includes the *byVal* keyword, you will often see it included in event handlers.

While technically you pass by value, the object passed is itself a reference. A copy of the reference is made, but it refers to the original object, and changes made within the method will affect the original object through that copy of the reference. This book refers to passing a reference by value as “pass by reference.”

Some events use the base class *EventArgs*, but *EventArgs* objects contain no useful additional information about the event. The controls that do require their event handlers to know additional information about the event will pass in an object of a type derived from *EventArgs*.

For example, the *TreeView* *AfterCollapse* event handler receives an argument of type *TreeViewEventArgs*, derived from *EventArgs*. *TreeViewEventArgs* has the properties *Action* and *Node*, each of which has values pertaining to the actual event. The specifics of the event argument for each control are detailed when that control is discussed later in this book.

## Control Events

Every form and control used in Windows Forms derives from *System.Windows.Forms.Control*, so they inherit all of the more than 50 public events contained by the *Control* object. Some of the most commonly used *Control* events are listed in

Table 4-1 through Table 4-3. For each (public) event, a protected method handles the event.

Many controls support other events, in addition to the inherited events. For example, the `TreeView` class exposes several events for handling node expansion and collapse. Control-specific events are detailed in the relevant sections.

*Table 4-1. Common Control events*

Event	Event argument	Description
Click	EventArgs	Raised when a control is clicked by the mouse.
ControlAdded	ControlEventArgs	Raised when a new control is added to <code>Control.ControlCollection</code> .
ControlRemoved	ControlEventArgs	Raised when a control is removed from <code>Control.ControlCollection</code> .
DockChanged	EventArgs	Raised if the <code>Dock</code> property— i.e., which edge of the parent container the control is docked to—is changed, either by user interaction or program control.
DoubleClick	EventArgs	Raised when a control is double-clicked. If a control has both a <code>Click</code> and <code>DoubleClick</code> event handler, the <code>DoubleClick</code> will be preempted by the <code>Click</code> event.
Enter	EventArgs	Raised when a control receives focus. Suppressed for <code>Form</code> objects. For nested controls, cascades up and down the container hierarchy.
Layout	LayoutEventArgs	Raised when any change occurs that affects the layout of the control (e.g., the control is resized or child controls are added or removed).
Leave	EventArgs	Raised when focus leaves the control.
Move	EventArgs	Raised when a control is moved.
Paint	PaintEventArgs	Raised when a control is redrawn.
ParentChanged	EventArgs	Raised when the parent container of a control changes.
Resize	EventArgs	Raised when a control is resized. Generally preferable to use the <code>Layout</code> event.
SizeChanged	EventArgs	Raised when the <code>Size</code> property is changed, either by user interaction or programmatic control. Generally preferable to use the <code>Layout</code> event.
TextChanged	EventArgs	Raised when the <code>Text</code> property changes, either by user interaction or programmatic control.
Validating	CancelEventArgs	Raised when a control is validating. If <code>CancelEventArgs.Cancel</code> property set <code>true</code> , then all subsequent focus events are suppressed. Suppressed if <code>Control.CausesValidation</code> property set <code>false</code> .
Validated	EventArgs	Raised when a control completes validation. Suppressed if the <code>CancelEventArgs.Cancel</code> property passed to the <code>Validating</code> event is set <code>true</code> . Suppressed if <code>Control.CausesValidation</code> property set <code>false</code> .



Although all Forms controls inherit from `System.Windows.Forms.Control`, not all controls necessarily expose all the events contained in `Control`. For example, the Windows user interface does not allow you to double-click a button control. Yet the `Button` class inherits from `Control` via the `ButtonBase` class. In fact, even though Visual Studio .NET does not expose a `DoubleClick` event for a `Button` control, you can add the event and hook it up manually. Your program will compile and run, but the `DoubleClick` event will never be raised for a button.

.NET can suppress the `Click` and `DoubleClick` events on selected controls by setting the `StandardClick` and `StandardDoubleClick` values, respectively, of the `ControlStyles` enumeration.

The events listed in Table 4-2 implement drag-and-drop.

*Table 4-2. Control Drag-and-Drop events*

Event	Event argument	Description
<code>DragDrop</code>	<code>DragEventArgs</code>	Raised when a drag-and-drop operation is completed.
<code>DragEnter</code>	<code>DragEventArgs</code>	Raised when an object is dragged onto the control. At the time this event is raised, the drag operation is still in progress (i.e., the user hasn't yet let the mouse button go up).
<code>DragLeave</code>	<code>DragEventArgs</code>	Raised when an object is dragged off of the control.
<code>DragOver</code>	<code>DragEventArgs</code>	Raised when an object is dragged over the control.
<code>GiveFeedback</code>	<code>GiveFeedbackEventArgs</code>	Raised during a drag operation to allow modification to the mouse pointer.

The events listed in Table 4-3 are raised when a mouse interacts with a control. Some of these low-level events, in addition to being raised, are synthesized into the higher-level `Click` and `DoubleClick` events.

*Table 4-3. Control Mouse events*

Event	Event argument	Description
<code>MouseEnter</code>	<code>EventArgs</code>	Raised when mouse pointer enters control.
<code>MouseMove</code>	<code>MouseEventArgs</code>	Raised when mouse pointer moved over control.
<code>MouseHover</code>	<code>EventArgs</code>	Raised when mouse hovers over control.
<code>MouseDown</code>	<code>MouseEventArgs</code>	Raised when mouse button pressed while mouse pointer is over control.
<code>MouseWheel</code>	<code>MouseEventArgs</code>	Raised when mouse wheel moved while control has focus.
<code>MouseUp</code>	<code>MouseEventArgs</code>	Raised when mouse button released while mouse pointer is over control.
<code>MouseLeave</code>	<code>EventArgs</code>	Raised when mouse pointer leaves control.

## Web Controls Versus Windows Controls

ASP.NET web applications are also event driven. There are many similarities between web form and Windows Forms events. The main difference is that there are nearly 60 different Windows control events, but only six different Web Control events.

The reason for this is that all web form events are processed on the server, rather than locally on the client. As such, each event must be posted back to the server for processing, necessitating a roundtrip between the server and the client. If Web Forms supported the full complement of user interaction events, such as Mouse or Key events, performance would be severely impacted.

For a complete discussion of web form events, please see *Programming ASP.NET*, Second Edition (O'Reilly).

## Implementing an Event

Events were demonstrated back in Chapter 2. There, a Button was added to a form and the Click event for the button was handled. Handling the event was demonstrated both in a text editor and in Visual Studio .NET. Those examples also showed that the syntax and mechanics of handling events is somewhat different in C# and VB.NET, although the underlying fundamentals are the same.

The code samples shown in Example 4-1 (in C#) and Example 4-3 (in VB.NET) are duplicates of those shown in Examples 2-5 and 2-6.

### In C#

Example 4-1, reproduced here from Example 2-5, demonstrates the basic principles of implementing an event handler in C# by using a text editor.

*Example 4-1. Hello World Windows application with button control in C# (HelloWorld-win-button.cs)*

```
C# using System;
using System.Drawing;
using System.Windows.Forms;

namespace ProgrammingWinApps
{
    public class HelloWorld : System.Windows.Forms.Form
    {
        private Button btn;

        public HelloWorld()
        {
            Text = "Hello World";
```

*Example 4-1. Hello World Windows application with button control in C# (HelloWorld-win-button.cs) (continued)*

```
C#
    btn = new Button();
    btn.Location = new Point(50,50);
    btn.Text = "Goodbye";
    btn.Click += new System.EventHandler(btn_Click);

    Controls.Add(btn);
}

static void Main()
{
    Application.Run(new HelloWorld());
}

private void btn_Click(object sender, EventArgs e)
{
    Application.Exit();
}
}
```

Handling an event in C# involves two steps:

#### *Implement the event handler method*

The event handler method, highlighted in Example 4-1, is called `btn_Click`. It has the required signature (two parameters: `sender` of type `object` and `e` of type `EventArgs`), and, as required, it returns `void`.

The code in the body of the event handler method performs whatever programming task is required to respond to the event. In this example, the event handler closes the application with the static method `Application.Exit()`.

#### *Hook up the event handler method to the event*

This is done by instantiating an `EventHandler` delegate, which encapsulates the `btn_Click` method, then using the `+=` operator to add that delegate to the button's `Click` event. This is done in Example 4-1 with the following line of code:

```
C#
    btn.Click += new System.EventHandler(btn_Click);
```

As easy as this is, Visual Studio .NET makes it even simpler. The fundamentals in working with events in the IDE will be shown with a simple application.

Open Visual Studio .NET and create a new Visual C# Windows Application project. Name it `csEvents`. Put a `Label` and `Button` control on the form. Using the `Properties` window, set the control properties to the values listed in Table 4-4.

*Table 4-4. HelloWorld-Events Control properties*

Control type	Property	Value
Form	(Name)	Form1
	Text	Events Demonstrator
	Size	250,200

Table 4-4. HelloWorld-Events Control properties (continued)

Control type	Property	Value
Label	(Name)	lblTitle
	Text	Events Demonstrator
	Size	150,25
Button	(Name)	btnTest
	Text	Do It!

After the controls are placed and the properties set, Visual Studio .NET should look similar to Figure 4-1.

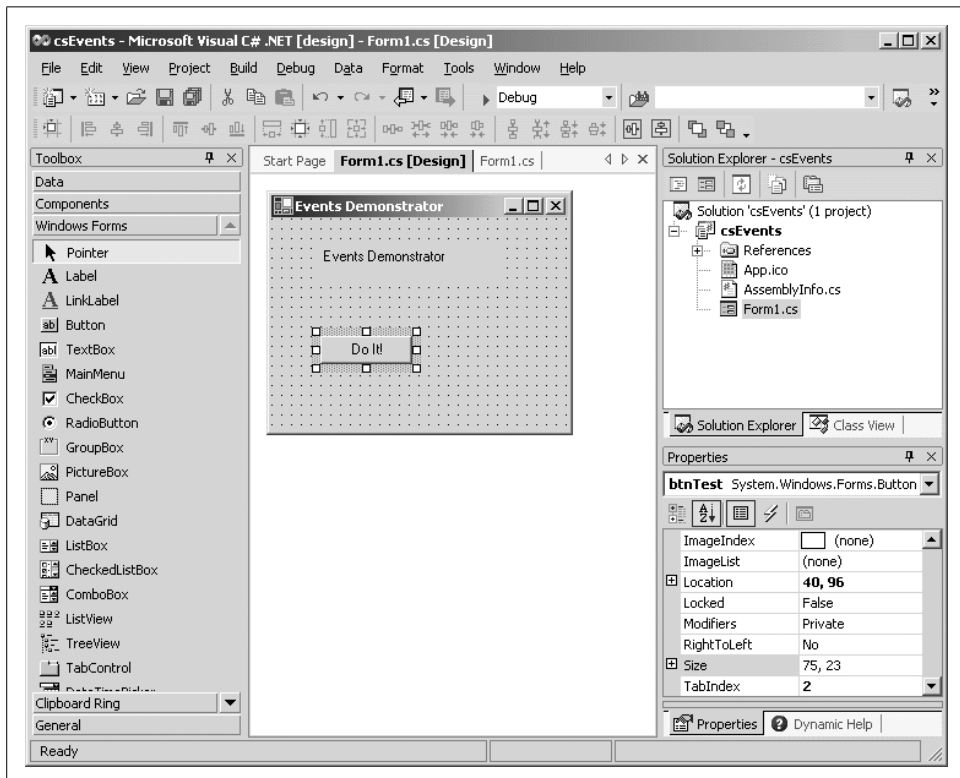


Figure 4-1. csEvents layout in Visual Studio .NET

There are several different ways to implement an event in Visual Studio .NET with C#.



C# and VB.NET differ in the user interface used by Visual Studio .NET to implement events, although the underlying class libraries and technology are the same. These differences are primarily a nod to backward compatibility for VB6 programmers. The VB.NET version will be detailed shortly.

**Double-click the control.** Double-click the button control. Visual Studio.NET takes this as an indication that you want to implement the default event handler for the button (the click event). Visual Studio.NET creates and registers an event handler, and moves you to the code window with the cursor placed in the body of the event handler:

```
C# private void btnTest_Click(object sender, System.EventArgs e)
{
}
}
```

Visual Studio.NET has created a method declaration that follows the event handler prototype exactly. The method name defaults to the name of the control with an underscore character and the default event name concatenated on the end.



Each control has a default event (whichever event is most commonly used with that control). For many controls, it is the Click event, but not always. The default event for the TreeView control is AfterSelect, although that control does have a Click event. The default event for each control will be detailed when the control is covered.

Enter a line of code to pop up a message box in response to the button click:

```
C# private void btnTest_Click(object sender, System.EventArgs e)
{
    MessageBox.Show("Click event just handled.", "Event Demo");
}
}
```

Running the form will produce the application shown in Figure 4-2. Clicking the button will pop up a message dialog with the words “Click event just handled,” as shown in Figure 4-3.

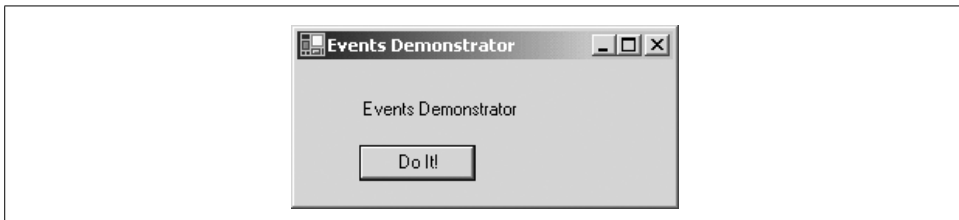


Figure 4-2. Events Demonstrator application

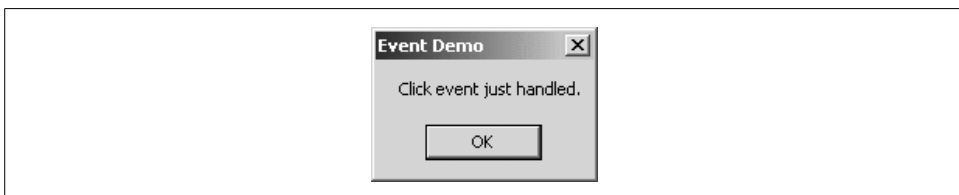


Figure 4-3. Event Demonstrator MessageBox

When you developed Example 4-1 in a text editor, you saw that in addition to implementing the event handler method, you also had to add a delegate encapsulating that method to the event. In Example 4-1, it was done with the following line of code:

```
C# btn.Click += new System.EventHandler(btn_Click);
```

When using Visual Studio .NET, registering the event is done for you automatically. This can be seen by going to the code window for the form, finding and expanding the region of code labeled Windows Form Designer generated code, and looking for the following section of code inside the InitializeComponent method:

```
C# //  
// btnTest  
//  
this.btnTest.Location = new System.Drawing.Point(56, 128);  
this.btnTest.Name = "btnTest";  
this.btnTest.TabIndex = 2;  
this.btnTest.Text = "Do It!";  
this.btnTest.Click += new System.EventHandler(this.btnTest_Click);
```

The highlighted line of code adds the method to the EventHandler delegate.

**Use the lightning bolt icon in the Properties window.** Double-clicking on the control only allows you to handle the default event using the default event handler method name. You can also create event handlers for any of a control's events, and name them whatever you like. To do so in C#, highlight the control in the design window and view the Properties window for the control (select View → Properties Window from the menu, press F4, or right-click and select Properties).

At the top of the Properties window is a row of buttons, shown in Figure 4-4. The first two buttons on the left sort the window's contents by category or alphabetically. The right-most button displays Property pages, if any. Of most interest here are the remaining two buttons.



Figure 4-4. Property Window button bar

The Properties button (📄) causes the window to display all the properties for the control, while the Events button (⚡) causes the window to display all the supported events for the control, either alphabetically or by category.

If there is an event handler already defined for an event, it will be listed in the column next to the event name. Clicking on that name and pressing the Enter key will take you to that event handler in the code window.

If there is no event handler listed next to an event, highlight the event and press the Enter key to create an event handler with the default name. The method skeleton will be created, the event will be registered, and you will be taken to that method in the code window, where you can enter the body of the method.

Alternatively, you can enter any method name you wish. Pressing Enter will use the name you entered to create a skeleton event handler method and automatically hook up that event handler method to the event.

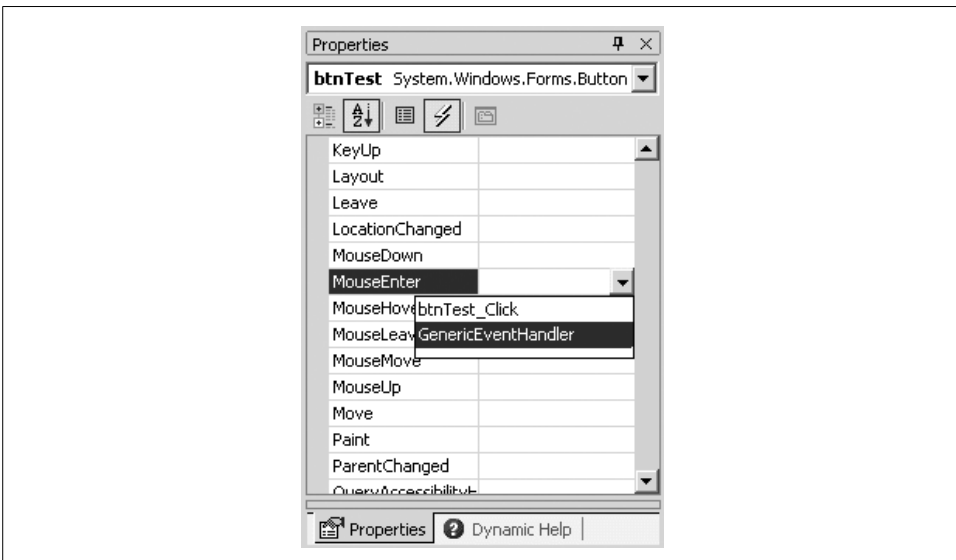
Finally, when an event is highlighted in the Properties window, a drop-down arrow will appear in the column for the method names. Clicking on the drop-down will display all the methods in the code available to be event handlers. This can be used to assign the same method to many different events, either for the same control or for different controls.

To demonstrate this last point, go to the code window for `csEvents`. Create a generic event handler method by adding the code shown in Example 4-2 to the `Form1` class.

*Example 4-2. Generic event handler in C#*

```
C# private void GenericEventHandler(object sender, EventArgs e)
{
    MessageBox.Show("Generic event handler", "Event Demo");
}
```

Now go back to the form designer and highlight the button. If the Events are not visible in the Property window, click on the Events button. Slide down to the `MouseEnter` event and click the drop-down arrow. You will see all the available methods, as shown in Figure 4-5.



*Figure 4-5. Event drop-down*

Click on `GenericEventHandler` to hook that handler method to the event.

Now add the same event handler to the Click method of the Label control named lblTitle. Click on the lblTitle control, find the Click event in the list of events in the Property window, click the drop-down arrow, and select GenericEventHandler. Run the program.

You will see that every time the mouse moves over the button, a dialog box similar to that shown in Figure 4-3 appears with the message Generic Event Handler. In fact, it is not possible to click on the button with your mouse because the MouseEnter event occurs before you have the opportunity to click on the button. (You can however click the button by pressing the Enter key once the button has focus.)

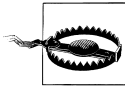
Clicking on the label containing the title also brings up the Generic Event Handler message.

Visual Studio .NET does all the work of hooking the GenericEventHandler method to both the lblTitle Click event and the btnTest MouseOver event. You can see how this was done by examining the region of code labeled Windows Form Designer generated code. In the section of code initializing the lblTitle control, the following line hooks the GenericEventHandler method to the Click event:

```
C# this.lblTitle.Click += new System.EventHandler(this.GenericEventHandler);
```

Similarly, the GenericEventHandler method is hooked to the btnTest MouseEnter event with this line of code:

```
C# this.btnTest.MouseEnter +=  
    new System.EventHandler(this.GenericEventHandler);
```



It is very easy to add event handlers to a form by double-clicking on a control. Sometimes adding the handlers is too easy, since accidentally double-clicking on a control will create an empty event handler method for that control, if it does not already have one. You may notice some of these empty methods in your code. These empty methods impose a small performance penalty on your program and clutter your code.

If you simply delete the empty methods, your program will not compile. Remember that the event handler was added to the event delegate in the “hidden” code contained in the InitializeComponent method in the Windows Form Designer generated code region. To manually delete the empty method from your code, you must also delete any references to the method where it is added to an event delegate.

A simple way to delete an event handler is to use the Properties window: highlight the offending event and delete the event handler. This will remove both the method and the registration code.

By the same token, if you rename an event handler method manually, you must find the relevant line in InitializeComponent and rename the method reference there as well, but doing so on the Properties window will update the reference for you.

In the unusual case that you want more than one handler for a single event, you can add as many event handler methods to an event as you wish simply by implementing the event handler methods, and then adding those events to the delegate with additional += statements. (Even in Visual Studio .NET, this requires inserting the lines of code yourself.)

Similarly, you can remove an event handler method by using the -= operator. For example, the following code snippet adds three methods to a delegate for handling the Click event, then removes one of the methods from the delegate. In this way, it is possible to add and remove event handler methods dynamically and thereby start and stop event handling for specific events anywhere in your program:

```
C# btn.Click += new System.EventHandler(GenericEventHandler);
    btn.Click += new System.EventHandler(SpecialEventHandler);
    btn.Click += new System.EventHandler(ClickEventHandler);
    btn.Click -= new System.EventHandler(GenericEventHandler);
```

## In VB.NET

Example 4-3, reproduced from Example 2-6, demonstrates the basic principles of implementing an event handler in VB.NET, using a text editor.

*Example 4-3. Hello World Windows application with button control in VB.NET (HelloWorld-win-button.vb)*

```
VB Imports System
Imports System.Drawing
Imports System.Windows.Forms

namespace ProgrammingWinApps
    public class HelloWorld : Inherits System.Windows.Forms.Form

        Private WithEvents btn As Button
        public sub New()
            Text = "Hello World"

            btn = new Button()
            btn.Location = new Point(50,50)
            btn.Text = "Goodbye"
            Controls.Add(btn)
        end sub

        public shared sub Main()
            Application.Run(new HelloWorld())
        end sub

        private sub btn_Click(ByVal sender as object, _
            ByVal e as EventArgs) _
            Handles btn.Click
            Application.Exit()
        end sub
    end class
end namespace
```

As with the C# example shown in Example 4-1, there are two steps to handling an event in VB.NET. However the syntax used in VB.NET is somewhat different than that in C#.

#### *Implement the event handler method*

The event handler method, highlighted in Example 4-3, is called `btn_Click`. It has the required signature (two objects: sender of type object and `e` of type `EventArgs`). Since this method is a subroutine, denoted by the `sub` keyword, it does not return a value. Event handlers never return a value.

The `Handles` keyword specifies which event this method will handle. The identifier following the keyword indicates that this method will handle the Click event for the Button called `btn`.

The code in the body of the event handler method performs whatever programming chore is required. In this example, it closes the application with the static method `Application.Exit()`.

#### *Instantiate the control using the WithEvents keyword*

Unlike in C#, there is no code here to explicitly add the method to the delegate. Instead, a Button is declared as a private member variable using the keyword `WithEvents`. This keyword tells the compiler that this object will raise events:

**VB**      `Private WithEvents btn As Button`

The compiler automatically creates delegates for any events referred to by a `Handles` clause and adds the event handler methods to the appropriate delegate.

Implementing events in VB.NET is made even easier when using Visual Studio .NET. To demonstrate this, open Visual Studio .NET and create a new VB.NET Windows application project called `vbEvents`.

Put a Label control and a Button control on the form. Using the Properties window, set the control properties to the values listed in Table 4-4 (the same values used in the C# example).

You can use VB.NET in several different ways to implement events in Visual Studio .NET.

**Double-click the control.** Double-click the Button control. You will be brought immediately to the code-editing window for the control, ready to enter code for the default event. The following code skeleton for the event handler method will be in place, with the cursor properly placed for you to commence typing the body of the method:

**VB**      `Private Sub btnTest_Click(ByVal sender As System.Object, _  
                                    ByVal e As System.EventArgs) Handles btnTest.Click  
End Sub`



In Visual Studio .NET, the method declaration will be on a single line, not wrapped with a line-continuation character, as printed here.

This method declaration exactly follows the event handler method prototype for VB.NET. The method name defaults to the name of the control with an underscore character and the default event name concatenated on the end. You can, however, use any name you wish, since the actual relationship between the method and the event it handles is dictated by the `Handles` keyword.



Each control has a default event (whichever event is most commonly used with that control). For many controls, it is the `Click` event. The default event for the `TreeView` control, however, is `AfterSelect`, although that control does have a `Click` event. The default event for each control will be detailed when the control is covered.

Enter a line of code to pop up a message box so that the `btnTest_Click` method now looks like:

```
VB Private Sub btnTest_Click(ByVal sender As System.Object,  
    ByVal e As System.EventArgs) Handles btnTest.Click  
    MessageBox.Show("Click event just handled.", "Event Demo")  
End Sub
```

Running the form will produce the same application previously developed in C#, shown in Figure 4-2. Clicking the button will pop up a message dialog with the words "Click event just handled," as shown in Figure 4-3.

In Example 4-3, which was developed in a text editor, you saw that in addition to implementing the event handler method, you must also instantiate the control using the `WithEvents` keyword. In Example 4-3, that was done with the following line of code:

```
VB Private WithEvents btn as Button
```

When using Visual Studio .NET, this step is done for you automatically. You can see it by going to the code window for the form, finding and expanding out the region of code labeled `Windows Form Designer generated code`, and looking for the following line of code:

```
VB Friend WithEvents btnTest As System.Windows.Forms.Button
```

In Example 4-3, the `Private` access modifier was used for the object, which restricts access to the object to the class of which it is a member, i.e., `HelloWorld`. Visual Studio .NET uses the `Friend` access modifier, which is somewhat more expansive, allowing access from any class within the project in which it is defined.



It is very easy to add event handlers to a form by double-clicking on a control. Sometimes it is too easy, since accidentally double-clicking on a control will create an empty event-handler method for that control, if it does not already have one. You may notice some of these empty methods in your code. In VB.NET applications, they do not usually cause any problems other than clutter.

If you simply delete the empty methods, the VB.NET program will still compile (unlike with C#). This is because there is no explicit connection between the event-handler method and the event in the code. The compiler makes the connection at compile time only if both halves of the connection are present.

If the control itself is deleted from the form designer, the lines of code instantiating the control with the `WithEvents` keyword will be removed, but the event-handler method will remain. You might want to delete these methods manually if they are no longer used, to minimize clutter and confusion in your code.

**Use the drop-down lists at the top of the code window.** Double-clicking on the control allows you to add code to the default event only by using the default event-handler method name. You can also create event handlers for any of a control's events. To do so in VB.NET, view the code window. At the top of the window are two drop-down lists, as shown in Figure 4-6. The drop-down on the left lists all the controls on the form, while the drop-down on the right lists all the possible events for each control (plus (Declarations), which moves the cursor to the top of the code window).

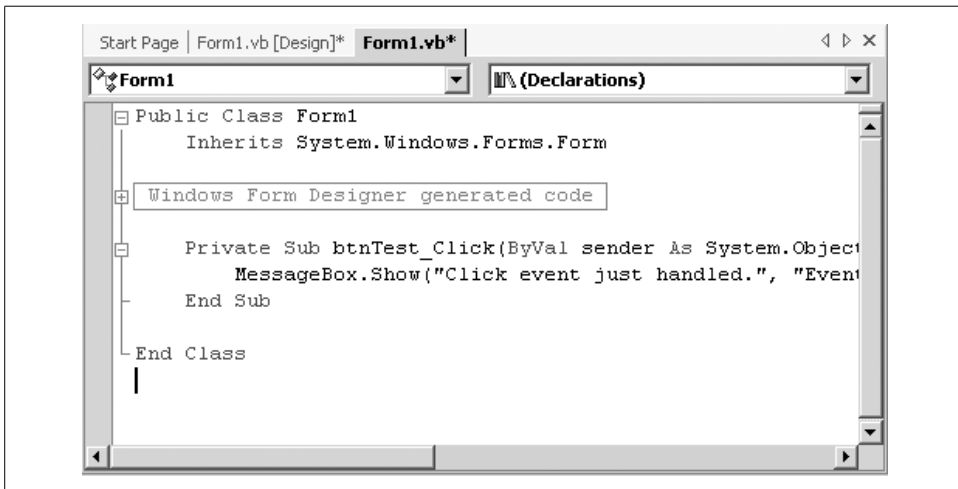


Figure 4-6. VB.NET Object and event drop-down lists

First select the control whose events you wish to handle from the left drop-down. Select the event to handle from the right drop-down. If the event handler already exists, the cursor will move to the subroutine. If the event handler subroutine does

not exist, then it will be created, with the cursor located inside the code skeleton, ready to enter your code.



It is a good idea to use meaningful, nondefault names for all the controls on your form that will be referenced elsewhere in the code to enhance readability and maintainability. Default names of controls are always the name of the control with a number appended, such as `Button23`, and default event-handler names are the name of the control, with an underscore and the event name, such as `Button23_Click`. This becomes especially important when assigning events to controls.

An event handler method can easily be renamed simply by editing the method declaration. There is no need to edit any other line of code, since the method name is not associated with the method until compile time.

It is not possible to directly assign the same event handler method to multiple events on a form using this syntax, as can be done in *C#*. This is because each event-handler method uses the `Handles` keyword to directly associate the method with a specific event. One way to accomplish this indirectly would be to call the same method from within multiple event handlers.

It is possible, however, to have multiple event-handler methods respond to the same event. This is accomplished by having the `Handles` keyword on each of the event-handler methods refer to the same event. In this situation, the order in which the multiple events will fire is not defined. If the order is important, then you need to use the dynamic event implementation and the `AddHandler` statement, described next.



The drop-down at the top of the code window is somewhat different for the Form than for the controls contained within the Form. When a control other than the Form is selected in the left drop-down, the right drop-down lists all the possible events for that control.

If the Form is selected in the left drop-down menu, the only thing listed besides (Declarations) are existing methods in the Form class plus the protected method `Finalize`.

There are two other entries in the left drop-down menu that are not objects: (Overrides) and (Base Class Events). (Overrides) causes the right drop-down to display all the virtual methods and properties of the Form class that can be overridden. (Base Class Events) causes the right drop-down to display all the events for the Form class that can be implemented. In either case, selecting an entry from the right drop-down causes Visual Studio .NET to insert the skeleton code for the appropriate property, method, or event to be inserted in the code window. If an event is selected, Visual Studio .NET will also hook the event-handler method to the event.

**Dynamic event implementation.** In the VB.NET syntax described so far, there is no sign of delegates in the code. The compiler automatically creates the necessary delegates

at compile time and adds the methods marked with the Handles keyword to the appropriate delegate for each event.

You can explicitly add the event handler methods to the delegate in VB.NET, as is done in C#, even though it is not the default way in which Visual Studio .NET handles events in VB.NET programs. This is done with the AddHandler statement. AddHandler does not provide any significant performance benefits over the Handles keyword, but does provide greater flexibility. AddHandler and RemoveHandler allow you to add, remove, and change the event handler associated with an event dynamically (in your code at runtime). You can also add multiple event handlers to a single event using AddHandler.

The listing in Example 4-4 shows the modifications to Example 4-3 necessary to use the AddHandler statement rather than the Handles keyword for implementing events. The new or modified lines are highlighted.

*Example 4-4. Using AddHandler to implement events*

```
VB imports System
imports System.Drawing
imports System.Windows.Forms

namespace ProgrammingWinApps
    public class EventsDemo : inherits System.Windows.Forms.Form

        Private btn as Button
        public sub New()
            Text = "Events Demonstration - AddHandler"

            btn = new Button()
            btn.Location = new Point(50,50)
            btn.Text = "Test"

            Controls.Add(btn)
            AddHandler btn.Click, AddressOf btn_Click
        end sub

        public shared sub Main()
            Application.Run(new EventsDemo())
        end sub

        private sub btn_Click(ByVal sender as object, _
            ByVal e as EventArgs)
            MessageBox.Show("btn_Click method","Events Demonstration")
        end sub
    end class
end namespace
```

Three code changes are necessary to dynamically add event handlers in a VB.NET program.

- The line instantiating the control no longer includes the WithEvents keyword. (This is not mandatory, but the keyword is no longer needed.)
- The event handler method declaration no longer uses the Handles keyword.
- An AddHandler statement is included to add the event-handler method to the delegate that handles the event.

The AddHandler statement takes two comma-separated arguments. The first argument is the name of the event to be handled, using dot notation. It is a two-part name consisting of the name of the object and the name of the event. The second argument is the AddressOf keyword followed by the name of the method that handles the event.

The RemoveHandler statement uses the same syntax as the AddHandler statement. Together they let you start or stop event handling for any specific event anywhere in the program.

It is possible to use both techniques for event handling in the same program, even for the same event in the same control. Consider the program in Example 4-5, which uses both techniques to handle the Click event for the button.

*Example 4-5. Using both AddHandler and Handles to handle events*

```
VB imports System
imports System.Drawing
imports System.Windows.Forms

namespace ProgrammingWinApps
    public class EventsDemo : inherits System.Windows.Forms.Form

        Private WithEvents btn as Button
        public sub New()
            Text = "Events Demonstration - AddHandler"

            btn = new Button()
            btn.Location = new Point(50,50)
            btn.Text = "Test"

            Controls.Add(btn)
            AddHandler btn.Click, AddressOf btn_Click
        end sub

        public shared sub Main()
            Application.Run(new EventsDemo())
        end sub

        private sub btn_Click(ByVal sender as object, _
            ByVal e as EventArgs)
            MessageBox.Show("btn_Click method","Events Demonstration")
        end sub

        private sub btn_ClickHandles(ByVal sender as object, _
```

*Example 4-5. Using both AddHandler and Handles to handle events (continued)*

```
VB
    ByVal e as EventArgs) _
        Handles btn.Click
        MessageBox.Show("btn_ClickHandles method", "Events Demonstration")
    end sub
end class
end namespace
```

The WithEvents keyword was added back to the line instantiating the Button object. It has no effect on the AddHandler statement functionality. However, it does enable the btn\_ClickHandles method, which utilizes the Handles keyword, to also handle the button Click event.

When the program is compiled and run, the btnClickHandles method, which the compiler adds to the delegate behind the scenes, is called first. It is followed by the btn\_Click method, which is added to the delegate by the AddHandler statement.

The following two statements are equivalent:

```
C#    btn.Click += new System.EventHandler(GenericEventHandler);
```

```
VB    AddHandler btn.Click, AddressOf GenericEventHandler
```

## Performance

As mentioned throughout this chapter, events are implemented in the .NET Framework using delegates. This has a performance cost. For most events, especially those involved with user interaction such as Click and MouseOver, the performance hit is negligible. For some events in some applications, such as the Paint event in very high-performance or drawing-intensive applications, the performance penalty may be significant.

Creating a custom control and overriding the protected event method without adding a delegate has many benefits, one of which may be a small reduction in this performance penalty. This technique will be covered in Chapter 17.

## Some Examples

In this section, you will see examples of events in use. In the first example, you will use keyboard events to capture keystrokes, showing what information is available about each keystroke. The next example will use keystroke information and the Validating event to control and validate the contents entered into a text box.

## Keyboard Events

It is often useful or necessary to capture keystrokes and then take action based on the details related to that keystroke. For example, you may want to disallow certain characters or convert all lowercase characters to uppercase. Keyboard events provide access to this type of functionality.

The three events listed in Table 4-5 are raised when the user presses a key on the keyboard.

*Table 4-5. Key Events for all controls*

Event	Event data	Description
KeyDown	EventArgs	Raised when a key is pressed. The KeyDown event occurs prior to the KeyPress event.
KeyPress	KeyPressEventArgs	Raised when a character generating key is pressed. The KeyPress event occurs after the KeyDown event and before the KeyUp event.
KeyUp	EventArgs	Raised when a key is released.

The KeyDown and KeyPress events may seem somewhat redundant, but they fire at different points in the keyboard event stream and contain different information in the EventArgs object.

The EventArgs event data associated with the KeyDown and KeyUp events provides low-level information about the keystroke, listed in Table 4-6. This information allows you to determine, for example, if an upper- or lowercase character was pressed. It also tells you if any modifier keys (Alt, Ctrl, or Shift) were pressed and in which combination. (You will also get a KeyDown and a KeyUp event if a modifier key is pressed and released on its own.)

*Table 4-6. EventArgs properties (KeyDown and KeyUp)*

Property	Data type	Description
Alt	Boolean	Read-only value indicating if the Alt key was pressed. <code>true</code> if pressed, <code>false</code> otherwise.
Control	Boolean	Read-only value indicating if the Ctrl key was pressed. <code>true</code> if pressed, <code>false</code> otherwise.
Shift	Boolean	Read-only value indicating if the Shift key was pressed. <code>true</code> if pressed, <code>false</code> otherwise.
Modifiers	Keys	Read-only flags indicating the combination of modifier keys (Alt, Ctrl, Shift) pressed. Modifier keys can be combined using the bitwise OR operator.
Handled	Boolean	Value indicating if the event was handled. <code>false</code> until set otherwise.
KeyCode	Keys	Read-only value containing the key code for the key pressed. Typical values include the A key, Alt, and BACK (backspace).
KeyData	Keys	Read-only value containing the key code for the key pressed, combined with modifier flags to indicate combination of modifier keys (Alt, Ctrl, Shift).
KeyValue	integer	Key code property expressed as a read-only integer.



The state of the modifier keys can also be retrieved from the read-only `Control.ModifierKeys` property. This property is static in C# and shared in VB.NET. Like the `Modifiers KeyEventArgs` property, it is of type `Keys`.

The `Modifiers`, `KeyCode` and `KeyData` properties are of type `Keys`. The `Keys` enumeration, listed in Table A-2 in the Appendix, is comprised of constants identifying all the possible keys on a keyboard. The decimal key code value in Table A-2 corresponds to the virtual-key codes familiar to Windows programmers.

## Unicode and ASCII Characters

Each character in the American Standards Committee for Information Interchange (ASCII) character set is represented by a single byte (8 bits) of data, representing 256 characters. The first 128 characters (represented by 7 bits) are standardized and usually referred to as low-order ASCII characters. The upper 128 characters are not standardized, although many well established character sets use all 256 characters. lists the low order ASCII characters.

Unicode characters are a superset of the ASCII character set. They are represented by two bytes, which allows a maximum of 65,536 characters. The Unicode technology was introduced to allow easier representation of languages other than English, especially Asian languages such as Chinese and Japanese, which do not have limited alphabets. Unicode also allows for character sets containing many more characters than an ASCII character set, such as special symbols and stylings of characters.

You can insert Unicode characters or determine the Unicode character code for any character in Windows using the Character Map tool, which is accessible by clicking on the Start menu and then `Programs → Accessories → System Tools`.

The `KeyPress` event exposes two properties contained in `KeyPressEventArgs`, listed in Table 4-7. The `KeyChar` property is used to retrieve the composed ASCII character. In other words, if an uppercase character is pressed, the `KeyChar` property tells you that directly, as opposed to telling you a character was pressed in combination with the Shift key.

Table 4-7. *KeyPressEventArgs* properties (*KeyPress*)

Property	Description
Handled	Boolean value indicating if the event was handled. false until set otherwise. When true, the key-stroke is not displayed.
KeyChar	Read-only value of type char containing the composed ASCII character.

In the next example, you will create a simple Windows application with a single-line TextBox for entering keystrokes. A larger multiline TextBox will display the keystroke events and event argument properties so that you can see what is going on. Two Labels will simultaneously display the character in both upper- and lowercase, irrespective of how it was entered. Finally, a Reset button will clear all fields. During the course of the example, you will also see how to translate keystrokes from one character to another.

Open Visual Studio .NET and create a new project. Call it KeyEvents. (Since both C# and VB.NET examples are shown here, the examples will be saved as csKeyEvents and vbKeyEvents.)

Drag all the controls listed in Table 4-8 onto the form. Set the properties of the form and the controls to the values shown in Table 4-8. When done, the form should look something like Figure 4-7.

*Table 4-8. KeyEvents controls*

Control	Name	Property	Value
Form	Form1	Size	425,320
		Text	Key Event Demonstrator
TextBox	txtInput	Location	8,8
		Multiline	False
		Size	100,20
		Text	<blank>
TextBox	txtMsg	Location	8,40
		MultiLine	True
		ScrollBars	Vertical
		Size	304,232
		TabStop	False
		Text	<blank>
Button	btnReset	Location	328,8
		Size	75,23
		Text	Reset
Label	label1	Location	320,104
		Size	40,16
		Text	Lower:
Label	label2	Location	320,56
		Size	40,16
		Text	Upper:
Label	lblUpper	BorderStyle	Fixed3D
		Location	368,56

Table 4-8. KeyEvents controls (continued)

Control	Name	Property	Value
Label	lblLower	Size	32,23
		Text	<blank>
		BorderStyle	Fixed3D
		Location	368,104
		Size	32,23
		Text	<blank>

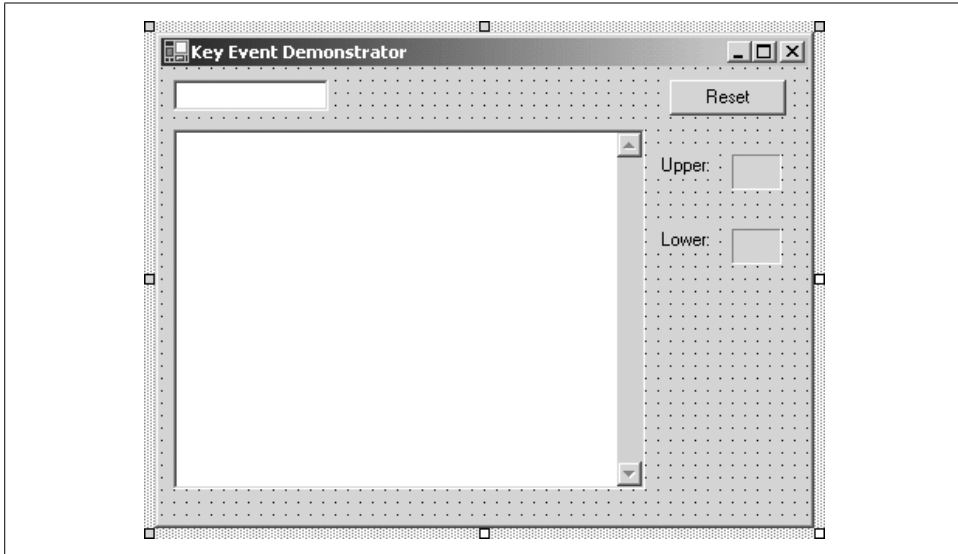


Figure 4-7. KeyEvents form layout

The Reset button will clear the Text properties of the TextBoxes and Labels. To implement this functionality, add an event handler for the Reset button. The easiest way to do this in either C# or VB.NET is to double-click on the control. Alternatively, you could use any of the language-specific techniques described earlier in this chapter. In any case, this will bring up a code window with an empty skeleton for the `btnReset_Click` event in place and the cursor placed for code entry.

Add the highlighted lines of code shown in Example 4-6 for C# and in Example 4-7 for VB.NET to the event handler skeletons in the code window.

Example 4-6. `btnReset` Click event handler in C#

```
C# private void btnReset_Click(object sender, System.EventArgs e)
{
    strMsg = "";
    txtMsg.Text = strMsg;
    txtInput.Text = "";
}
```

Example 4-6. *btnReset Click event handler in C# (continued)*

```
C#    lblUpper.Text = "";
    lblLower.Text = "";
}
```

Example 4-7. *btnReset Click event handler in VB.NET*

```
VB Private Sub btnReset_Click(ByVal sender As System.Object, _
                                ByVal e As System.EventArgs) _
                                Handles btnReset.Click

    strMsg = ""
    txtMsg.Text = strMsg
    txtInput.Text = ""
    lblUpper.Text = ""
    lblLower.Text = ""
End Sub
```

You may notice the variable `strMsg` underlined in the Visual Studio .NET code window. It is underlined because the editor recognizes that this variable name has not yet been declared. You must declare `strMsg` as a member variable of the `Form1` class so that it is visible to all of the methods of the class. To do this, add the appropriate line of code inside the `Form1` class declaration:

```
C#    private string strMsg = "";
```

```
VB    Dim strMsg As String = ""
```

Now you will implement an event handler for the `KeyDown` event for the `TextBox` named `txtInput`. Do *not* double-click on the `TextBox` control, or an empty code skeleton will be inserted for the `TextChanged` event, which is the default event for the `TextBox` control.

Instead, use the techniques described previously in this chapter to insert a code skeleton for a nondefault event, in this case the `KeyDown` event for `txtInput`. In C#, highlight the control in the design window, and then click on the Events icon (⚡) in the Properties window. Scroll down to the `KeyDown` event, highlight the event, and press Enter. In VB.NET, use the drop-down lists at the top of the code window. In the left drop-down, select the control: `txtInput`. In the right drop-down, select `KeyDown`.

To implement the `KeyDown` event handler, add the highlighted code shown in Example 4-8 (for C#) or in Example 4-9 (for VB.NET) to the empty code skeletons. The `KeyDown` event handler will get the character from the `KeyEventArgs` event argument, extract various properties from the event argument, and then append that information to the `TextBox` `txtMsg`.

*Example 4-8. txtInput\_KeyDown event in C#*

```
C# private void txtInput_KeyDown(object sender, System.Windows.Forms.KeyEventArgs e)
{
    txtMsg.AppendText("KeyDown event." + "\r\n");
    txtMsg.AppendText("\t" + "KeyCode name: " + e.KeyCode + "\r\n");
    txtMsg.AppendText("\t" + "KeyCode key code: " + ((int)e.KeyCode) +
        "\r\n");
    txtMsg.AppendText("\t" + "KeyData name: " + e.KeyData + "\r\n");
    txtMsg.AppendText("\t" + "KeyData key code: " + ((int)e.KeyData) +
        "\r\n");
    txtMsg.AppendText("\t" + "KeyValue: " + e.KeyValue + "\r\n");
    txtMsg.AppendText("\t" + "Handled: " + e.Handled + "\r\n");
    txtMsg.AppendText("\r\n");
}
```

*Example 4-9. txtInput\_KeyDown event in VB.NET*

```
VB Private Sub txtInput_KeyDown(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles txtInput.KeyDown
    txtMsg.AppendText("KeyDown event." + vbCrLf)
    txtMsg.AppendText(vbTab + "KeyCode name: " + e.KeyCode.ToString() + _
        vbCrLf)
    txtMsg.AppendText(vbTab + "KeyCode key code: " + _
        CInt(e.KeyCode).ToString() + vbCrLf)
    txtMsg.AppendText(vbTab + "KeyData name: " + e.KeyData.ToString() + _
        vbCrLf)
    txtMsg.AppendText(vbTab + "KeyData key code: " + _
        CInt(e.KeyData).ToString() + vbCrLf)
    txtMsg.AppendText(vbTab + "KeyValue: " + e.KeyValue.ToString() + _
        vbCrLf)
    txtMsg.AppendText(vbTab + "Handled: " + e.Handled.ToString() + vbCrLf)
    txtMsg.AppendText(vbCrLf)
End Sub
```

Run the application, make certain the input TextBox has focus, and enter an upper-case G (i.e., a shifted G). The result is shown in Figure 4-8.

Figure 4-8 shows that two KeyDown events were handled. The first was the pressed Shift key; the second was the letter G. The first two lines of data displayed for each event show the contents of the KeyEventArgs.KeyCode property. This is accomplished with the following two lines of code:

```
C# txtMsg.AppendText("\t" + "KeyCode name: " + e.KeyCode + "\r\n");
    txtMsg.AppendText("\t" + "KeyCode key code: " + ((int)e.KeyCode) +
        "\r\n");

VB txtMsg.AppendText(vbTab + "KeyCode name: " + e.KeyCode.ToString() + _
    vbCrLf)
    txtMsg.AppendText(vbTab + "KeyCode key code: " + _
        CInt(e.KeyCode).ToString() + vbCrLf)
```

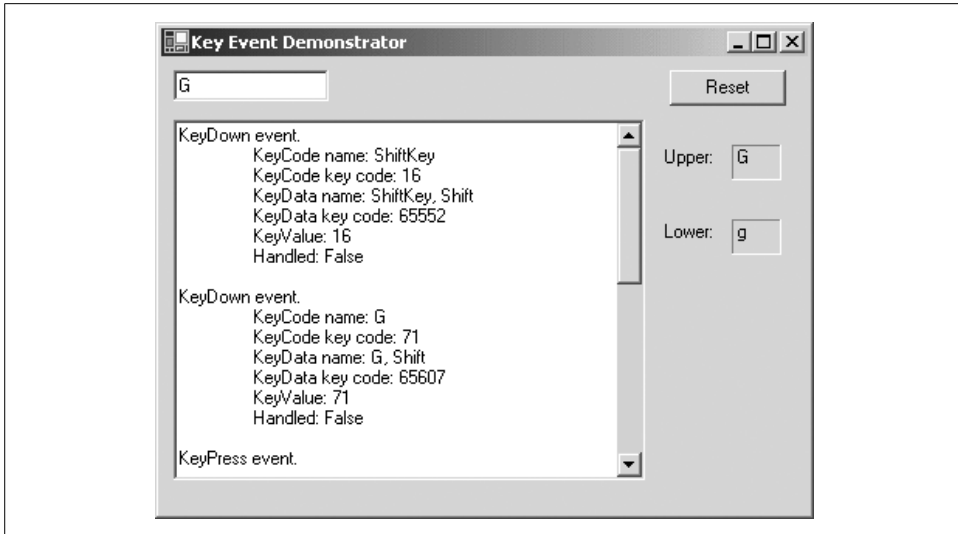


Figure 4-8. KeyEvents application showing a shifted G

The object `e` refers to the instance of `KeyEventArgs` passed in as one of the method arguments. It contains the properties listed in Table 4-6. The `KeyCode` property contains a member of the `Keys` enumeration (listed in Table A-2 in the Appendix) that identifies which key generated the `KeyDown` event.

`e.KeyCode` contains the name of the key. In VB.NET, the `ToString()` method must be used to include it as part of a string. That is not necessary in C#, although it would not do any harm.

Casting `e.KeyCode` to an integer returns the `KeyCode` key code, which corresponds to the virtual-key code familiar to Windows programmers, which itself corresponds (for the lower 127 characters) to the decimal ASCII value for the key. (The ASCII characters are listed in Table A-2.) The cast is done in C# using the cast operator `(( ))` and in VB.NET using the `CInt` function.

There is another significant difference between the two languages here. The C# version embeds tab characters and new lines using escape sequences in string literals, while the VB.NET version uses VB.NET constants for the purpose. The commonly used VB.NET constants and their C# equivalent are listed in Table 4-9.

Table 4-9. Commonly used VB.NET constants and C# escape sequences

VB.NET constant	C# escape sequence	KeyCode value (decimal)	Meaning
<code>vbCr</code>	<code>\r</code>	13	Carriage return
<code>vbCrLf</code>	<code>\r\n</code>	13 & 10	Carriage return/line-feed combination
<code>vbFormFeed</code>	<code>\f</code>	12	Form feed
<code>vbLf</code>	<code>\n</code>	10	Line feed (new line)
<code>vbTab</code>	<code>\t</code>	9	Tab

The next two lines displayed in the output report on the `EventArgs.KeyData` property. This is accomplished with the following lines of code:

```
C# txtMsg.AppendText("\t" + "KeyData name: " + e.KeyData + "\r\n");
    txtMsg.AppendText("\t" + "KeyData key code: " + ((int)e.KeyData) +
        "\r\n");
```

```
VB txtMsg.AppendText(vbTab + "KeyData name: " + e.KeyData.ToString() + _
    vbCrLf)
    txtMsg.AppendText(vbTab + "KeyData key code: " + _
        CInt(e.KeyData).ToString() + vbCrLf)
```

The `KeyData` property returns the same information as the `KeyCode` property combined with flags to indicate which modifier keys were pressed, if any. In this example, the `ShiftKey` was pressed in combination with the `Shift` modifier key (that does seem redundant since they are the same key) and the `G` key was pressed, also in combination with the `Shift` modifier key.

The next line reports the value of the `KeyValue` property. This is the key code corresponding to the key pressed. It is redundant with the `KeyCode`:

```
C# txtMsg.AppendText("\t" + "KeyValue: " + e.KeyValue + "\r\n");
```

```
VB txtMsg.AppendText(vbTab + "KeyValue: " + e.KeyValue.ToString() + _
    vbCrLf)
```

The final line displayed in the `KeyDown` event handler tells the status of the `Handled` property, which is `false` until specifically set otherwise:

```
C# txtMsg.AppendText("\t" + "Handled: " + e.Handled + "\r\n");
```

```
VB txtMsg.AppendText(vbTab + "Handled: " + e.Handled.ToString() + vbCrLf)
```

Looking ahead, the `KeyUp` and the `KeyDown` events both use the same event argument, `EventArgs`, so it is reasonable that both event handlers will want to display the same information. To do this, abstract out the contents of the event handler into a helper method, passing the event argument in, and then call the helper method in the event handler. This process is shown in Example 4-10 for C# and in Example 4-11 for VB.NET.

*Example 4-10. Handling `KeyDown` and `KeyUp` with helper method in C#*

```
C# private void KeyMsgBox(string str, EventArgs e)
{
    txtMsg.AppendText(str + " event." + "\r\n");
    txtMsg.AppendText("\t" + "KeyCode name: " + e.KeyCode + "\r\n");
    txtMsg.AppendText("\t" + "KeyCode key code: " + ((int)e.KeyCode) +
        "\r\n");
    txtMsg.AppendText("\t" + "KeyData name: " + e.KeyData + "\r\n");
    txtMsg.AppendText("\t" + "KeyData key code: " + ((int)e.KeyData) +
        "\r\n");
}
```

*Example 4-10. Handling KeyDown and KeyUp with helper method in C# (continued)*

```
C#    txtMsg.AppendText("\t" + "KeyValue: " + e.KeyValue + "\r\n");
    txtMsg.AppendText("\t" + "Handled: " + e.Handled + "\r\n");
    txtMsg.AppendText("\r\n");
}

private void txtInput_KeyDown(object sender,
                               System.Windows.Forms.KeyEventArgs e)
{
    KeyMsgBox("KeyDown", e);
}
```

*Example 4-11. Handling KeyUp and KeyDown with helper method in VB.NET*

```
VB    Private Sub KeyMsgBox(ByVal str As String, ByVal e As KeyEventArgs)

        txtMsg.AppendText(str + " event." + vbCrLf)
        txtMsg.AppendText(vbTab + "KeyCode name: " + e.KeyCode.ToString() + _
                           vbCrLf)
        txtMsg.AppendText(vbTab + "KeyCode key code: " + _
                           CInt(e.KeyCode).ToString() + vbCrLf)
        txtMsg.AppendText(vbTab + "KeyData name: " + e.KeyData.ToString() + _
                           vbCrLf)
        txtMsg.AppendText(vbTab + "KeyData key code: " + _
                           CInt(e.KeyData).ToString() + vbCrLf)
        txtMsg.AppendText(vbTab + "KeyValue: " + e.KeyValue.ToString() + _
                           vbCrLf)
        txtMsg.AppendText(vbTab + "Handled: " + e.Handled.ToString() + vbCrLf)
        txtMsg.AppendText(vbCrLf)
    End Sub

    Private Sub txtInput_KeyDown(ByVal sender As Object, _
                                   ByVal e As System.Windows.Forms.KeyEventArgs) _
        Handles txtInput.KeyDown

        KeyMsgBox("KeyDown", e)
    End Sub
```

The helper method is called `KeyMsgBox`. It takes two arguments: a string which should contain the name of the event and the instance of `KeyEventArgs`. The first argument is used in the first line in the method to display which event is being handled. `e` is used just as it was in the actual event handler method, described above.

The call to the helper method is simple; it involves passing in the name of the event and event argument:

```
C#    KeyMsgBox("KeyDown", e);
```

```
VB    KeyMsgBox("KeyDown", e)
```

Now that you have the `KeyDown` event handler implemented with a helper method to do the work, it is very simple to implement the `KeyUp` event handler in a similar fashion because both events use the same `KeyEventArgs` event argument. Again, remember not to double-click on the `txtInput` control, since that will implement the

default event, which is not what you want here. Instead, use the techniques described above for the language you are using. The KeyUp event handler is shown implemented in C# in Example 4-12 and in VB.NET in Example 4-13.

*Example 4-12. KeyUp event in C#*

```
C# private void txtInput_KeyUp(object sender,
                                   System.Windows.Forms.KeyEventArgs e)
{
    KeyMsgBox("KeyUp", e);
}
```

*Example 4-13. KeyUp event in VB.NET*

```
VB Private Sub txtInput_KeyUp(ByVal sender As Object, _
                              ByVal e As System.Windows.Forms.KeyEventArgs) _
    Handles txtInput.KeyUp
    KeyMsgBox("KeyUp", e)
End Sub
```

When the application is run with the implemented KeyUp event handler and an uppercase G is again entered in the TextBox, you will get the results shown in Figure 4-9 (scrolling down to the bottom half of the displayed text). The two KeyDown events, for the Shift key and for the G key, are the same as seen previously in Figure 4-8. They are followed by the KeyUp event for the G key, and followed by the KeyUp event for the Shift key. The event data for the KeyUp event is identical to the event data for the KeyDown event.

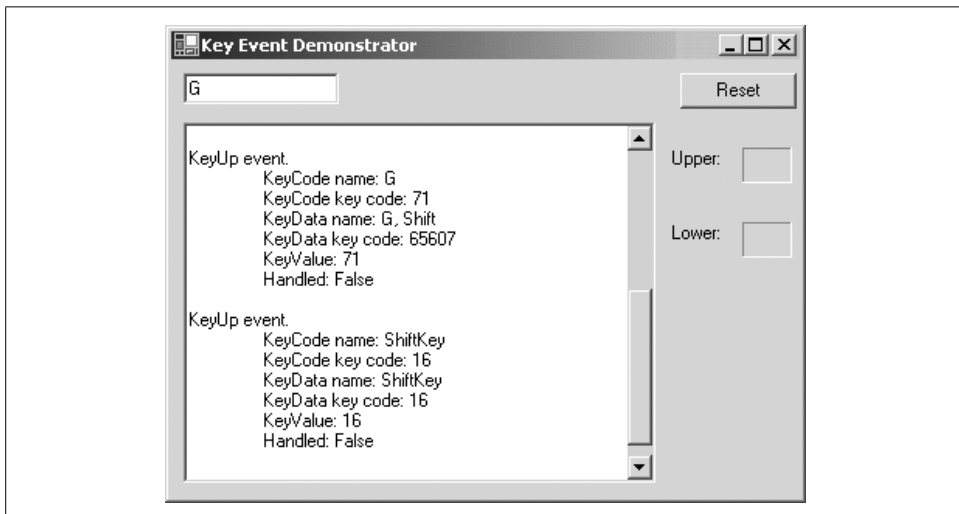


Figure 4-9. KeyDown and KeyUp events for shifted G

The KeyDown and KeyUp events provide a lot of information, but often you really care about the ASCII value of the keystroke—i.e., how the operating system

interprets the keystroke, not what key was actually pressed. For example, the G key will result in the same Keys enumeration of G, with a key code value of 71, irrespective of whether the Shift key was pressed (to produce an uppercase G) or not (to produce a lowercase g). To determine the case, you need to process additional properties. Similarly, the number 5 along the top of the keyboard will return a Keys enumeration of D5 with a key code value of 53, while the 5 on the numeric keypad will return a Keys enumeration of NumPad5 with a key code value of 101. In many applications, you won't care which key was pressed, you just want the ASCII value for the number 5, which is 53.

The KeyPress event provides exactly this sort of information. In addition, you can use the KeyPressEventArgs.Handled property to suppress a keystroke from being processed by the operating system. This will be demonstrated later.

To implement the KeyPress event handler, use the described techniques to add a KeyPress code skeleton to the ongoing example. Add the highlighted code shown in Example 4-14 (for C#) or in Example 4-15 (for VB.NET) to the empty code skeletons. This event handler will get the character from the KeyPressEventArgs event argument, and then append various pieces of information about the character to a string displayed in the txtMsg TextBox. It also populates the lblUpper label with an uppercase version of the character and lblLower label with a lowercase version.

*Example 4-14. txtInput KeyPress event handler code in C#*

```
C# private void txtInput_KeyPress(object sender,
                                   System.Windows.Forms.KeyPressEventArgs e)
{
    char keyChar;
    keyChar = e.KeyChar;

    txtMsg.AppendText("KeyPress event." + "\r\n");
    txtMsg.AppendText("\t" + "KeyChar: " + keyChar + "\r\n");
    txtMsg.AppendText("\t" + "KeyChar Code: " + (int)keyChar + "\r\n");
    txtMsg.AppendText("\t" + "Handled: " + e.Handled + "\r\n");
    txtMsg.AppendText("\r\n");

    // Fill in the Upper and Lower labels
    lblUpper.Text = keyChar.ToString().ToUpper();
    lblLower.Text = keyChar.ToString().ToLower();
}
```

*Example 4-15. txtInput KeyPress event handler code in VB.NET*

```
VB Private Sub txtInput_KeyPress(ByVal sender As Object, _
                                ByVal e As System.Windows.Forms.KeyPressEventArgs) _
                                Handles txtInput.KeyPress
    Dim keyChar As Char
    keyChar = e.KeyChar

    txtMsg.AppendText("KeyPress event." + vbCrLf)
```

Example 4-15. `txtInput` `KeyPress` event handler code in VB.NET (continued)

```
VB      txtMsg.AppendText(vbTab + "KeyChar: " + keyChar + vbCrLf)
      txtMsg.AppendText(vbTab + "KeyChar Code: " + _
          AscW(keyChar).ToString() + vbCrLf)
      txtMsg.AppendText(vbTab + "Handled: " + e.Handled.ToString() + vbCrLf)
      txtMsg.AppendText(vbCrLf)

      ' Fill in the Upper and Lower labels
      lblUpper.Text = keyChar.ToString().ToUpper()
      lblLower.Text = keyChar.ToString().ToLower()
End Sub
```

Running the application now and entering the letter G without the Shift key produces the results shown in Figure 4-10. Notice that the `KeyPress` information returns a lowercase g and a key code of 103, rather than the key code of 71 returned by the `KeyDown` event. 103 is the ASCII value for lowercase g while 71 is the ASCII value for uppercase G.

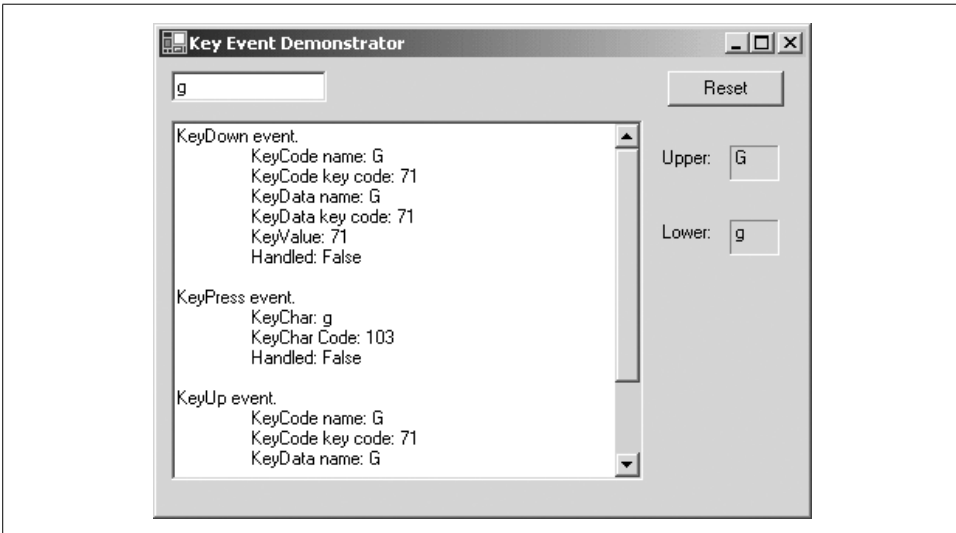


Figure 4-10. `KeyPress` event handler output

The first two lines in the event handler get the character entered at the keyboard from the `KeyPressEventArgs` event argument and assign it to a variable `keyChar`, declared as type `char`, since the `KeyPressEventArgs.KeyChar` property is of type `char` (i.e., it is a Unicode character).

The next several lines in the event handler use the `KeyPressEventArgs.KeyChar` property to retrieve the composed ASCII character, i.e., already taking into account modifier keys. Both the character name and integer value are displayed:

```
C#      txtMsg.AppendText("\t" + "KeyChar: " + keyChar + "\r\n");
      txtMsg.AppendText("\t" + "KeyChar Code: " + (int)keyChar + "\r\n");
```

**VB**

```
txtMsg.AppendText(vbTab + "KeyChar: " + keyChar + vbCrLf)
txtMsg.AppendText(vbTab + "KeyChar Code: " + _
    AscW(keyChar).ToString() + vbCrLf)
```

Since the `KeyChar` property is of type `char`, there is no need to use the `ToString` method in either language to concatenate it into a string. The character code value, on the other hand, is an integer and must be cast as such, and then converted to a string using the `ToString` method.



Objects of type `char` can implicitly convert to a string, since there is no loss of data in such a conversion. This is true even in VB.NET with the type checking switch on (Option Strict On). You can not implicitly convert from string to `char`, as information would be lost.

To declare a literal character in C#, enclose it in single quotes:

**C#**

```
char myChar = 'A';
```

In VB.NET, you append the letter `c`, as in:

**VB**

```
Dim myChar as Char = "A"c
```

The VB.NET version uses the `AscW` method rather than the more common `CInt` method to cast the value, since `Char` values in VB.NET cannot be converted to `Integer`. The `AscW` method returns an integer value representing the character code of a character.

The final three lines of code in Example 4-14 and Example 4-15 take the character entered, convert it to both upper- and lowercase, and fill in the appropriate labels.

Suppose you want to intercept the keystroke and selectively replace it with a different character. For example, suppose you want to intercept all dollar signs (\$) and replace them with a number sign (#). You could do this by handling the `Validating` event (demonstrated in the next section), but often a better way would be to change the character before it is even displayed on the screen. To do this, modify the `KeyPress` event-handler method to add the highlighted code shown in Example 4-16 (in C#) and Example 4-17 (in VB.NET).

*Example 4-16. Character substitution in `KeyPress` event in C#*

**C#**

```
private void txtInput_KeyPress(object sender,
    System.Windows.Forms.KeyPressEventArgs e)
{
    char keyChar;
    keyChar = e.KeyChar;

    txtMsg.AppendText("KeyPress event." + "\r\n");
    txtMsg.AppendText("\t" + "KeyChar: " + keyChar + "\r\n");
    txtMsg.AppendText("\t" + "KeyChar Code: " + (int)keyChar + "\r\n");
    txtMsg.AppendText("\t" + "Handled: " + e.Handled + "\r\n");
    txtMsg.AppendText("\r\n");

    // Fill in the Upper and Lower labels
    lblUpper.Text = keyChar.ToString().ToUpper();
```

Example 4-16. Character substitution in KeyPress event in C# (continued)

```
C# lblLower.Text = keyChar.ToString().ToLower();

// Change $ to #
if (keyChar.ToString() == "$")
{
    txtInput.AppendText("#");
    e.Handled = true;
}
}
```

Example 4-17. Character substitution in KeyPress event in VB.NET

```
VB Private Sub txtInput_KeyPress(ByVal sender As Object, _
    ByVal e As System.Windows.Forms.KeyPressEventArgs) _
    Handles txtInput.KeyPress

    Dim keyChar As Char
    keyChar = e.KeyChar

    txtMsg.AppendText("KeyPress event." + vbCrLf)
    txtMsg.AppendText(vbTab + "KeyChar: " + keyChar + vbCrLf)
    txtMsg.AppendText(vbTab + "KeyChar Code: " + _
        AscW(keyChar).ToString() + vbCrLf)
    txtMsg.AppendText(vbTab + "Handled: " + e.Handled.ToString() + vbCrLf)
    txtMsg.AppendText(vbCrLf)

    ' Fill in the Upper and Lower labels
    Dim str As String = e.KeyChar.ToString()
    lblUpper.Text = keyChar.ToString().ToUpper()
    lblLower.Text = keyChar.ToString().ToLower()

    ' Change $ to #
    If (keyChar.ToString() = "$") Then
        txtInput.AppendText("#")
        e.Handled = True
    End If
End Sub
```

When this code is run and a dollar sign (a shifted 4 on a U.S. English keyboard) is entered in the input field, the events displayed are KeyDown for Shift, KeyDown for 4, and KeyPress for \$, just as before. The Upper and Lower labels both display \$, since that character is unaffected by converting case. Before the event finishes, though, the character is tested to see if it is a \$. If so, the AppendText instance method appends the desired character, the # sign, to the text box. Then e.Handled is set to true. This suppresses all further handling of the original keypress.

## TextBox Validation

Several events can play a role in validating the contents of a TextBox, including the key events seen in the previous example. The sequence of events that occurs when a TextBox gains and loses focus are:

1. Enter
2. GotFocus
3. Leave
4. Validating
5. Validated
6. LostFocus

Of these, the GotFocus and LostFocus events are low-level events that are not typically used for validation. The Enter event is not useful for validation because it occurs before any data entry can occur. The Leave event is also not usually used for validation because its event argument, EventArgs, does not expose any properties for influencing the event.

Table 4-10 summarizes the most useful events for validating a TextBox.

*Table 4-10. TextBox events available for validation*

Event name	Event argument	Description
KeyPress	KeyPressEventArgs	Use the KeyPressEventArgs.Handled property to suppress keystrokes.
TextChanged	EventArgs	Raised if the Text property changed, either by user interaction or under programmatic control. Fires with every character entered in a TextBox.
Validating	CancelEventArgs	Raised after focus leaves the control and enters a control that has CausesValidation set to true. If the CancelEventArgs.Cancel property is set to true, then the current event is canceled, the Validated event is suppressed, and the focus is forced to remain in the control.
Validated	EventArgs	Raised after control is finished validating (after the Validating event).

In the following example, you will see the KeyPress and Validating events used to control and validate data entered in a TextBox. The example will allow a user to enter an ISBN number, which will then be validated.

International Standard Book Number (ISBN) numbers are used by the book industry to track and uniquely identify book titles. They are typically found on the back cover of books, often in conjunction with a bar code. There is more to ISBN numbers than the information discussed here (for example, the meaning of the different portions of the number and how they are assigned). All you need to know for this example, however, is that an ISBN number consists of nine digits, called the true number, plus one check digit or the letter X (for check-digit value 10). The digits may be separated into sections separated by hyphens. The hyphens must be allowed but are ignored.

The algorithm for calculating the check digit is as follows: Multiply the first digit in the true number by 10, the next digit by 9, the next by 8, and so on until the last digit is multiplied by 2. Add all these products together. The number needed to increase that sum to the next multiple of 11 is the check digit (that is, the check digit

is the sum of the products modulo 11). If the check “digit” turns out to be 10, use the letter X instead.

To demonstrate how this works, open Visual Studio .NET and create a new Windows application project called *IsbnValidate*. Add the controls and set the properties listed in Table 4-11.

Table 4-11. *IsbnValidate* controls

Control	Name	Property	Value
Form	Form1	Size	272,320
		Text	ISBN Validation
Label	label1	Location	48,16
		Font	Tahoma, 14.25pt, Bold Italic
		Size	176,23
		Text	ISBN Validation
TextBox	txtInput	Location	72,64
		Size	100,20
		Text	<blank>
Label	label2	Location	24,104
		Size	80,23
		Text	True Number:
Label	label3	Location	32,152
		Size	72,23
		Text	Check Digit:
Label	lblTrue	BorderStyle	Fixed3D
		Location	112,104
		Size	100,23
		Text	<blank>
Label	lblCheck	BorderStyle	Fixed3D
		Location	112,152
		Size	100,23
		Text	<blank>
Label	lblResults	Location	56,192
		Size	152,24
		Text	<blank>
Button	btnClear	Location	88,240
		Size	75,23
		Text	Clear

When all the controls are in place, the form layout should look similar to Figure 4-11.

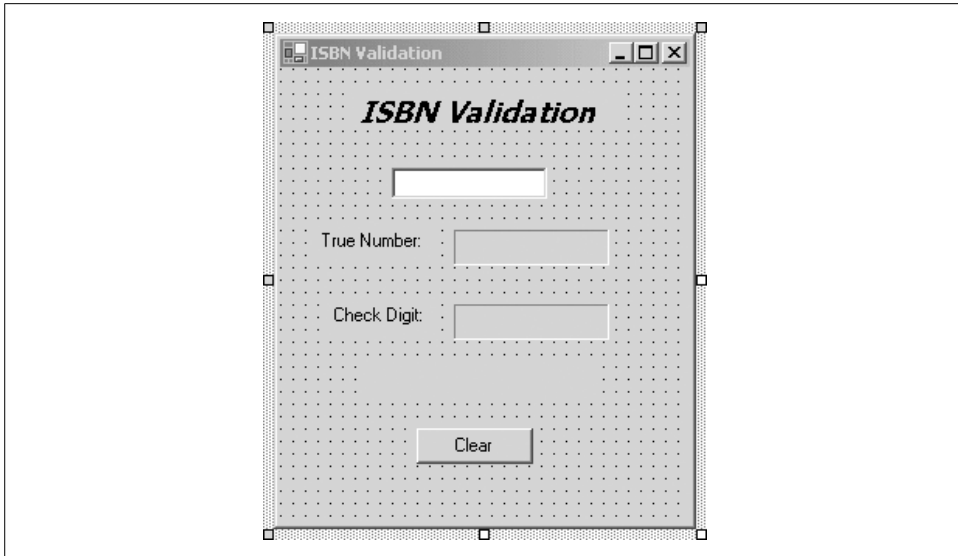


Figure 4-11. ISBN validator design layout

Most validation work will occur in the Validating event of the input TextBox, which takes `CancelEventArgs` as its event argument. `CancelEventArgs` has one property: `Cancel`. When set to true, all events that would normally occur after the Validating event are suppressed. This means that the Validated and LostFocus events do not fire, and the cursor cannot leave the control.

Implement the Validating event handler in C# by highlighting the input TextBox control, clicking on the Events icon (🔗) in the Properties window, scrolling to the Validating event, and entering the event handler method name: `IsbnValidate`. In VB.NET, go to the code-editing window, select the `txtInput` control from the drop-down list at the top left of the window, then scroll to the Validating event in the right drop-down. The method skeleton will have the default name of `txtInput_Validation`. Change it to `IsbnValidate`.

Enter the highlighted code from Example 4-18 into the C# `IsbnValidation` code skeleton or the highlighted code from Example 4-19 for into the VB.NET code skeleton.

Example 4-18. `IsbnValidation` event handler in C#

```
C# private void IsbnValidate(object sender,
                                System.ComponentModel.CancelEventArgs e)
{
    string strTrue;
    string strCheck;
    string strIsbn = "";
    string strPad;
    int sum = 0;
    int pad;
```