

Craig Utley

**A Programmer's Introduction to
Visual Basic.NET**

SAMS

201 West 103rd Street
Indianapolis, IN 46290 USA

A Programmer's Guide to Visual Basic.NET

Copyright © 2001 by Sams Publishing

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-32203-X

Library of Congress Catalog Card Number: 2001087650

Printed in the United States of America

First Printing: May 2001

04 03 02 01 4 3 2 1

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Sams Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

EXECUTIVE EDITOR

Shelley Kronzek

DEVELOPMENT EDITOR

Kevin Howard

MANAGING EDITOR

Charlotte Clapp

PROJECT EDITOR

Carol Bowers

COPY EDITOR

Michael Henry

INDEXER

Eric Schroeder

TECHNICAL EDITOR

Boyd Nolan

TEAM COORDINATOR

Pamalee Nelson

INTERIOR DESIGNER

Gary Adair

COVER DESIGNER

Gary Adair

PAGE LAYOUT

Gloria Schurick

Overview

	Foreword	viii
	Introduction	1
1	Why Should You Move to Visual Basic.NET?	3
2	Your First VB.NET Application	21
3	Major VB.NET Changes	49
4	Building Classes and Assemblies with VB.NET	73
5	Inheritance with VB.NET	91
6	Database Access with VB.NET and ADO.NET	105
7	Building Web Applications with VB.NET and ASP.NET	133
8	Building Web Services with VB.NET	153
9	Building Windows Services with VB.NET	165
10	Upgrading VB6 Projects to VB.NET	175
A	The Common Language Specification	187
	Index	191

Contents

INTRODUCTION	1
1 WHY SHOULD YOU MOVE TO VISUAL BASIC.NET?	3
Visual Basic.NET: A New Framework	3
The Common Language Runtime.....	6
Managed Execution	8
Microsoft Intermediate Language (MSIL)	8
The Just-In-Time Compiler	9
Executing Code	9
Assemblies	10
The Common Type System	12
Classes	13
Interfaces	13
Value Types	14
Delegates	14
The .NET Framework Class Library	14
Self-Describing Components	15
Cross-Language Interoperability	16
The Catch.....	17
Security	17
Code Access Security (CAS).....	18
Role-Based Security	18
Summary	18
2 YOUR FIRST VB.NET APPLICATION	21
The Start Page.....	21
Creating a New Project.....	23
Examining the IDE	25
Creating Your First VB.NET Application.....	31
Windows Application Enhancements	36
Resizing Controls Automatically.....	36
Anchoring Controls to the Form Edges	38
Easier Menus	41
Setting Tab Order	42
Line and Shape Controls: You're Outta Here	44
Form Opacity.....	45
Summary	48
3 MAJOR VB.NET CHANGES	49
General Changes	49
Default Properties	49
Subs and Functions Require Parentheses	50
Changes to Boolean Operators	51

Declaration Changes	52
Support for New Assignment Operators	52
ByVal Is Now the Default	53
Block-Level Scope	53
While...Wend Becomes While...End While	54
Procedure Changes	54
Array Changes.....	57
Option Strict.....	58
Data Type Changes	59
Structured Error Handling.....	62
Structures Replace UDTs	64
IDE Changes	66
New Items	66
Constructors and Destructors	66
Namespaces	67
Inheritance	69
Overloading	69
Free Threading.....	70
Garbage Collection	72
Summary	72

4 BUILDING CLASSES AND ASSEMBLIES WITH VB.NET 73

Creating Your First Class Library.....	74
Adding a “Souped-Up” Class	74
Creating Properties	75
Building a Test Client	76
Read-only and Write-only Properties	79
Parameterized Properties.....	79
Default Properties	80
Constructors in Your Classes.....	80
Classes Without Constructors	81
Adding Methods to Classes.....	82
Adding Events	82
The “Final” Code	84
Compiling the Assembly.....	86
Reusing the Assembly in Other Applications	87
How .NET Locates Assemblies.....	88
Summary	90

5 INHERITANCE WITH VB.NET 91

What Is Inheritance?.....	91
Interface Inheritance in VB6	92
VB.NET’s Implementation Inheritance.....	93
A Quick Inheritance Example	94
Shared Members	95
Inheritance Keywords	96

	Forcing or Preventing Inheritance	96
	Overriding Properties and Methods	97
	Polymorphism	99
	Polymorphism with Inheritance	100
	Polymorphism with Interfaces.....	101
	When to Use and When Not to Use Inheritance	102
	Summary	103
6	DATABASE ACCESS WITH VB.NET AND ADO.NET	105
	Accessing a Database from a Windows Application	106
	Using the DataAdapter Configuration Wizard	107
	ADO.NET	122
	About ADO.NET	122
	DataSets	122
	Working with the ADO.NET Objects	123
	XML Integration	128
	The XML Designer	129
	Summary	131
7	BUILDING WEB APPLICATIONS WITH VB.NET AND ASP.NET	133
	Your First ASP.NET Application	134
	How ASP.NET Works	137
	Web Pages and Code	138
	Server Controls	138
	Validation Controls	142
	Data Binding	149
	Handling Re-entrant Pages	151
	Summary	152
8	BUILDING WEB SERVICES WITH VB.NET	153
	Creating Your First Web Service	154
	Testing the Web Service	155
	Creating a Web Service Client	156
	How Web Services Work	162
	And You Thought Disco Was Dead.....	163
	Accessing Web Services	163
	Summary	164
9	BUILDING WINDOWS SERVICES WITH VB.NET	165
	Creating Your First Windows Services Project.....	166
	Adding Installers to Your Service	168
	Configuring Your Service	169
	Understanding Windows Services	170
	Service Lifetime and Events	171
	Debugging Your Service	172
	Summary	173

10 UPGRADING VB6 PROJECTS TO VB.NET 175

Upgrading Your First VB6 Application	175
The Visual Basic Upgrade Wizard	176
Examining the Upgraded Forms and Code	178
Modifications	179
Differences in Form Code	180
The Visual Basic Compatibility Library	181
The Upgrade Process	182
Learn VB.NET	182
Pick a Small Project and Make Sure That It Works	182
Upgrade the Project and Examine the Upgrade Report	183
Fix Any Outstanding Items in VB.NET	183
Helping Your VB6 Applications Upgrade	183
Do Not Use Late Binding	183
Specify Default Properties	184
Use Zero-Bound Arrays	184
Examine API Calls	184
Form and Control Changes	185
Summary	185

A THE COMMON LANGUAGE SPECIFICATION 187

What Is the Common Language Specification?	187
VB.NET Data Types and the CLS	188

INDEX 191

Foreword

Do you remember the moment when you wrote your first Visual Basic application? For some people, that moment happened ten years ago, when Microsoft released Visual Basic 1.0 in 1991. For others, that moment comes today, when they use Visual Basic.NET for the first time. Whenever it happens, you experience a feeling familiar to all VB programmers: “Wow! This makes development easy!” It happened to me in 1994, when I wrote my first application using Visual Basic 3.0. The application was a data-entry form with a data control, some text boxes, and an OK button—a simple application that read and wrote data to a Microsoft Access database. It took only a quarter of an hour to develop, and most importantly: I had fun doing it! When I finished, I realized that in fifteen minutes, VB had turned me into a Windows programmer, and my head started filling up with ideas of amazing programs I could write using VB. Suddenly, I was hooked.

I wasn’t alone. Since its inception in 1991, more than three million other developers have become hooked on VB. Visual Basic 1.0 revolutionized the way people developed software for Windows; it demystified the process of Windows application development and opened up programming to the masses. In its more than seven versions, Visual Basic has continued to provide us with the features we need to create rich, powerful Windows applications and as our needs evolved, so too did the Visual Basic feature set. In VB 1.0, database programming was limited to CardFile, the editor did not support Intellisense, and there were no Web development capabilities. Over the years, features such as these have been introduced and enhanced: VB 3.0 introduced the DAO data control and enabled us to easily write applications that interact with information in Access databases. When Windows 95 was released, VB 4.0 opened the door to 32-bit development and delivered the ability to write class modules and DLLs. VB 5.0 delivered productivity improvements with Intellisense in code and ActiveX control authoring. VB 6.0 introduced us to Internet programming with WebClasses and ActiveX DHTML pages.

Just as Visual Basic 1.0 opened the door to Windows development, Visual Basic.NET again opens up software development—this time to the more than three million Visual Basic developers. It makes it easier than ever before for VB developers to build scalable Web and server applications. It provides technology to bridge the gap from traditional client-side development to the next generation of Web services and applications. It extends the RAD experience that is the heart of Visual Basic to the server and to the Internet.

It has been a pleasure working with Craig Utley on this book. Visual Basic.NET introduces some new concepts; concepts such as assemblies, Web services, ADO.NET, and the .NET Framework. Craig explores these concepts and explains them in terms that will be familiar and relevant to VB developers. Craig is no

stranger to Visual Basic: He wrote his first VB application using VB 1.0, and in the years since, has written numerous books and articles on Visual Basic, ASP, and SQL Server programming. Craig also has worked extensively in the IT industry developing custom applications and providing consultancy and training services based around Visual Basic, ASP, COM+, and SQL Server. Adding to Craig's industry experience, the Microsoft Visual Basic Program Management team—the very people who designed the features of Visual Basic.NET—helped with the technical content of this book. The result is a concise and accurate introduction to Visual Basic.NET, an invaluable resource for the Visual Basic developer who wants to program the Web, use inheritance, access Web Services, upgrade projects, create Windows services, and begin using all the powerful new features of Visual Basic.NET.

When you write Visual Basic code, you join the three million developers who, for the past 10 years, have been the most productive programmers in the industry. With Visual Basic.NET, you enter the growing community of developers who have the most powerful and productive version of Visual Basic ever: a Visual Basic for both Windows and Web application development; a Visual Basic for creating and consuming next generation Web services; a Visual Basic that is redefining rapid application development in our connected world.

Ed Robinson
Program Manager
Microsoft Visual Basic.NET

About the Author

Craig Utley is President of CIOBriefings LLC, a consulting and training firm focused on helping customers develop enterprise-wide solutions with Microsoft technologies. Craig has been using Visual Basic since version 1.0, and he has guided customers through the creation of highly scalable Web applications using Active Server Pages, Visual Basic, MTS/Component Services, and SQL Server. Craig's skills in analyzing and designing enterprise-wide solutions have been used by large corporations and start-up companies alike. A frequent conference speaker as well as a book, courseware, and article author, Craig has recently spent much time writing about VB.NET and ASP.NET for both Sams and Volant Training.

Dedication

In memory of my grandparents: William and Kathryn Utley and Aubrey and Helen Prow

Acknowledgments

I have to start off by thanking Shelley Kronzek of Sams Publishing. She started talking to me a while ago about writing for Sams. She and I discussed a number of ideas, but I was hardheaded about only wanting to write a book to help Visual Basic developers move to VB.NET, because I saw it as such a fundamental shift in the way VB developers would work. Shelley finally got tired of hearing me talk about it, and said, “Do it. You have three weeks.”

Without Shelley’s trust, this could never have happened. She put together a fantastic team to help me: Mike Diehl provided valuable early input. Boyd Nolan has been a great technical editor, gently pointing out when I got things wrong (and I did). Kevin Howard, the development editor, has been great at keeping the process moving and helping make the book better by catching many of my mistakes. Mike Henry made sure my English teachers felt that I had learned what they taught me. Carol Bowers, the Project Editor, made sure I always had a chapter or two to review and kept us all on schedule.

Just as I couldn’t have done this without help from the team at Sams, I couldn’t have done it without the help of people at Microsoft. My main contact there was Ed Robinson, who provided brilliant support for me, even though I know he was swamped with trying to make sure that VB.NET was coming along. He coordinated most of my contact with people at Microsoft, and for that I owe him many thanks. The list of people at Microsoft who provided support is long, but I want to mention them all: Ari Bixhorn, Jim Cantwell, Alan Carter, Michael Day, Chris Dias, Steve Hoag, Andrew Jenks, Srivatsan Parthasarathy, Steven Pratschner, Sam Spencer, Susan Warren, Ben Yu Pan Yip, and Paul Yuknewicz all provided feedback on the chapters, in every case making them better. Dave Mendlen and Rob Copeland also got interested in the project and made sure that I had support from their teams. Finally, I’d like to thank David Keogh, who put together a great Author’s Summit for authors to learn about .NET.

This book would have been impossible without help from those listed earlier, and my good friend Martha McMahon. I’m sure there are more, and I apologize if I left out your name. Despite all the reviews done by various people, any mistakes in this book are strictly mine.

Tell Us What You Think!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As an Executive Editor for Sams, I welcome your comments. You can fax, e-mail, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax: 317-581-4770

E-mail: feedback@sampublishing.com

Mail: Shelley Kronzek
Executive Editor
Sams Publishing
201 West 103rd Street
Indianapolis, IN 46290 USA

Introduction

Why Does This Book Exist?

This book is meant to give you a head start on the changes from Visual Basic to Visual Basic.NET (VB.NET). Most of the book assumes that you are comfortable with Visual Basic 6.0 (VB6), so the book endeavors to be a quick introduction to the major differences between VB6 and the new VB.NET.

I've been using Visual Basic since version 1.0. The most dramatic shift had been in the move from VB3 to VB4, when class modules were introduced, and VB started on its long, slow path to becoming object oriented. For the first time, you could build COM components in VB, leading to an explosion in n-tier application development. VB4 brought COM development to the average programmer, so it was no longer a technology known only to a few C++ developers.

When I first started looking at the differences between VB6 and VB.NET, I realized that the change would be even more significant than it had been from VB3 to VB4. I thought it would be good to put together a book that helped VB6 developers transition to VB.NET. To that end, I pitched the idea for a book named something like *Migrating from VB to VB.NET* to a couple of different companies. Sams Publishing liked the idea, and one day they called me and asked me about doing a miniature version of the book...in three weeks.

I don't know who was crazier: Sams, for asking for the book in three weeks, or me, for agreeing to do it. Then, Sams said they were giving the book away, and I thought they had really lost it. Still, the mission was clear: create a book that targets Visual Studio.NET, Beta 1. Then, *the day after* I finished the book on Beta 1, Sams made the decision to release a book based on Beta 2 instead. Although I can't say I was thrilled, I think it was the right decision. There were significant changes between Beta 1 and Beta 2. Microsoft says there will be far fewer changes between Beta 2 and the final product, so this book should have a much longer shelf life than a book based on Beta 1.

There is no doubt: VB.NET will be an exciting change for us all. There is so much new material to learn that it can be somewhat daunting at first. However, the benefits of the .NET Framework are significant, and in the end can greatly reduce the effort required today to build enterprise-ready distributed applications.

This book will be followed by a much more comprehensive book based on the final version of Visual Studio.NET (VS.NET). The good news is that, as previously mentioned, the changes between Beta 2 and the final product should be far less dramatic than those changes between Beta 1 and Beta 2. Having said that, however, realize that there will be changes before Visual Studio is released.

Why VB.NET Instead of C#?

A lot of press has been given to the new language Microsoft has created: C# (pronounced “C-Sharp”). This is a new language, based on C/C++. C#, like VB.NET, is built specifically for the .NET Framework, and much has been written about it. Given all the hype, some people might wonder why they should choose VB.NET over C#.

Although both VB.NET and C# projects are created in the Visual Studio.NET environment, VB.NET was created specifically for VB developers and has a number of unique features that make it a great choice for building .NET applications. VB.NET is still the only language in VS.NET that includes background compilation, which means that it can flag errors immediately, while you type. VB.NET is the only .NET language that supports late binding. In the VS.NET IDE, VB.NET provides a drop-down list at the top of the code window with all the objects and events; the IDE does not provide this functionality for any other language. VB.NET is also unique for providing default values for optional parameters, and for having a collection of the controls available to the developer. Don’t forget that C#, like its C and C++ brethren, is case sensitive, something that drives most experienced VB developers crazy. In addition, C# uses different symbols for equality (=) and comparison (==). Finally, let’s face it: If you know VB, you are further down the road with VB.NET than you are with C#. Even though much has changed, the basic syntax of VB.NET is similar to VB, so you already know how to declare variables, set up loops, and so on.

As you can see, VB.NET has some advantages over the other .NET languages. If you’re curious about the advantages of VB.NET over traditional VB, you’ll have to read this book.

Who Should Read This Book?

This book is targeted at current VB developers. If you don’t know VB, parts of the book might not make sense to you. The goal here is to cover what has changed. So, if something hasn’t changed, I have to assume that you already know it. If you know VB, and want to learn VB.NET or at least see what it can do for you, this book is for you.

If you are currently using Visual InterDev to create Web applications, this book is also for you, because Visual InterDev has been integrated throughout Visual Studio.NET. This means you can create Visual InterDev–like Web Applications using VB.NET (and C#). You get several advantages with this new approach, including the ability to write in full VB.NET instead of VBScript, and the advantages of .NET’s Web application architecture (ASP.NET) over the current ASP model are significant.

No matter what you will be doing with VB.NET, the place to start is with the .NET Framework. Without understanding the .NET Framework, you won’t be able to write good VB.NET applications, regardless of whether they are Windows or Web applications. Therefore, prepare to get started by examining the .NET Framework.



CHAPTER 1

Why Should You Move to Visual Basic.NET?

One of the most common questions today is, “Why should I move to .NET?” .NET is new, and there are many questions about what it can do for you. From a Visual Basic standpoint, it’s important to understand some of the dramatic benefits that can be achieved by moving to VB.NET.

Visual Basic.NET: A New Framework

Many people have looked at VB.NET and grumbled about the changes. There are significant changes to the language: a new optional error handling structure, namespaces, true inheritance, free threading, and many others. Some see these changes as merely a way that Microsoft can place a check mark next to a certain feature and be able to say, “Yeah, we do that.” However, there are good reasons for the changes in VB.NET.

The world of applications is changing. This is merely a continuation of what has occurred over the past several years. If you took a Visual Basic 1.0 developer and showed him an n-tier application with an ASP front end, a VB COM component middle tier, and a SQL Server back end full of stored procedures, it would look quite alien to him. Yet, over the past few years, the vast majority of developers have been using Visual Basic to create COM components, and they have

become quite versed in ADO as well. The needs for reusability and centralization (a way to avoid distributing components to the desktop) have driven this move to the n-tier model.

The move to the Web revealed some problems. Scalability was an issue, but more complex applications had other requirements, such as transactions that spanned multiple components, multiple databases, or both. To address these issues, Microsoft created Microsoft Transaction Services (MTS) and COM+ Component Services. MTS (in Windows NT 4) and Component Services (an updated MTS in Windows 2000) acted as an object-hosting environment, allowing you to gain scalability and distributed transactions with relative ease. However, VB components could not take full advantage of all that Component Services had to offer, such as object pooling, because VB did not support free threading.

In the ASP/VB6 model, Microsoft had developers building a component and then calling it via an ASP. Microsoft realized that it would be a good idea to make the component directly callable over HTTP, so that an application anywhere in the world could use that component. Microsoft threw their support behind SOAP, Simple Object Access Protocol, which allows developers to call a component over HTTP using an XML string, with the data returning via HTTP in an XML string. Components sport URLs, making them as easy to access as any other Web item. SOAP has the advantage of having been a cross-industry standard, and not just a Microsoft creation.

At this point, you might be tempted to think that SOAP is all you need, and that you can just stick with VB6. Therefore it is important to understand what VB.NET gives you, and why it makes sense for you, and many other developers, to upgrade to .NET. For example, you create components and want them to be callable via SOAP, but how do you let people know that those components exist? .NET includes a discovery mechanism that allows you to find components that are available to you. You'll find out more about this mechanism, including the "disco" file, in Chapter 8, "Building Web Services with VB.NET." .NET also provides many other features, such as garbage collection for freeing up resources, true inheritance for the first time, debugging that works across languages and against running applications, and the ability to create Windows services and console applications.

Before proceeding, it's important to understand a little bit more about what is meant by ".NET." There are many ".NETs" here. There is VB.NET, which is the new version of Visual Basic. There is Visual Studio.NET, an Integrated Development Environment that hosts VB.NET, C#, and C++.NET. Underlying all this is the .NET Framework and its core execution engine, the Common Language Runtime.

In the .NET model, you write applications that target the .NET Framework. This gives them automatic access to such benefits as garbage collection (which destroys

objects and reclaims memory for you), debugging, security services, inheritance, and more. When you compile the code from any language that supports the .NET Framework, it compiles into something called MSIL, or Microsoft Intermediate Language. This MSIL file is binary, but it is not machine code; instead, it is a format that is platform independent and can be placed on any machine running the .NET Framework. Within the .NET Framework is a compiler called the Just-In-Time, or JIT, compiler. It compiles the MSIL down to machine code specific to that hardware and operating system.

In looking at the fundamental changes, it's important to understand that the number-one feature request from Visual Basic developers, for years, has been inheritance. VB has had *interface inheritance* since VB4, but developers wanted *real* or *implementation inheritance*. Why? What are the benefits? The main benefit of inheritance is the ability to create applications more quickly. This is an extension of the promise of component design and reusability. With implementation inheritance, you build a base class and can inherit from it, using it as the basis for new classes. For example, you could create a `Vehicle` class that provides basic functionality that could be inherited in both a `Bicycle` class and a `Car` class. The important point here is that `Bicycle` and `Car` inherit the *functionality*, or the actual code, from the `Vehicle` class. In VB4, the best you could do was inherit the structure, minus any implementation code. In VB.NET, the functionality in that base class is available to your other classes as is, or you can extend and modify it as necessary.

.NET provides you with integrated debugging tools. If you've ever debugged an ASP application that had VB COM components, you know that you had to use Visual InterDev to debug the ASPs and VB to debug the components. If you also had C++ components in the mix, you had to use the C++ debugger on those components. With .NET, there is one debugger. Any language that targets the .NET Framework can be debugged with that single debugger, even if one part of your application is written in VB.NET and calls another part written in C# (pronounced "C-Sharp"), or any other language built to target the .NET Framework.

.NET supplies a standard security mechanism, available to all parts of your application. .NET provides a possible solution to DLL Hell, and removes much of the complexity of dealing with COM and the registry. .NET allows you to run components locally, without requiring the calling application to go to the registry to find components.

There are also things that VB.NET can do that you cannot do today in VB. For example, Web Applications are a new form of project. Gone is Visual InterDev with its interpreted VBScript code. Instead, you now build your ASP.NET pages with VB.NET (or C# or C++), and they are truly compiled for better performance. VB.NET lets you create Windows services natively for the first time by providing a Windows Services project type. And yes, VB.NET lets VB developers build truly free-threaded components and applications for the first time.

Finally, you need to realize that the new language is actually going to have a version number on it, although the final name is undecided. It might well be called VB.NET 2002. This implies that at some point, there will be new versions of VB.NET, just as there were new versions of VB. In this book, references to previous versions of VB will be either VB or VB6. References to VB.NET 2002 will be just VB.NET.

You have decided you need to move from Visual Basic 6 to VB.NET, and you picked up this book to find out about the changes. Yet, the first thing you see is a chapter about the .NET Framework. Why start with the .NET Framework? The truth is that you cannot understand VB.NET until you understand the .NET Framework. You see, the .NET Framework and VB.NET are tightly intertwined; many of the services you will build into your applications are actually provided by the .NET Framework and are merely called into action by your application.

The .NET Framework is a collection of services and classes. It exists as a layer between the applications you write and the underlying operating system. This is a powerful concept: The .NET Framework need not be a Windows-only solution. The .NET Framework could be moved to any operating system, meaning your .NET applications could be run on any operating system hosting the .NET Framework. This means that you could achieve true cross-platform capabilities simply by creating VB.NET applications, provided the .NET Framework was available for other platforms. Although this promise of cross-platform capability is a strong selling point to .NET, there has not yet been any official announcement about .NET being moved to other operating systems.

In addition, the .NET Framework is exciting because it encapsulates much of the basic functionality that used to have to be built into various programming languages. The .NET Framework has the code that makes Windows Forms work, so any language can use the built-in code in order to create and use standard Windows forms. In addition, Web Forms are part of the framework, so any .NET language could be used to create Web Applications. Additionally, this means that various programming elements will be the same across all languages; a Long data type will be the same size in all .NET languages. This is even more important when it comes to strings and arrays. No longer will you have to worry about whether or not a string is a BStr or a CStr before you pass it to a component written in another language.

The Common Language Runtime

One of the major components of the .NET Framework is the Common Language Runtime, or CLR. The CLR provides a number of benefits to the developer, such as exception handling, security, debugging, and versioning, and these benefits are available to any language built for the CLR. This means that the CLR can host a variety of languages, and can offer a common set of tools across those languages. Microsoft

has made VB, C++, and C# "premier" languages for the CLR, which means that these three languages fully support the CLR. In addition, other vendors have signed up to provide implementations of other languages, such as Perl, Python, and even COBOL.

When a compiler compiles for the CLR, this code is said to be *managed code*. Managed code is simply code that takes advantage of the services offered by the CLR. For the runtime to work with managed code, that code must contain metadata. This metadata is created during the compilation process by compilers targeting the CLR. The metadata is stored with the compiled code and contains information about the types, members, and references in the code. Among other things, the CLR uses this metadata to

- Locate classes
- Load classes
- Generate native code
- Provide security

The runtime also handles object lifetimes. Just as COM/COM+ provided reference counting for objects, the CLR manages references to objects and removes them from memory when all the references are gone, through the process known as *garbage collection*. Although garbage collection actually gives you slightly less control than you had in VB, you gain some important benefits. For example, your errors should decrease because the number of objects that end up hanging around due to circular references should be reduced or completely eliminated. In addition, garbage collection ends up being much faster than the old way of destroying objects in VB. Instances of objects you create that are managed by the runtime are called *managed data*. You can interact with both managed and unmanaged data in the same application, although managed data gives you all the benefits of the runtime.

The CLR defines a standard type system to be used by all CLR languages. This means that all CLR languages will have the same size integers and longs, and they will all have the same type of string—no more worrying about BStrings and CStrings! This standard type system opens up the door for some powerful language interoperability. For example, you can pass a reference of a class from one component to another, even if those components are written in different languages. You also can derive a class in C# from a base class written in VB.NET, or any other combination of languages targeted to the runtime. Don't forget that COM had a set of standard types as well, but they were binary standards. This meant that with COM, you had language interoperability at run time. With .NET's type standard, you have language interoperability at design time.

After it is compiled, managed code includes metadata, which contains information about the component itself, and the components used to create the code. The runtime can check to make sure that resources on which you depend are available. The metadata removes the need to store component information in the registry. That means moving a component to a new machine does not require registration (unless it will be a global assembly, which is described in Chapter 4, “Building Classes and Assemblies with VB.NET”), and removing components is as simple as deleting them.

As you can see, the Common Language Runtime provides a number of benefits that are not only new, but should enhance the experience of building applications. Other benefits that you will see in more detail include some of the new object-oriented features to VB.NET. Many of these new features are not so much additions to the language as they are features of the runtime that are simply being exposed to the VB.NET.

Managed Execution

To understand how your VB.NET applications work, and just how much the code differs from the VB code that Dorothy wrote in Kansas, it's important to understand managed code and how it works. To use managed execution and get the benefits of the CLR, you must use a language that was built for, or *targets*, the runtime. Fortunately for you, this includes VB.NET. In fact, Microsoft wanted to make sure that VB.NET was a premier language on the .NET platform, meaning that Visual Basic could no longer be accused of being a “toy” language.

The runtime is a language-neutral environment, which means that any vendor can create a language that takes advantage of the runtime's features. Different compilers can expose different amounts of the runtime to the developer, so the tool you use and the language in which you write might still appear to work somewhat differently. The syntax of each language is different, of course, but when the compilation process occurs, all code should be compiled into something understandable to the runtime.

NOTE

Just because a language targets the runtime doesn't mean that the language can't add features that are not understood by other languages. To make sure that your components are completely usable by components written in other languages, you must use only types that are specified by the Common Language Specification. The Common Language Specification elements will be examined in Appendix A, “The Common Language Specification.”

Microsoft Intermediate Language (MSIL)

One of the more interesting aspects of .NET is that when you compile your code, you do not compile to native code. Before you VB developers panic and fear that you are

returning to the days of interpreted code, realize that the compilation process translates your code into something called Microsoft intermediate language, which is also called MSIL or just IL. The compiler also creates the necessary metadata and compiles it into the component. This IL is CPU independent.

After the IL and metadata are in a file, this compiled file is called the PE, which stands for either *portable executable* or *physical executable*, depending on whom you ask. Because the PE contains your IL and metadata, it is therefore self-describing, eliminating the need for a type library or interfaces specified with the Interface Definition Language (IDL).

The Just-In-Time Compiler

Your code does not stay IL for long, however. It is the PE file, containing the IL, that can be distributed and placed with the CLR running on the .NET Framework on any operating system for which the .NET Framework exists, because the IL is platform independent. When you run the IL, however, it is compiled to native code for that platform. Therefore, you are still running native code; you are not going back to the days of interpreted code at all. The compilation to native code occurs via another tool of the .NET Framework: the Just-In-Time (JIT) compiler.

With the code compiled, it can run within the Framework and take advantage of low-level features such as memory management and security. The compiled code is native code for the CPU on which the .NET Framework is running, meaning that you are indeed running native code instead of interpreted code. A JIT compiler will be available for each platform on which the .NET Framework runs, so you should always be getting native code on any platform running the .NET Framework. Remember, today this is just Windows, but this could change in the future.

NOTE

It is still possible to call operating system-specific APIs, which would, of course, limit your application to just that platform. That means it is still possible to call Windows APIs, but then the code would not be able to run within the .NET Framework on a non-Windows machine. At this point in time, the .NET Framework exists only on the Windows platform, but this will probably change in the future.

Executing Code

Interestingly, the JIT compiler doesn't compile the entire IL when the component is first called. Instead, each *method* is compiled the first time it is called. This keeps you from having to compile sections of code that are never called. After the code is compiled, of course, subsequent calls use the compiled version of the code. This natively compiled code is stored in memory in Beta 2. However, Microsoft has

provided a PreJIT compiler that will compile all the code at once and store the compiled version on disk, so the compilation will persist over time. This tool is called `ngen.exe` and can be used to precompile the entire IL. If the CLR cannot find a precompiled version of the code, it begins to JIT compile it on-the-fly.

After the code starts executing, it can take full advantage of the CLR, with benefits such as the security model, memory management, debugging support, and profiling tools. Most of these benefits will be mentioned throughout the book.

Assemblies

One of the new structures you will create in VB.NET is the assembly. An *assembly* is a collection of one or more physical files. The files are most often code, such as the classes you build, but they could also be images, resource files, and other binary files associated with the code. Such assemblies are known as *static* assemblies because you create them and store them on disk. *Dynamic* assemblies are created at runtime and are not normally stored to disk (although they can be).

An assembly represents the unit of deployment, version control, reuse, and security. If this sounds like the DLLs you have been creating in Visual Basic for the past six years, it is similar. Just as a standard COM DLL has a type library, the assembly has a *manifest* that contains the metadata for the assembly, such as the classes, types, and references contained in the IL. The assembly often contains one or more classes, just like a COM DLL. In .NET, applications are built using assemblies; assemblies are not applications in their own rights.

Perhaps the most important point of assemblies is this: All runtime applications must be made up of one or more assemblies.

The Assembly Manifest

The manifest is similar in theory to the type library in COM DLLs. The manifest contains all the information about the items in the assembly, including what parts of the assembly are exposed to the outside world. The manifest also lists the assembly's dependencies on other assemblies. Each assembly is required to have a manifest.

The manifest can be part of a PE file, or it can be a standalone file if your assembly has more than one file in it. Although this is not an exhaustive list, a manifest contains

- Assembly name
- Version
- Files in the assembly
- Referenced assemblies

In addition, a developer can set custom attributes for an assembly, such as a title and a description.

An End to DLL Hell?

One of the great benefits of COM was supposed to be an end to DLL Hell. If you think back for a moment to the days of 16-bit programming, you'll remember that you had to distribute a number of DLLs with a Windows application. It seemed that almost every application had to install the same few DLLs, such as `Ctrl3d2.dll`. Each application you installed might have a slightly different version of the DLL, and you ended up with multiple copies of the same DLL, but many were different versions. Even worse, a version of a particular DLL could be placed in the `Windows\System` directory that then broke many of your existing applications.

COM was supposed to fix all that. No longer did applications search around for DLLs by looking in their own directories, and then search the Windows path. With COM, requests for components were sent to the registry. Although there might be multiple versions of the same COM DLL on the machine, there would be only one version in the registry at any time. Therefore, all clients would use the same version. This meant, however, that each new version of the DLL had to guarantee compatibility with previous versions. This led to interfaces being immutable under COM; after the component was in production, the interface was never supposed to change. In concept that sounds great, but developers released COM components that broke binary compatibility; in other words, their components modified, added, or removed properties and methods. The modified components then broke all existing clients. Many VB developers have struggled with this exact problem.

The .NET Framework and the CLR attempt to address this problem through the use of assemblies. Even before .NET, Windows 2000 introduced the capability to have an application look in the local directory for a DLL, instead of going to the registry. This ensured that you always had the correct version of the DLL available to the application.

The runtime carries this further by allowing components to declare dependencies on certain versions of other components. In addition, multiple versions of the same component can be run simultaneously in what Microsoft calls *side-by-side instancing* or *side-by-side execution*.

The Global Assembly Cache (GAC)

Even though components in .NET do not have to be registered, there is a similar process if you have an assembly that is to be used by multiple applications. The CLR actually has two caches within its overall code cache: the download cache and the global assembly cache (GAC). An assembly that will be used by more than one application is placed into the global assembly cache by running an installer that

places the assembly in the GAC. If an assembly is not in the local directory and not in the GAC, you can have a codebase hint in a configuration file. The CLR then downloads the assembly, storing it in the download cache and binding to it from there. This download cache is just for assemblies that have to be downloaded, and will not be discussed further.

The GAC is where you place a component if you want multiple applications to use the same component. This is very similar to what you have with registered COM components in VB6.

Placing assemblies in the GAC has several advantages. Assemblies in the GAC tend to perform better because the runtime locates them faster and the security does not have to be checked each time that the assemblies are loaded. Assemblies can be added to or removed from the GAC only by someone with administrator privileges.

Where things get interesting is that you can actually have different versions of the same assembly in the GAC at the same time. Notice that I avoided saying, “registered in the GAC” because you aren’t placing anything in the registry. Even if a component is running in the GAC, you can add another version of the same component running alongside it, or you can slipstream in an emergency fix. This is all based on the version number and you have control over whether an update becomes a newly running version or merely an upgrade to an existing version.

Assemblies can be placed in the GAC only if they have a shared name. Assemblies and the GAC will be discussed in more detail in Chapter 4.

The Common Type System

The Common Type System specifies the types supported by the CLR. The types specified by the CLR include

- **Classes**—The definition of what will become an object; includes properties, methods, and events
- **Interfaces**—The definition of the functionality a class can implement, but does not contain any implementation code
- **Value Types**—User-defined data types that are passed by value
- **Delegates**—Similar to function pointers in C++, delegates are often used for event handling and callbacks

The type system sets out the rules that language compilers must follow to produce code that is cross-language compatible. By following the type system, vendors can produce code that is guaranteed to work with code from other languages and other compilers because all languages are consistent in their use of types.

Classes

Most Visual Basic developers are familiar with classes. Classes are definitions or blueprints of objects that will be created at runtime. Classes define the properties, methods, fields, and events of objects. If the term *fields* is new to you, it simply means public variables exposed by the class; fields are the “lazy way” to do properties. Together, properties, methods, fields, and events are generically called *members* of the class.

If a class has one or more methods that do not contain any implementation, the class is said to be *abstract*. In VB.NET, you cannot instantiate abstract classes directly; instead, you must inherit from them. In VB6, it was possible to create a class that was just method definitions and then to use the `Implements` keyword to inherit the interface. You could actually instantiate the interface in VB6, but because it did not have any implementation code, there was no point in doing so.

In VB.NET, you can create a class that has implementation code instead of just the interface, and then mark the class as abstract. Now, other classes can inherit from that abstract class and use the implementation in it or override the implementation as needed. These are new concepts to VB developers. In the past, VB had only interface inheritance, but VB.NET has “real” inheritance, known as *implementation inheritance*.

In VB.NET, interfaces are separate from classes. In VB6, you created interfaces by creating classes with method definitions, but no implementation code inside those methods. You will see more on interfaces in the next section, but realize that although a VB.NET class can implement any number of interfaces, it can inherit from only one base class. This will be examined in more detail throughout the book.

Classes have a number of possible characteristics that can be set, and that are stored in the metadata. In addition, members can have characteristics. These characteristics include such items as whether or not the class or member is inheritable. These will be discussed in more detail in Chapter 4.

Interfaces

Interfaces in VB.NET are like the interfaces in previous versions of VB: They are definitions of a class without the actual implementation. Because there is no implementation code, you cannot instantiate an interface, but must instead implement it in a class.

There is one exception to the “no implementation code in an interface” rule: In VB.NET, you can define what are called *static* members. These can have implementation code, and you will see more about them in Chapter 4.

Value Types

In .NET languages, a standard variable type, such as an integer, is native to the language, and it is passed by value when used as an argument. Objects, on the other hand, are always passed by reference. However, a value type is a user-defined type that acts much like an object, but is passed by value. In reality, value types are stored as primitive data types, but they can contain fields, properties, events, and both static and nonstatic methods. Value types do not carry the overhead of an object that is being held in memory.

If this seems confusing, think about enums. Enumerations are a special type of value type. An enum simply has a name and a set of fields that define values for a primitive data type. Enums, however, cannot have their own properties, events, or methods.

Delegates

Delegates are a construct that can be declared in a client. The delegate actually points to a method on a particular object. Which method it points to on which object can be set when the instance of the delegate is created at declaration. This allows you to define calls to various methods in different objects based on logic in your code.

Delegates are most often used to handle events. Using delegates, you can pass events to a centralized event handler. Due to the small size of this book, delegates will not be examined in more detail.

The .NET Framework Class Library

The .NET Framework provides a number of types that are already created and ready for use in any language that targets the Common Language Runtime. These types include such items as the primitive data types, I/O functions, data access, and .NET Framework security.

Perhaps one of the biggest changes in the way developers will work with VB.NET is the entire area of namespaces. Namespaces will be covered in Chapter 3, “Major VB.NET Changes,” but it is important to understand that the .NET Framework provides a host of utility classes and members, organized within a hierarchy called a *namespace*. At the root of the hierarchy is the System namespace. A namespace groups classes and members into logical nodes. This way, you can have the same name for a method in more than one namespace. The `Left()` method could therefore exist in the `System.Windows.Forms` namespace and the `Microsoft.VisualBasic` namespace.

One advantage of namespaces is that similar functions can be grouped within the same namespace, regardless of the assembly in which they are physically located.

Any language targeting the runtime can use the System namespaces. For example, if you need to perform data access, you do not set a reference to the ADO.NET component. Instead, you reference, or import, the System.Data namespace. Often, you will see the following line:

```
Imports System.Data
```

This does not import all the methods and cause code bloat, however. Instead, it instructs the compiler to treat the methods and types within the namespace as part of your project's own namespace, so instead of having to write this:

```
System.Data.SqlClient()
```

You can simply make a call to

```
SqlClient()
```

There are many other System namespaces. System namespaces include the functionality for security, threading, text and binary I/O, and Web services. How to use various System namespaces is introduced throughout the book.

Don't let the term *namespaces* scare you. It is one of the most fundamental changes in VB.NET, but it is a service provided by the runtime that gives common, rich functionality to any language built on the runtime. As long as you inherit from the namespaces provided to you by the runtime, your application can run on any platform that supports .NET. In this way, you are working on learning the environment as much as you are learning the language. That's one reason why this chapter is the first in the book.

Self-Describing Components

In traditional VB, compiled components created a type library that attempted to define what was in the component as far as classes, interfaces, properties, methods, and events. Communication occurred through a binary interface at the COM level. Unfortunately, one language could expect as parameters data types or structures that are not available to other languages, or that are at least difficult to implement. For example, C++ components often expect pointers or structures to be passed in, and this could be problematic if the calling program is written in Visual Basic. The .NET Framework attempts to solve this by compiling additional data into all assemblies. This additional data is called *metadata* and allows compiled components to interact seamlessly. Couple this with a common type system so that all runtime-compatible languages share the same types, and you can see that cross-language compatibility is enhanced.

The metadata that is stored in the components is binary, and contains all types, members, and references in that file or assembly. The metadata is compiled into the PE file, but when the file is used at runtime, the metadata is moved into memory so that it can be accessed more quickly.

One of the most important things to understand is that it is the metadata that allows the runtime to find and execute your code. The metadata is also used by the runtime to create a valid binary native code version when the MSIL is compiled. The runtime also uses metadata to handle the messy details of memory cleanup and security.

When it comes to the benefits of the metadata for a developer, realize that there is so much information in the metadata that you can inherit from a PE created in a different language because the metadata provides such a rich level of detail. This means that you are actually inheriting from a compiled component, not an IDL file as was required previously. In fact, because metadata is compiled into the PE or assembly, you no longer need IDL files or type libraries. All the information is now contained in the PE.

By default, the metadata contains the following information:

- PE or assembly identity: name, version, culture, public key
- Dependencies on other assemblies
- Security roles and permissions
- Exported types

Each type has the following metadata:

- Name, visibility, base class, implemented interfaces
- Members

In addition, you can extend the metadata using attributes. These are created by the developer and allow additional information to be placed in the metadata.

Cross-Language Interoperability

If you've been building COM components for a while, you know that one of the great promises of COM is that it is language independent. If you build a COM component in C++, you can call it from VB, and vice versa. However, to reach that point, your code had to be compiled to a COM standard. Much of this was hidden from the VB developer, but your component had to implement the IUnknown and IDispatch interfaces. Without these interfaces, they would not have been true COM components. COM is only giving you cross-language interoperability at the binary level, however. This means that you can only take advantage of this interoperability at run time.

Now, however, the CLR gives you much better language interoperability. Not only can you inherit classes from one PE written in language A and use them in language B, but debugging now works across components in multiple languages. This way, you can step through the code in a PE written in C# and jump to the base class that was

written in VB.NET. This means that your cross-language interoperability is happening at design time and run time, not just the run time given to you by COM. In addition, you can raise an error (now called an *exception*) in one language and have it handled by a component in another language. This is significant because now developers can write in the language with which they are most comfortable, and be assured that others writing in different languages will be able to easily use their components.

The Catch

This all sounds great, and you are probably getting excited about the possibilities. There is a catch, however: To make use of this great cross-language interoperability, you must stick to only those data types and functions common to all the languages. If you're wondering just how you do that, the good news is that Microsoft has already thought about this issue and set out a standard, called the Common Language Specification, or CLS. If you stick with the CLS, you can be confident that you will have complete interoperability with others programming to the CLS, no matter what languages are being used. Not very creatively, components that expose only CLS features are called *CLS-compliant components*.

To write CLS-compliant components, you must stick to the CLS in these key areas:

- The public class definitions must include only CLS types.
- The definitions of public members of the public classes must be CLS types.
- The definitions of members that are accessible to subclasses must be CLS types.
- The parameters of public methods in public classes must be CLS types.
- The parameters of methods that are accessible to subclasses must be CLS types.

These rules talk a lot about definitions and parameters for public classes and methods. You are free to use non-CLS types in private classes, private methods, and local variables. Even if you have a public class that you want to be CLS compliant, the implementation code *inside* that class does not have to be CLS compliant; as long as the definition is compliant, you are safe.

The CLS is still in flux, but the basics are well established. See Appendix A for the basics of the CLS.

Security

If you create a VB component today, your choices for implementing security are somewhat limited. You can use NTFS to set permissions on the file itself. You can

place it in MTS/COM+ Component Services and turn on role-based security. You can call it over DCOM and use DCOMCNFG to set permissions. You can always just code your own security.

One of the runtime's main benefits is that an entire security infrastructure is built right in. In fact, two major security models are set up in the .NET Framework: code access security and role-based security.

Code Access Security (CAS)

This security does not control who can access the code; rather, it controls what the code itself can access. This is important because it allows you to build components that can be trusted to varying degrees. If you build a VB component today and want to perform database access, you are free to call ADO and connect to a database (provided, of course, that you have a valid user ID and password). With .NET, however, you can actually specify, with the tools in the .NET Framework, what actions your component can and, more importantly, cannot perform. This has the benefit of preventing others from using the code in ways that you did not intend.

Perhaps the main benefit of CAS is that you can now trust code that is downloaded from the Internet. Security can be set up so that it becomes impossible for the code to perform any mischievous actions. This would prevent most of the macro viruses that are spread via e-mail today.

Role-Based Security

Role-based security is the same type of security you get when you use MTS or COM+ Component Services. In .NET, the Framework determines the caller, called a *principal*, and checks the principal's individual and group permissions. Unlike COM/COM+ role-based security, however, .NET cannot make an assumption that the user will have a valid NT user account and token to pass in. Therefore, .NET allows for generic and custom principals, as well as standard Windows principals. You can define new roles for each application if you want.

Summary

You have just driven by some of the most important .NET features at about 60 miles per hour. If some of the terminology seems foreign to you, do not worry. Some of it will be covered in more detail in this book. Terminology that isn't covered in more detail in this book will be covered in other books, but this book will cover the basics of what you need to know to get started using VB.NET.

For those questioning whether to move to .NET, it is important to understand the benefits .NET gives you. Having a unified debugger, gaining free threading, and

finally having full inheritance are major leaps forward. The runtime also gives you a nice security framework, and you don't have to worry about registering components anymore. Web Services allows you to easily create services that are consumable over a standard HTTP connection.

As you work with VB.NET, understand that you are also working closely with the .NET Framework. It is hard to separate the two, and the more you understand about the Framework, the better VB.NET developer you will be.



CHAPTER 2

Your First VB.NET Application

It's time to jump in and start working with VB.NET. First, you need to learn a little bit about the new IDE. The new VB.NET IDE might look somewhat familiar to you, but there are some significant changes that make it a more useful environment. However, these changes can be frustrating to experienced VB developers because many of the keystrokes have changed, windows have different names, and the debugging tools work differently. VB.NET is part of Visual Studio.NET (or VS.NET), which finally consolidates all the development languages into one place: VB.NET, C++.NET, and C#. You can even create a single solution, containing multiple projects, in which the individual projects are written in separate languages.

The Start Page

The very first time you start Visual Studio.NET, you are taken to a screen that allows you to configure the IDE. That screen is the My Profile page discussed later in the chapter. After your first visit to the My Profile page, all subsequent starts of Visual Studio.NET begin with the Start Page, as shown in Figure 2.1. The start page contains a number of sections, as indicated by the links along the left side. These sections are

- **Get Started**—This option allows you to open a recent or existing project, or create a new one. No recent projects are listed on the Get Started area shown in Figure 2.1. As you create projects in VB.NET, this area will display the four most recently opened projects. This area also contains links to open an existing

project, to create a new project, and to log a bug report. Expect this last option to disappear after the final product is released.

- **What's New**—This option covers new language features in Visual Studio.NET, including each individual language and the Visual Studio.NET environment. There are links to topics in the help files on new features for the VS.NET languages, the .NET SDKs, and a link to check for VS.NET upgrades.
- **Online Community**—This provides links to the Microsoft newsgroups. These are newsgroups accessible with any newsreader, but they are served from Microsoft's news server (msnews.microsoft.com) and not normal Usenet news servers. This page appears blank in some of the interim builds of VS.NET, but expect it to be fixed for the released version of Beta 2.
- **Headlines**—Provides a place for links to news about .NET. In some interim builds of Beta 2, this page simply generates an error. However, by the time Beta 2 is released, expect this page to include a link to MSDN Online at least.
- **Search Online**—Searches the MSDN Online library.
- **My Profile**—This screen lets you choose the overall layout of Visual Studio.NET. You can set the keyboard mappings to the same scheme as in previous versions of Visual Studio, such as Visual Basic 6. You can also set the window layout to match previous versions of Visual Studio projects, and you can automatically filter help using the profile. Throughout this book, I will assume that the profile is set to Visual Studio Developer, and that all other settings are left at their default values.

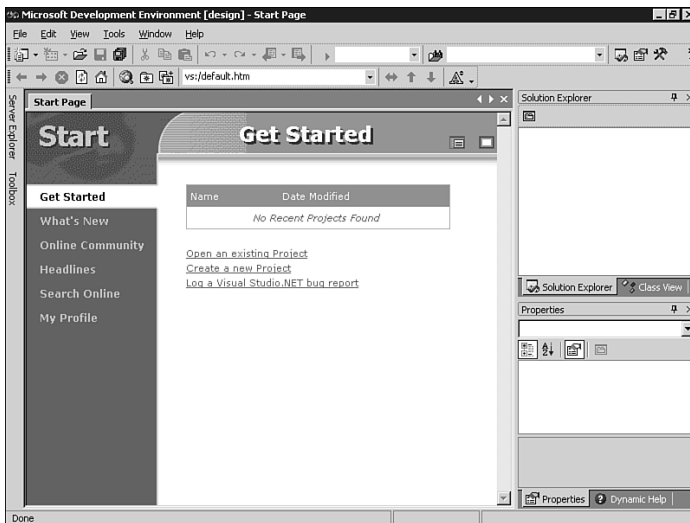


Figure 2.1

The Visual Studio.NET Start Page.

NOTE

The Start Page is HTML and will likely change by the time VS.NET is released.

Help is now more tightly integrated into Visual Studio. To see this, click the What's New in Visual Basic link on the What's New page. Notice that when you do this, the help is added into the same window that held the Start Page, as shown in Figure 2.2. As you will see later, it is even possible to have help running continuously while you work, searching for topics associated with whatever you are working on at the moment.

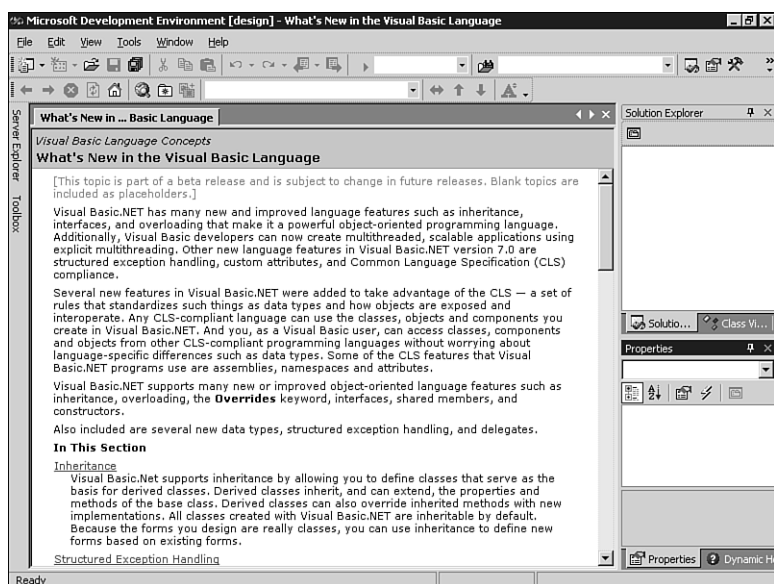


Figure 2.2

Help is shown within the IDE instead of as a separate window.

Creating a New Project

Return to the Start Page, identifiable by the tab at the top, and click on Get Started. Now, click on the Create New Project link. Doing so opens the New Project dialog shown in Figure 2.3. Notice that there are different languages you can use to create applications in Visual Studio.NET. This book will focus on only the Visual Basic projects.

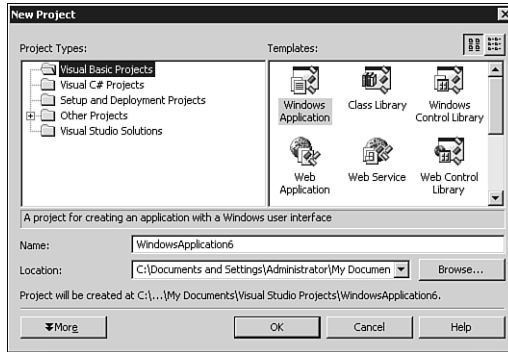


Figure 2.3

The New Project dialog box.

If you examine the Visual Basic project types, you'll see that many of them are different from what you are used to with VB6. Some of the major project types are

- **Windows Application**—This is a *standard executable*, in VB6 terminology. It is the way to create applications with a Windows interface, using forms and controls. This is as close to “your father’s VB” as you’ll get in VB.NET.
- **Class Library**—This project type allows you to create classes that will be used in other applications. Think of it as similar to the COM components that you have been building, which VB6 called the ActiveX DLL and ActiveX EXE project types.
- **Windows Control Library**—This project type is for creating what used to be called ActiveX controls. This type allows you to create new controls to be used in Windows applications.
- **Web Application**—Goodbye, Visual InterDev. Goodbye, server-side, interpreted scripting languages for Active Server Pages. Visual Basic now has Web Application projects, which use ASP.NET to create dynamic Web applications. These projects allow you to create HTML, ASP.NET, and VB files. You will now code your Web applications using a powerful, event-driven model instead of the request/response model.
- **Web Service**—If you’ve used VB6 to create COM components and then made them available over HTTP with SOAP, you understand the concept of Web Services. Web Service projects are components that you make available to other applications via the Web; the underlying protocol is HTTP instead of DCOM, and you pass requests and receive responses behind the scenes using XML. Some of the major promises of Web Services are that they are all standards-based and are platform independent. Unlike DCOM, which was tied to a COM (that is, Windows) infrastructure, Web Service projects can be placed on

any platform that supports .NET, and can then be called by any application using just the HTTP protocol.

- **Web Control Library**—As with Web Service projects, there's no exact match back in VB6 for the Web Control Library projects. Thanks to the new Web Application projects in VB.NET, you can add controls to Web pages just as you would in a standard Windows Application project, but VB.NET makes them HTML controls at runtime. You can design your own controls that can then be used by Web applications.
- **Console Application**—Many of the Windows administrative tools are still console (or command-line, or DOS) applications. Previously, you didn't have a good way to create console applications in VB, and instead had to rely on C++. Now, console applications are natively supported by VB.NET.
- **Windows Service**—As with console applications, there was no good way to create Windows services in previous versions of VB. Windows services, of course, are programs that run in the background of Windows, and can automatically start when the machine is booted, even if no one logs in.

Those are the basic types of applications you can create. You can also create an Empty project (for Windows applications, class libraries, and services) or an empty Web Application (for Web applications).

Examining the IDE

If you are still on the New Project dialog, choose to create a Windows Application. Name it LearningVB and click the OK button. After a time, a new project will open up. Notice this adds a Form1.vb tab to the main window. In the main window, you now have an empty form. This is commonly referred to as the Form Designer. In fact, there are various types of designers that can get loaded into this work area. So far, you should feel pretty much at home.

One difference that has occurred, perhaps without you noticing, is that the files created have already been saved on your machine. In VB, you could create a project, do some quick coding, and then exit without saving, and nothing was stored on your machine. Now, however, the files are saved at creation, so each project you create does store something on the hard drive. Visual InterDev developers are used to this, but straight VB developers will see this as a change.

If you look at the right side of the IDE, you'll see a window called the Solution Explorer. This works like the Project Explorer in VB6, showing you the projects and files you have in the current *solution* (what VB6 called a *group*). The Solution Explorer currently lists the solution name, the project name, and all the forms and modules. Right now, there is just one form, named Form1.vb. In addition, the window will have a file called AssemblyInfo.vb, which is part of the metadata that will

be compiled into this assembly. You also see a new node, called References, in the list. If you expand the References node, you will see all the references that are already available to your project when you start. You can see the Solution Explorer in Figure 2.4. For now, don't worry too much about the references.

A second tab, labeled Class View, exists at the bottom of the Server Explorer window. If you click on the Class View tab, you will see the LearningVB project listed. If you expand the project node, you will see the namespaces for this project listed. It so happens that there is just one namespace at the moment, and by default, it is the same as the project name. Expand the LearningVB namespace and you will see that just Form1 is listed below it. Expand Form1 and you will see some of the form's methods, as well as a node for Bases and Interfaces. If you expand that node and the Form node under it, you will see a long list of properties, methods, and events available to you in the form. You can see a small part of this list in Figure 2.5. Again, don't worry about these for now. Just understand that this is certainly different from anything you saw in Visual Basic 6.

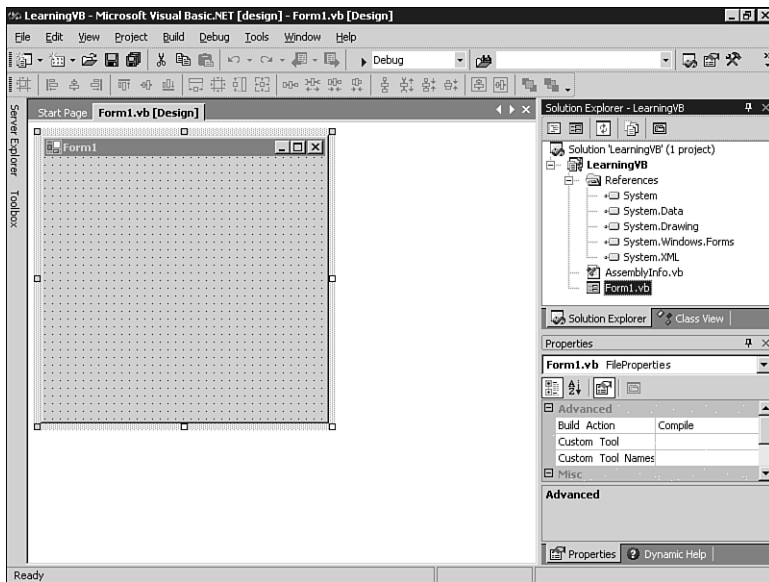


Figure 2.4

The Solution Explorer window.

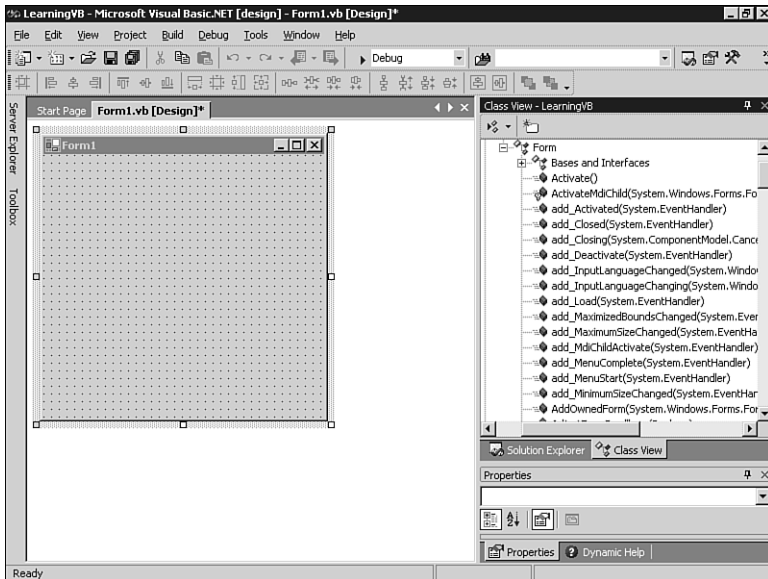


Figure 2.5

The new Class View window.

If you want to know more about what one of those properties or methods can do for you, it's easy to look it up in the Object Browser. For example, scroll down the Bases and Implemented Interfaces list until you find `FocusInternal`. Right-click on it and choose `Browse Definition`. You will see that the Object Browser opens as a tab in the main work area, and that you are on the definition for the `FocusInternal` method. You can see that `FocusInternal` returns a `Boolean`. Figure 2.6 shows what this should look like in the IDE.

Below the Server Explorer/Class View windows is something that will be quite familiar to you: the Properties window. If you close the Object Browser and go back to the `Form1.vb [Design]` tab, you should see the properties for `Form1`. You might actually have to click on the form for it to get the focus. After the form has the focus, you will see the properties for the form. Most of these properties will look very familiar to you, although there are some new ones. What some of these new properties are, and what they can do for you, will be examined later in this chapter.

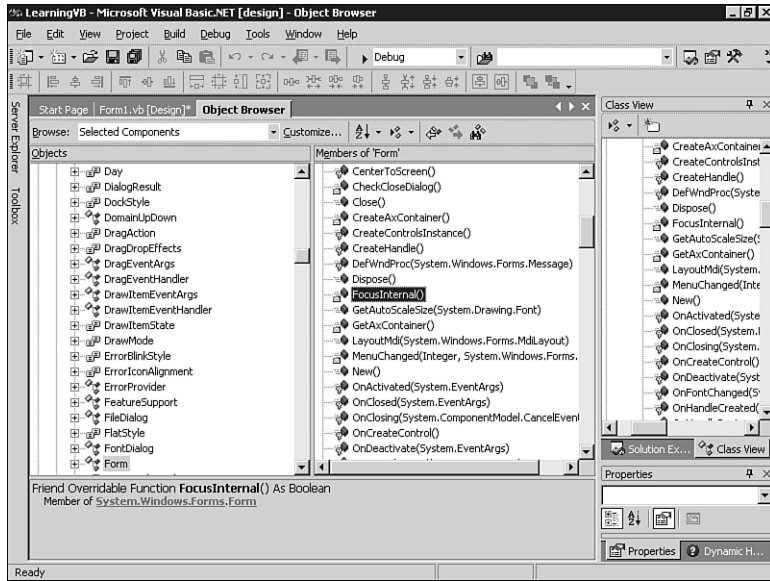


Figure 2.6

The Object Browser is now a tab in the main work area.

NOTE

The Properties window is sorted by category instead of alphabetically, by default. Therefore, it can be difficult to find certain properties. In the toolbar for the Properties window are buttons that let you switch between a categorized and alphabetical listing of the properties.

In the same area as the Properties window is a tab labeled Dynamic Help. This is a new Visual Studio.NET feature that allows you to have constantly updating help while you work. It monitors what you are doing in the IDE and provides a list of help topics for your current activity. For example, with only the form open and active, click on the Dynamic Help tab. You will get a list of help topics associated with the Forms Designer, how to add controls to a form, and similar topics. Figure 2.7 shows what this list will look like. Feel free to click around on various IDE windows and watch the dynamic help change. Just realize that the Dynamic Help feature does consume some resources. On more powerful computers, this will not be a big deal, but it could be a problem on machines with slow processors or too little RAM.

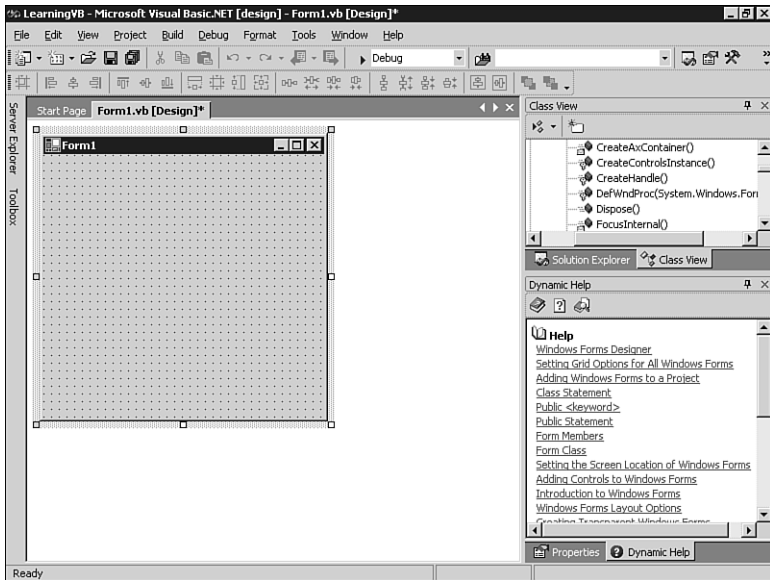


Figure 2.7

The Dynamic Help window.

Along the left side of the IDE are two sideways tabs. The first tab is labeled Server Explorer and the second tab is labeled Toolbox. To get either to appear, just hover the mouse over the tab.

The Server Explorer is a new feature to the IDE. It allows for discoverable services on various servers. For example, if you want to find machines that are running Microsoft SQL Server, there is a SQL Server Databases node under each server. In Figure 2.8, you can see that I have one server registered to my Server Explorer: laptop. This server does have SQL Server, and you can see a list of the databases. Within Server Explorer, you can perform the actions that you used to perform with the Data View window. You can view the data in a table, you can drop or create tables, you can create, drop, and edit stored procedures, and all the other activities you performed with the Data View window in VB6. If you did much with Visual InterDev, these tools will feel even more comfortable to you.

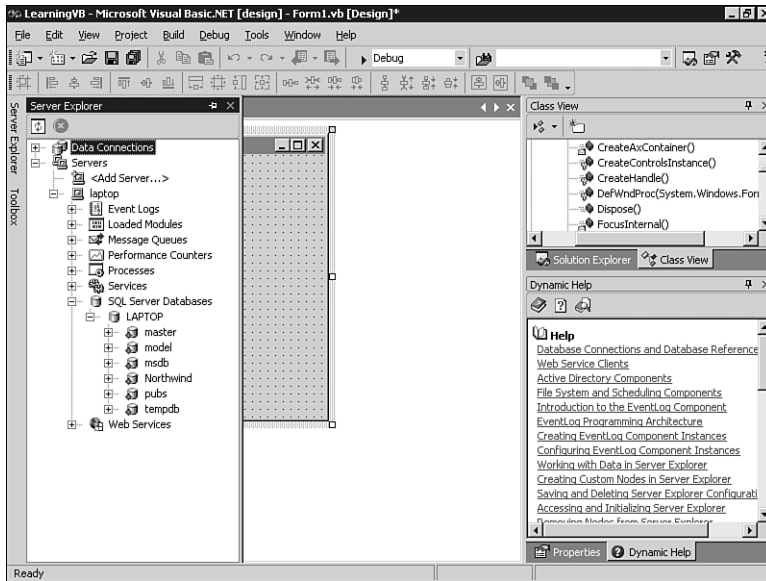


Figure 2.8

The new Server Explorer window.

You can also use the Server Explorer to find and connect to message queues, monitor another machine with the performance counters, and find Web services on other machines. The Server Explorer is a handy tool, with many of the administrative functions you need to perform all in one place.

The second sideways tab is Toolbox. The Toolbox is just what its name implies: It is where you find the controls that you want to put on a form. Notice, however, that this Toolbox is a little different from the Toolbox in VB6. In fact, it looks much more like the Toolbox from Visual InterDev. The Toolbox is split into horizontal tabs. Figure 2.9 shows the Windows Forms tab open, but there are also Data, Components, Clipboard Ring, and General tabs. Don't worry too much about those other tabs for now.

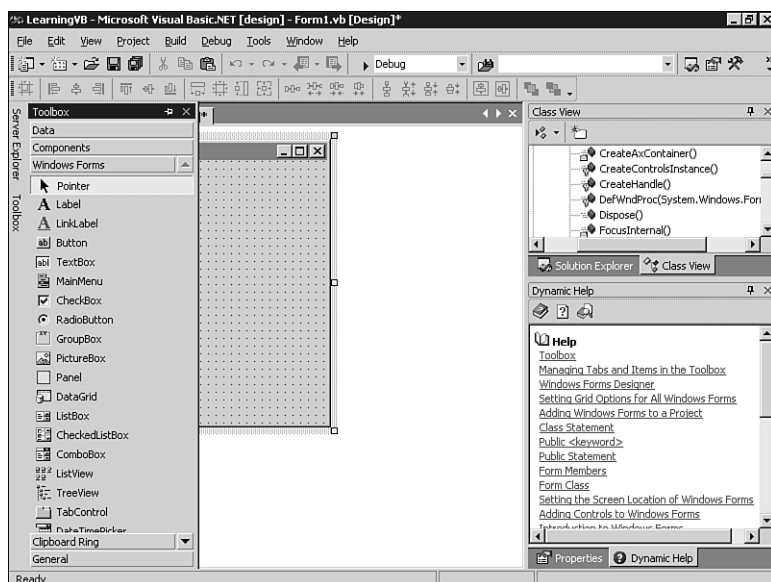


Figure 2.9

The improved Toolbox.

Creating Your First VB.NET Application

You have an open project with one form in it. So far, you haven't done anything to it, so you now need to create the obligatory Hello World application. I remember when Microsoft was running around showing the world how you could create a Hello World application in VB by typing just one line of code. Well, it's still that easy, but things certainly look different now.

Make sure that the form designer is the current tab in the work area, and open the Toolbox. Click and drag a button onto the form. Place it wherever you want. So far, this is just like VB6. Now, double-click on the button.

Double-clicking on the button causes the code window to open, just as it did in VB. However, in VB.NET, the code is added as a tab in the work area. The new tab is labeled Form1.vb. You have a lot of code in this window, and it's code that you haven't seen before. In fact, before typing any code at all, you have the code shown in Figure 2.10. Notice that I have turned on line numbers just for easier referencing. If you want to turn on line numbers as well, go to Tools, Options. Expand the Text Editor node and choose Basic. Check the Line Numbers check box.

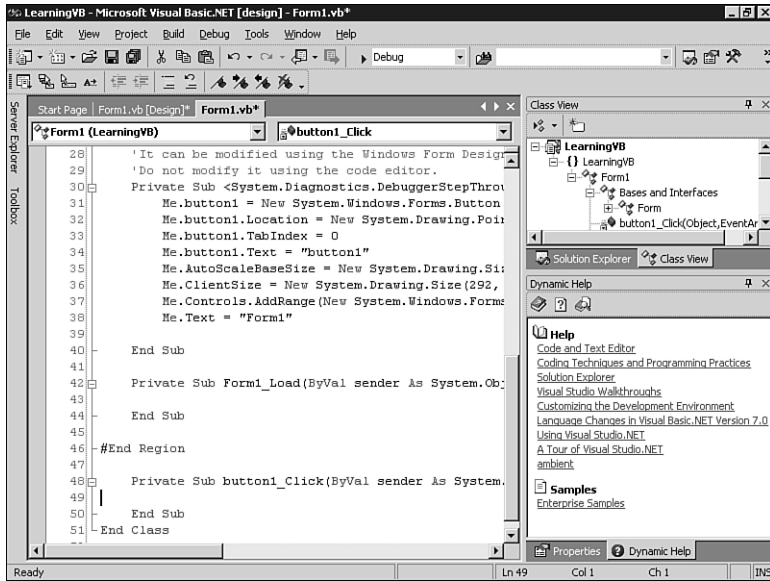


Figure 2.10

The code window shows a significant amount of code before you even start typing.

The first line of code shows that the form, Form1, is actually just a class. This is one of the biggest changes to VB.NET: Forms are truly classes. Why? Because forms are really just classes, and they have been forever. You might not have ever thought of it that way, but a form is just another class. When a form is displayed, you have just instantiated a class.

The next line is an Inherits statement. Any form you create just inherits from the base Form object. You are calling one of the .NET System namespaces; in this case, System.Windows.Forms.Form. It is this namespace that gives you the base functionality of forms, such as the methods and events you will access in your code. The Inherits statement is your first indication of true inheritance in VB.NET. You can inherit from this base class and then extend it if you want to do so.

The next piece inside the class is a public sub named New. Notice that it calls a sub named InitializeComponent. The InitializeComponent routine is created for you by VS.NET. The New routine is similar to the Form_Load event procedure you got used to in VB6. Notice that you need to add any of your own code after the call to InitializeComponent.

After the New sub is a Dispose sub, much like the Form_Unload found in VB6. This is the area for the cleanup of anything you might have open. Don't worry about the Overrides keyword for now; it will be covered when you learn more about inheritance.

Next is where things start to get interesting. A gray line runs down the left side of the code, with minus signs at the Class and Sub definitions. The next line has some text that says Windows Form Designer generated code with a minus sign next to it. You have an expanded block of code that was generated for you. One thing that the VB.NET editor lets you do is to collapse or expand blocks of code. This can make it easier to see a cleaner view of your code. If you collapse the code at line 4 (at least that's the line in Figure 2.10), you will collapse the code region containing New, Dispose, and the InitializeComponent sub. In fact, the entire region of code is shown in Listing 2.1.

Listing 2.1 Code Generated for You by the Windows Form Designer

#Region " Windows Form Designer generated code "

```
Public Sub New()
    MyBase.New()

    'This call is required by the Windows Form Designer.
    InitializeComponent()

    'Add any initialization after the InitializeComponent() call
End Sub

'Form overrides dispose to clean up the component list.
Public Overrides Sub Dispose()
    MyBase.Dispose()
    If Not (components Is Nothing) Then
        components.Dispose()
    End If
End Sub
Private WithEvents button1 As System.Windows.Forms.Button

'Required by the Windows Form Designer
Private components As System.ComponentModel.Container

'NOTE: The following procedure is required by the _
        Windows Form Designer
'It can be modified using the Windows Form Designer.
'Do not modify it using the code editor.
Private Sub <System.Diagnostics.DebuggerStepThrough()> _
    InitializeComponent()
        Me.button1 = New System.Windows.Forms.Button()
        Me.button1.Location = New System.Drawing.Point(104, 72)
        Me.button1.TabIndex = 0
        Me.button1.Text = "button1"
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(292, 273)
        Me.Controls.AddRange(New System.Windows.Forms.Control() _
```

Listing 2.1 continued

```
{Me.button1})  
Me.Text = "Form1"  
End Sub  
#End Region
```

As you can see, the `InitializeComponent` routine is in this block, and it is what sets up the controls on the form. Notice that the button has properties set, such as the location and size, and then is added to a controls collection.

Finally, below this region of code is the routine to handle the click event on the button. This looks different from what you have seen before as well. In fact, if you started a new Standard EXE project in VB6, added a button to the form, and then double-clicked the button, your entire project would show just this code:

```
Private Sub Command1_Click()  
  
End Sub
```

However, in VB.NET, you will see a lot more code. In fact, the `Click` event is not even handled until line 57, at least in this example. The declaration of the event procedure is quite different as well. In VB.NET, the event procedure looks like this:

```
Private Sub button1_Click(ByVal sender As System.Object, _  
    ByVal e As System.EventArgs) Handles button1.Click  
End Sub
```

Again, don't worry about all the changes for now. Instead, type in the following line of code:

```
msgbox "Hello, World"
```

When you move off this line, you will notice that VB.NET automatically adds parentheses around your argument, leaving you with this:

```
MsgBox("Hello, World")
```

This is one of the first fundamental language changes: Subs and functions require parentheses around the argument. The `MsgBox` command is a function, but in VB6, you did not need parentheses around a parameter if you were ignoring the return value. In VB.NET, you always need the parentheses, so get used to adding them or at least get used to seeing VB.NET add them for you.

Now, it is time to run this rather exciting demonstration and see whether it works. You can click the Start button on the toolbar (it should look familiar) or you can go to the Debug menu and choose Start.

Form1 should load. Click on the button, and you should get a message box to pop up, with the text `Hello, World`. Notice that the title of the message box is the same as the project name, just as it was in VB6. Close the application and return to the IDE.

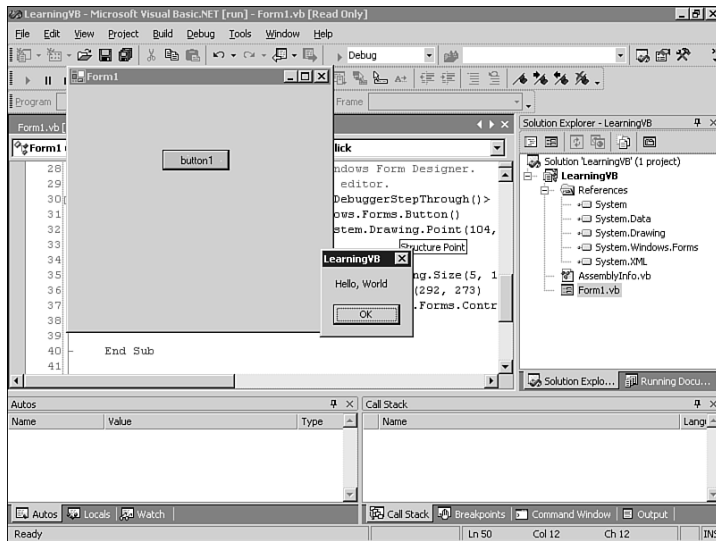


Figure 2.11

The Hello World application running.

Notice that a new window is showing. At the bottom of the screen, an Output window is now open. Figure 2.12 shows this window. Currently, it shows you all the debug statements. You did not put any `debug.print` statements into your code, but certain actions by the compiler automatically put comments into the Debug window.

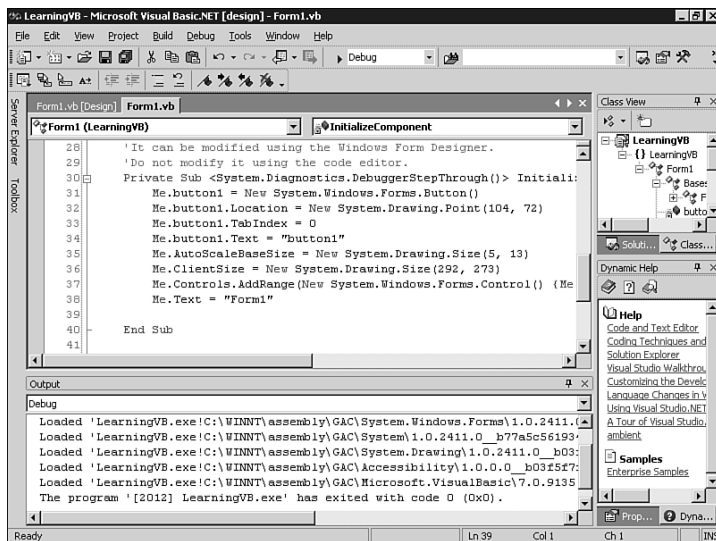


Figure 2.12

The new Debug window.

Before moving on, you might want to change the title of the message box. Return to the one line of code you've written so far and modify it to look like this:

```
MsgBox("Hello, World", , "My first VB.NET App")
```

Now that you have added a title to the message box, run the project again, and the title of your message box will be My first VB.NET App. If you think it doesn't get any better than this, hang on.

Windows Application Enhancements

Visual Studio.NET has added a variety of features to VB.NET to make the IDE more powerful and to enhance the functionality of Windows forms. There is improved support for features such as building menus, automatically adjusting to changes in the size of the text to display, better anchoring of controls for resized windows, and a much-improved mechanism for setting the tab order.

Resizing Controls Automatically

You can set certain controls to resize automatically based on what they need to display. This is easy to examine with a simple label. Open the Toolbox and add a label to your form. Drag it over to the left side of the form, so that it is not in line with the button.

Now, change some properties of that label. Change the `BorderStyle` to `FixedSingle`, and set `AutoSize` to `True`. Change the `Text` property to `This is a test`. You might have noticed that you changed the `Text` property, and not a `Caption` property as you did in VB versions 1–6. Welcome to another one of those little changes that might trip you up.

Now, modify the code for the `button1_Click` event. Remove the `MsgBox` call and add the following code:

```
label1().Text = "Put your hand on a hot stove " & _  
    "for a minute, and it seems like an hour. " & _  
    "Sit with a pretty girl for an hour, and " & _  
    "it seems like a minute. THAT'S relativity."
```

Before you run this, realize one more thing about the code. In VB6, it would have been legal to say this:

```
Label1 = "Put your hand on a hot stove..."
```

This would run just fine in VB6 because the default property of `Label1` was the `Caption` property. Well, in VB.NET, there is no such thing as a default property. Therefore, you always have to specify the property when you write your code.

Go ahead and run the project. A form will appear that is similar to what you see in Figure 2.13. Now, click Button1 and notice that the text in the label changed, which is no big surprise. However, notice that the label has grown to try to hold all the text. If you resize the form, you will see that the label has indeed grown to show all the text. Figure 2.14 shows this to be the case.

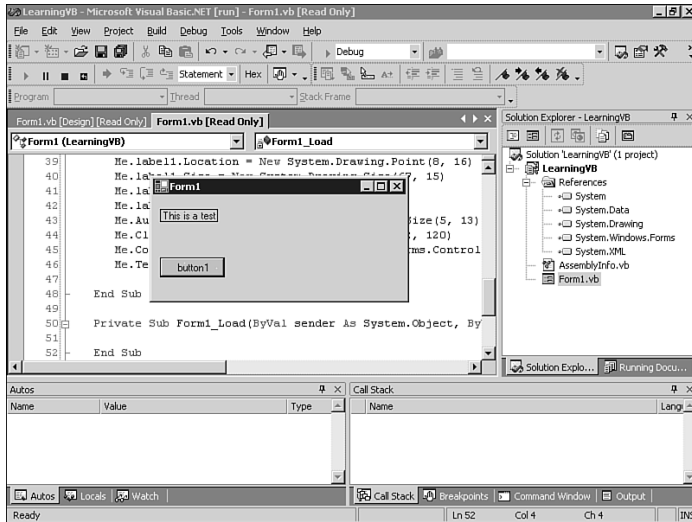


Figure 2.13

A form with a small amount of text in a label.

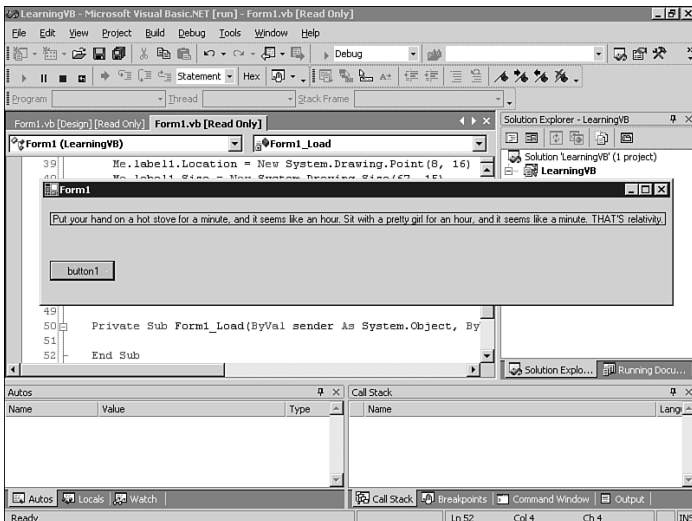


Figure 2.14

The same form, showing the label has grown to contain all the text.

If you're saying that this isn't a big deal, think of the code you had to write before. You had to check the length of the string, but you also had to be aware of what font was in use. A group of characters in Arial is not the same length as the same group of characters in Courier. Therefore, resizing could be a hit-or-miss proposition. The `AutoSize` property removes that guesswork for you.

Anchoring Controls to the Form Edges

How many times did you create a VB form and set the border style to fixed so that people couldn't resize it? If you placed a series of buttons along the bottom of the form, you didn't want people to resize the form and suddenly have this bottom row of buttons in the middle of the form.

VB.NET allows you to anchor controls to one or more sides. This can allow a control to move as the form is resized, so that it appears to stay in its proper place.

Back on `Form1`, delete the label you added, and remove the code inside the `Button1_Click` event procedure. Make the form slightly bigger and move `Button1` toward the lower-right corner. Now, add a `TextBox` to the form. Change the `TextBox`'s `Multiline` property to `True` and resize the text box so that it fills up most of the form, except for the bottom portion that now contains the button. Your form should now look something like the one shown in Figure 2.15.

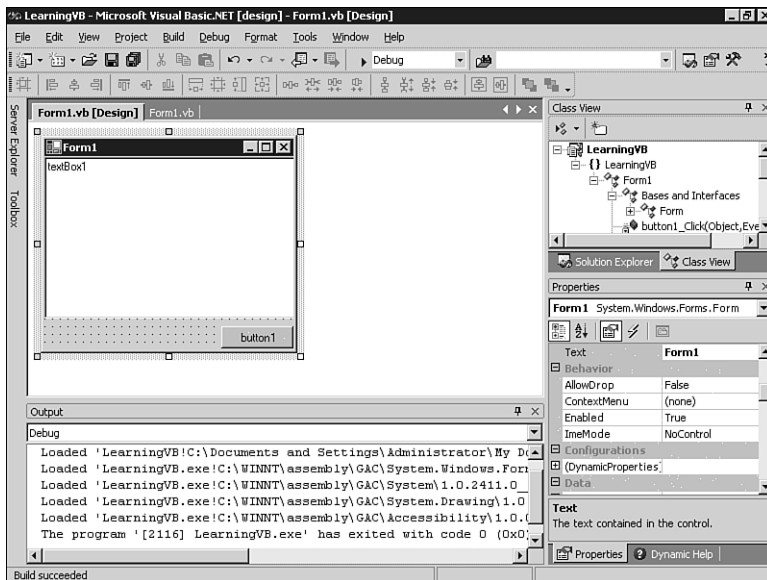


Figure 2.15

The form as it looks at design time.

Run the project. After the window is loaded, resize it by moving the lower-right corner down and to the right. You will end up with something that looks like Figure 2.16.

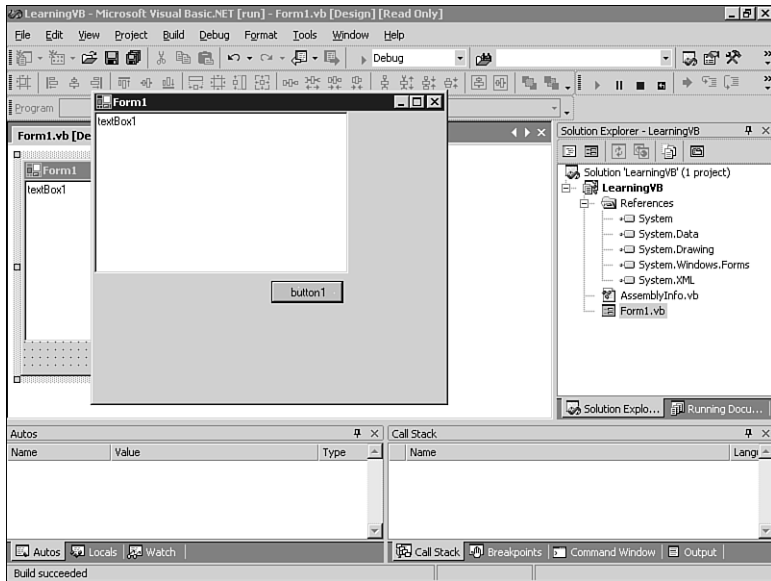


Figure 2.16

The form at runtime, showing what happens when the form is resized.

Obviously, this looks pretty strange, and it is exactly what you are used to with VB6. Your answer in VB6, short of a third-party ActiveX control, is to write a lot of code that runs when the form's `Resize` event is fired. You use that code to move the button and resize the text box. VB.NET has a better way of handling this, so stop the application and return to the IDE.

Make sure that you are on the `Form1.vb [Design]` tab and click on `Button1`. In the Properties window, scroll down and find the `Anchor` property. When you drop down this list, you get a strange box that shows a gray something in the middle, and four small rectangles around it. By default, the top and left rectangles are darkened, whereas the right and bottom rectangles are empty. The darkened rectangles indicate that right now, the button is anchored to the top and left sides of the form. As you resize the form, the button will stay the same distance from the top and left sides. That is not what you want, so click the top and left rectangles to clear them, and then click the bottom and right rectangles to select them. When you are done, the `Anchor` property should look like Figure 2.17. After you collapse the drop-down list, the `Anchor` property should be `Bottom, Right`.

Next, click on `TextBox1` and choose its `Anchor` property. Click on the bottom and right rectangles, but leave the left and top rectangles chosen. Now, the text box is tied

to all borders, and it will remain the same distance from them. The Anchor property for TextBox1 should now be Top, Bottom, Left, Right.

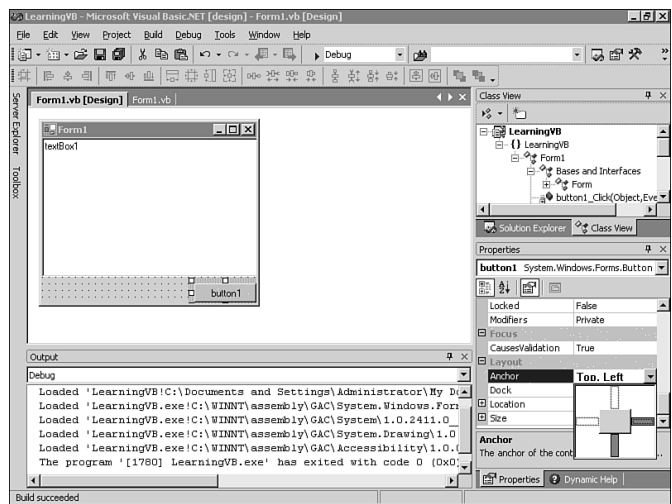


Figure 2.17
The Anchor property tool.

Run the project again. After the form is open, resize it. Notice that the button now stays in the lower-right corner, and the text box automatically resizes with the form. You can see this behavior in Figure 2.18. The resizing and movement were accomplished without writing a line of code.

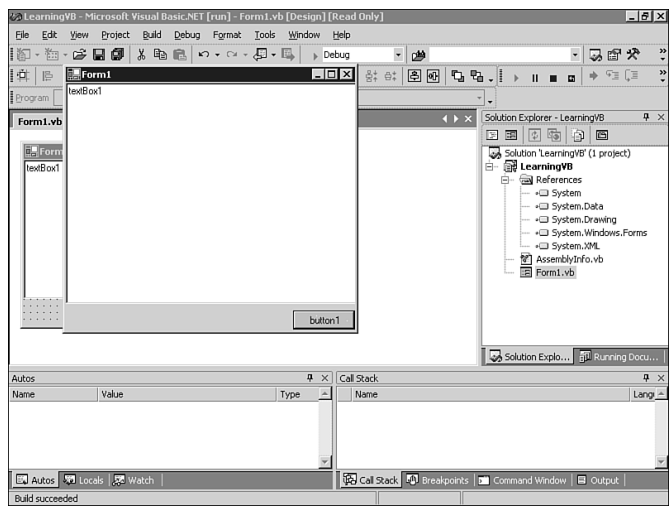


Figure 2.18
The form showing how the controls can resize or move as the form is resized.

Easier Menus

If you hated building menus under previous versions of VB, you weren't alone. The VB menu editor has never won any awards for ease of use nor for user friendliness. The new Menu Editor in VB.NET might just make you enjoy building menus.

Using the same form as in the previous example, go to the Toolbox and double-click the MainMenu control. This adds a menu bar to the form (and simply pushes down the text box). The menu is also added to an area below the form, called the Component Tray, which you'll see more of later. The menu shows one item that says Type Here, as shown in Figure 2.19.

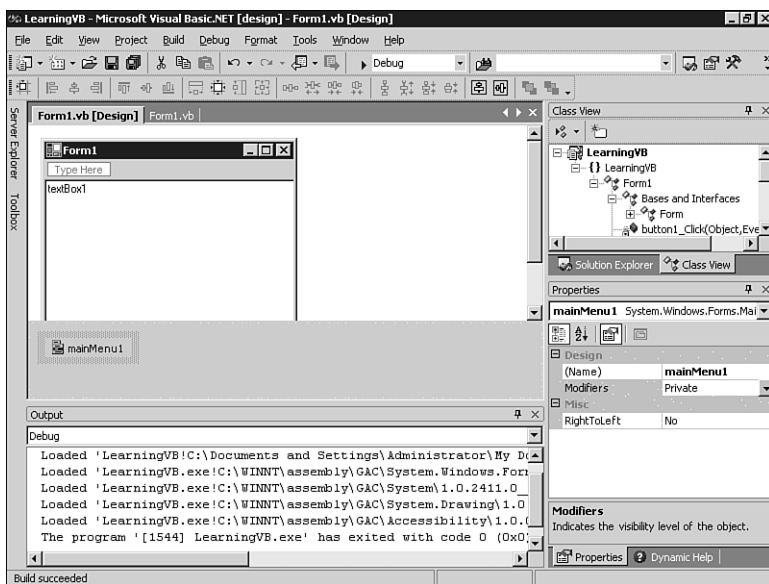


Figure 2.19

The new Menu Editor in action.

Click once in the box that says Type Here, and type **&File**. This creates a new Type Here box to the right and a new Type Here box below. Click once in the Type Here below the word **&File** and type **&Open**. Just as in previous versions of VB, the ampersand (&) is used to signify an Alt+key selection. Typing **&Open** creates a new Type Here box to the right and a new Type Here below. As you can see, you are graphically building a menu. Click in the box below the Open and type **&Close**.

You can click in the Type Here box just to the right of the File menu choice to add another top-level menu. For example, you could add **&Edit** to the right of the File menu. Now, the Edit menu gets new blank entries, and you could add items for copying and pasting.

Click on the Open menu choice under the File menu (on the menu you just created, not VB.NET's menus). If you look in the Properties window, you will see that the actual name of the object is `MenuItem2`. You can change that if you want, but don't worry about it for now. Instead, click on the drop-down for the `Shortcut` property. Scroll through the list until you find the `Ctrl+O` choice. The `ShowShortcut` property is set to `True` by default, but you won't see the shortcut at design time. Instead, run the application. Figure 2.20 shows you roughly what you should see. Notice that the underlining of the letters (signified by the ampersand) might not appear unless you hold down the `Alt` key.

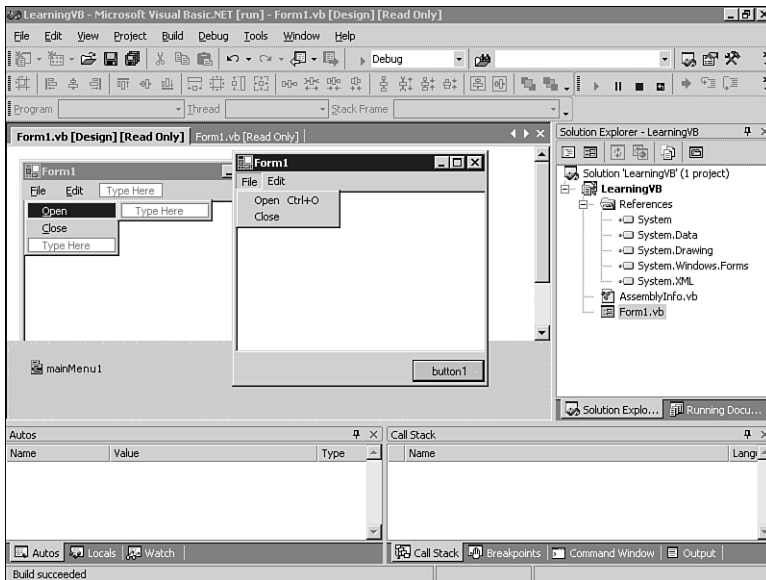


Figure 2.20

The new menu being displayed at runtime.

Setting Tab Order

If you disliked creating menus in previous versions of VB, it's a safe bet that you hated setting the tab order, especially on complex forms. You had to click each control, one at a time, and make sure that the `TabIndex` property was correct. Take the form you've been working with and remove the textbox. Leave `Button1` in the lower-right corner.

Add three more buttons to the form. Place them in such a way as to make Button1 the last in the series. In fact, mix them up, if you prefer. If you take a look at Figure 2.21, you'll see that the buttons have been placed in such a way that you will want the tab order to be Button2, Button4, Button3, and Button1. If you start the project and run it as is, however, the order will still be Button1, Button2, Button3, and Button4.

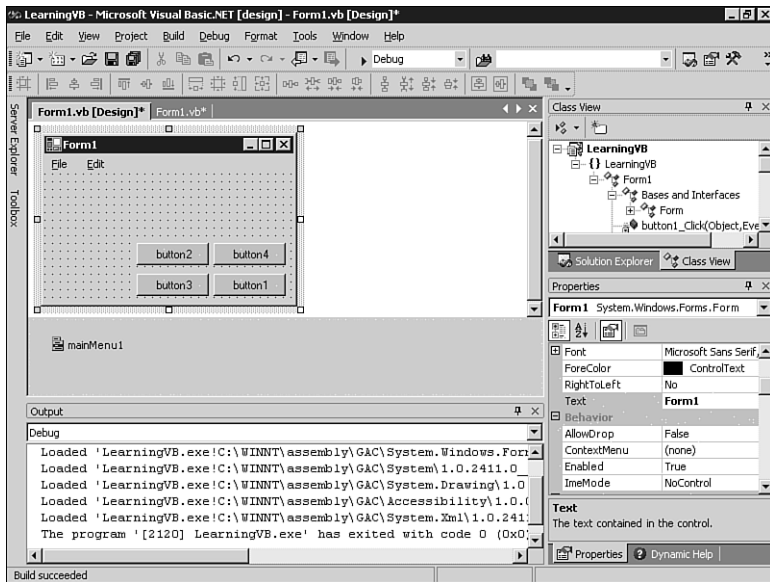


Figure 2.21

The form with buttons in the wrong order.

To set the tab order, click on the View menu and choose Tab Order. You now get small numbers on each control that can receive the focus, as shown in Figure 2.22. To reset the tab order, simply click on the controls in the order that you want them to receive focus. In this example, you just click on Button2, Button4, Button3, and Button1, in that order, and the tab order is set for you, as evidenced by the number in the upper-left corner of each button.

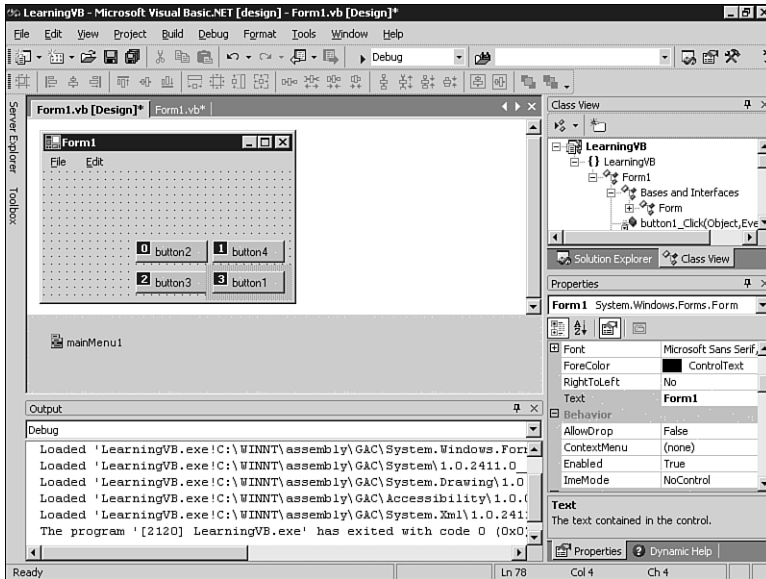


Figure 2.22

The new way to set the tab order.

You can choose View, Tab Order to turn off the display of the tab order. You can run the project to verify the new tab order.

Line and Shape Controls: You're Outta Here

If you examine the Toolbox, you might just find yourself searching and searching for the Line and Shape controls. They're gone. Yes, no more line and shape controls.

Microsoft removed them because they were actually windowless controls in VB6; that is, they didn't have an associated `hWnd`, but were painted directly on the form. In the new VB.NET forms engine, all controls must be windowed, and you no longer have control transparency.

Now that you've heard the technical explanation, what do you do about it? The easiest way to fake a line is to use a label. Turn on the border and then set the height (or width) to 1. This makes the label look like a line, and it works fine. For more complex shapes, Microsoft recommends that you use the GDI+ objects, which are very powerful. This is not something that will be covered in this book, but be aware of the change, and what you need to know to get around it.

Form Opacity

Forms now have an `Opacity` property. Is it a necessary feature? Probably not, but it could be used for some useful purpose. At least while you're working with it, you'll look busy.

On the `Form1` you've been working with so far, find its `Opacity` property and change it to 50%. Now, run the project and you'll notice that the form is now quite see-through. The form still works fine (not that this current form is doing anything). You can click and drag the form; you can press the buttons. It works fine, but it is translucent at the moment. Close the project and you can add some code to take advantage of this new, if not quite critical, feature.

Go to the Toolbox and add a timer to your form. Unlike VB6, the timer doesn't actually appear on the form, in the Component Tray for the form. Click on the timer in this window and then set the timer's `Enabled` property to `True`. The interval of 100 is fine.

Now, click on the `Form1.vb` tab to get to the code. In the `Sub New` routine, add the following line of code right after the `Add any initialization...` comment:

```
Me.Opacity = 0
```

This merely sets the opacity to 0, which means that the form will be completely invisible when the form loads.

Next, use the Class Name and Method Name drop-down boxes at the top of the code window to choose the `Tick` event of the `Timer1` control. This adds an event procedure for the `Timer1_Tick` event. Inside that event procedure, enter the following code:

```
Form1.Opacity = form1.Opacity + 0.01
If Me.Opacity >= 1 Then
    timer1.Enabled = False
    beep()
End If
```

This code merely increments the opacity by .01 each time the `Tick` event fires. When it finally reaches 1 (100%), the timer is shut off by setting its `Enabled` property to `False`, and the `beep` is called just to make a sound so that you know when it is finished.

If you are at all confused by the code, Listing 2.2 shows how the whole listing will look (with the Windows Form Designer generated code shown).

Listing 2.2 Your First VB.NET Application, with All the Code Discussed So Far

```
Public Class Form1
    Inherits System.Windows.Forms.Form

    #Region " Windows Form Designer generated code "

        Public Sub New()
            MyBase.New()

            'This call is required by the Windows Form Designer.
            InitializeComponent()

            'Add any initialization after the InitializeComponent() call
            Me.Opacity = 0
        End Sub

        'Form overrides dispose to clean up the component list.
        Public Overrides Sub Dispose()
            MyBase.Dispose()
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End Sub

        Private WithEvents button1 As System.Windows.Forms.Button
        Private WithEvents label1 As System.Windows.Forms.Label
        Private WithEvents menuItem1 As System.Windows.Forms.MenuItem
        Private WithEvents menuItem2 As System.Windows.Forms.MenuItem
        Private WithEvents menuItem3 As System.Windows.Forms.MenuItem
        Private WithEvents mainMenu1 As System.Windows.Forms.MainMenu
        Private WithEvents menuItem4 As System.Windows.Forms.MenuItem
        Private WithEvents button2 As System.Windows.Forms.Button
        Private WithEvents button3 As System.Windows.Forms.Button
        Private WithEvents button4 As System.Windows.Forms.Button
        Private WithEvents timer1 As System.Windows.Forms.Timer
        Private components As System.ComponentModel.IContainer

        'Required by the Windows Form Designer

        'NOTE: The following procedure is required by the _
            Windows Form Designer
        'It can be modified using the Windows Form Designer.
        'Do not modify it using the code editor.
        <System.Diagnostics.DebuggerStepThrough()> _
        Private Sub InitializeComponent()
            Me.components = New System.ComponentModel.Container()
            Me.menuItem1 = New System.Windows.Forms.MenuItem()
            Me.menuItem2 = New System.Windows.Forms.MenuItem()
            Me.menuItem3 = New System.Windows.Forms.MenuItem()
            Me.timer1 = New System.Windows.Forms.Timer(Me.components)
```

```

Me.button2 = New System.Windows.Forms.Button()
Me.button3 = New System.Windows.Forms.Button()
Me.button4 = New System.Windows.Forms.Button()
Me.button1 = New System.Windows.Forms.Button()
Me.mainMenu1 = New System.Windows.Forms.MainMenu()
Me.menuItem4 = New System.Windows.Forms.MenuItem()
Me.menuItem1.Index = 0
Me.menuItem1.MenuItems.AddRange(New System.Windows.Forms.MenuItem() _
    {Me.menuItem2, Me.menuItem3})
Me.menuItem1.Text = "&File"
Me.menuItem2.Index = 0
Me.menuItem2.Shortcut = System.Windows.Forms.Shortcut.CtrlO
Me.menuItem2.Text = "&Open"
Me.menuItem3.Index = 1
Me.menuItem3.Text = "&Close"
Me.timer1.Enabled = True
Me.button2.Location = New System.Drawing.Point(96, 72)
Me.button2.TabIndex = 0
Me.button2.Text = "button2"
Me.button3.Location = New System.Drawing.Point(96, 104)
Me.button3.TabIndex = 2
Me.button3.Text = "button3"
Me.button4.Location = New System.Drawing.Point(176, 72)
Me.button4.TabIndex = 1
Me.button4.Text = "button4"
Me.button1.Location = New System.Drawing.Point(176, 104)
Me.button1.TabIndex = 3
Me.button1.Text = "button1"
Me.mainMenu1.MenuItems.AddRange(New System.Windows.Forms.MenuItem() _
    {Me.menuItem1, Me.menuItem4})
Me.menuItem4.Index = 1
Me.menuItem4.Text = "&Edit"
Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
Me.ClientSize = New System.Drawing.Size(259, 134)
Me.Controls.AddRange(New System.Windows.Forms.Control() _
    {Me.button4, Me.button3, Me.button2, Me.button1})
Me.Menu = Me.mainMenu1
Me.Opacity = 0.5
Me.Text = "Form1"
End Sub

#End Region

Private Sub button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles button1.Click End Sub

Private Sub timer1_Tick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles timer1.Tick Me.Opacity =
Me.Opacity + 0.01

```

Listing 2.2 continued

```
        If Me.Opacity >= 1 Then
            timer1().Enabled = False
            Beep()
        End If
    End Sub
End Class
```

Run the project. You should see your form fade in slowly as the opacity is increased. You will hear a beep when the form is fully opaque.

Summary

This chapter sought to get you into the new VB.NET IDE quickly. You have created your first VB.NET application, and you have seen some of the new features presented in the IDE. It is now easier to have controls maintain their positions even when the form is resized. Help is more readily available. The Server Explorer makes it easier to locate and use resources throughout your enterprise. New project types open up a new world for VB.NET developers.

In addition to all these changes, there are some fundamental language changes as well. A few of those have been mentioned in this chapter, and if you looked at the code, you saw some others. The next chapter is dedicated to presenting some of the more important language changes.



CHAPTER 3

Major VB.NET Changes

VB.NET introduces major changes to the VB language. Some are modifications to existing ways of working, whereas others are brand new. This chapter will cover some of those changes, but this is by no means an exhaustive list of all changes from VB to VB.NET. First, you'll see some of the features that have changed. Then you will see some of the new features.

General Changes

There are a number of general changes to be aware of when moving from VB to VB.NET. Among them are topics such as the removal of default properties, subs and functions requiring parentheses, `ByVal` being the default method for passing parameters, and changes to the logical operators. These changes, and others, are detailed in this section.

Default Properties

In VB6, objects could have default properties. For example, the following code is perfectly valid in VB6, if you assume that `Text1` is a text box:

```
Text1="Hello, World"
```

This code takes the string `"Hello, World"` and sets the default property of the text box, the `Text` property, to the string. The major drawback to default properties is that they require you to have a `Set` command in VB. For example, take a look at the following block of VB6 code:

```
Dim txtBillTo as TextBox
Dim txtShipTo as TextBox
txtShipTo = txtBillTo
```

The line of code `txtShipTo = txtBillTo` sets the `Text` property of `txtShipTo` to the value in the `Text` property of `txtBillTo`. But what if that isn't what you wanted? What if, instead, you wanted to create an object reference in `txtShipTo` that referred to the object `txtBillTo`? You'd have to use this code:

```
Set txtShipTo = txtBillTo
```

As you can see, default properties require you to use the `Set` keyword to set references from one object variable to another.

VB.NET gets around this problem by getting rid of default properties. Therefore, to copy the `Text` property from `txtBillTo` into the `Text` property of `txtShipTo`, you'd have to use this code:

```
txtShipTo.Text = txtBillTo.Text
```

Setting the two variables equal to each other sets a reference from one to the other. In other words, you can set an object reference without the `Set` keyword:

```
txtShipTo = txtBillTo ' Object reference in VB.NET
```

To be more precise, default properties *without parameters* are no longer supported. Default properties that require parameters are still valid. Default properties with parameters are most common with collection classes, such as in ADO. In an ADO example, if you assume that `rs` is an ADO Recordset, check out the following code:

```
rs.Fields.Item(x).Value ' OK, fully qualified
rs.Fields(x).Value ' OK, because Item is parameterized
rs.Fields(x) ' Error, because Value is not parameterized
```

The easy solution is to fully qualify everything. This avoids any confusion about which properties are parameterized and which are not. However, as you know from your VB days, the number of dots you have should be minimized. The reason is that each dot requires an OLE lookup that slows you down. Therefore, you should code carefully when dealing with parameters.

Subs and Functions Require Parentheses

As you saw in the last chapter when you used the `MsgBox` function, you must now always use parentheses with functions, even if you are ignoring the return value. In addition, you must use parentheses when calling subs, which you did not do in VB6. For example, assume that you have this sub in both VB6 and VB.NET:

```
Sub foo(ByVal Greeting As String)
    ' implementation code here
End Sub
```

In VB6, you could call this sub in one of two ways:

```
foo "Hello"
```

```
Call foo("Hello")
```

In VB.NET, you also could call this sub in one of two ways:

```
Foo("Hello")
```

```
Call foo("Hello")
```

The difference, of course, is that the parentheses are always required in the VB.NET calls, even though you aren't returning anything. The `Call` statement is still supported, but it is not really necessary.

Changes to Boolean Operators

The `And`, `Not`, and `Or` operators were to have undergone some changes. Microsoft originally said that the operators would short-circuit, but now they are staying the way they worked in VB6. This means that in VB.NET, as in VB6, if you had two parts of an `And` statement and the first failed, VB6 still examined the second part. Examine the following code:

```
Dim x As Integer
Dim y As Integer
x = 1
y = 0
If x = 2 And y = 5/y Then
...
```

As a human, you know that the variable `x` is equal to 1. Therefore, when you look at the first part of the `If` statement, you know that `x` is not equal to 2, so you would logically think it should quit evaluating the expression. However, VB.NET examines the second part of the expression, so this code would cause a divide-by-zero error.

If you want short-circuiting, VB.NET has introduced a couple of new operators: `AndAlso` and `OrElse`. In this case, the following code would not generate an error in VB.NET:

```
Dim x As Integer
Dim y As Integer
x = 1
y = 0
If x = 2 AndAlso y = 5/y Then
...
```

This code does not cause an error; instead, because `x` is not equal to 2, VB.NET does not even examine the second condition.

Declaration Changes

You can now initialize your variables when you declare them. You could not do this in VB6. In VB6, the only way to initialize a new variable was to do so on a separate line, like this:

```
Dim x As Integer
x = 5
```

In VB.NET, you can rewrite this into one line of code:

```
Dim x As Integer = 5
```

Another significant, and much-requested, change is that of declaring multiple variables, and what data type they assume, on one line. For example, you might have the following line:

```
Dim x, y As Integer
```

As you're probably aware, in VB6, *y* would be an *Integer* data type, but *x* would be a *Variant*. In VB.NET, this has changed, so both *x* and *y* are *Integers*. If you think, "It's about time," there are many who agree. This should remove a number of bugs and strange type conversions experienced by new VB developers. It should also make the code more efficient by making variables the expected type instead of using the *Object* type (which replaces the *Variant* type, as you will see later).

Support for New Assignment Operators

VB.NET now supports shortcuts for performing certain assignment operations. In VB6, you incremented *x* by 1 with the following line of code:

```
x = x + 1
```

In VB.NET, you can type an equivalent statement like this:

```
x += 1
```

Not only can you use the plus sign, but VB.NET now also supports `-=`, `*=`, `/=`, `\=`, and `^=` from a mathematical standpoint, and `&=` for string concatenation. If all this looks like C/C++, that's where it came from. However, the `++` operator is not supported. Microsoft made a decision not to include the `++` operator because they felt it made the code more difficult to read.

Because VB.NET is in beta and has not yet been performance tuned, it is unclear whether these new assignment operators will be more efficient. These operators did tend to be more efficient in C/C++, due to a more efficient use of the CPU's registers. Therefore, it will be interesting to test these new operators when the final, tuned version of VB.NET is released.

ByVa1 Is Now the Default

In what many consider a strange decision, the default way to pass parameters in VB has always been by reference. The decision was actually made because passing by reference is faster within the same application, but can be costly if you are calling components across process boundaries. If you're a little rusty, *by reference* means that you are passing only the address of a variable into the called routine. If the called routine modifies the variable, it actually just updates the value in that memory location, and therefore the variable in the calling routine also changes. Take this VB6 example:

```
Private Sub Command1_Click()
    Dim x As Integer
    x = 3
    foo x
    MsgBox x
End Sub

Sub foo(y As Integer)
    y = 5
End Sub
```

If you run this example in VB6, the message box shows the value 5. That happens because in VB6, when you pass `x` to `foo`, you are just sending the memory address of the variable `x`, so when `foo` modifies `y` to 5, it is changing the value in the same memory location to which `x` points, and this causes the value of `x` to change as well.

If you tried to type this example into VB.NET, you'd see something happen. First, of course, you'd have to add parentheses around `x` in your call to `foo`. However, when you tried to type the definition of `foo`, VB.NET would automatically add the word `ByVa1` into the definition, so it would end up looking like this:

```
Sub foo(ByVa1 y As Integer)
```

If you wanted to pass by reference, you would have to add the `ByRef` keyword yourself, instead of VB.NET using the new default of `ByVa1`. This is a benefit to those of you calling procedures across process boundaries, something that is common in the world of distributed applications. In addition, this should cut down on errors like those seen by novice users who didn't understand the concept of passing by reference in previous versions of VB.

Block-Level Scope

VB.NET adds the ability to create variables that are visible only within a block. A *block* is any section of code that ends with one of the words `End`, `Loop`, or `Next`. This means that `For...Next` and `If...End If` blocks can have their own variables. Take a look at the following code:

```
While y < 5
    Dim z As Integer
    ...
End While
```

The variable `z` is now visible only within the `While` loop. It is important to realize that although `z` is visible only inside the `While` loop, its lifetime is that of the procedure. That means if you re-enter the `While` statement, `z` will have the same value that it did when you left. Therefore, it is said that the *scope* of `z` is block level, but its *lifetime* is procedure level.

While...Wend Becomes While...End While

The `While` loop is still supported, but the closing of the loop is now `End While` instead of `Wend`. If you type `Wend`, the editor automatically changes it to `End While`. This change finally move the `While` loop into synch with most other VB “block” structures, which all end with an `End <block>` syntax.

Procedure Changes

VB.NET has changes that affect how you define and work with procedures. Some of those changes are mentioned in this section.

Optional Arguments Require a Default Value

In VB6, you could create an optional argument (or several optional arguments) when you defined a procedure. You could, optionally, give them a default value. That way, if someone chose not to pass in a value for an argument, you had a value in it. If you did not set a default value and the caller did not pass in a value, the only way you had to check was to call the `IsMissing` statement.

`IsMissing` is no longer supported because VB.NET will not let you create an optional argument that does not have a default value. `IsMissing` is not needed because an optional argument is guaranteed to have a value. For example, your declaration might look like this:

```
Sub foo(Optional ByVal y As Integer = 1)
```

Notice that the `Optional` keyword is shown, just as it was in VB6. This means a parameter does not have to be passed in. However, if it is not passed in, `y` is given the default value of 1. If the calling routine does pass in a value, of course, `y` is set to whatever is passed in.

Static Not Supported on Subs or Functions

In VB6, you could put `Static` in the declaration of a sub or function. Doing so made every variable in that sub or function *static*, which meant that they retained their values between calls to the procedure. For example, this was legal in VB6:

```
Static Sub foo()
    Dim x As Integer
    Dim y As Integer
    x = x + 1
    y = y + 2
End Sub
```

In this example, `x` will retain its value between calls. So, the second time this procedure is called, `x` already has a value of 1, and the value of `x` will be incremented to 2. The variable `y` would have the value of 2, and therefore the second time in it would be incremented to 4.

VB.NET does not support the `Static` keyword in front of the sub or function declaration anymore, as you saw in the example above. In fact, if you want an entire sub or function to be static, you need to place `Static` in front of each variable for which you want to preserve the value. This is the only way to preserve values in variables inside a sub or function. In VB.NET, the equivalent sub would look like this:

```
Sub foo()
    Static x As Integer
    Static y As Integer
    x = x + 1
    y = y + 2
End Sub
```

The Return Statement

The `Return` statement can be used to produce an immediate return from a called procedure, and can optionally return a value. For example, look at this block of code:

```
Function foo() As Integer
    While True
        Dim x As Integer
        x += 1
        If x >= 5 Then
            Return x
        End If
    End While
End Function
```

In this code, you enter what normally would be an infinite loop by saying `While True`. Inside the loop, you declare an integer called `x` and begin incrementing it. You check the value of `x` and when it becomes greater than or equal to 5, you call the

Return statement and return the value of `x`. This causes an immediate return, and does not wait to hit the `End Function` statement.

The old way of doing it still works, however. For example, to rewrite this procedure using the “old” approach to return a value, you would make it look like the following code. This sets the name of the procedure, `foo`, to the value to be returned. Then you have to exit the loop so that you’ll hit `End Function`. You also could have used `Exit Function` instead of `Exit Do`.

```
Function foo() As Integer
    Do While True
        Dim x As Integer
        x += 1
        If x >= 5 Then
            foo = x
            Exit Do
        End If
    Loop
End Function
```

ParamArrays Are Now Passed ByVal

A parameter array, or `ParamArray`, is used when you do not know how many values you will pass into a procedure. A `ParamArray` allows for an unlimited number of arguments. A parameter array automatically sizes to hold the number of elements you are passing in.

A procedure can have only one `ParamArray`, and it must be the last argument in the definition. The array must be one-dimensional and each element must be the same data type. However, the default data type is `Object`, which is what replaces the `Variant` data type in VB.NET.

In VB6, all the elements of a `ParamArray` are always passed `ByRef`. This cannot be changed. In VB.NET, all the elements are passed `ByVal`. This cannot be changed. Again, `ByRef` was used in previous versions of VB because it was more efficient for passing parameters within a program all running in the same application space. When working with out-of-process components, however, it is more efficient to pass parameters `ByVal`.

Properties Can Be Modified ByRef

In VB6, if a class had a property, you could pass that property to a procedure as an argument. If you passed the property `ByVal`, of course, it just copied the value. If you passed it `ByRef`, the called procedure might modify the value, but this new value was *not* reflected back in the object’s property.

In VB.NET, however, you can pass a property to a procedure `ByRef`, and any changes to the property in the called procedure *will* be reflected in the value of the property for the object.

Take the following VB.NET example. Don't worry if the Class syntax looks strange; just realize that you are creating a class called `Test` with one property: `Name`. You instantiate the class in the `button1_Click` event procedure, and pass the `Name` property, by reference, to `foo`. `foo` modifies the variable and ends. You then use a message box to print out the `Name` property.

```
Private Sub button1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles button1.Click
    Dim x As New Test()
    foo(x.Name)
    MsgBox(x.Name)
End Sub

Sub foo(ByRef firstName As String)
    firstName = "Torrey"
End Sub

Public Class Test
    Dim firstName As String
    Property Name() As String
        Get
            Name = firstName
        End Get
        Set(ByVal Value As String)
            firstName = Value
        End Set
    End Property
End Class
```

When you run this example, the message box will display "Torrey". This shows that the property of the object is being passed by reference, and that changes to the variable are reflected back in the object's property. This is new in VB.NET.

Array Changes

Arrays have undergone some changes as well. Arrays could be somewhat confusing in previous versions of VB. VB.NET seeks to address any confusion by simplifying the rules and removing the capability to have nonzero lower boundaries.

Array Size

In VB6, if you left the default for arrays to start at 0, declaring an array actually gave you the upper boundary of the array, not the number of elements. For example, examine the following code:

```
Dim y(2) As Integer
y(0) = 1
y(1) = 2
y(2) = 3
```

In this VB6 code, you declare that `y` is an array of type `Integer`, and the upper boundary is 2. That means you actually have three elements: 0–2. You can verify this by setting those three elements in code.

In VB.NET, array declaration was going to change, so that the parameter was the number of elements. However, due to the possibility of breaking existing code in the upgrade process, this has been changed back to the way it worked in VB.

Lower Boundary Is Always Zero

VB6 allowed you to have a nonzero lower boundary in your arrays in a couple of ways. First, you could declare an array to have a certain range. If you wanted an array to start with 1, you declared it like this:

```
Dim y(1 To 3) As Integer
```

This would create an array with three elements, indexed 1–3. If you didn't like this method, you could use `Option Base`, which allowed you to set the default lower boundary to either 0 (the default) or 1.

VB.NET removes those two options from you. You cannot use the `1 to x` syntax, and `Option Base` is no longer supported. In fact, because the lower boundary of the array is always 0, the `Lbound` function is no longer supported.

Array Assignment Is ByRef Instead of ByVal

In VB6, if you had two array variables and set one equal to the other, you actually were creating a copy of the array in a `ByVal` copy. Now, in VB.NET, setting one array variable to another is actually just setting a reference to the array. To copy an array, you can use the `Clone` method.

Option Strict

`Option Strict` is a new statement that specifically disallows any type conversions that would result in data loss. You can use only widening conversions; this means you could convert from an `Integer` to a `Long`, but you could not convert from a `Long` to an `Integer`. `Option Strict` also prevents conversions between numbers and strings.

`Option Strict` is on by default in Visual Studio.NET Beta 2. To turn it off, go to the top of the code window and type `Option Strict Off`.

You can see that the IDE recognized potential problems with the following lines of code:

```
Dim longNumber As Long
Dim integerNumber As Integer
integerNumber = longNumber
```

With this code, the IDE will underline `longNumber` and a tooltip will tell you the following: `Option Strict` disallows implicit conversions from `Long` to `Integer`.

Data Type Changes

There are several changes to data types that are important to point out. These changes can have an impact on the performance and resource utilization of your code. The data types in VB.NET correspond to the data types in the `System` namespace, which is important for cross-language interoperability.

All Variables Are Objects

Technically, in VB.NET, all variables are subclassed from the `Object` base class. This means that you can treat all variables as objects. For example, to find the length of a string, you could use the following code:

```
Dim x As String
x = "Hello, World"
MsgBox(x.Length)
```

This means that you are treating `x` as an object, and examining its `Length` property. Other variables have other properties or methods. For example, an `Integer` has a `ToString` method that you can use, which you will see in a moment.

Short, Integer, and Long

In VB.NET, the `Short` data type is a 16-bit integer, the `Integer` is a 32-bit integer, and the `Long` is a 64-bit integer. In VB6 code, you might have used `Long` in many places because it was 32-bit and therefore performed better on 32-bit operating systems than `Integer`, which was just 16-bit. On 32-bit operating systems, 32-bit integers perform better than 64-bit integers, so in VB.NET, you will most likely want to return to using the `Integer` data type instead of the `Long` data type.

Automatic String/Numeric Conversion Not Supported

In VB6, it was easy to convert from numbers to strings and vice versa. For example, examine this block of code:

```
Dim x As Integer
Dim y As String
x = 5
y = x
```

In VB6, there is nothing wrong with this code. VB will take the value 5 and automatically convert it into the string "5". VB.NET, however, disallows this type of conversion by default. Instead, you would have to use the `CStr` function to convert a number to a string, or the `Val` function to convert a string to a number. You could rewrite the preceding code for VB.NET in this manner:

```
Dim x As Integer
Dim y As String
x = 5
```

```
y = CStr(x)
y = x.ToString ' This is equivalent to the previous line
```

Fixed-Length Strings Not Supported

In VB6, you could declare a fixed-length string by using a declaration like the one shown here:

```
Dim y As String * 30
```

This declared `y` to be a fixed-length string that could hold 30 characters. If you try this same code in VB.NET, you will get an error. All strings in VB.NET are variable length.

All Strings Are Unicode

If you got tired of worrying about passing strings from VB to certain API calls that accepted either ANSI or Unicode strings, you will be happy to hear that all strings in VB.NET are Unicode.

The Value of True

Long ago, someone at Microsoft decided that `True` in VB was equal to `-1`. The .NET Framework declares that `True` should be equal to `1`.

In reality, `0` is `False` and any nonzero is `True`. However, if you ask for the value of `True`, you will get `-1` inside of VB.NET. You can verify that any nonzero is `True` by using this code:

```
Dim x As Integer
x = 2
If CBool(x) = True Then
    MsgBox("Value is True")
End If
```

If you make `x` any positive or negative integer, the `CBool` function will make it `True`. If you set `x` equal to `0`, `CBool` will return `False`. Even though `True` is actually shown as `-1` in VB.NET, if you pass it out to any other .NET language, it is seen as `1`.

The important thing to understand is this: Do not code for the value (`1` or `-1`). Just use the keyword `True` and you will avoid any problems.

The Currency Data Type Has Been Replaced

The `Currency` data type has been replaced by the `Decimal` data type. The `Decimal` data type is a 12-byte signed integer that can have up to 28 digits to the right of the decimal place. It supports a much higher precision than the `Currency` data type, and has been designed for applications that cannot tolerate rounding errors. The `Decimal` data type has a direct equivalent in the .NET Framework, which is important for cross-language interoperability.

The Variant Data Type Has Been Replaced

The Variant data type has been replaced. Before you start thinking that there is no longer a catch-all variable type, understand that the Variant has been replaced by the Object data type. The Object data type takes up only four bytes because all it holds is a memory address. Therefore, even if you set the variable to an integer, the Object variable holds four bytes that point to the memory used to store the integer. The integer is stored on the heap. It will take up 8 bytes plus 4 bytes for the integer, so it consumes a total of 12 bytes. Examine the following code:

```
Dim x
x = 5
```

In this case, x is a variable of type Object. When you set x equal to 5, VB.NET stores the value 5 in another memory location and x stores that memory address. When you attempt to access x, a lookup is required to find that memory address and retrieve the value. Therefore, just like the Variant, the Object data type is slower than using explicit data types.

According to the VB.NET documentation, the Object data type allows you to play fast-and-loose with data type conversion. For example, look at the following code:

```
Dim x
x = "5" ' x is a string
x -= 2
```

In this example, x will now hold the numeric value 3. The only way you can run this example is if you set `Option Strict Off` in the file. If you change the code to use string concatenation, the conversion also works fine. For example, the following code will run without an error, and x will end up holding the string "52":

```
Dim x
x = "5" ' x is a string
x &= 2
```

An Object data type can be Nothing. In other words, it can hold nothing, which means it doesn't point to any memory address. You can check for Nothing by setting the Object variable equal to Nothing or by using the `IsNothing` function. Both checks are shown in the following code:

```
Dim x
If x = Nothing Then MsgBox("Nothing")
If IsNothing(x) Then MsgBox("Still nothing")

Dim x As Object
If x Is Nothing Then MsgBox("Nothing")
If IsNothing(x) Then MsgBox("Still Nothing")
```

Structured Error Handling

Error handling has changed in VB.NET. Actually, the old syntax still works, but there is a new error handling structure called `Try...Catch...Finally` that removes the need to use the old `On Error Goto` structure.

The overall structure of the `Try...Catch...Finally` syntax is to put the code that might cause an error in the `Try` portion, and then catch the error. Inside the `Catch` portion, you handle the error. The `Finally` portion runs code that happens after the `Catch` statements are done, regardless of whether or not there was an error. Here is a simple example:

```
Dim x, y As Integer ' Both will be integers
Try
    x \= y ' cause division by zero
Catch ex As Exception
    msgbox(ex.Message)
End Try
```



NOTE

The `ex As Exception` part of the `Catch` statement is optional.

Here, you have two variables that are both integers. You attempted to divide `x` by `y`, but because `y` has not been initialized, it defaults to 0. That division by zero raises an error, and you catch it in the next line. The variable `ex` is of type `Exception`, which holds the error that just occurred, so you simply print the `Message` property, much like you printed `Err.Description` in VB6.

In fact, you can still use `Err.Description`, and the `Err` object in general. The `Err` object will pick up any exceptions that are thrown. For example, assume that your logic dictates that an error must be raised if someone's account balance falls too low, and another error is raised if the balance drops into the negative category. Examine the following code:

```
Try
    If bal < 0 Then
        Throw New Exception("Balance is negative!")
    ElseIf bal > 0 And bal <= 10000 Then
        Throw New Exception("Balance is low; charge interest")
    End If
Catch
    MessageBox.Show("Error: " & Err().Description)
Finally
    MessageBox.Show("Executing finally block.")
End Try
```

In this case, your business logic says that if the balance drops below zero, you raise an error informing the user that the balance is below zero. If the balance drops below 10,000 but remains above zero, you notify the user to start charging interest. In this case, `Err().Description` picks up the description you threw in your Exception.

You can also have multiple Catch statements to catch various errors. To have one Catch statement catch a particular error, you add a When clause to that Catch.

Examine this code:

```
Try
    x \= y ' cause division by zero
Catch ex As Exception When Err().Number = 11
    MsgBox("You tried to divide by zero")
Catch ex As Exception
    MsgBox("Acts as catch-all")
End Try
```

In this example, you are looking for an `Err().Number` of 11, which is the error for division by zero. Therefore, if you get a division-by-zero error, you display a message box that says `You tried to divide by zero`. However, if a different error were to occur, you would drop into the Catch without a When clause. In effect, the Catch without a When clause acts as an “else” or “otherwise” section to handle anything that was not handled before.

Notice that the code does not fall through all the exceptions; it stops on the first one it finds. In the preceding example, you will raise an `Err().Number` of 11. You will see the message `You tried to divide by zero`, but you will skip over the catch-all Catch. If you want to run some code at the end, regardless of which error occurred (or, indeed, if no error occurred), you could add a Finally statement. The code to do so follows:

```
Try
    x \= y ' cause division by zero
Catch ex As Exception When Err().Number = 11
    MsgBox("You tried to divide by zero")
Catch ex As Exception
    MsgBox("Acts as catch-all")
Finally
    MsgBox("Running finally code")
End Try
```

In this code, whether or not you hit an error, the code in the Finally section will execute.

There is also a documented way to exit from a Try statement early. The `Exit Try` keyword is supposed to break you out of the Try early, ignoring any Finally code. The syntax would look like this:

```
Try
    x \= y ' cause division by zero
Catch ex As Exception When Err.Number = 11
    MsgBox("You tried to divide by zero")
Exit Try
Catch ex As Exception
    MsgBox("Acts as catch-all")
Finally
    MsgBox("Running finally code")
End Try
```

Structures Replace UDTs

User-defined types, or UDTs, were a way to create a custom data type in VB6. You could create a new data type that contained other elements within it. For example, you could create a Customer data type using this syntax:

```
Private Type Customer
    Name As String
    Income As Currency
End Type
```

You could then use that UDT in your application, with code like this:

```
Dim buyer As Customer
buyer.Name = "Martha"
buyer.Income = 20000
MsgBox(buyer.Name & " " & buyer.Income)
```

The Type statement is no longer supported in VB.NET. It has been replaced by the Structure statement. The Structure statement has some major changes, but to recreate the UDT shown earlier, the syntax is this:

```
Structure Customer
    Dim Name As String
    Dim Income As Decimal
End Structure
```

Notice that the only real difference so far is that you have to Dim each variable inside the structure, which is something you did not have to do in the Type statement, even with Option Explicit turned on. Notice also that Dim is the same as Public here, meaning that the variables are visible to any instance of the structure.

Structures have many other features, however. One of the biggest differences is that structures can support methods. For example, you could add a Shipping method to the Customer structure to calculate shipping costs based on delivery zone. This code adds a DeliveryZone property and a DeliveryCost function:

```
Structure Customer
    Dim Name As String
```

```

Dim Income As Decimal
Dim DeliveryZone As Integer

Function DeliveryCost() As Decimal
    If DeliveryZone > 3 Then
        Return 25
    Else
        Return CDec(12.5)
    End If
End Function
End Structure

```

Here, you have a built-in function called `DeliveryCost`. To use this structure, your client code would look something like this:

```

Dim buyer As Customer
buyer.Name = "Martha"
buyer.Income = 20000
buyer.DeliveryZone = 4
msgbox(buyer.DeliveryCost)

```

In this case, the message box would report a value of 25.

If you think this looks like a class, you are correct. In fact, structures can have properties, methods, and events. They also support implementing interfaces and they can handle events. There are some caveats, however. Some of those stipulations include the following:

- You cannot inherit from a structure.
- You cannot initialize the fields inside a structure. For example, this code is illegal:


```

Structure Customer
    Dim Name As String = "Martha"
End Structure

```
- Properties are public by default, instead of private as they are in classes.

Structures are value types rather than reference types. That means if you assign a structure variable to another structure variable, you get a copy of the structure and not a reference to the original structure. The following code shows that a copy is occurring because an update to `seller` does not update `buyer`:

```

Dim buyer As Customer
Dim seller ' Object data type
seller = buyer
seller.Name = "Linda"
MsgBox(buyer.Name)
MsgBox(seller.Name)

```

Note that the preceding code will not work if you have `Option Strict` turned on. If you want to test this, you'll have to enter `Option Strict Off`.

IDE Changes

There are numerous IDE changes. Several of those have been addressed already: the lack of the Line and Shape controls, the new menu builder, the new way of setting the tab order, and the Dynamic Help window. There are many other changes, such as the debugging windows, but they will not be discussed in this small book.

New Items

In addition to changes to the core language and some of the tools, there are some completely new features to VB.NET. The major new items include such features as constructors and destructors, namespaces, inheritance, overloading, free threading, and garbage collection. This is not an exhaustive list by any means, but these six features are worth discussing to one degree or another. One feature that should be mentioned as well is something you've seen before: auto-indentation. As soon as you create a block, such as a Sub or If block, the IDE indents the next line of code automatically.

Constructors and Destructors

Constructors and destructors are potentially new concepts for VB programmers. They are similar to the `Class Initialize` and `Class Terminate` events you have had with classes. At their simplest level, constructors and destructors are procedures that control the initialization and destruction of objects, respectively. The procedures `Sub New` and `Sub Destruct` replace the VB6 `Class_Initialize` and `Class_Terminate` methods. Unlike `Class_Initialize`, `Sub New` runs only once, when the object is first created. `Sub New` cannot be called explicitly except in rare circumstances.

`Sub Destruct` is called by the system when the object is set to `Nothing` or all references to the object are dropped. However, you cannot ensure when `Sub Destruct` actually will be called, thanks to the way VB.NET does garbage collection, which is discussed at the end of this chapter.

One of the reasons for constructors is that you can create an object and pass in some initialization parameters. For example, assume that you want to create a class dealing with a training course, and you want to initialize the class with a course number so that it can retrieve certain information from a database. In your class, you create a `Sub New` method that accepts an argument for the course ID. The first line inside the `Sub New` is a call to another constructor, usually the constructor of the base class on which this class is based. Fortunately, VB.NET gives you an easy way to call the constructor of the base class for your current class: `MyBase.New`.

If you want to create a class for a training course and initialize it with a course ID, your class would look something like this:

```
Class Course
    Sub New(ByVal CourseID As Integer)
```

```

        MyBase.New()
        FindCourseInfo(CourseID)
        ...
    End Sub
End Class

```

To call this class from the client code, your call would look something like this:

```
Dim TrainingCourse as Course = New Course(5491)
```

Namespaces

Perhaps one of the most confusing aspects of VB.NET for VB developers is the concept of a namespace. A *namespace* is a simple way to organize the objects in an assembly. When you have many objects, such as in the `System` namespace provided by the runtime, a namespace can simplify access because it is organized in a hierarchical structure, and related objects appear grouped under the same node.

For example, say you wanted to model the objects, properties, and methods for a pet store. You might create the `PetStore` namespace. You could then create some sub-namespaces. For example, you might sell live animals and supplies as two major categories. Within the live animals category, you might sell dogs, cats, and fish. If this sounds like an object model, it can be thought of as similar. However, none of these are actual objects. You could have a namespace called `PetStore.Animals.Dogs`, and in this namespace you'd find the classes and methods necessary for dealing with dogs. If you wanted to handle the inventory, you might find that in `PetStore.Supplies`. If you wanted to look at the kinds of dog food you have in stock, you might look in `PetStore.Supplies.Dogs`. Where the physical classes exist is up to you; they all can be in one big assembly, but logically separated into these various namespaces.

If you want to use the objects in an assembly without having to fully qualify them each time, use the `Imports` statement. Doing so allows you to use the names of the objects without fully qualifying their entire namespace hierarchy.

For example, when you created your first VB.NET project in Chapter 2, “Your First VB.NET Application,” you saw some `Imports` statements at the top of the form's code module. One of those statements was `Imports System.Windows.Forms`. That statement made all the classes, interfaces, structures, delegates, and enumerations available to you without you having to qualify the full path. That means the following code is legal:

```
Dim x as Button
```

Without the `Imports System.Windows.Forms` statement, your line of code would have to look like this:

```
Dim x as System.Windows.Forms.Button
```

In fact, the `MsgBox` that you know and love can be replaced by the `System.Windows.Forms.MessageBox` class. `MsgBox` still works for compatibility, but it is actually part of the `Microsoft.VisualBasic.Interaction` namespace, which contains a number of methods that approximate functions in previous versions of VB.

If you're wondering why Microsoft pulled out some language elements and made them part of the runtime, it should be fairly obvious: so that any language targeting the runtime, on any platform, would have access to common functionality. That means that you can go into C# and have a message box, just by importing `System.Windows.Forms` and calling the `MessageBox` class. `MessageBox` is quite powerful—the `Show` method has twelve variations of it.

Creating Your Own Namespaces

You are free to create your own namespaces inside your assemblies. You can do this simply by inserting your own `Namespace...End Namespace` block. Inside the namespace block, you can have structures, classes, enums, interfaces, and other elements. You must name the namespace and it becomes what someone would import. Your code might look like this:

```
Namespace VolantTraining
    Public Class Customer
        'code here
    End Class

    Public Class Student
        'code here
    End Class
End Namespace
```

Namespaces can be nested within other namespaces. For example, your namespace might look something like this:

```
Namespace VolantTraining
    Namespace Customer
        Class Training
            ...
        End Class
        Class Consulting
            ...
        End Class
    End Namespace
    Namespace Student
        ...
    End Namespace
End Namespace
```

Here you have one namespace, `VolantTraining`, that holds two other namespaces: `Customer` and `Student`. The `VolantTraining.Customer` namespace holds the

Training and Consulting classes, so you could have customers who have used your training services and customers who have used your consulting services.

If you choose not to create explicit namespaces, all your classes and modules still belong to a namespace. This namespace is the default namespace and is the name of your project. You can see this namespace, and change it if you want, by clicking viewing the Project Properties dialog box for your project. The text box labeled Root Namespace represents the root namespace for your project. If you declare a namespace within your project, it is subordinate to this root namespace. Therefore, if the root namespace of your applications Project1, then the full namespace for VolantTraining would be Project1.VolantTraining.

Inheritance

The most-requested feature in VB for years has been inheritance. Microsoft often countered that VB did inheritance; VB did interface inheritance, which meant that you could inherit (what VB called *implement*) an interface. However, the interface you implemented did not have any code in it, or if it did, the code was ignored. Therefore, the class implementing the interface had to provide methods for all the methods in the interface, and the code had to be rewritten in each class that implemented the interface. VB developers wanted to be able to write that code once, in a base class, and then to inherit that class in other classes, which would then be called *derived classes*. The derived classes should be able to use the existing code in the base class. With VB.NET, developers have their wish.

Not only does your derived class inherit the properties and methods of the base class, it can extend the methods and, of course, create new methods (in the derived class only). Derived classes can also override any existing method in the base class with a new method of the same name, in a process called *overriding*. Forms, which are really just classes, can be inherited to create new forms.

There are many concepts to inheritance, and seeing it in practice is important enough to make it a chapter unto itself. Chapter 5, “Inheritance with VB.NET,” is all about inheritance and how to use it in VB.NET.

Overloading

Overloading is another feature that some VB developers have been requesting for a long time. In short, overloading allows you to define the same procedure multiple times. The procedure has the same name but a different set of arguments each time.

You could fake this in an ugly way in VB6. You could pass in an argument as a Variant, and then use the VarType command to check the type of variable that was passed in. This was cumbersome, and the code could get nasty if your procedure had to accept an array or a collection.

VB.NET gives you a nice way to handle this issue. Imagine that you have a procedure that can accept a string or an integer. The functionality inside the procedure would be quite different depending on whether what is passed is a string or an integer. Your code might look something like this:

```
Overloads Function FindCust(ByVal psName As String) As String
    ' search name field for %psName%
End Function
```

```
Overloads Function FindCust(ByVal piCustNo As Integer) As String
    ' search CustID field
End Function
```

You now have a function called `FindCust` that can be passed either an integer or a string. Your calls to it could look like this:

```
Dim x As String
x = FindCust("Hello")
x = FindCust(1)
```

As long as `Option Strict` is turned on (which is the default, at least in Beta 2), you cannot compile an invalid call. For example, there is no overloaded `FindCust` that accepts a floating-point value of any kind. VB.NET would not let the following code compile:

```
x = FindCust(12.5)
```

Free Threading

For the first time, VB.NET has given VB developers the ability to write truly free-threaded applications. If your application is going to perform a task that could take a long time, such as parsing through a large recordset or performing a complex series of mathematical calculations, you can push that processing off to its own thread so that the rest of your application is still accessible. In VB6, the best you could do to keep the rest of the application from appearing to be locked was to use the `DoEvents` method.

Examine this code, which is written for VB.NET. Here you have some code for `button4`. This code calls the `BeBusy` routine, which has a loop in it to just to take up time. However, while in this loop, you are consuming the thread for this application, and the UI will not respond while the loop is running.

CAUTION

This takes about eight seconds to run on my machine. It might run a significantly longer or shorter time on your machine. The good news is the VB.NET IDE runs on a separate thread, so if you find yourself waiting forever, just click on the IDE and choose `Stop Debugging` from the `Debug` menu.

```

Private Sub button4_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles button4.Click
    BeBusy()
End Sub

Sub BeBusy()
    Dim i As Decimal
    For i = 1 To 10000000
        'do nothing but tie up app
    Next
    Beep()
End Sub

```

To create a thread, you must use the `System.Threading.Thread` class. In the creation of the class, you pass in the name of the procedure or method you want to run on that thread. You preface the procedure or method name with the `AddressOf` operator. Your code would look like this:

```
Dim busyThread As New System.Threading.Thread(AddressOf BeBusy)
```

To fix the code and keep `BeBusy` from consuming the main program thread, you have now created a new thread and will run `BeBusy` on that thread. However, that line of code isn't enough. Next, you must call the `Start` method on that new thread. With VB.NET, calling `BeBusy` on its own thread would look like this:

```

Private Sub button4_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles button4.Click
    Dim busyThread As New System.Threading.Thread(AddressOf BeBusy)
    busyThread.Start()
End Sub

```

No changes are required to the `BeBusy` procedure. If you now run this code, the interface will remain active while `BeBusy` is running. The `Beep` in `BeBusy` will let you know when the procedure has finished running.

There are some caveats to using free threading. They seem significant, but each one has a fairly reasonable workaround. Some of those caveats and workarounds are as follows:

- The procedure or method you run on a new thread cannot have any arguments. To get around this problem, you have a couple of choices. You could use global variables, but that is not an elegant solution. Instead, create properties or fields in the class whose method you are calling, and set those properties after the object is created. Obviously, that wouldn't help you in the earlier example because the call was simply to a sub in the same program.
- The procedure or method you run on a new thread cannot return a value. To get around that issue, you could use global variables, but again, this is not an elegant solution. One major problem is that your application would have to keep checking the thread to see when it was done, before you would be safe in using that global variable. Instead, you should consider raising an event with the return value as a parameter in the event.

Synchronization is also an issue with free-threaded applications. If you are performing a complex series of calculations on one thread, other parts of your application must wait until that thread is finished before they can use the results. You can monitor threads to see when they are finished or you can have the methods on those threads raise events to notify you when they are done. VB.NET provides an `IsAlive` property for each thread, so you can check to see when the thread is running.

There is much more to free threading, but it is not a topic that will be covered in more detail in this book.

Garbage Collection

Garbage collection is now being handled by the runtime. In VB6, if you set an object to `Nothing`, it was destroyed immediately. This is no longer true in VB.NET. Instead, when you set an object to `Nothing` or it loses all its references, it becomes garbage collectable. It's still in memory, taking up resources. The garbage collector runs on a separate thread, and it passes by occasionally looking for objects to clean up (destroy). This lag is often less than a second, but it could be longer on a system in heavy use. Even though the object has been marked for garbage collection, any resources it has opened are still open, including any data or file locks that it might have obtained.

Microsoft calls this “no deterministic finalization” because the developer is no longer truly in control of when the object will be destroyed. The `Sub Dispose` is called when the object is truly destroyed by the garbage collector. You can call the garbage collector by using the `Collect` method of the `GC` class in the `System` namespace.

Summary

There are a number of changes to the VB language in the move to VB.NET. It is important to understand them as you move forward. Some changes are minor, such as the requirement to use parentheses and the lack of default properties. More significant changes on the list are features such as the new `Try...Catch...Finally` error handling.

Finally, there are some critical new features. Namespaces are the most dramatic change that will affect every VB.NET developer. Inheritance and free threading will also make major changes in the way applications are written.

Now that you have seen many of the significant changes in the language itself, each of the next chapters covers a single topic, showing either new functionality (such as Windows Services and Web Services) or changes to how things are done (such as building classes and assemblies).



CHAPTER 4

Building Classes and Assemblies with VB.NET

During the last three or so years, most VB developers have spent a great percentage of their time building COM components. These components are used as the middle-tier components in n-tier systems. The benefits of building n-tier applications are well known, and include:

- Code reuse
- Elimination of many or all distribution headaches
- Encapsulation of business logic to control business processes and access to databases

Not surprisingly, VB.NET lets you build components, but they are no longer COM components. COM components have certain elements such as class IDs (CLSIDs), type libraries, and interface IDs (IIDs). Each class in a COM component has to support `IUnknown` and `IDispatch`.

VB.NET refers to one or more classes compiled into a file as a *class library*, rather than a COM component. Class libraries are compiled into an assembly, which often has a `.DLL` extension. You can use the classes from the class library much like you would the classes from a COM component: You instantiate the objects in the client application and then call properties and methods and respond to events. However, assemblies are *not* COM components; instead, they are .NET assemblies.

Creating Your First Class Library

To see how to build your first class library, start Visual Studio.NET and from the Start Page, click Create New Project. From the New Project dialog box, choose Visual Basic Projects in the Project Types list box, and then choose Class Library in the Templates list box. Name the project Healthcare and click the OK button.

At this point, a new class is created for you. The first thing you might notice is that the class does not have a designer by default, making it different from most VB.NET application types. Instead, you start with basically the same thing you had in VB6: an empty class.

Right now, you have one class, named `Class1`. Here is where things start to diverge from VB6. In VB6, you had one class per class module, and these were compiled into a single component. Class modules had a `.CLS` extension. In VB.NET, your module has a `.VB` extension, and a single source code file can contain more than one class. One or more source code files can be compiled into an assembly. You can create a new class at any time using the `Class...End Class` block.

In the code window, change the class definition line to name the class `Patient`. In other words, change this line:

```
Public Class Class1
```

to this:

```
Public Class Patient
```

You have now changed the class name, but if you look in the Solution Explorer window, or just look at the tab on the current code window, you see the filename is still `Class1.vb`.

Right-click on the `Class1.vb` icon in the Solution Explorer and choose Rename. Name the file `Healthcare.vb`. You should see the tab in the main code window change to reflect the new filename. What you have done is change the name of the file that holds one or more classes.

Adding a “Souped-Up” Class

You already have your first class, which was created for you when you created the class library. This class is now called `Patient`, but it does not have a constructor (`Public Sub New`) as most new files in VB.NET do. You can add your own constructor if you want your class to have one.

It is also possible to create a new class that comes with a designer and a constructor already created. In the Solution Explorer, right-click on the project name, choose

Add, and then choose Add Component. From the dialog that opens, choose Component class and click Open. This adds a new file to the project, which has only a class in it. This class, however, includes a designer, as shown in Figure 4.1.

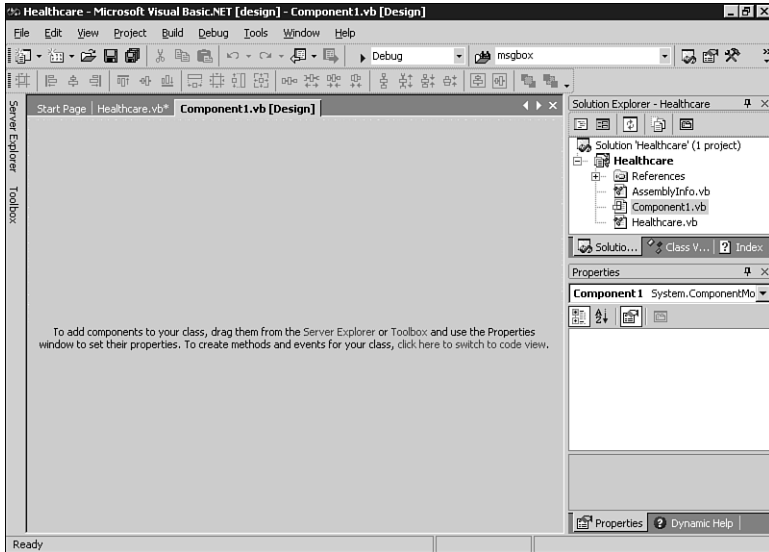


Figure 4.1

A new class created with a designer and a constructor already created.

The new class includes a designer, so you can perform actions such as dragging database connections to the designer to make it easier to create data components.

Double-clicking on the designer opens the code window. You can see that there is more code in this class than in the class that was created with the class library project. If you examine the code, you'll notice the following line:

```
Inherits System.ComponentModel.Component
```

This line makes the class inherit from the base Component class in System.ComponentModel. If you added that line to your Healthcare class, you would have access to the designer there.

For now, you'll return to the generic Healthcare class you created earlier.

Creating Properties

You can now start adding properties to your class, but be aware that the syntax for creating Visual properties has changed.

Add a `FirstName` property by first creating a private variable. Inside the class, add the following code:

```
Dim msFirstName as String
```

Now, add the following code. Realize that as soon as you type the first line, most of the rest of the code will be filled in for you by the IDE. You'll still have to add the `Return` and the assignment.

```
Public Property FirstName() As String
    Get
        Return msFirstName
    End Get
    Set(ByVal Value As String)
        msFirstName = Value
    End Set
End Property
```

Notice the new syntax for creating properties. No longer do you create matching `Public Property Get`/`Public Property Let` procedures. Instead, you create a `Public Property` block and then add a `Get` and a `Set` section to it. There is no `Let` anymore, which makes life easier.

Also notice that this property does not accept an argument, as you would have had to do with a `Public Property Let` in VB6. This means that you no longer have to have the value passed in as a parameter; instead, if the user passes in a value, it is put in the `Value` variable automatically.

Building a Test Client

Now, it is time to test this class. True, it has only one property, but you need to see how to call this class in a client application. From the `File` menu, choose `New` and then `Project`. This time, add a `Windows Application`. Name it `HealthcareClient` but before you click `OK`, make sure that you select the `Add to Solution` radio button. The default is to close the current solution and open a new one. By choosing to add this new project to the current solution, you have the equivalent of a VB6 group.

After you click the `OK` button, the new project is loaded into the `Solution Explorer`, as shown in Figure 4.2. As in VB6, the project name that appears in bold in the `Solution Explorer` is the project that will start when you start the application. Simply right-click on the `HealthcareClient` project and choose `Set as StartUp Project` from the pop-up menu.

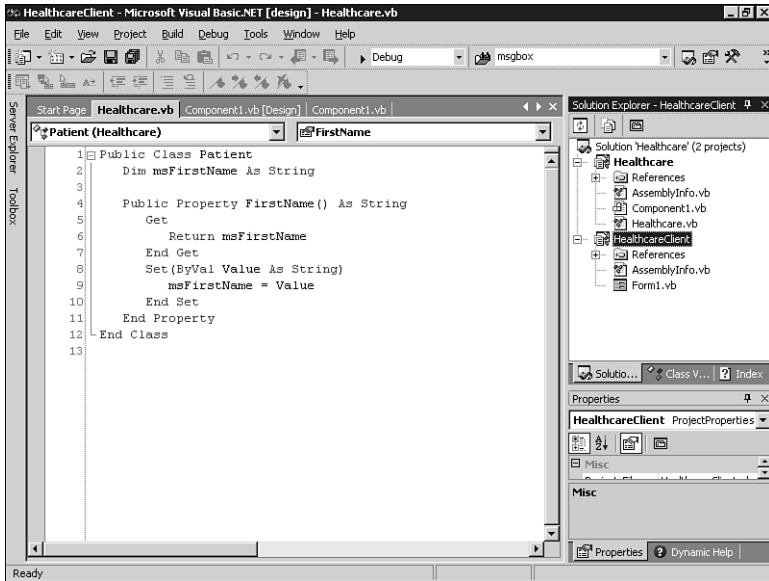


Figure 4.2

The Solution Explorer showing two projects loaded at once.

In the Solution Explorer, right-click on the References node for the HealthcareClient project and choose Add Reference. The Add Reference dialog box will appear. Click on the Projects tab and you should see your Healthcare project. It is already highlighted, but the OK button is disabled, as you can see in Figure 4.3. Click the Select button to move Healthcare into the Selected Components box, and then click the OK button.

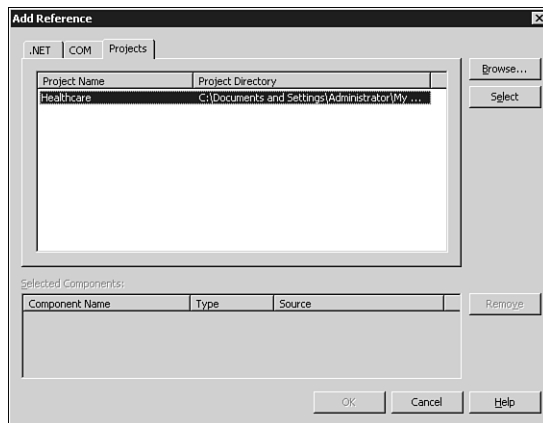


Figure 4.3

The Add Reference dialog box.

Now, on the form in `HealthcareClient`, add a button. Double-click the button to get to the code window, and enter the following code for the `Button1_Click` event procedure:

```
Protected Sub button1_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles button1.Click  
    Dim Patient As New Healthcare.Patient()  
    Patient.FirstName = "Bob"  
    msgbox(Patient.FirstName)  
End Sub
```

This code should look very familiar to VB6 developers. After adding a reference, you instantiate the object by setting the variable `Patient` to a `New Healthcare.cPatient`. If you are used to VB6, you might be tempted to try this shortcut and type this line of code:

```
Dim Patient As New Patient()
```

If you type the line this way, however, you'll get an error that says `User-defined type not defined: Patient`. This shows that you need to have the name of the component fully qualified. The word `Healthcare` in this case is not the assembly name; instead, it is the namespace.

To get around this problem, you can import the namespace containing the `Patient` class. Remember that all projects have a default namespace, and the default name of this default namespace is the same as the project. Therefore, if you go to the top of the code module and add an `Imports` statement, the "shortcut" reference to the `Patient` class will work. Your `Imports` statement must go at the top of the module, and it will look like this:

```
Imports Healthcare
```

You might have noticed in the code that you used the `New` keyword to create the object. In VB6, you could use the `New` keyword in two ways. Here is the first way:

```
Dim Patient as New Healthcare.Patient
```

In VB6, this code works, but it is not the best way to create objects. When you use this method, the object is not actually created until you call the first property or method. In fact, each call to a property or method requires a check to see whether the object has already been instantiated. To avoid this overhead, you should have been using this method in VB6:

```
Dim Patient as Healthcare.cPatient  
Set Patient = New Healthcare.cPatient
```

In VB.NET, however, the following two methods are considered equivalent:

```
Dim Patient As New cPatient()
```

```
Dim Patient As cPatient = New cPatient()
```

In both of these lines, the object is created in memory immediately. That means you never have to worry about the overhead of checking whether the object was instantiated on each call to a property or method that you could have in VB6.

Read-only and Write-only Properties

Returning to the class library you are creating, you should notice that the property you created, `FirstName`, has both a `Get` and `Set` section in the Property block. In VB6, to make a property read-only, you simply did not create a `Public Property Let` statement. To create a write-only property, you did not create the `Public Property Get`.

You might be tempted to try to create a read-only property by simply leaving out the `Set...End Set` block. However, VB.NET handles read-only and write-only properties differently: You must add a keyword to the property declaration. To create a read-only property named `Age` (you can assume that it's calculated from the person's date of birth), your code would look like this:

```
Public ReadOnly Property Age() As Single
    Get
        'get Date of Birth (DOB)
        'calculate age from DOB
        'return age
    End Get
End Property
```

Creating a write-only property is equally simple. Just put the keyword `WriteOnly` in the property declaration, and have only a `Set...End Set` block in the property procedure.

Parameterized Properties

It is possible to create a parameterized property. Using the example of a patient, consider that a patient is likely to have several physicians attending to him at any one time. Therefore, although your `Healthcare` class library might have a `Physician` class, the `Patient` will likely hold a collection of `Physician` objects. You would be able to access this collection through a parameterized property and walk through this collection.

Creating a parameterized property is fairly straightforward: You simply add a parameter to the property procedure. If you have a `Physicians` property that walks through a collection of `Physician` objects, the code would look something like this:

```
Dim PhysiciansList As New Collection()  
Public ReadOnly Property Physicians(ByVal iIndex As Integer) _  
    As Physician  
    Get  
        Return CType(PhysiciansList(iIndex), Physician)  
    End Get  
End Property
```

You might notice several unusual things in this code. First of all, the basic functionality is there, in that you pass in an index value and get back a particular object in a collection. Notice also that even though the `PhysiciansList` is defined as a `Collection` data type, `Option Strict` prevents an automatic conversion from a `Collection` type to a `Physician` type. Therefore, you must run the `CType` function and pass both the object you want to convert and the type of the class (or object) into which it should be converted. Only then will the return actually succeed.

Default Properties

At the beginning of Chapter 3, you learned that default properties were gone, with the caveat that default properties *without parameters* were gone. However, if a property has one or more parameters, it can be the default property. Therefore, the `Physicians` property in `Patient` could be a default property.

Making a property the default property is as simple as adding the word `Default` in front of the property declaration. To make the `Physicians` property the default, your declaration would look like this:

```
Default Public ReadOnly Property _  
    Physicians(ByVal iIndex As Integer) As Physician
```

Now, in your client program, you could call the default property on the `Patient` object. The last two lines of this code snippet are equivalent:

```
Imports Healthcare  
...  
Dim Patient As New Patient()  
Dim Phys As New Physician()  
Phys = Patient.Physicians(1)  
Phys = Patient(1) 'equivalent to line above
```

Constructors in Your Classes

The one class you have built so far, `Patient`, did not come with a constructor; in other words, there was no `Sub New` available when you first created the class library. Constructors can be quite useful because they allow you to instantiate an object with some values already in it.

For example, assume that you wanted the Patient to allow you to pass in a PatientID when you instantiated the object. That means you could write code that would create the object and, at creation time, read a database and fill in the properties for that patient. Your new definition for the Sub New would look like this:

```
Public Sub New(Optional ByVal iPatientID As Integer = 0)
```

Your client code could now instantiate the object and pass in a value at instantiation. Either of the following lines would allow you to create an instance of the object with a value already set:

```
Dim Patient As New Patient(1)
```

```
Dim Patient As Patient = New Patient(1) 'equivalent to line above
```

Classes Without Constructors

Obviously, your classes do not have to have constructors. If you just use the Class...End Class block to create a new class, you will not be provided with a Sub New. Whether or not a class has constructors or implements System.ComponentModel.Component, the class can still have properties and methods, and can be created just like any other class. The Physicians class that has been used in previous examples could look something like this:

```
Public Class Physician
    Dim miPhysID As Integer
    Public Property PhysicianID() As Integer
        Get
            Return miPhysID
        End Get
        Set
            miPhysID = Value
        End Set
    End Property

    Public ReadOnly Property Age() As Single
        Get
            'get Date of Birth (DOB)
            'calculate age from DOB
            'return age
        End Get
    End Property
End Class
```

Adding Methods to Classes

Adding a method to your class is done the same way it was done in VB6. If you want to create an `Admit` method in the `Patient` class, your code might look something like this:

```
Public Function Admit() As Boolean
    'add patient to database, notify billing, etc.
    Return True
End Function
```

The call to this from the client is equally simple:

```
If Patient.Admit Then...
```

Before you begin thinking that there aren't any changes in methods, understand that there are major changes in defining methods. However, the changes are specific to inheritance, and they will be covered in Chapter 5.

Adding Events

To add an event to your class, use the `Event` keyword. Imagine that you wanted to add an event to the class that notified you when you had pending lab results. You could create the event using code like this inside the class that needs to fire the event:

```
Event LabResult(ByVal LabType As String)
```

This just creates the event definition in your code. To actually cause the event to fire, you'll have to add a `RaiseEvent` statement elsewhere in the code. For example, if you set a `PatientID` property, you can go check a database for any new lab results for that patient. If there are new lab results, you could raise an event. Your code would look similar to this:

```
Dim miPatientID As Integer
Public Property PatientID() As Integer
    Get
        Return miPatientID
    End Get
    Set
        miPatientID = Value
        'check labs database for this patient
        'if there are new lab results
        RaiseEvent LabResult("CBC")
    End Set
End Property
```

If this were a real procedure, you wouldn't hard-code "CBC" into the event, but would instead pull the lab type from the database.

Handling the Event in the Client

You have two options for handling this event in the client. The first way to handle the events is using the `WithEvents` keyword. The second way is to use the `AddHandler` statement.

To use the `WithEvents` keyword, you need to declare the object and include `WithEvents`. Notice that if you are using the `WithEvents` keyword, the declaration cannot be local to a sub or function. Therefore, this code will go outside any sub or function, at the “module” level in your client:

```
Dim WithEvents Patient As Patient
```

This line assumes that you have imported the `HealthCare` namespace. Notice that the keyword `New` is *not* in the preceding statement. You cannot use the `WithEvents` keyword in a declaration inside a procedure, nor can you use the `New` keyword in a declaration outside a procedure to create an object that has events. Therefore, you have to declare the object using the `WithEvents` keyword outside of any procedure, and then you have to use the `New` keyword inside a procedure, as shown here:

```
Dim WithEvents Patient As Patient
...
Public Sub Foo()
    Patient = New cPatient()
    ...
End Sub
```

Now, to add an event procedure, click on the Class Name drop-down list box at the top of the code window and choose `Patient`. In the Method Name drop-down list box, choose `LabResult`. This creates an event procedure named `Patient_LabResult` that looks like this:

```
Public Sub Patient_LabResult(ByVal LabType As System.String) _
    Handles Patient.LabResult
    ...
End Sub
```

The second way to handle events is to use the `AddHandler` statement. You now do not have to define the class using the `WithEvents` keyword. Instead, you define it as you did before. You must also have a sub or function that will act as the event procedure.

Next, you use the `AddHandler` statement to tie a particular event from the object to the procedure you created to handle the event.

For example, assume that you wanted to create an event handler sub called `LabHandler`. This procedure would be defined as a standard sub. It would have to take as arguments any parameters defined in the `LabResult` event. In the example here, the event named `LabResult` passes along a `lab` parameter with a data type of `string`. Therefore, your procedure would have to accept a `string` as an argument.

When you use `AddHandler`, you specify the name of the event in the `Patient` class you want it to handle (`LabResult`), but you must also refer to the procedure; in this case, `LabHandler`. However, you don't refer directly to `LabHandler`; instead, you refer to the address of `LabHandler`, using the `AddressOf` operator. Your code to set up event handling with the `AddHandler` statement would look like this:

```
Protected Sub button1_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs) Handles button1.Click  
    Dim Patient As New Patient()  
    AddHandler Patient.LabResult, AddressOf Me.LabHandler  
    ...  
End Sub  
  
Private Sub LabHandler(ByVal LabType As String)  
    ...  
End Sub
```

Notice that in the `AddHandler`, you refer to the object variable (`Patient`) and the event name (`LabResult`) using the *Object.Event* syntax.

The advantage of this approach is that you can have one procedure handling multiple events, even if the events are from different components. You can also hook events into objects that you create at runtime. For example, you might get a collection of objects passed to you, but still be able to handle their events using this procedure.

The “Final” Code

The following code doesn't actually do anything, and some earlier changes are undone in this code. Still, if you are trying to keep up with the code, and want to make sure that the compilation works in the next section, here is the code inside the `Healthcare` project:

```
Public Class Patient  
    Dim msFirstName As String  
    Dim PhysiciansList As New Collection()  
    Dim miPatientID As Integer  
  
    Public Property FirstName() As String  
        Get  
            Return msFirstName  
        End Get  
        Set(ByVal Value As String)  
            msFirstName = Value  
        End Set  
    End Property
```

```

Default Public ReadOnly Property Physicians _
(ByVal iIndex As Integer) As Physician
    Get
        Return CType(PhysiciansList(iIndex), Physician)
    End Get
End Property

Public Function Admit() As Boolean
    'add patient to database, notify billing, etc.
    Return True
End Function

Event LabResult(ByVal LabType As String)

Public Property PatientID() As Integer
    Get
        Return miPatientID
    End Get
    Set(ByVal Value As Integer)
        miPatientID = Value
        'check labs database for this patient
        'if there are new lab results
        RaiseEvent LabResult("CBC")
    End Set
End Property
End Class

Public Class Physician
    Dim miPhysID As Integer
    Public Property PhysicianID() As Integer
        Get
            Return miPhysID
        End Get
        Set(ByVal Value As Integer)
            miPhysID = Value
        End Set
    End Property

    Public ReadOnly Property Age() As Single
        Get
            'get Date of Birth (DOB)
            'calculate age from DOB
            'return age
        End Get
    End Property
End Class

```

Compiling the Assembly

Now that you have created a class library with two classes (Patient and Physician), it is time to compile your assembly. In VB6, you would compile a COM component, but you aren't in VB6 anymore. Instead, you are writing for the .NET Framework, and that means you will be compiling an assembly. The assembly might have a .DLL extension, but it is not a traditional DLL in the Windows API sense, nor is it a COM DLL in the VB6 sense.

Building the assembly is fairly easy. The build option is no longer on the File menu, but is now a separate menu labeled Build. If you click on the Build menu, you will see several choices. Because you have a solution in this example, with a class library (Healthcare) and Windows application (HealthcareClient), you will see options to build or rebuild the solution. You also have an option to deploy the solution. In the next section of the Build menu, you have the option to build or rebuild one of the projects in the solution, depending on which project is highlighted in the solution explorer when you click the menu. Finally, there is a choice for a batch build and one for the Configuration Manager.

In .NET, you can compile your assemblies in either Debug or Release mode, or you can create your own custom modes. Debug mode compiles in symbolic debug information and does not use any compiler optimizations. Release mode does not compile in any of the symbolic debug information, and it applies code optimizations. Obviously, you will use Debug mode while developing and debugging your application. After you have the bugs worked out and are ready to deploy the application, you switch to Release mode and recompile your assembly. Realize that you can modify these modes or add your own. For example, you could add debug information into projects compiled under Release mode, if you chose to modify Release mode in that fashion.

You can see that the Debug option is set if you look at the toolbar. The Solutions Configuration drop-down list box allows you to choose Debug or Release, and to open the Configurations Manager. You can see this in Figure 4.4.

For now, just click on the Healthcare project one time in the Solution Explorer window. This will select it as the project to build if you choose not to build the entire solution. Now, click on the Build menu and choose Build Healthcare. The Output window at the bottom of the IDE should report when the build is done, and show how many projects succeeded, how many failed, and how many were skipped. In this case, the HealthcareClient is not counted as skipped, so you should see that one project succeeded, with no failures and no projects skipped.

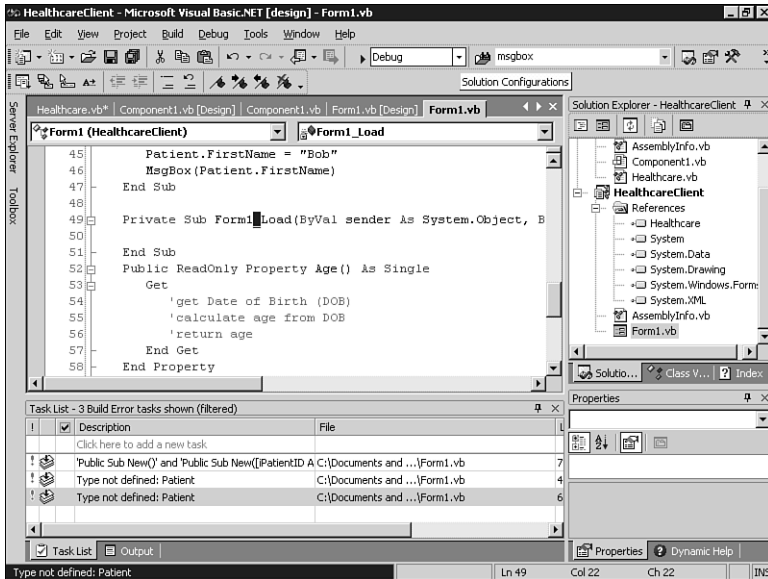


Figure 4.4

The Solutions Configuration option on the toolbar.

The compilation process creates two files. The default location for the project is My Documents\Visual Studio Projects\Healthcare. If you look in that folder, you'll see a bin folder below it. Inside the bin folder are two files: Healthcare.dll and Healthcare.pdb. The .DLL is the assembly, and the .PDB is the debug file that goes along with the .DLL because you compiled it with the Debug configuration. .PDB stands for *program database*.

Reusing the Assembly in Other Applications

One of the other big differences from this and VB6 is that VB.NET assemblies do not use the registry. In fact, VB.NET does not put any information about the assembly in the registry. If you search through the registry, you won't find any mention of the Healthcare assembly. You will find information about the project and solution, but that's just so that Visual Studio.NET can show them in the recent files list.

Given that there is nothing about the assembly in the registry, how do you reuse it in another application? If you start a new Windows Application project in VB.NET and right-click on the References node in the Solution Explorer window, the Add Reference dialog box is shown. However, the Healthcare component is not listed. This shouldn't be too surprising; it showed up earlier because HealthcareClient was part of the same solution. Now, however, you have started a new project, which creates a new solution. This solution is in a different directory, and therefore has no knowledge of the Healthcare component.

To have the new project use the Healthcare component, make sure that you are on the .NET Framework tab of the Add Reference dialog box and click the Browse button. Navigate to the My Documents\Visual Studio Projects\Healthcare\bin folder and choose Healthcare.dll. The References node in the Solution Explorer window now shows the Healthcare component.

Your application can now use the Healthcare component. You can create objects by typing the following code:

```
Dim Patient As New Healthcare.Patient()
```

You can always use the `Import` command to import the Healthcare namespace, so you could instantiate objects with just this code:

```
Dim Patient As New Patient()
```

How .NET Locates Assemblies

Think back to COM: When a program makes a call to a component, the Service Control Manager goes to the registry, looks up the component information, locates the object, creates it, and returns to the calling application a pointer to the newly created object. .NET doesn't use the registry, so it has to use a different mechanism for locating assemblies that are referenced in an application (or in another assembly). The full details are not critical to understanding how to develop components, but a basic understanding of the process is important for troubleshooting.

Step 1: Getting the Reference Information

Your application attempts to call a referenced assembly. The reference contains the following information about the assembly:

- Name
- Version
- Culture
- Public Key (for strongly named assemblies)

The runtime uses this information to try to locate the assembly, using the following steps.

Step 2: Looking at the Configuration File

The configuration file is a powerful concept. You can create a configuration file to handle any reference information that might change. For example, if you think you will need to update the particular version of an assembly used by your application,

you can store that information in a configuration file. You can also force an application to bind to only the versions of components that were available when the application was built. This is called Safe Mode. The configuration file is an XML file.

Step 3: Using CodeBases or Probing

If you want to prevent probing, you can specify a `CodeBase` value in the configuration file. When the runtime loads the file pointed to by the `CodeBase`, it checks the version information to make sure it matches what is in the application's reference. By default, if the assembly pointed to by the `CodeBase` is the same version or higher, the binding will occur.

If there is no `CodeBase`, the runtime looks first in the GAC if the assembly is strongly named. If the assembly is private, the search is performed only in the application's root directory, referred to as the *AppBase*. The runtime sees a reference to an assembly in the application, such as `Healthcare`, and looks for the assembly by first searching for `Healthcare.DLL`.

If the assembly is not found in the *AppBase* directory, the runtime searches based on the path set up in the configuration file. This path is specified with the `<AppDomain>` tag, as such:

```
<probing privatePath="MyAssemblies" />
```

The runtime automatically appends some other information to the search path. For example, if the assembly you are searching for is `Healthcare`, the runtime looks in `AppBase\Healthcare`. It also looks in the `bin` directory, so it searches `AppBase\bin`. Because you specified `MyAssemblies` in the `<AppDomain>` tag, the runtime searches `AppBase\MyAssemblies`. Finally, it appends the locale, so if the locale is `de`, the search path will include `AppBase\Bin\de`, `AppBase\Healthcare\de`, and `AppBase\MyAssemblies\de`.

Step 4: The GAC

Recall that the Global Assembly Cache is a cache for assemblies to be used by multiple applications on the same machine. Even if an assembly is found through probing or `CodeBases`, the runtime looks for updated versions in the GAC, provided you have not turned off this functionality. If a higher version exists in the GAC, it is used.

If no assembly was found using a `CodeBase` or probing, the runtime searches the GAC for a match. If a matching assembly is found, it is used.

To add assemblies to, remove assemblies from, or view assemblies in the GAC, use the `gacutil.exe` console application, provided by the Framework. The GAC is also a directory, so you can add assemblies to the GAC merely by copying them into the directory.

Step 5: The Administrator Steps In

It is possible for the administrator to create a file that specifies a particular version of an assembly to be used. If an application is searching for version 2.1.4.0 of Healthcare, the Administration policy file can point the application to another version, forcing an upgrade or downgrade.

Summary

Creating classes (or objects) is one of the most common things for a VB developer to do. Class creation is different in VB.NET, as you can tell from this chapter. It changes much more when you start looking at inheritance, as you will in the next chapter.

The main differences are how you create properties, how you handle read-only, write-only, and default properties, and how you can handle events. There is also a big difference in the fact that you compile assemblies instead of COM components. Finally, the way files are located is also very different.

There is an important concept in dealing with assemblies that is not covered due to time and size constraints, and that is how assemblies handle versioning. Look for this information in an updated book after Visual Studio.NET is released.



CHAPTER 5

Inheritance with VB.NET

What Is Inheritance?

For years, one of the most requested features from Visual Basic developers was inheritance. Microsoft has had one form of inheritance, *interface inheritance*, available in VB for several years. Developers, however, wanted to have what is known as *implementation inheritance*. With VB.NET, developers finally get their wish, thanks to the CLR. And, if you're unclear on what interface and implementation inheritance are, read on.

Inheritance is a very powerful form of code reuse. With inheritance, you write one class that contains some properties and methods. That class becomes the basis for other classes, and is known as the *base class*. Classes that then use this base class are known as *derived classes*. Most often, the derived classes extend the functionality of the base class.

Imagine if you wanted to create an application to model part of a university. As part of your application, you wanted to create two objects, representing students and professors. Students and professors both have a number of things in common: Each has a name, an address, and so forth. However, each also has some differences. Students have declared majors, whereas professors have classes they can teach. Students are charged tuition, whereas professors are paid for their work. Professors submit grades, whereas students receive them.

As you can see, there are similarities and differences that must be taken into account when these objects are built. The goal of inheritance is to write the code for the similarities once, and then use it in both the Student and Professor objects.

Interface Inheritance in VB6

In VB6, you had what was called *interface inheritance*. With interface inheritance, you could create the interface (or shell, or definition) of a reusable component. Suppose that you wanted to create a Person object that had all the shared attributes of both students and professors. In the case of VB6, you would create a class module that contained the definitions of those attributes, but no actual code. Your interface class might be named `iPerson` and look like this:

```
Public Property Get Name() As String
End Property
Public Property Let Name(psName As String)
End Property

Public Property Get Address() As String
End Property
Public Property Let Address(psAddress As String)
End Property

Public Function Enroll() As Boolean
End Function
```

In VB6, you could create an interface using a class module. Technically, VB could not create interfaces, but you could use the hack of creating a class module with only definitions for properties and methods, and then implement it in another class. This was actually just creating a class, but there would be no implementation code in it, so it “looked like” an interface. For example, the `Enroll` method does not have any code to actually handle the enrollment. And if you are curious why there is an `Enroll` function in the `iPerson` interface, it is because professors can also take courses at the university, which some do because they can do so for free at most institutions.

To use this interface in other classes in VB6, you had to put this line of code in your class:

```
Implements iPerson
```

This line set up your new class to inherit the interface from the `iPerson` class. In VB6, you could actually put implementation code into the interface class, but it was ignored when you implemented the interface in your derived class.

Imagine that you have used the `Implements` keyword in a class named `cStudent`. This class must now implement every public property, method, and event found in `iPerson`. Therefore, your code in `cStudent` would include code similar to the following:

```
'VB6 Code
Public Property Get iPerson_Name() As String
    iPerson_Name = msName
End Property

Public Property Let iPerson_Name(psName As String)
    msName = psName
End Property

Public Property Get iPerson_Address() As String
    iPerson_Address = msAddress
End Property

Public Property Let iPerson_Address(psAddress As String)
    msAddress = psAddress
End Property

Public Function iPerson_Enroll() As Boolean
    'open database
    'check to see if class is filled
    'if not, add student
End Function
```

The code in `cStudent` contains the implementation of the interface designed in `iPerson`.

Looking at this, you'll probably realize that most of the time, the `cStudent` implementation code will end up being the same in `cProfessor`, at least for these class members. Wouldn't it be easier to create this implementation code in `iPerson` and then just have that code brought into both `cStudent` and `cProfessor`? That's what implementation inheritance is all about.

VB.NET's Implementation Inheritance

In VB.NET, you don't have to create a separate interface, although you can. Instead, you can just create a class with implementation code and inherit that in other classes. This means you could add the code for the `Name` and `Address` properties and the `Enroll` method in a `Person` class. Then, `Professor` and `Student` could both inherit from the base class `Person`. `Student` and `Professor` would both have access to the code inside of `Person`, or they could replace it with their own code if needed.

One thing to realize is that in VB.NET, all classes are inheritable by default. This includes forms, which are just a type of class. This means you can create a new form based on an existing form.

Even though VB.NET provides true implementation inheritance, interfaces are not gone. In fact, they are greatly improved in VB.NET; together, inheritance and interfaces give you the ability to use polymorphism. This will be examined at the end of this chapter.

A Quick Inheritance Example

Create a new VB.NET Windows Application project and name it InheritanceTest. Add a button to the form and go to the code window. In the code, add the following class. Make sure that you add it *outside* the class for the form!

```
Public Class Person

    Dim localName, localAddress As String

    Property Name() As String
        Get
            Name = localName
        End Get
        Set(ByVal Value As String)
            localName = Value
        End Set
    End Property

    Property Address() As String
        Get
            Address = localAddress
        End Get
        Set(ByVal Value As String)
            localAddress = Value
        End Set
    End Property

    Public Function Enroll() As Boolean
        'check class enrollment
        'if enrollment < max. class size then
        'enroll person in class
        Enroll = True
    End Function
End Class
```

This code creates a Person class with two properties, Name and Address, and an Enroll method. So far, there is nothing about this class that you haven't seen before.

Now, add a second class, called `Student`. Your code should look like this:

```
Public Class Student
    Inherits Person
End Class
```

As you can see, there isn't any implementation code in `Student` at all. There are no properties or methods defined. Instead, all you do is inherit from `Person`.

Now, in the `Button1_Click` event handler on the form, add the following code:

```
Dim Student As New Student()
MsgBox(Student.Enroll)
```

Your form is creating an instance of the `Student` class. The only code in the `Student` class is an `Inherits` statement that inherits the `Person` class. However, because VB.NET supports inheritance, you'll be able to call the `Enroll` method of the `Person` class from within the `Student` class, even though the `Person` class does not explicitly define an `Enroll` method. Instead, the `Enroll` method is present because `Student` is inheriting the method from `Person`. Go ahead and run the project to verify that the message box does report back a value of `True`.

Is it possible to instantiate `Person` directly? In this case, yes. For example, you could modify the `Button1_Click` event handler to look like this:

```
Dim Student As New Student()
Dim Person As New Person()
MsgBox(Student.Enroll & " - from Student")
MsgBox(Person.Enroll & " - from Person")
```

Both of these calls to the `Enroll` method will work fine.

This might raise a host of questions. For example, could `Student` redefine the `Enroll` method so that it is not using the code in `Person`? The answer is yes. Could `Person` refuse to let someone instantiate it directly, instead forcing people to instantiate other classes that inherit it? Again, the answer is yes. You will see this and more as you examine more that you could do with inheritance.

Shared Members

VB.NET introduces the concept of shared members. *Shared members* are a way of creating a member (a property, procedure, or field) that is shared among all instances of a class. For example, you could create a property for a database connection string. This will be the same for each class, so you can fill it in for one class and all classes can then see that one property. Shared properties are most often used in inheritance, so that all objects created from derived classes can share the same member across all instances of the class. However, shared members can be used without regard to inheritance.

Imagine that you have an XML file or other persistence mechanism for the student data listed earlier, and you want to define a method to be able to find a student, given his name, and return the student object. Rather than the COM model of having a factory to create the object, you could add the `Find` functionality as a shared method to the student class. For example:

```
Public Class Student
    Inherits Person

    Public Shared Function Find(ByVal studentName As String) _
        As Student
        Dim Student As New Student()
        Dim xmlobj As Xml.XmlElement
        'get the xml object
        'fill in the mName field
        'fill in the mAddress field
        Return Student
    End Function
End Class
```

Shared methods are commonly used in the runtime for this ability to create a specific instance of an object, for example `System.IO.CreateDirectory()` that will create a directory in the file system, and return a `DirectoryInfo` object. The same `System.IO.Directory` class also provides a shared method, `Move`, to rename a directory.

Inheritance Keywords

There are a number of keywords associated with inheritance. Remember that by default, all classes you create are inheritable. You inherit from classes using the `Inherits` keyword. The class from which you inherit is then known as the *base class*. The `Inherits` keyword can be used only in classes and interfaces. It is important to point out that a derived class can only inherit from one base class.

Forcing or Preventing Inheritance

The `NotInheritable` modifier is used to mark a class as not inheritable; in other words, it cannot be used as a base class. If you were to modify the `Person` class in your `InheritanceTest` project, it would look like this:

```
Public NotInheritable Class Person
```

If you change the `Person` definition to look like the preceding line, `Student` will no longer be able to inherit from `Person`.

In contrast to the `NotInheritable` modifier, there is also a `MustInherit` modifier. `MustInherit` says that a class cannot be instantiated directly. Instead, it must be inherited by a derived class, and the derived class can be instantiated.

If you changed `Person` to include the `MustInherit` keyword, it would look like this:

```
Public MustInherit Class Person
```

Now, you would not be able to use the following line of code in the client:

```
Dim Person As New Person
```

However, you would still be able to inherit `Person` in your `Student` class, and your client could instantiate `Student`.

Overriding Properties and Methods

When you inherit a base class, the properties and methods cannot be overridden, by default. Given your earlier example in `InheritanceTest`, you could not have created a function named `Enroll` in `Student` because one existed in `Person`. There is a modifier, `NotOverridable`, that says a particular property or method cannot be overridden. Although methods are normally not overridable, you can use this keyword only in a unique case: If you have a method that is already overriding a base method, you can mark the new, derived method as `NotOverridable`.

If you want to allow a property or method to be overridden, you can mark it with the `Overridable` modifier, as shown here:

```
Public Overridable Function Enroll() As Boolean
```

Now, your `Student` can create its own `Enroll` method. To do so, your client will have to use the `Overrides` modifier, so it would look like this:

```
Public Overrides Function Enroll() As Boolean
```

In the VB.NET IDE, it is also possible to use the drop-down lists at the top of the code window to override methods or implement interfaces.

There is also a `MustOverride` modifier. This forces a derived class to override the property or method. The `MustOverride` modifier changes the structure of the property or method. Because the property or method must be overridden, you do not put any implementation code in it. In fact, there is not even an `End Property` or `End Sub` or `End Function` when using the `MustOverride` modifier. If a class has even a single property or method with the `MustOverride` modifier, that class must be marked as `MustInherit`.

For example, here you have a base `Transportation` class that has a method, `Move`, that is marked as `MustOverride`. This means that `Transportation` must be marked as `MustInherit`. The `Train` class inherits from `Transportation`, and then overrides the `Move` method.

```
Public MustInherit Class Transportation
    MustOverride Function Move() As Boolean
End Class

Public Class Train
    Inherits Transportation
    Public Overrides Function Move() As Boolean
        'code goes here
    End Function
End Class
```

Although you can override a base class property or method in your derived class, you can still access the properties or methods in the base class using the `MyBase` keyword. This allows you to call base classes members even though you have overridden them in your derived class.

For example, assume you wanted to have a `Transportation` class, with an overridable function called `Move`. `Train` then inherits from `Transportation`, and implements its own `Move` method, overriding the one in `Transportation`. However, `Train` can call its `Move` method or the one in `Transportation`. The method `CallMethods` calls first the `Move` in `Train`, and then the `Move` in `Transportation`. The user will see two message boxes. The first will have the text `Hello from the Train class`, whereas the second will have the text `Hello from the Transportation class`.

```
Public Class Transportation
    Overridable Function Move() As Boolean
        MsgBox("Hello from the Transportation class")
    End Function
End Class

Public Class Train
    Inherits Transportation
    Public Overrides Function Move() As Boolean
        MsgBox("Hello from the Train class")
    End Function

    Public Sub CallMethods()
        Move()
        MyBase.Move()
    End Sub
End Class
```

Related to `MyBase` is `MyClass`. Assume that, in your base class, you have method `A` calling an overridable method `B`. If you want to verify that the method `B` you call is the one you wrote in the base class, and not the derived, overridden method `B` in the derived class, call method `B` with the `MyClass` qualifier, as in `MyClass.B`.

For example, assume that you have a `Transportation` class that has two methods: `MakeReservation` and `BuyTicket`. `MakeReservation` calls `BuyTicket`.

MakeReservation and BuyTicket are both overridable. Your Train class can inherit Transportation, and create a BuyTicket method that overrides the BuyTicket in Transportation. If you don't create a MakeReservation in Train, your call to MakeReservation will use the code in the Transportation class. However, if the code in Transportation.MakeReservation calls BuyTicket, by default you'll call the BuyTicket you've created in Train. Here is the code:

```
Public Class Transportation
    Overridable Function MakeReservation() As Boolean
        'CheckSchedule
        BuyTicket()
        'etc
    End Function
    Overridable Function BuyTicket() As Boolean
        MsgBox("Generic Transportation implementation")
    End Function
End Class

Public Class Train
    Inherits Transportation

    Public Overrides Function BuyTicket() As Boolean
        MsgBox("Train-specific implementation")
    End Function
End Class
```

Now, suppose that you want the MakeReservation to call the BuyTicket method in Transportation, even if Train overrides BuyTicket. To accomplish this, just change Transportation.MakeReservation to this:

```
Public Class Transportation
    Overridable Function MakeReservation() As Boolean
        'CheckSchedule
        MyClass.BuyTicket()
        'etc
    End Function
```

In this case, if your client calls Train.MakeReservation, the MakeReservation method will call the BuyTicket in the base class (Transportation) instead of the overridden BuyTicket (if one exists in Train). However, it's important to note that if your Train class overrides MakeReservation, MyClass will not come into play. This is because you will be calling the overridden MakeReservation, which won't include the MyClass keyword.

Polymorphism

Polymorphism is the ability to change the implementation of a base class for different objects. For example, if you have a bicycle and a car, both can move, but they do

so in very different ways. They use different mechanisms for movement, and the distance that each can move in an hour is significantly different. Yet, both a Car and a Bike class might inherit from a base Transportation class, which could also be used as the basis for a Plane class, a Train class, a HotAirBalloon class, and so on.

Polymorphism with Inheritance

Polymorphism was possible in VB6 using interfaces, which will be examined in a moment. VB.NET allows you to perform polymorphism using inheritance. The difference from what you have done so far is that you can actually use the base class as a variable type, and you can handle any derived class with that new variable.

For example, examine the following code. The Transportation class contains just one method, Move. Two classes inherit the Transportation class: Car and Bicycle. Both have overridden the Move method. The code looks like this:

```
Public MustInherit Class Transportation
    Public MustOverride Function Move() As Boolean
End Class
```

```
Public Class Bicycle
    Inherits Transportation
    Overrides Function Move() As Boolean
        'code here
        Move = True
    End Function
End Class
```

```
Public Class Car
    Inherits Transportation
    Overrides Function Move() As Boolean
        'different code here
        Move = True
    End Function
End Class
```

So far, this looks similar. However, notice now what your client can do. It cannot directly create an instance of Transportation because it is marked as MustInherit. But, you can declare a variable of type Transportation. You can be assured that any object that inherits Transportation has a Move method, and you don't have to worry about what kind of object it is. Your client code might look like this:

```
Protected Sub Button1_Click _
    (ByVal sender As Object, ByVal e As System.EventArgs)
    Dim MyCar As New Car()
    Dim MyBike As New Bicycle()
```

```

        PerformMovement(MyCar)
        PerformMovement(MyBike)
    End Sub

Public Sub PerformMovement(ByVal Vehicle As Transportation)
    If Vehicle.Move() Then
        'do something
    End If
End Sub

```

You'll notice that the `PerformMovement` sub accepts an argument of type `Transportation`. This sub doesn't care if you pass it an object of `Car` or `Bicycle` type. Because the object being passed in inherits from `Transportation`, it is guaranteed to support the `Move` method, so the code will run without problems.

Polymorphism with Interfaces

Interfaces still exist in VB.NET. In VB6 and COM, they were most often used when you needed to be able to modify your code without breaking existing clients. You could actually modify the structure of your classes, but provide an interface that looks like the old version of the component. This kept existing client applications happy while at the same time you were able to modify the classes to enhance functionality.

In VB.NET, you can still use interfaces this way. However, interfaces are best used when you want to be able to use different object types in the same way, and the objects don't necessarily share the same base type. For example, the `IEnumerable` interface is used to expose an enumerator for a class. Using this interface, you can move through a class using `For Each`, even if the class is not based on a `Collection` object.

You'll see an example of polymorphism with interfaces using the same example of the `Car` and the `Bicycle`. First, `Transportation` will be created as an interface instead of a base class, which might make more sense: The `Move` method is most likely to be quite different between a car and bicycle. Then, you'll implement the `Transportation` interface. Finally, you'll call the objects from the client.

```

Interface Transportation
    Function Move() As Boolean
End Interface

Public Class Bicycle
    Implements Transportation
    Function Move() As Boolean Implements Transportation.Move
        'code here
        Move = True
    End Function
End Class

```

```
Public Class Car
    Implements Transportation
    Function Move() As Boolean Implements Transportation.Move
        ' different code here
        Move = True
    End Function
End Class
```

You'll notice that now, when you implement from an interface, you don't have to use the `Overrides` modifier. In addition, the method definition is followed by an `Implements` keyword that specifies which method in the interface the current method is implementing. This allows you to have different names for the methods in the class that is implementing the interface. The client code here will be the same as it is when you use inheritance polymorphism.

When to Use and When Not to Use Inheritance

When inheritance was first introduced into some other languages, it caused problems as users went too far, and had very deep inheritance trees. Inheritance is powerful and useful, but should not be overdone. I remember working with a developer. He spent an entire day working on the design of objects and finally said, "I can't figure out where to use polymorphism." My suggestion was that if he couldn't find where to use it, maybe he didn't need to.

There are actually three ways to build an object model:

- **Encapsulation**—If you want to use functionality from an object, but not really look like the object or override its functionality, don't inherit; instead encapsulate an instance of the object. Example: You have a calculator application, and you want to show the number display in red when it is negative. You could define a new type of control from text box or label that shows the number in red when negative, or you could just hook onto the text changed event of the text box, and perform the functionality there. The former mechanism would be overkill.
- **Interface implementation**—If the derived objects don't share very much in terms of implementation, use an interface instead of a common base class. There is not much point in having a base class for which all the methods must be overridden; an interface would probably be a better way to go. Classes can also implement multiple interfaces, so a car may implement both the `GasPowered` and the `Drivable` interfaces, whereas an airplane might implement the `GasPowered` and the `Fly` interfaces. A bird, on the other hand, implements `EatWorms` and `Fly`, but does not implement `GasPowered` or `Drivable`.

- **Inheritance**—Use inheritance if you want to utilize functionality in the base class and extend it while trying to look like the base class. If some of the methods in the base class are not appropriate for your derived class, that is probably a good indication that inheritance was not the right move. In some cases, merely encapsulating another class instance and proxying calls to it might be a cleaner solution.

Summary

As you can see, setting up inheritance is not difficult. However, the challenge comes when you design your components and your object model. Inheritance is quite powerful, but you must intelligently define your base classes to make them broad enough to be inheritable, but general enough to be useful to many derived classes.

With the introduction of implementation inheritance, VB.NET moves into the realm of object-oriented (OO) languages for the first time. VB6 had some OO constructs, but the addition of implementation inheritance, one of the most basic OO concepts, greatly strengthens VB.NET's OO portfolio.



CHAPTER 6

Database Access with VB.NET and ADO.NET

The vast majority of applications today have to perform some sort of database access. Whether you are building Windows applications and want to perform simple database access, or you are creating middle-tier components to handle database access, understanding how to connect to a database, retrieve information, and manipulate data are critical to understanding how to move to VB.NET.

First, VB.NET introduces new tools for connecting to databases. Some of these tools are controls that act as wrappers around the database access objects, called ADO.NET. These controls simplify the code you write, and allow you to bind graphical controls to DataSets you generate from the database. These controls require a new way of working with data-bound controls in your application.

Second, ADO has evolved into ADO.NET. The ADO.NET functionality is different from what you are used to with ADO. Although similar in concept, the ADO.NET objects follow a disconnected paradigm. One of the major differences is that you are not necessarily retrieving the equivalent of a recordset, but an entire schema structure. These objects will be examined in this chapter.

Accessing a Database from a Windows Application

VB.NET includes a number of new controls for accessing data from your application. You can see this in action by creating a new Windows Application project in VB.NET and naming it LearningVBdata. Click on the Toolbox and notice that there is a Data tab at the top. Click on the Data tab, and you will see a series of controls as shown in Figure 6.1. In the past, controls dragged from the Toolbox and dropped on the form were shown on the form. In VB.NET, however, controls that do not have a visual interface appear in the component tray, a window below the form. This is where the majority of your data controls will appear when added to a form.

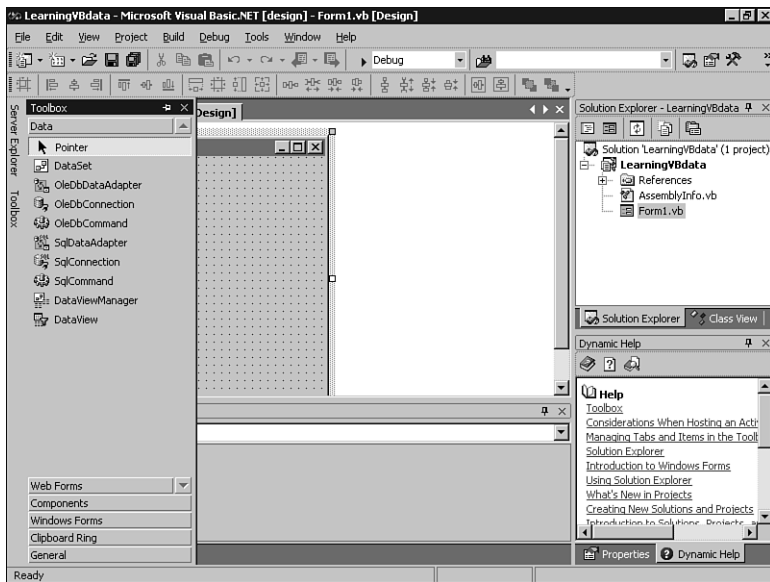


Figure 6.1

The new data controls.

There is a second option for data access that helps separate things and keeps them a little cleaner. You can use *components*, or forms without a visual part; these are classes with a designer. All your forms can share these components. Thus, you can have a single component that's responsible for all the data access in your project and your forms can connect to it. This is the model you will follow in the next few pages.

Select Add New Item from the Project menu and double-click the Component Class icon. This will add a new component, named Component1, to your project.

Using the DataAdapter Configuration Wizard

To create a data-driven form easily, VB.NET includes a DataAdapter Configuration Wizard. To start the wizard, open the Toolbox, choose the Data tab, and drag an `OleDbDataAdapter` control to the component. When you drop the control, the DataAdapter Configuration Wizard will launch automatically. The first screen of the wizard is simply information, so after reading it, click the Next button to advance into the wizard.

NOTE

The examples in this chapter will use the Northwind database in SQL Server 2000. If you do not have access to SQL Server, choose an Access MDB file to try these samples. In addition, because SQL Server is being used, you could use the `SqlDataAdapter` control, which is optimized for accessing SQL Server 7.0 and SQL Server 2000. However, the `OleDbDataAdapter` control is able to access any OLE DB data source, so it is more generic and will work fine for those of you using Access.

The next screen in the wizard asks you to choose the Data Connection to use with this DataAdapter. Assuming that you are truly new to VB.NET, you probably don't have any data connections created yet. To correct this, click the New Connection button. This launches the standard Data Link Properties dialog box you most likely are used to seeing in VB6 (see Figure 6.2).

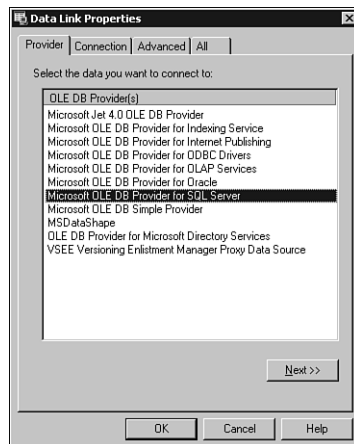


Figure 6.2

The first page of the Data Link Properties dialog box.

On the Data Link Properties dialog box, click the Connection tab and in the Server Name box, enter the name of the server on which your SQL Server resides. Next,

enter the login information, and finally choose the name of the database, which in this case is Northwind. Your final Connection tab should look similar to Figure 6.3. After you're done, feel free to test the connection, and then click the OK button.

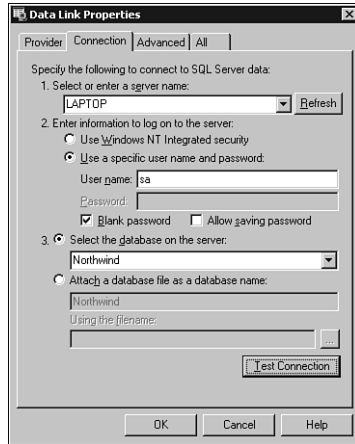


Figure 6.3

The second page of the Data Link Properties.

The DataAdapter Configuration Wizard should now show the connection you just set up, as shown in Figure 6.4. Click the Next button.



Figure 6.4

The DataAdapter Configuration Wizard showing you the new connection.

The wizard now asks you what type of query you want to perform. Your choices are to use a SQL statement, create a new stored procedure, or use an existing stored

procedure, as shown in Figure 6.5. Although stored procedures are the way to go 99% of the time, for this demonstration just choose Use SQL Statements and click the Next button.

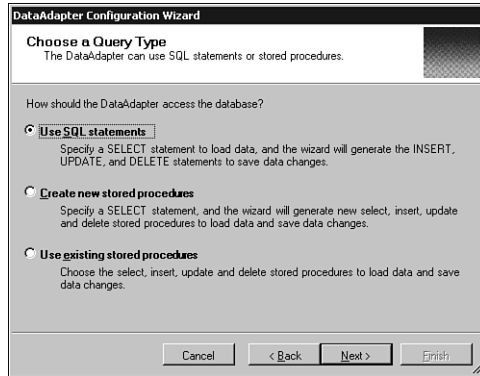


Figure 6.5

Choosing to use a SQL statement for this command.

Next, you are asked to enter the SQL statement you want to use. There are two buttons on this page of the wizard. The Advanced Options handles issues dealing with inserts, updates, and deletes, so don't worry about those for this example. The second button launches the SQL Builder. The SQL Builder is a very nice, graphical tool for creating queries. If you have used the SQL Builder in Visual InterDev or the Microsoft Query Tool, this tool will be familiar to you. If you have used Access, this tool is very similar to the Query Designer in Access. For now, just type in the SQL statement in the text box. Your query should be `Select * from Products` as shown in Figure 6.6.

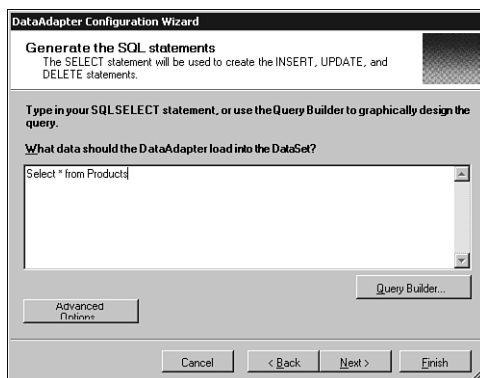


Figure 6.6

Your query entered into the wizard.

Click Next and you will be on the last screen, which again is just for information. Click the Finish button and you will exit the wizard.

You will now be returned to the Component1.vb [Design] tab. If you look at the design surface, two controls have been added: `OleDbDataAdapter1` and `OleDbConnection1`. These controls represent the connection and the query you have added to this form. If you think about it, the `DataAdapter` is similar to a `Command` object in ADO.

The `OleDbConnection1` control describes the database connection; if you click on it once and look at the Properties window, you will see the connection string, database, server, provider, and other information about the connection. The `OleDbDataAdapter1` control describes your actual query. If you don't see that in the Properties window, expand the `SelectCommand` property, and the `CommandText` (your SQL statement) is revealed.

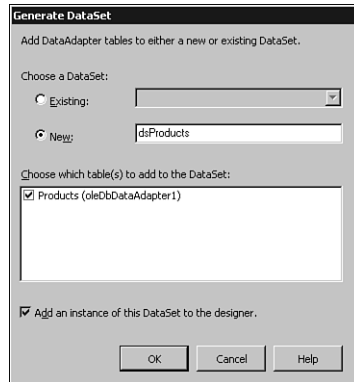
If you are keeping score, what you have done so far is basically equivalent to creating an ADO Connection object, and then an ADO Command object. What has to happen next is to actually get the records into memory, which is the equivalent of creating an ADO Recordset. In VB.NET, the object you use to store data is called a `DataSet`.

Click once on the form. Notice that on the menu bar, a new menu item has appeared: Data. If you don't click on the form first, you won't see Data on the menu bar. If you click on the Data menu, you'll see two choices: Generate `DataSet` and Preview Data. You can see these same choices, and an additional one, by selecting the `OleDbDataAdapter1` control and looking in the Properties window.

The Generate `DataSet` option creates a `DataSet` object, which is the object that holds the data in memory, just like the ADO Recordset.

Click on the Data menu and choose Generate `DataSet`. The Generate `DataSet` dialog box opens, and asks you for the name of the `DataSet`. Name the `DataSet` `dsProducts`. Leave the option to add an instance of the class to the designer unchecked. Your dialog box should look like the one shown in Figure 6.7. Click the OK button.

If you look in the Solution Explorer window, you will notice that a new file has been added: `dsProducts.xsd`. The XSD file is an XML Schema Definition file. This is your first indication that the data access in .NET relies heavily on XML.

**Figure 6.7**

Creating the DataSet.

Next, you'll need to add code to load the DataSet. Open the code window of the component by selecting Code from the View menu. Inside Form1, add the following methods:

```
Public Function FillDataSet() as LearningVBdata.dsProducts
    Dim ds as new LearningVBdata.dsProducts

    Me.oleDbConnection1.Open()

    Try
        Me.oleDbDataAdapter1.Fill(ds)
    Catch fillException As System.Exception
        msgBox("Caught exception: " & fillException.Message)
    End Try

    Me.oleDbConnection1.Close()
    Return ds
End Sub

Public Sub UpdateDataSource(ByVal ds As LearningVBdata.dsProducts)
    oleDbConnection1.Open()

    Dim UpdatedRows As System.Data.DataSet
    Dim InsertedRows As System.Data.DataSet
    Dim DeletedRows As System.Data.DataSet

    UpdatedRows = ds.GetChanges(System.Data.DataRowState.Modified)
    InsertedRows = ds.GetChanges(System.Data.DataRowState.Added)
    DeletedRows = ds.GetChanges(System.Data.DataRowState.Deleted)
    Try
        oleDbDataAdapter1.Update(UpdatedRows)
        oleDbDataAdapter1.Update(InsertedRows)
```

```
        OleDbDataAdapter1.Update(DeletedRows)
    Catch updateException As System.Exception
        MsgBox("Caught exception: " & updateException.Message)
    End Try

    Me.OleDbConnection1.Close()
End Sub
```

These methods might appear complex, but they're actually quite easy to understand. The first of these two methods is used to get data from the datasource described by the `DataAdapter` and `DataConnection`. When called, it returns a `DataSet` that has already been filled. The second of the methods takes a parameter of a `DataSet`. It is used to update the datasource. Any rows that the `DataSet` has added, deleted, or changed since the last time it was filled will be sent to the datasource. Because ADO.NET is built to work in a disconnected fashion, you will make changes to a single disconnected `DataSet`. Then ADO.NET will figure out whether a particular record has been added, updated, or deleted, and carry out the appropriate actions against the back-end database.

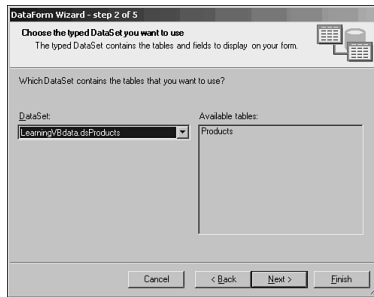
Using the DataForm Wizard

Although most VB developers scoff at the concept of bound controls, they are very useful for quickly building a prototype or proof-of-concept. In addition, bound controls work well to demonstrate some of the functionality in VB.NET.

To start the DataForm Wizard, right-click the `LearningVBData` project node, select **Add**, and then **Add New Item**. Double-click the DataForm Wizard icon. This starts the DataForm Wizard. The first screen is just information, so after reading it, click the **Next** button.

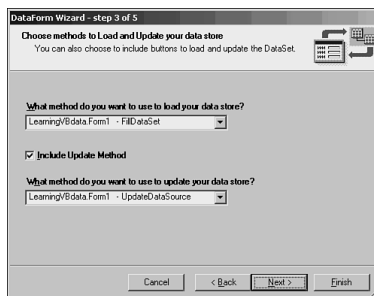
Step 2 of the DataForm Wizard asks you to choose whether to use a new `DataSet` or to use one you've already created. Because you have already created the `DataSet` you will be using, select the **Existing** radio button. If you drop down the list, you should see only one `DataSet`, which is the one you just finished creating: `dsProducts`. The `DataSet` is listed as `LearningVBdata.dsProducts`, and the list of available tables is shown in the list box on the right side. Figure 6.8 shows what your form should look like. Click the **Next** button after choosing your `DataSet`.

Step 3 of the wizard asks you what method you want to use to load the data store. Earlier, when you added the controls and set the SQL statement, a `FillDataSet` method was created in `Component1`. Choose the `FillDataSet` method for the first box.

**Figure 6.8**

Step 2 of the DataForm Wizard.

If you want, you can check the Include Update Method box, but be aware that doing so enables you to manipulate the data as you work with it, and this might not be something you want to do. However, if this is just a test machine, feel free to check this box and then select `UpdateDataSource` as the method to use to update the data store. Figure 6.9 shows what this screen should look like when you are done. Click Next to move forward.

**Figure 6.9**

Step 3 of the DataForm Wizard.

Step 4 of the DataForm Wizard displays a list of the tables in your DataSet, as well as the columns in each table. If you had a DataSet with relationships between tables, you could even display a hierarchical relationship with this screen. Make sure that all the fields are chosen and click the Next button.

Finally, Step 5 asks whether you want to display all the records (which will show them in a grid) or just a single record (which will show the record in text boxes).

If you choose the grid display, the generated form displays the DataSet in a `Datagrid`. A `Datagrid` has the functionality to navigate, add, and remove records already built into it. Thus, choosing to view the records in a grid removes the options for adding

navigation controls, as well as buttons for Add, Delete, and Cancel. Select the Single Record radio button and leave all the buttons enabled, as shown in Figure 6.10. Click the Finish button to exit the wizard, have it create a form, and work its magic.

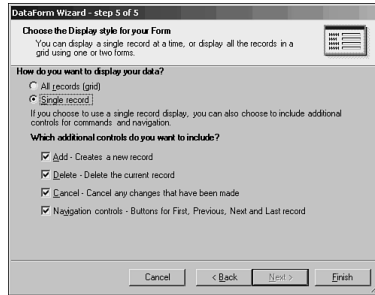


Figure 6.10

Choosing how to view the data on the form.

The form sports a large number of controls: buttons, text boxes, and panels. Before running the result, you need to set DataForm1 as the Startup object for your application. Select Properties from the Project menu. In the Startup object drop-down, select DataForm1 and press the OK button. Go ahead and run the form by clicking on the Start button.

After the form is loaded, click the Load button. This actually fills the DataSet and then displays the first record on the form. Use the navigation buttons to walk through the records. Figure 6.11 shows how your form should look.

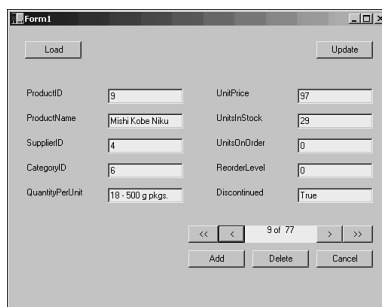


Figure 6.11

Your first data-driven VB.NET form.

If you update the data, you will get an interesting lesson in just how ADO.NET works. For example, in Figure 6.11, the current record is ProductID 9, Mishi Kobe Niku. The UnitsInStock is 29. If you change the UnitsInStock to 28, and then click the Next (>) button, you scroll to ProductID 10. Then, if you click the Previous (<)

button, ProductID 9 shows a UnitsInStock of 28. “Aha!” you think. “I’ve updated that record.” And you have. But you haven’t.

ADO.NET uses a disconnected paradigm. That means that the data in the DataSet is not connected to the underlying database in real-time. The data you have changed is in the DataSet, and not in the SQL Server table. You can verify this by opening SQL Server’s Query Analyzer and running the following SQL statement against the Northwind database:

```
Select * from Products where ProductID=9
```

If you run this SQL statement, you will see that the UnitsInStock is still 29, even though you now see 28 in the application you built.

This disconnected mode of working is why the DataForm Wizard adds an Update button. The Update button calls the UpdateDataSource procedure that you selected in the DataForm Wizard. As soon as you click the Update button, the data is updated in SQL Server. Using this disconnected model, you could make multiple inserts, updates, and deletes, and then send them to the underlying database all at once.

Some of you might have used disconnected recordsets in ADO. If so, this sounds familiar to you. For those of you who didn’t use ADO’s disconnected recordsets, this might sound strange. Just be aware that this is the default behavior in ADO.NET.

You can also see the data in the form of a grid. Select Add New Item from the Project menu. Double-click the DataForm Wizard icon to start the DataForm Wizard. Choose the dsProducts DataSet that you created earlier. In the method you want to use to load the data store, pick the FillDataSet in Component1. Select the Products table and all the fields. Finally, choose to display All Records (grid) on Step 5 of the wizard.

VB.NET adds a DataGrid control to the form, along with a Load button. Don’t start the project yet; if you do, you will see only Form1. To see DataForm2 at startup, right-click the project in the Solution Explorer window and choose Properties. In the Startup Object box, choose LearningVBdata.DataForm2. Now, start the project and click the Load button. You should see the records from the Products table in the form, as shown in Figure 6.12.

ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	Units
1	Chai	1	1	10 boxes x 20	18	39	0
2	Chang	1	1	24 - 12 oz bott	19	17	40
3	Aniseed Syrup	1	2	12 - 550 ml bo	10	13	70
4	Chef Anton's	2	2	48 - 6 oz jars	22	53	0
5	Chef Anton's	2	2	36 boxes	21.35	0	0
6	Grandma's Bo	3	2	12 - 8 oz jars	25	120	0
7	Uncle Bob's O	3	7	12 - 1 lb pkgs	30	15	0
8	Northwind's C	3	7	12 - 12 oz jars	40	6	0

Figure 6.12

Your DataForm using the grid.

Binding to Data Without the DataForm Wizard

What if you wanted to bind fields to the DataSet manually? This step involves placing the controls on the form and binding them through setting properties. In addition, you have to write the navigation code yourself. Still, this is a common request, so it is important to see how it works.

Add a new Windows form to the LearningVBdata project. Name the form `BoundControls.vb`. After the form is ready, open the Toolbox and add a `ComboBox` to the form. Name this control `cboCountry`. Change the `DropDownStyle` property to `DropDownList`. Find the `Items` property and click the ellipses (...) in the property box to open the String Collection Editor. Typing one per line, enter the following countries: USA, France, Germany, Brazil, UK, Mexico, and Spain. Figure 6.13 shows what this should look like.

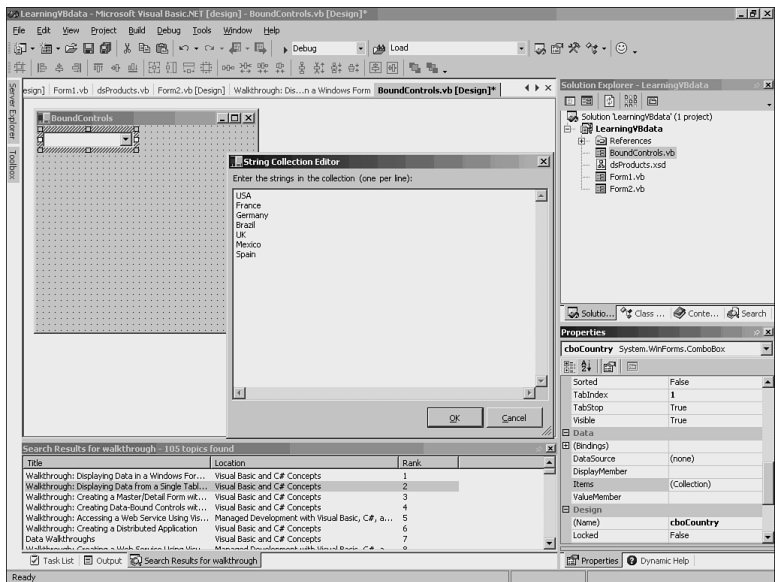


Figure 6.13

Filling in the countries for the `cboCountry` box.

Next, add three text boxes to the form, and a label for each text box. Name the three text boxes `txtCustName`, `txtContName`, and `txtPhone`; clear out the `Text` property of each. Set the `Text` properties for the labels to `Customer Name`, `Contact Name`, and `Phone`, respectively. Finally, add a label for the combo box and label it `Country`.

Now, it's time to add some data to the form. Open `Component1`, and go to the `Toolbox`. In the `Data` tab, drag over an `oleDbDataAdapter` to the component. This

starts the DataAdapter Configuration Wizard you saw earlier. Choose the data connection you created earlier, and then choose to use a SQL statement. In the screen that asks you to enter a SQL statement, enter the following command:

```
Select CompanyName, ContactName, Phone
From Customers
Where (Country=?)
```

This code creates a parameterized query. It can't run properly unless you provide a value to replace the question mark. This way, you will choose a country from the combo box, and only the records that match that country will be retrieved and used to populate the DataSet.

Finish the wizard and you will see that you have a new `OleDbDataAdapter` on the component (because you used the same connection as last time, there was no need to create a new `OleDbConnection`). You still need a `DataSet`, so click on the Data menu and choose `Generate DataSet`. Select the option to create a new `DataSet`, and name it `dsCustomers`. Make sure that only the `Customers` table is selected and click OK. As happened the last time, a file named `dsCustomers.xsd` is added to the Solution Explorer.

Next, you'll have to write new methods to fill and update the new `DataSet`. Add the following methods to `Component1`:

```
Public Function FillCustomers(country As String) _
    as LearningVBdata.dsCustomers
    Dim ds as new LearningVBdata.dsCustomers

    Me.OleDbConnection1.Open()

    Try
        Me.OleDbDataAdapter2.SelectCommand.Parameters.Item(0).Value _
            = country
        Me.OleDbDataAdapter2.Fill(ds)
    Catch fillException As System.Exception
        MsgBox("Caught exception: " & fillException.Message)
    End Try

    Me.OleDbConnection1.Close()
    Return ds
End Sub

Public Sub UpdateCustomers(ByVal ds As LearningVBdata.dsCustomers)
    OleDbConnection1.Open()

    Dim UpdatedRows As System.Data.DataSet
    Dim InsertedRows As System.Data.DataSet
    Dim DeletedRows As System.Data.DataSet
```

```
UpdatedRows = ds.GetChanges(System.Data.DataRowState.Modified)
InsertedRows = ds.GetChanges(System.Data.DataRowState.Added)
DeletedRows = ds.GetChanges(System.Data.DataRowState.Deleted)
Try
    OleDbDataAdapter2.Update(UpdatedRows)
    OleDbDataAdapter2.Update(InsertedRows)
    OleDbDataAdapter2.Update(DeletedRows)
Catch updateException As System.Exception
    MsgBox("Caught exception: " & updateException.Message)
End Try

Me.OleDbConnection1.Close()
End Sub
```

Now go back to the `BoundControls` form. This time, you have to add a `DataSet` to the form by hand. You didn't have to do this the last time because the `DataForm Wizard` handled that for you. Drag a `DataSet` object from the `Toolbox` onto your form. Select `LearningVBdata.dsCustomers` from the typed `DataSet` drop-down and click `OK`. You should see a `dsCustomers1` added to the frame below the form.

Now that you have created a `DataSet`, you need to tie the controls to the `DataSet`. The first thing you need to do is wait for the user to select a country. After he does, you need to pass that country into the parameter you created in the `SQL` string inside the `DataAdapter`. You then need to execute the query and fill the text boxes with the appropriate values.

Double-click the `cboCountry` combo box to open the code for the `SelectedIndexChanged` event. Enter the code shown so that your procedure looks like this:

```
Protected Sub cboCountry_SelectedIndexChanged _
    (ByVal sender As Object, ByVal e As System.EventArgs)
    dsCustomers1.Clear()
    dim con as new Component1
    con.Me.FillCustomersDataSet(dsCustomers1, cboCountry.Text)
End Sub
```

The `Clear` method clears out the `DataSet`. This is important, because the `DataSet` is disconnected. If you fail to clear it out before calling `FillDataSet`, any returned data is appended to the data already in the `DataSet`! The second call, to `FillDataSet`, passes in the value for the country parameter.

This code brings back the data, but you now need to show this data in the text boxes. Return to the form and click once on the `txtCustName` box. In the `Properties` window, scroll down until you find the `DataBindings` node. You have to click the plus next to it to expand the node to see all the properties. In the `Text` property, you get a drop-down window, and in it you have the ability to choose a field from the `Customers`

table. Expand the nodes until you locate the `dsCustomers1 - Customers.CompanyName` field.

For the `txtContName` box, bind it to `dsCustomers1 - Customers.ContactName`. The `txtPhone` box should be bound to `dsCustomers1 - Customers.Phone`.

You can now test the project. Before doing so, however, you need to make the `BoundControls` form the default. Right-click on the project in the Solution Explorer window and choose Properties. In the Startup Object box, choose `LearningVBdata.BoundControls` and click OK. Now, run the project. When it is running, drop down the list and choose USA. It might take a moment, but your form should look something like the one shown in Figure 6.14.

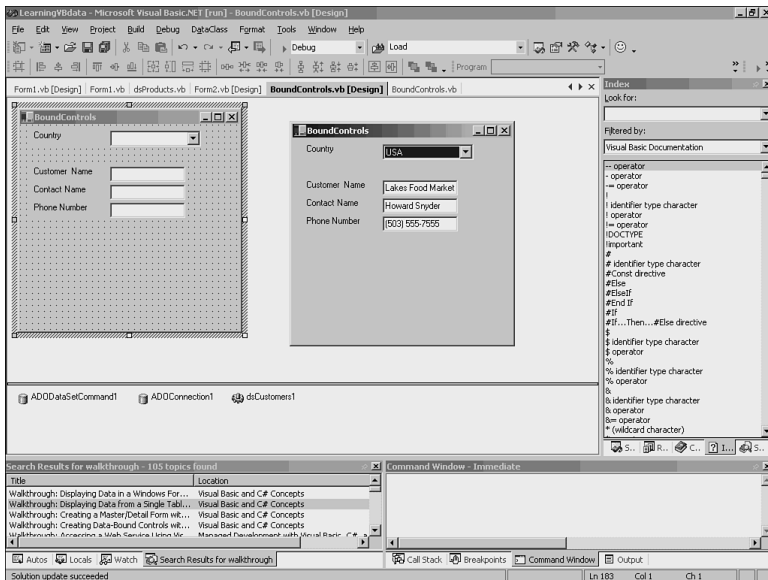


Figure 6.14

The data displayed in the unbound controls.

This works well if you only want to see the first record for each country. If, however, you want to scroll through the records for each country, you can add your own navigation buttons.

Before adding your navigation buttons, there is an object about which you should be aware: the `BindingManager` object. This object is provided to you by the Windows form, and it contains information about the current data source. One of the main reasons you will use it is for position information within a `DataSet`.

Add four buttons to the form, and make them small. Going from left to right, change their text properties to the following symbols: <<, <, >, and >>, respectively. These, of course, represent first, previous, next, and last. Name the buttons cmdFirst, cmdPrevious, cmdNext, and cmdLast. Add a label to the form, and name it lblRecCount. Position the controls as you prefer.

Now it is time to add code to your project. You have to add code to each of the four buttons. Then, you have to create a routine to update the current record position in the label. You also have to call this current record procedure from each button and the cboCountry you created earlier.

When you finish, the code you entered should look like this:

```
Sub PositionIndicator()  
    Dim recNumber As Integer  
    Dim recCount As Integer  
    recNumber = Me.BindingManager(dsCustomers1, "Customers").Position + 1  
    recCount = Me.BindingManager(dsCustomers1, "Customers").Count  
    lblRecCount.Text = recNumber & " of " & recCount  
End Sub
```

```
Protected Sub cmdLast_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs)  
    Me.BindingManager(dsCustomers1, "Customers").Position = _  
        Me.BindingManager(dsCustomers1, "Customers").Count - 1  
    PositionIndicator()  
End Sub
```

```
Protected Sub cmdNext_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs)  
    If Me.BindingManager(dsCustomers1, "Customers").Position < _  
        Me.BindingManager(dsCustomers1, "Customers").Count - 1 Then  
        Me.BindingManager(dsCustomers1, "Customers").Position += 1  
    End If  
    PositionIndicator()  
End Sub
```

```
Protected Sub cmdPrevious_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs)  
    If Me.BindingManager(dsCustomers1, "Customers").Position > 0 Then  
        Me.BindingManager(dsCustomers1, "Customers").Position -= 1  
    End If  
    PositionIndicator()  
End Sub
```

```
Protected Sub cmdFirst_Click(ByVal sender As Object, _  
    ByVal e As System.EventArgs)  
    Me.BindingManager(dsCustomers1, "Customers").Position = 0
```

```

    PositionIndicator()
End Sub

Protected Sub cboCountry_SelectedIndexChanged(ByVal sender As
    Object, ByVal e As System.EventArgs)
    dsCustomers1.Clear()
    Me.FillDataSet(dsCustomers1, cboCountry.Text)
    PositionIndicator()
End Sub

```

Some of the techniques in this code might look strange to you. The BindingManager code is what gives you programmatic access to the DataSets being used by your form. You must specify which table in the DataSet to use, because a DataSet can have several tables. The Position property runs from 0 (the first record) to Count - 1 (the last record).

Run this form. It should work, and you should not be able to move past the end of the records at either end of the DataSet. Your form should look similar to the one in Figure 6.15.

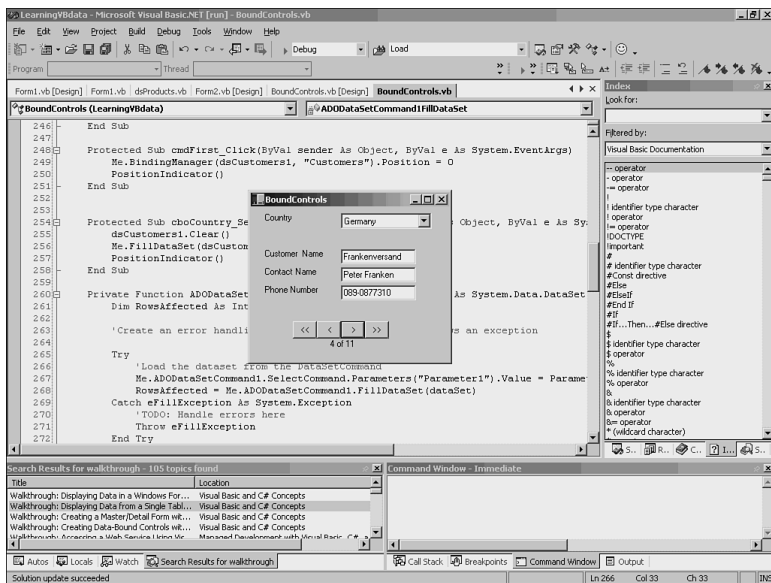


Figure 6.15

Navigating the form with unbound controls.

ADO.NET

So far, all your programming has revolved around displaying data on a form. This has been accomplished by binding controls to a DataSet. Other than the navigation code you wrote in the previous example, you haven't had to code against the ADO.NET objects. Obviously, you need to work against these objects when you start using ADO.NET in components, where you no longer have a user interface. Therefore, it is important to understand something about the ADO.NET controls and how they work.

About ADO.NET

As I've mentioned many times before, ADO.NET uses a disconnected architecture to operate. The reason for this is that traditional applications that maintained an open database connection did not scale well. If a component opened a connection to the database and held it open for the lifetime of the application, it consumed expensive database resources, when the connection probably was needed for only a small percentage of that time. As the number of users grows, the overhead of the database connections can begin to affect the database performance negatively.

Therefore, Microsoft decided to use a disconnected architecture for ADO.NET. This not only solves the problems with scalability by reducing the number of active connections, it makes it much easier to transfer data from one component to another. You do not need to have both components connected to the database, nor do you have to have them both understand some binary format of the data, as you will see in a moment.

Additionally, Microsoft recognized that in much programming today, you basically have a disconnected application architecture, thanks to the Web. Think of a typical Web application: The user requests a page and the page is generated on the server, with a mix of HTML and data from the database. By the time the page is sent back to the user and rendered in his browser, the connection to the database is no longer needed. Therefore, if it takes one second to render a page but the user views it for twenty-nine seconds, a database connection is needed for only one-thirtieth of the total time the page is used.

If you still need to use a connected architecture, Microsoft recommends you use ADO. ADO.NET is inherently disconnected, so ADO is still a better approach if you need a continuous connection to the underlying database.

DataSets

ADO.NET uses the DataSet to store disconnected data. This new structure is similar to an ADO recordset, but it has some important differences. For example, an ADO

recordset looks like a single table, even if the data is from multiple tables. The final recordset is merely rows and columns. A DataSet, however, can store multiple tables in its cache. You can define the relationships between these tables, and then read from individual tables or join tables and retrieve data. A DataSet, in a sense, works like a miniature copy of the database, although it usually contains only a small subset of the data or the tables.

A DataSet knows nothing about the underlying database. In fact, multiple underlying databases could make up the data in a DataSet. There is no reason why some of the data couldn't be from SQL Server while other data comes from Oracle. Another object, the DataSet adapter, holds the information regarding the connection to the underlying database. The DataSet adapter has methods that allow you to retrieve and update data, but these methods actually rely on you to provide the proper SQL statements or stored procedure names.

Datasets know all about XML. If you need to pass a DataSet's data from one component to another, the data is passed using XML. If you want to write a DataSet to disk to be able to retrieve it later, it is stored in XML format. ADO.NET uses XML because it is a standard format, and any consumer that understands XML can use that XML stream, transforming it as desired into whatever format is necessary. Contrast this to an ADO recordset, and you can see the advantages. An ADO recordset uses a binary format that has no meaning on another platform. XML, however, is a text-based standard that can be consumed by any platform. Further, that fact that XML is text means that it is quite easy to transfer it over HTTP.

NOTE

Actually, ADO has allowed you to persist a recordset as XML for a while. However, this is not the default format; instead, you must call the save method and pass a parameter to specify the XML format. XML is the default in ADO.NET.

Another advantage to using XML as the transport is that unlike ADO's disconnected recordsets, passing a DataSet to another component is as simple as transferring text. With a disconnected recordset, you had a binary file that had to be marshaled across process boundaries, which was an expensive proposition.

Working with the ADO.NET Objects

There are a number of new objects in ADO.NET that are different from what ADO developers have been using. Some of the names sound similar to the names of ADO objects, but be careful that you don't assume certain capabilities that might not exist.

You've already seen a discussion of the DataSet, but this is like starting an ADO discussion with the recordset: It's where you do most of the work, but it's the endgame as far as the coding goes. Therefore, with ADO.NET, you need to understand how to

make the database connection, and how to execute the statements that end up filling in the DataSet.

Connections: OleDbConnection and SqlConnection

The OleDbConnection object uses OLE DB to connect to a data source. ADO.NET also includes a new type of connection: SqlConnection. This works like the OleDbConnection object, but it does not use OLE DB. Instead, SqlConnection uses a native driver to connect to SQL Server and offers better performance than the OLE DB provider. Because some people reading this book might not have access to SQL Server, I will continue to use OleDbConnection.

Connecting to a data source is straightforward: Define the connection string, create the connection object, and open the connection.

In the LearningVBdata project, add a new Windows form and name it DataForm3.vb. For now, just add a button to the form and double-click the button to get to the code window. Go to the top of the code window and enter the following line of code:

```
Imports System.Data.OleDb
```

Now, in the procedure for the button's click event, add the following code:

```
Dim connString As String = _  
    "Provider= SQLOLEDB.1;Data Source=localhost;" & _  
    "uid=sa;pwd=;Initial Catalog=northwind;"  
Dim myConn As New OleDbConnection()  
myConn.ConnectionString = connString  
myConn.Open()
```

Don't forget to change the connection string to match your environment as necessary. Also, you could have written the second and third lines this way:

```
Dim myConn As OleDbConnection = New OleDbConnection(sConnString)
```

OleDbCommand and SqlCommand

When you call the Open method, you open a connection to the database. Now, you need to create a command to run against the database. You do this by creating an OleDbCommand object (or SqlCommand object if you are using a SqlConnection). Add the following code to the procedure right after the myConn.Open command:

```
Dim sqlStatement As String = "SELECT * FROM Products"  
Dim myComm As OleDbCommand= New OleDbCommand (sqlStatement, myConn)
```

For the second line, you could have used this alternative syntax:

```
Dim myComm As New OleDbCommand ()  
myComm.CommandText = sqlStatement  
myComm.ActiveConnection = myConn
```

You are now seeing the power of constructors: What could have taken three lines can now be done on one line.

You have now established a connection to the database, and you have created a command to retrieve the records. As in ADO, however, you can't store those records in a connection or a command. Instead, you must have an object that can hold those records.

The DataReader

ADO.NET actually provides a couple of ways to access the records that result from a query. One is the DataSet you have already examined in some detail and will see again in a moment. This allows you to store the records in memory in a disconnected fashion.

What if the amount of data you are retrieving is quite large, and you don't want to consume all that memory? ADO.NET provides another object, called the DataReader. The DataReader is just that: an object that reads the data. It reads the data one record at a time, in a forward-only, read-only stream. This allows you to examine one record at a time, and then move to the next. Only one record at a time is actually in memory, so this significantly cuts down on the memory required.

The following code sets up a DataReader and then enters a loop that lets you see each record. The code actually exits the loop after the first time through, so you don't have to look at 77 message boxes.

```
Dim myReader As OleDbDataReader = Nothing
myReader = myComm.ExecuteReader()
While myReader.Read
    MsgBox(myReader.GetString(1))
Exit While
End While
```

The GetString method returns a string value for the indexed field. Because you start with 0, the field with the index of 1 is the ProductName field. No conversion takes place, so you have to use the correct Get method, as there are such methods as GetString, GetBoolean, and GetDateTime. There is also a GetDataTypeName to let you discover the data type of a particular field.

If you choose to run the example at this point, don't forget to make DataForm your startup object for the project.

This works fine if you want to move through a result set in a sequential manner, and it uses very few resources. However, if you want to be able to scroll through the data or make changes to it, you must store the data in a DataSet.

The DataSet

The DataSet is the object you have been dealing with for some time. The DataSet actually contains one or more DataTable objects. A DataSet command can pull data from the database and store it in a DataTable inside a DataSet. Therefore, with ADO, a recordset is just what its name implies: a set of records. In ADO.NET, the DataSet acts as a collection for one or more data tables, and each data table may contain data from multiple tables in the underlying database.

To create and fill a DataSet, you first must create a DataSetAdapter object and then set a reference back to the OleDbCommand or SqlCommand that will provide the records. You can do this by adding the following code:

```
Dim dsCommand As New OleDbDataAdapter ()  
dsCommand.SelectCommand = myComm
```

Although it isn't shown here, you will soon see that the DataSetAdapter object has a Fill method that actually executes the statement in the command object.

Next, you must create a DataSet object. You then call the Fill method on the DataSetAdapter and pass in this new DataSet as one of the parameters. The second parameter is the name of the DataTable inside the DataSet that will be created by this DataSetAdapter.

Note that to get the following code to run, you must comment out the lines that deal with the DataReader. Enter the following code to have the DataSetCommand create a data table inside the DataSet:

```
Dim dsNWind As New DataSet()  
dsCommand.Fill (dsNWind, "Products")
```

When you execute this code, a table is created inside the DataSet. This table is now named Products, but it could have been named anything. When you call Fill, if the table name you specify does not already exist, it is created automatically, and the schema matches that of the result you get back from your query. It is possible to define the table schema ahead of time, in which case the Fill method simply inserts the data into the table.

Notice, too, that there is no property in the DataSetAdapter tying it to a particular DataSet, nor is there a property in the DataSet tying it to a DataSetAdapter. This should be familiar to ADO programmers, where all the objects are independent of each other.

If you have multiple tables in a DataSet, you can even define the relationships between them. Think of your DataSet as a miniature database in memory. You can insert, update, and delete records in the tables in the DataSet. These changes can optionally be made back in the underlying database.

To see the values in a table in a DataSet, you have to go through quite a number of collections and properties. For example, you now have a Products table in the DataSet. If you want to retrieve the first record (record zero) and the second field (field 1, the ProductName) your code could look like this:

```
msgbox(dsNWind.Tables.Item("Products").Rows(0).Item("ProductName").ToString)
```

Here, you specify Tables.Item("Products"). You could also have used Tables.Item(0). Then, you specify the first row and the ProductName field. You could have also said Rows(0).Item(1). You then apply a ToString method to convert the value explicitly into a string. A simpler version of this syntax is

```
msgbox(dsNWind.Tables.Products.Rows(0).ProductName.ToString)
```

An alternative to this rather cumbersome line is to create a DataRow object. You can loop through the rows in a table and retrieve the values, as shown in this example:

```
Dim drProduct As DataRow
For Each drProduct In dsNWind.Tables.Item("Products").Rows
    MsgBox(drProduct.Item("ProductName").ToString)
Exit For
Next
```

Again, there is an Exit statement so that you don't have to run through all 77 products. As you can see in the code, you simply use a DataRow object to point to the individual rows, and then you can access the fields as needed.

Before moving on, here is the complete code (remember that the DataReader code had to be commented out so the rest would work):

```
Protected Sub Button1_Click _
    (ByVal sender As Object, ByVal e As System.EventArgs)
    Dim sConnString As String = "Provider= SQLOLEDB.1;" & _
        Data Source=localhost;uid=sa;pwd=;Initial Catalog=northwind;"
    Dim myConn As New OleDbConnection()
    myConn.ConnectionString = sConnString
    myConn.Open()

    Dim sqlStatement As String = "SELECT * FROM Products"
    Dim myComm As OleDbCommand = New OleDbCommand(sqlStatement, myConn)
    Dim myReader As OleDbDataReader = Nothing
    Dim dsCommand As New OleDbDataAdapter()
    dsCommand.SelectCommand = myComm
    Dim dsNWind As New DataSet()
    dsCommand.Fill(dsNWind, "Products")

    MsgBox(dsNWind.Tables.Products.Rows(0).ProductName.ToString)

    Dim drProduct As DataRow
    For Each drProduct In dsNWind.Tables.Item("Products").Rows
```

```
        MsgBox(drProduct.Item("ProductName").ToString)
    Exit For
Next
End Sub
```

XML Integration

As you are aware, XML is currently “the big thing.” Everyone is talking about XML, even if they aren’t quite doing it yet. The idea behind XML is to provide a way to transfer structured or relational information in a text-only format. Beyond that, though, the data is self-describing and easy to manipulate into a different format. XML is seen as the ideal way for businesses to pass data back and forth.

Microsoft has made most of .NET XML based, and that is clearly evident with the inclusion of two classes: `XmlReader` and `XmlWriter`. Microsoft has created two objects to implement these two classes, called `XmlTextReader` and `XmlTextWriter`. These classes can be found in the `System.Xml` namespace. Working together, .NET and ADO.NET can read or write data in the same manner, whether it is XML or relational data.

Microsoft has created an `XmlDataDocument` object to tie relational data from a `DataSet` with the XML Document Object Model (DOM). If you need to work with XML, the `XmlDataDocument` can load either relational or XML data and manipulate it. If the `XmlDataDocument` is loading relational data, it uses a `DataSet` as its source. After the relational data is loaded by the `XmlDataDocument`, it is accessed using the .NET XML classes and functions.

One of the things that .NET will do for you is validate your XML against an XML schema. If you aren’t familiar with an XML schema, here’s your one-paragraph crash course: Just as your database has a schema that defines the tables, columns, constraints, and so forth, an XML schema defines the structure of an XML document to make sure that the document is properly formed. This way, you can share the schema with others and when they get an XML document from you, they can compare it to the schema. Pretty soon, if everyone starts using the same schemas, you have universal DataFormats. It isn’t quite the universal language translator from Star Trek, but it’s a step in the right direction for businesses that need to share data across disparate systems.

NOTE

There are several types of XML schemas, including DTDs (Document Type Definitions). However, XSD is the standard as of March 16, 2001, and it is what Visual Studio.NET produces.

You have already created an XML schema; in this case, an XSD file. In fact, if you look in your Solution Explorer window, you will see two XSD files: `dsCustomers.xsd` and `dsProducts.xsd`. If you double-click the `dsCustomers.xsd` file, you will see a new tab added to the main work area. The first view of the XSD is the schema view, which shows the DataFormat looking very much like a table in a database. Along the bottom of this window are two buttons: Schema and XML. If you click the XML button, you will actually see the XML schema that has been generated for you by Visual Studio.NET. Here is part of the schema:

```
<xsd:schema id="dsCustomers" targetNamespace=
  "http://www.tempuri.org/dsCustomers.xsd" _
  xmlns="http://www.tempuri.org/dsCustomers.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" _
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  elementFormDefault="qualified">
<xsd:element name="Customers">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="CompanyName" type="xsd:string"/>
      <xsd:element name="ContactName" minOccurs="0" type="xsd:string"/>
      <xsd:element name="Phone" minOccurs="0" type="xsd:string"/>
      <xsd:element name="CustomerID" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:key name="Constraint1" msdata:PrimaryKey="true">
    <xsd:selector xpath="."//Customers" />
    <xsd:field xpath="CustomerID" />
  </xsd:key>
...

```

If you examine this code, you will see the name of the table (`<xsd: element name="Customers">`) and the fields and their data types (`<xsd: element name="CompanyName" type="xsd:string" />` and so on). At the end of this snippet, the XSD defines the primary key.

The XML Designer

You can create an XSD schema from scratch, using the XML Designer. In your LearningVBdata project, click the Add New Item button. In the Add New Item dialog box, make sure that Local Project Items is selected and then choose XML Schema. Name the schema `University.xsd`. A nearly blank form will greet you.

If you click on the Toolbox, you'll notice it now consists of a tab labeled XSD Schema. Drag a `simpleType` over to the designer. This will add a graphical representation for a simple type. A *simple type* is just a type based on the base types in XML, such as strings, time, integers, and so on. However, by declaring a simple type, you

can limit certain elements, such as the size of the field. In the simpleType you just added, change the name to stState. Leave the type as string, and in the box below stState, choose length. Set a length of 2 in the box to the right. You have taken the standard string data type and limited it to exactly two characters.

Now, click and drag a complexType from the Toolbox. A *complexType* acts basically as a subtype that can then be used by other elements in the schema. Name this complex type Person. Add a Name attribute (the A you see stands for “attribute”) of type string. Add Address and City attributes, both of type string. Finally, add a State attribute, but this time use the type stState, which is the simple type you created earlier.

Next, it’s time to create a relational table. To create a relational table, drag an element from the Toolbox to the designer. Name the element Student. Now, you must define the fields of the table. Person has most of what you need, and it was created as a complex type because a Professor table would use the same Person structure.

Change the A in the first column to E for element, and then type StudentID, and make it of type int. Next, add another element named StudentInfo, but make the type Person. The designer automatically adds a reference to an element named StudentInfo. Figure 6.16 shows what the Designer should look like.

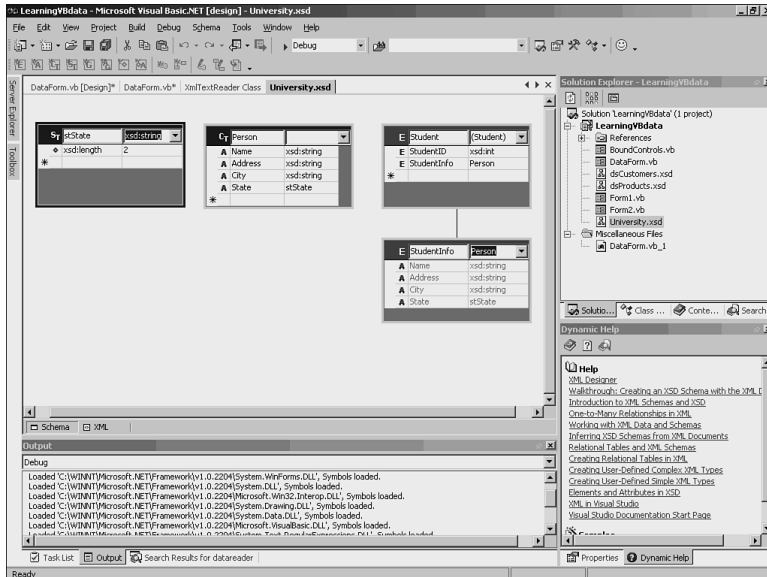


Figure 6.16

The XML Designer after you have created your schema.

Is there an easier way to create a schema? Yes, if you have an existing table in a database. Simply create a new XSD schema in your project, and drag and drop a table from the Server Explorer. This generates the XSD for you from the table definition.

What can you do with this XSD schema after you have it? You can load it just as you would a DataSet. In fact, if you set the proper path in code below and then ran it, the system will report that the DataSet name is *University*. You could then add records to the table in memory, or merge the data from another DataSet that actually gets the data from the underlying database.

```
Dim xdd As New XmlDataDocument()  
xdd.LoadDataSetMapping("c:\...\University.xsd")  
MsgBox(xdd.DataSet.DataSetName.ToString)
```

Summary

As you can see, there are a number of changes to working with data. There are new controls, but gone are the data controls that so many developers complained about in previous versions of VB. In fact, most of the controls here just generate code for you, taking a page from Visual InterDev's book. You can use the wizards to generate DataSets for you, but then programmatically access those DataSets if you don't want to use bound controls.

ADO.NET represents a major shift in focus. Instead of retrieving records and maintaining an open connection to the database, you are expected to create your own miniature database in memory, and cache the records there. This is a powerful concept because you can create custom databases for your applications without having to modify a thing in the underlying database. The disconnected paradigm makes your .NET applications much more scalable.

The XML integration is exciting, too. Expect to see much more on this as it is fleshed out further as .NET evolves.



CHAPTER 7

Building Web Applications with VB.NET and ASP.NET

Most developers today are building Web applications. For the past three and half years or so, Microsoft developers have been building Web applications using Active Server Pages, or ASP. ASP is a technology in which the pages are a mix of HTML and a scripting language, such as VBScript or JavaScript. The HTML was basically static, and was rendered as you typed it in the page. The script was interpreted on-the-fly, and generated additional HTML. This generated HTML was mixed in with the static HTML, and the page was sent to the browser.

Web applications, including Active Server Pages applications, follow a simple request/response metaphor because that is all that is allowed by HTTP. The user requests a page, and the page is sent to the browser to be rendered. The person can fill out data fields, and when he clicks a button, he is making a new request, and the response is generated on the server and returned.

ASP.NET has to use request/response, of course, because you're still using HTTP. However, ASP.NET seeks to simplify the coding model, by making it appear as an event-driven programming model. ASP.NET has the following advantages over ASP:

- Eliminates spaghetti code. Script is no longer inter-mixed with HTML. This makes the code much smaller, cleaner, and easier to maintain. This is made possible by the event-driven page processing.

- New controls have been introduced that promote user interface encapsulation. These controls give browser-independent rendering, which means that you write code only once for multiple clients.
- Page services have been introduced that reduce the grunt work involved in creating form pages that post back to themselves: ViewState and PostBack data processing.
- New application services make applications faster and more scalable. These include caching, farmable session state, and security to name a few.

Before going into the full details of how ASP.NET works, and how it differs from ASP, it might be helpful to build a quick ASP.NET application and examine the resultant code.

Your First ASP.NET Application

Start Visual Studio.NET and choose to create a new Visual Basic project using the Web Application project type (or template) and name the project WebAppTest. Notice, as shown in Figure 7.1, that the location of the project is an HTTP address, not a directory on the machine. The server to which you connect must have Internet Information Server (IIS) 4.0 or higher. Windows 2000 ships with IIS 5.0, so if your Web server is running Windows 2000, you are fine. The server must also have the .NET Framework loaded, so you might want to use your local machine as the Web server.

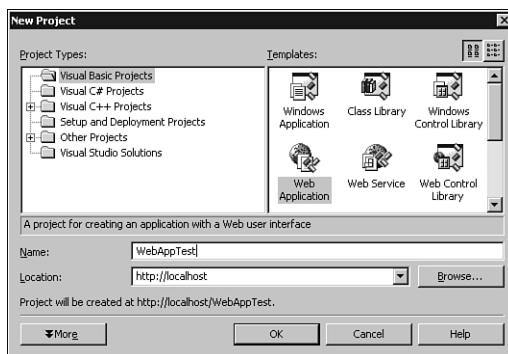


Figure 7.1

Creating a Web Application project requires a server running IIS and Visual Studio.NET (or at least the .NET Framework).

After you click the OK button, VS.NET attempts to communicate with the Web server. Provided this communication is successful, the project is created on the Web server, and you are ready to begin working with the project.

The page will open as a blank form in the designer, with a little descriptive text in the middle. If you look in the Solution Explorer, you will see that this page is named `WebForm1.aspx`. ASP uses the `.asp` extension, whereas ASP.NET files use the `.aspx` extension. ASP and ASP.NET can coexist in the same directory if necessary.

If you look at the editor, it is just a blank form right now. The default view is for the form to be in `GridLayout` mode, which means that you can drag and drop controls onto the form and easily position them by using the standard snap-to-grid feature of the designer. There is also a `FlowLayout` mode that allows you to get absolute positioning by placing the controls exactly where you want. To switch between `FlowLayout` and `GridLayout` modes, change the `pageLayout` property in the Properties window.

Go ahead and switch the `pageLayout` property to `FlowLayout`. This mode works like a word processor in some ways. If you click once on the form, you have a cursor blinking in the upper-left corner. Type `Welcome to my first ASP.NET page` and then press the Enter key. As you can see, the text is placed on the form, just as you would expect in a word processor. Highlight the text and look at the toolbar. There is a drop-down box that says `Normal`. Drop down this list and choose `Heading1`. The text enlarges significantly.

Along the bottom of this window are two buttons: `Design` and `HTML`. If you click the `HTML` button, you will be shown the HTML making up the page. Right now, the line that creates the `Heading1` is as follows:

```
<H1>Welcome to my first ASP.NET page</H1>
```

Now, go back to the `Design` view by clicking the `Design` button. Highlight the text again, and click the `Center` button to center the text. If you switch back to the `HTML` view, you will see that the earlier `Heading1` line has changed to this:

```
<H1 align=center>Welcome to my first ASP.NET page</H1>
```

At this point, you might think you have a high-powered HTML editor, not much different from `FrontPage` or a hundred other HTML editors. Now, however, it is time to see an example of some ASP.NET.

Go back to the `Design` view and move to the line below the `Heading1` line you added. Click on the `Toolbox` and notice that there is a tab for `Web Forms`. Click and drag a `Label` to your form. Next, click and drag a `button` to the form. You should now have a label and a button next to each other. Both the label and the button have a small green triangle in the upper-left corner.

Double-click on the button and you will open the code window. Notice that this code window creates a file with the same name as the `ASPX`, but the file has a `.VB` extension. As you will see, this is called a *code-behind page*. One of ASP.NET's design goals is to separate the code and the user interface.

In the code window, you will be in the `Button1_Click` event procedure. Type the following code:

```
Label1.Text = "Hello, World!"
```

Notice that you are programming this just as you would a standard Windows application. Go ahead and click the Start button. The page renders in the browser, as shown in Figure 7.2. What is interesting is not what you see in the browser, but the code behind it. Click on View, Source and you will see the HTML that is making up the page. Examine this HTML (which has some lines shown as two lines so that they fit in this book).

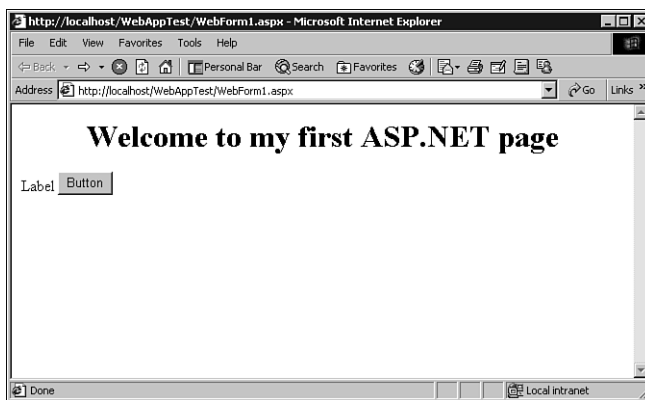


Figure 7.2

Your first ASP.NET page being rendered in the browser.

```
<HTML>
  <HEAD>
    <meta name="GENERATOR" content="Microsoft Visual Studio.NET 7.0">
    <meta name="CODE_LANGUAGE" content="Visual Basic 7.0">
    <meta name="vs_defaultClientScript" content="JScript">
    <meta name="vs_targetSchema" content="Internet Explorer 5.0">
  </HEAD>
  <body>
    <form name="WebForm1" method="post"
      action="WebForm1.aspx" id="WebForm1">
      <input type="hidden" name="__VIEWSTATE"
        value="dDwtMzQ3NzI5OTM0Qzs+" />

      <H1 align=center>Welcome to my first ASP.NET page</H1>
      <P align=left>
        <span id="Label1">Label</span>
        <input type="submit" name="Button1" value="Button" id="Button1" />
      </P>
```

```
</form>  
</body>  
</HTML>
```

For now, just notice that there is no VB code here. You created a procedure for the `Button1_Click` event and typed a line of code. However, none of that has made it to the client. This is because ASP.NET just sends HTML to the client, whereas the code is compiled and lives on the server. That means this page can be used by anyone, using any browser and operating system.

Test the page by clicking the button. You will notice the text `Hello, World!` now appears in the label. If you click View, Source again, you will notice that the `Label1` tag has changed from this:

```
<span id="Label1">Label</span>
```

to this:

```
<span id="Label1">Hello, World!</span>
```

How did this change? How was the click event handled?

How ASP.NET Works

Basically, ASP.NET works by using server-based components to generate HTML. The HTML is sent to the client and rendered in a browser. ASP.NET determines the capabilities of the client browser and generates HTML appropriate for that browser.

ASP.NET works by using server-based components to generate markup, such as HTML, and script. The HTML and script are sent to the client and rendered in a browser. ASP.NET determines the capabilities of the client browser and renders HTML appropriate for that browser. The type of markup sent to the client is determined by the controls. The markup code doesn't have to be HTML; for example, the mobile controls send WML to wireless devices.

ASP.NET user code (for example, the code that set the label text in your first project) is precompiled. This is in contrast to ASP, which interprets the script code that is intermingled with static HTML. Even if you were using compiled COM components with ASP, the calls to the components were late-bound. Using ASP.NET allows you to benefit from all the services of the .NET Framework, such as inheritance, security, and garbage collection.

ASP.NET also provides some of the functionality that has been coded by hand in the past. Like ASP, ASP.NET can provide automatic state management. Because HTTP is a stateless protocol, maintaining state in Web applications has always been a problem. ASP.NET provides state management that, unlike ASP, is scalable across Web farms, survives IIS crashes, and does not have to use cookies.

Web Pages and Code

The pages you create are divided into two parts: the user interface and the code. You can see this in the WebAppTest project because you have WebForm1.aspx and a WebForm1.vb. The VB file is a class file, called a *page class*, and it segregates your code from the HTML. When you create the page in VB.NET, you see the ASPX and the VB files as two views of the same page. When you compile the page, ASP.NET generates a new class and compiles it. This new class has the static HTML, ASP.NET server controls, and code from your form compiled in. Unlike ASP, all the HTML sent to the client is generated from the class on-the-fly. This class is actually an executable program, and whenever the page is called, the executable generates the HTML that is sent to the browser.

In the case of the page you created earlier, a compiled class was created from these two files. When someone browses the ASPX page, the class is executed and generates the HTML to send to the browser. If you look at the HTML sent to the browser, you'll notice that all the controls you added (the label and the form) are inside an HTML `<FORM>...</FORM>` block. This is because an HTML form is the only way for standard HTML to get data from an HTML page back to the server.

The code you wrote for the click event runs only on the server. When someone clicks the button, it acts as a submit button, which you can also see in the code. So, the user clicks the button, and the form is submitted to the server. In effect, you take the click event and send it to the server for processing. The generated class is instantiated, and the click event code is processed. A new HTML stream is generated and sent back to the client browser. This new HTML stream contains a new string to be placed in the label; in this case, the string is the text `Hello, World!`. ASP.NET works this way because you added server controls to the page. Server controls are discussed in the next section.

Server Controls

One of the biggest shifts in the creation of Web applications in VB.NET is the idea of the server control. When teaching students about Visual InterDev over the past few months, I've explained that although the product Visual InterDev is going away, its functionality is being added to products such as VB.NET, C#, and any other language in Visual Studio.NET. The server controls are one of the main ways in which this happens.

If you open Visual Studio.NET, click on the form, and then open the Toolbox, you'll see several interesting tabs. One tab is labeled HTML, and it shows a series of HTML tags. These are just elements that you can drag and drop to your form to create a static HTML page; each element maps directly to an HTML tag. For example,

if you drag and drop a Table from the HTML tab of the Toolbox and then look at the HTML, the following HTML is generated (shortened for brevity):

```
<TABLE cellSpacing=1 cellPadding=1 width=300 border=1>
  <TR>
    <TD>
    </TD>
    <TD>
    </TD>
    <TD>
    </TD>
  </TR>
</TABLE>
```

Even though you can work with the table in the designer just as you can any other control, the designer is really just writing HTML in the background. That's HTML in the static sense. That means that you can't start writing VB code against that table because it is just some HTML sitting in the page.

Because the elements in the HTML tab map to standard HTML tags, they aren't as powerful as the controls you added earlier. Although the button and text box you added earlier are mapped to standard HTML tags, these are ASP.NET server controls. They act as controls against which you can program just as you would in a Windows form; you double-click on a button, for example, and you wind up in the code window with a Button1_Click event procedure.

In the designer, click down to the next blank line. Drag over a button from the HTML tab of the Toolbox. Now, click on the Web Forms tab and drag over a button from the Toolbox. If you look in the designer, you'll have two buttons side by side, and they look identical except that the second one has the small green triangle in the upper-left corner, indicating that it is an ASP.NET server control.

If you look at the HTML, however, you will see a significant difference. Here are how the two buttons appear in the HTML view:

```
<INPUT type=button value=Button>
<asp:Button id=Button2 runat="server" Text="Button">
</asp:Button>
```

The first button is displayed with a standard HTML <INPUT> tag. The second button, however, is displayed with an <asp:Button> tag. This <asp:Button> tag is not standard HTML. Instead, when the .NET compiler sees this, it knows that the button is an ASP.NET server control, and you have all the functionality with this control that you would have with the same type of control in a Windows form.

In the Design view, double-click on the first button (the HTML button). When you do this, you'll see the dialog box shown in Figure 7.3. This message box informs you that this is an HTML element, which means you cannot write code for it. However,

the box offers you an option: You can convert the button to an HTML *server control*. There is a distinction between what it calls an HTML server control and an ASP.NET server control.

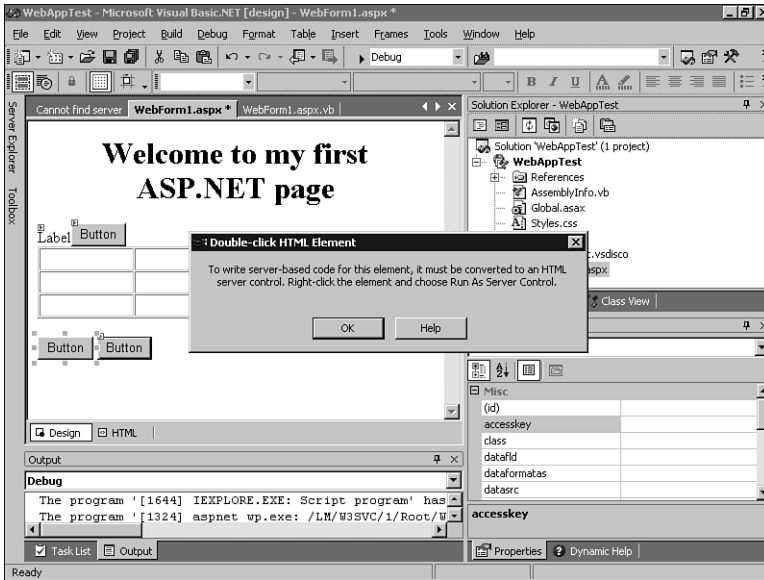


Figure 7.3

The message box explains that you cannot add code to an HTML element without first making it an HTML server control.

HTML server controls are standard HTML elements to which Microsoft has added server-side programming capabilities. When you add an HTML element, ASP.NET sees that as just text to be generated and passed to the client. HTML server controls, on the other hand, come with a programming model that exposes all the attributes of the HTML element on the server. Microsoft has made these controls quite powerful: They can maintain values between round trips to the server, they can respond to events on the server (or, optionally, on the client), and the controls can be bound to a data source.

ASP.NET server controls are more abstract controls in that they do not have a single corresponding HTML tag. For example, if you look at the Web Forms tab of the Toolbox, you'll notice controls such as the Calendar and Repeater. Although these controls end up being generated as HTML, they are often composed of many HTML tags. For example, the calendar is a table, with many rows and columns. ASP.NET server controls have all the benefits you saw with the HTML server controls, and they can also determine the capabilities of the client browser and render more or less

advanced HTML depending on the client. Some server controls have the ability to delay sending events to the server, instead caching them and sending them when the entire form is submitted.

You can take almost any element on a page and turn it into an HTML server control. For example, when you double-clicked the HTML button, the message box said that you had to convert it into an HTML server control to write code for it. The message box even said that you could right-click on the control and choose Run As Server Control. Behind the scenes, this would add `runat="server"` to the `<INPUT>` tag, and allow you to write server-side code based on that tag.

You can turn plain text into an HTML control. For example, you have a heading 1 at the top of the page that proudly proclaims, *Welcome to my first ASP.NET page*. Switch to the HTML view for a moment and you'll see that the code looks like this:

```
<H1 align=center>Welcome to my first ASP.NET page</H1>
```

Now, switch back to the Design view and highlight the text. Right-click on the highlighted text and choose Run As Server Control. Now, switch back to the HTML view and you will see that the HTML has changed to this:

```
<H1 align=center id=H11 runat="server">Welcome to my first ASP.NET page</H1>
```

Although this doesn't immediately look like a big change, two attributes have been added to the tag: `id=H11` and `runat="server"`. The first attribute makes this tag an object against which you can write some code. This is a common practice when writing Dynamic HTML (DHTML). The second attribute, however, tells the ASP.NET engine that this object's code will run on the server side. So, the server will handle any properties or methods or events on this element.

Double-click on the second HTML server control button you added. In the code window, in the `Button2_Click` event handler, type the following code:

```
H11.InnerText = "It is now " & Now
```

This code merely replaces the text inside the `<H1>` tag with the text `It is now` and then appends the current date and time. If you run the project and click the button, you will get results like those shown in Figure 7.4.

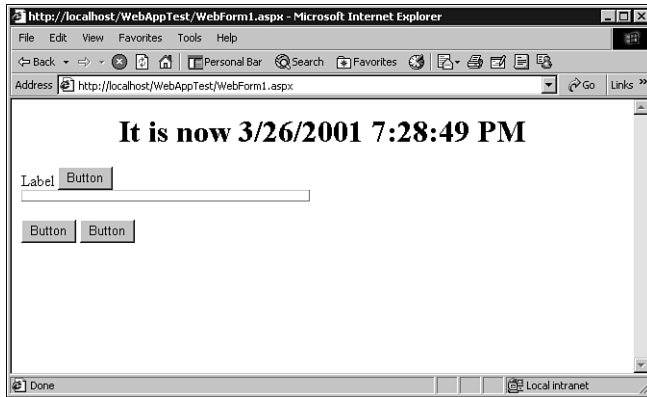


Figure 7.4

Almost any element can be turned into a server control, as this <H1> tag has been in this example.

If your curiosity is getting the better of you, feel free to drag a `Calendar` ASP.NET server control from the Web Forms tab of the Toolbox and drop it on your form. If you view the code in the HTML view of the designer, you'll see that you have added an `<asp:Calendar>` tag to the HTML. However, if you run the page and then view the source code that actually reaches the browser, you'll see that the calendar is actually just generated as an HTML table, as the following code snippet shows:

```
...
Sun
</td><td align="Center">
Mon
</td><td align="Center">
Tue
</td><td align="Center">
Wed
...
```

Some client-side JavaScript is mixed in as well because the ASP.NET engine realized that the browser was capable of handling it, and sent the code to the browser. Remember, server controls work hard to determine the capabilities of the client browser, and to render HTML and/or JavaScript to give the user the most robust experience.

Validation Controls

One of the most common requests for any Web application is the ability to perform client-side validation of input. It would have been nice if HTML were written with some sort of mask that could be applied to fields, but that isn't the case. In standard

HTML, there is no way to perform validation of data on the client. To get around this problem, most browsers let you mix in some client-side script code, which is capable of performing validation, but the code for this can be tedious to write.

There are a number of reasons for performing validation on the client. First, you give the user a better experience. If you can immediately notify the user that he did not fill in a required field, you just saved him the time it would have taken to submit the form, have the server generate a message to inform him of the problem, and return the error message to him. In addition, using client-side validation lessens network traffic and server load by never sending invalid data to the server.

Microsoft provides a series of validation controls in ASP.NET that automate client-side form validation. The validation controls in ASP.NET are smart; they will perform the validation at the client if possible. If the client can handle DHTML, the validation controls send the code down to the client. If the browser is less capable, the validation code is actually executed on the server.

From a development standpoint, however, you code the controls exactly the same way. This means you don't have to dumb down your validation code for less-capable browsers. You can have robust validation code and be assured that the same code will be run for all browsers. It's just that the location of where the code is run changes depending on the capabilities of each browser.

To try out some validation, it is useful to start from scratch. Add a new Web Form to your project and name it `UserInfo.aspx`. Change `UserInfo`'s `pageLayout` property to `FlowLayout`. Click on the Toolbox and drag a text box from the Web Forms tab and drop it on your new Web form. Now, on the form, click to the right of the text box so that you can see the cursor. Press the Enter key to move to the next line. Click on the Toolbox, drag a button from the Web Forms tab, and drop it on the line below the text box. Now, from the Web Forms tab of the Toolbox, drag a `RequiredFieldValidator` control and drop it next to the text box.

The `RequiredFieldValidator` control checks whether a particular field has been filled in. You place the `RequiredFieldValidator` on the page, and then tie it to a particular input control, such as a `TextBox`, `CheckBox`, or `DropDownList`. In this case, you want to tie it to the text box. Click once on the `RequiredFieldValidator` if it is not already the current object. In the Properties window, you will see a property named `ControlToValidate`. Click here and drop down the list. The only input control that will appear is `TextBox1`; this is correct, so choose it.

You are ready for your first test. Before you run the page, however, you might be tempted to go to the project properties and choose to make `UserInfo.aspx` the startup object. However, Web Applications work differently: There is no startup object, per se. Instead, right-click on `UserInfo.aspx` in the Solution Explorer window and choose `Set As Start Page`. This notifies Visual Studio.NET which page to

run when the user clicks the Start button. It does not change anything in the page itself. Now that you have set `UserInfo.aspx` as the start page, run the project. The page appears inside Internet Explorer.

After the page is running, click on the button without entering anything in the text box. Immediately, the message `RequiredFieldValidator` appears to the right of the text box. Now, enter anything into the text box and click the button. The `RequiredFieldValidator` text goes away, and the entered value stays in the text box. When the `RequiredFieldValidator` text disappears and the value you typed in the text box remains, it means that the submit action has taken place, which means you have made a round trip to the server.

The fact that the text stayed in the text box even after a round trip to the server is important. Before ASP.NET, you would have to have written server-side code to capture field values for you, and then write those values back into the text boxes. ASP.NET handles this automatically, meaning you get this advanced functionality without having to write the code.

If you use IE 4.0 or higher, the validation actually occurs on the client. This allows you to get immediate feedback that the field is blank, when in fact you specified that a value is required. After you fill in the value, you send the data to the server and perform a server round trip.

You can modify the error message by changing the properties. You will see this shortly.

Types of Validators

If you look at the Web Forms tab of the Toolbox, you'll see a variety of validator controls. Briefly, they are as follows:

- `RequiredFieldValidator`—This validator requires that its `ControlToValidate` property have a value. In other words, the control to which this validator is tied cannot be left blank.
- `CompareValidator`—This validator compares the value the user entered with a value you specify. Your specified value could be a constant, a calculated value, or a value from a database.
- `RangeValidator`—This validator requires that entered data be within a particular range. The range can be numeric, dates, currency, or alphabetical.
- `RegularExpressionValidator`—Regular expressions are also known as *masks*. This validator can make sure that entered data matches a particular format, such as the format of phone numbers and Social Security numbers.
- `CustomValidator`—This validator uses code you write yourself to validate the data.

- **ValidationSummary**—This validator simply reports all the errors encountered by the other validators. You will see an example of this validator shortly.

Applying Multiple Validators to the Same Field

It is possible to apply more than one validator to the same field. For example, you might have a Social Security number field, which is required and must be in a particular format. Drag a **RegularExpressionValidator** to the form, next to the **RequiredFieldValidator** you added earlier. On the **RegularExpressionValidator**, change the **ControlToValidate** property to **TextBox1**, and click on the ellipses on the **ValidateExpression** property. In the **Regular Expression Editor** dialog box that pops up, choose **U.S. Social Security Number** and then click **OK**.

Run the project, and try these three tests:

- Do not enter anything, and click the button. You will notice that the **RequiredFieldValidator** message appears.
- Enter **Hello** into the text box and press the button. The **RequiredFieldValidator** no longer appears because that condition is satisfied. However, the **RegularExpressionValidator** now appears because you are failing that condition.
- Enter **111-11-1111** into the text box and click the button. No validator text appears, and you will now see the form make a trip to the server because both validators are satisfied.

Modifying the Validators

Notice that the text in the validators is not the friendliest text that you could show your clients. If you click the **RequiredFieldValidator** in the **Design** pane and look at the properties, you'll notice a property named **ErrorMessage**. Change this property to **This field is required**. Next, click on the **RegularExpressionValidator** and change its **ErrorMessage** property to **SSN must be in ###-##-#### format**.

Now, run the page again, and repeat the same three tests from the last section. You should get validator messages for the first two tests, but the text displayed should be the new values you entered in the **ErrorMessage** property of each validator.

You'll also notice that even when the **RequiredFieldValidator** is not displayed, the **RegularExpressionValidator** appears far enough to the right that you can tell where the **RequiredFieldValidator** appears. That is because the validators have a **Display** property called, and its default value is **Static**. If you change this value to **Dynamic** for the **RequiredFieldValidator** and run the page again, you'll see that the **RegularExpressionValidator** now appears next to the text box when it is displayed.

The Dynamic value says that the field will not take up any space unless it is displayed. This is supported by IE 4.0 and higher, but support on other browsers might not exist. In those cases, the validator fields will work as they did when you had the property set to Static.

Providing a Validation Summary

It is possible simply to mark the fields that failed their validations with a symbol and provide a summary of all the validations at the top or bottom of the page. The `ValidationSummary` control allows you to build just such a list of all the errors that occurred.

To see this work, you will want to modify the `UserInfo.aspx` quite a bit. Add text before the first box identifying it as the Social Security Number field. Drop down to the next line and add the text First Name and then add a `TextBox` ASP.NET server control. After you add the text box, change its `ID` property to something meaningful, such as `txtFName`. Now, go to the next line, type Last Name, add a `TextBox` ASP.NET server control, and give it a meaningful name. Add the following additional labels and controls, and give the text boxes meaningful names:

- Address 1
- Address 2
- City
- State
- Zip
- Phone

Now, add a `RequiredFieldValidator` next to First Name, Last Name, Address 1, City, State, and Zip. That means the Address 2 and Phone fields are optional. Make sure that you bind the `RequiredFieldValidators` to the correct text boxes.

Next to the Phone text box, add a `RegularExpressionValidator` and set the `ValidationExpression` property to U.S. Phone Number, and bind this to the Phone Number text box. At this point, your form will look something like what you see in Figure 7.5.

Now, change the text on the validators so that they make sense. For example, on the `RequiredFieldValidator` for the First Name text box, set the `ErrorMessage` property to The First Name field is required. Change all the validators this way, including the description for the Social Security Number field you added a while ago. When you are done, run the project. After it is running, press the button without entering anything. You should get a page like the one shown in Figure 7.6.

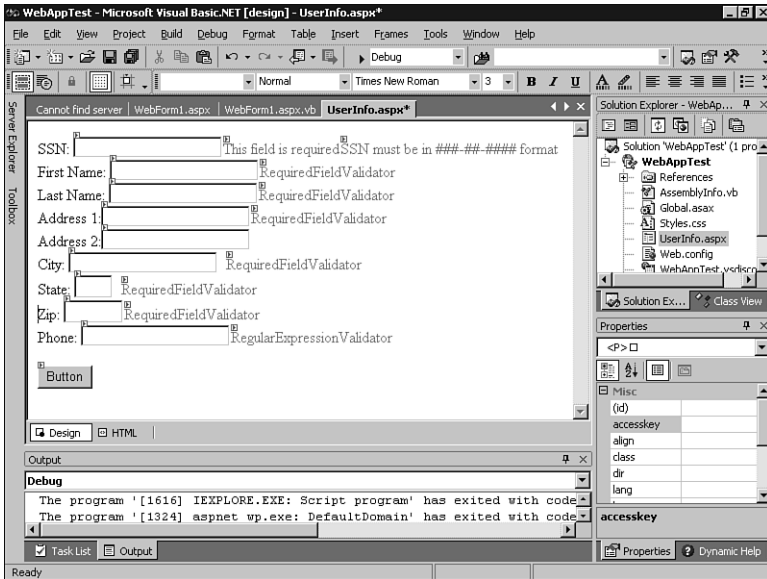


Figure 7.5

A more complex data entry screen with validator server controls.

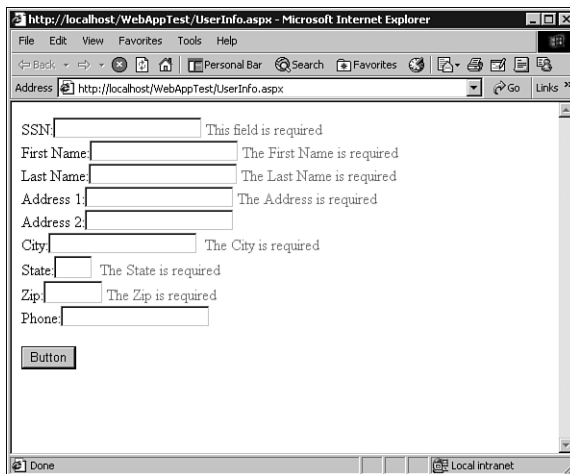


Figure 7.6

The data entry form is now being run, showing most of the validator messages.

As you saw in Figure 7.6, this can be a cumbersome way to show the errors. Instead, you might want to have a listing of all the errors in one place. That's where the `ValidationSummary` control comes into play. Drag a `ValidationSummary` control onto the form and drop it above the button but after the Phone text box and validator.

Highlight the control on the form, change the `HeaderText` property to The following errors occurred:, and then run the form. Click the button without entering anything, and you will see that the validator messages appear next to each text box, and the same messages appear in the `ValidationSummary` at the bottom of the page. Having this validation summary at the bottom can be handy, but do you need the same message beside each text box and in the summary? Usually not.

Return to the designer and on each validator tied to a field, change the `Text` property to an asterisk (*). Do not change anything in the `ValidationSummary`. Now, run the page again. If you press the button without filling in any fields, you will get an asterisk next to the required fields, but the `ErrorMessage` text you entered for each validator still appears in the `ValidationSummary` control.

In Figure 7.7, some fields have been entered, but some have not. In addition, the Phone field has been entered incorrectly. Notice the results in the `ValidationSummary` field.

The screenshot shows a web browser window titled "http://localhost/WebAppTest/UserInfo.aspx - Microsoft Internet ...". The address bar shows "http://localhost/WebAppTest/UserInfo.aspx". The form contains the following fields and values:

- SSN: 444-a1-bb33 *
- First Name: Martha
- Last Name: McMahon
- Address 1: *
- Address 2: *
- City: Anyplace
- State: KS
- Zip: *
- Phone: 5551212 *

Below the fields, the text "The following errors occurred:" is displayed, followed by a bulleted list of error messages:

- SSN must be in ###-##-#### format
- The Address is required
- The Zip is required
- The Phone Number must be in the (xxx) xxx-xxxx format

A "Button" is located at the bottom of the form.

Figure 7.7

The `ValidationSummary` control allows you to show all the validation errors in one place and, optionally, to mark each field.

You can see that the fields that have been entered properly do not have any indicator next to them. Those that are incorrect have an asterisk by them, and the error messages are listed in the validation summary at the bottom of the page. This provides a very easy, powerful mechanism for handling form validation. In the past, the alternative was client-side code that examined each field and then displayed error messages to the user. Here, you have gained the same functionality without writing any code.

Data Binding

One of the most common feats you will want to perform with your Web applications is presenting data from a database. Not surprisingly, Microsoft has added some ASP.NET controls specifically for displaying data. In addition, most of the regular ASP.NET server controls, such as the `TextBox`, can be bound to a data source.

The good news about data binding is that you use many of the same data controls you used in Chapter 6, “Database Access with VB.NET and ADO.NET,” which discussed the data controls and little bit about ADO.NET. There is a wrinkle with Web forms, however, and it is that you work with the controls in a slightly different way. In addition, the process that is outlined in this section might change slightly between Beta 2 and the final release of VS.NET. Still, the concepts should be the same.

Create a new Web Form in the project and name it `DataTest1.aspx`. This will open the form in the designer, and it is initially blank. To add some data-bound controls to this page, you must provide it with a `datasource`. Click on the Toolbox and from the Data tab, drag an `OleDbDataAdapter` to the page. This starts the `DataAdapter Configuration Wizard`, as you saw in Chapter 6. Click the Next button and you will have a choice of what data connection to use. Use the one you created in Chapter 6 that points to the Northwind database. Click the Next button.

On the next page, choose `Use SQL Statement` and click the Next button. In the text box, type in this SQL statement: `Select * from Products`. Click the Finish button.

NOTE

If all this seems like a lot of work, it is. There is a simpler way to achieve the same results. Using the Server Explorer, you can drag the `Products` table over from the appropriate server and drop it on your form. You're done.

Two controls end up being added to your form: an `OleDbConnection` and an `OleDbDataAdapter`. From the Data menu, choose `Generate DataSet`. A dialog box will open. Choose to create a new `DataSet` named `dsProduct`, and leave checked the box asking if you'd like to add an instance of the class to the designer.

Visual Studio now adds a `dsProduct` object, named `dsProduct1`, to your designer. At this point, you have a `DataSet` that can be filled with data.

In the Toolbox, change to the Web Forms tab. Drag a `DataGrid` control to the designer. The `DataGrid` is a control that automatically displays the data in a grid format, using HTML tables.

After the `DataGrid` is displayed on the form, right-click on it and choose `Property Builder`. This opens a `Properties` dialog box. On the General page, drop down the `DataSource` combo box and choose the `Products` table of `dsProducts1`. In the Data

Key Field combo box, drop down the list and choose the primary key of the Products table, ProductID. At this point, click the OK button.

Back in the designer, you will see that the grid has grown, and now has a column for each field in the database. You might be tempted to run this page now, but you aren't quite done.

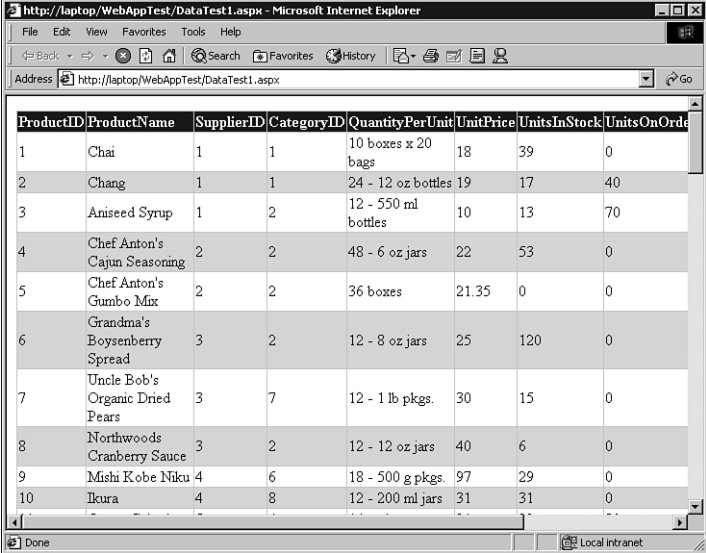
Double-click on the form (but not the DataTable) to get to the code window. You should see a DataTest1_Load event procedure. There is already some code in there as well. You need to add two lines of code, so you end up with this:

```
Protected Sub DataTest1_Load _  
    (ByVal Sender As System.Object, ByVal e As System.EventArgs)  
    If Not IsPostBack Then ' Evals true first time browser hits the page  
        Me.oleDbConnection1.Open()  
        Me.oleDbDataAdapter.Fill(dsProducts)  
        Me.oleDbConnection1.Close()  
        DataGrid1.DataBind()  
    End If  
End Sub
```

The first line of code you added, FillDataSet(dsProduct1), calls a routine that was created for you when you chose to generate the methods. This goes to the database, executes the statement in the oleDbDataAdapter, and stores the resulting records in the DataSet you pass in as a parameter (in this case, dsProduct1). The fourth line of code, DataGrid1.DataBind(), binds the grid to the DataView control, which is in turn bound to the DataSet.

Make sure that your DataTest1 form is set as the startup form and run the project. You should see the records from the Products table appear in a grid format on the page in IE, as shown in Figure 7.8. If you click View, Source in the browser, you will see a tremendous amount of state information at the top, but if you scroll past it, you will see that the data is displayed in a simple HTML table.

This just scratches the surface of what you can do with data binding in your ASP.NET applications. However, the discussion will stop here because you have seen the basic functionality provided, and future changes to Visual Studio.NET might require changes in how data binding is done.



ProductID	ProductName	SupplierID	CategoryID	QuantityPerUnit	UnitPrice	UnitsInStock	UnitsOnOrder
1	Chai	1	1	10 boxes x 20 bags	18	39	0
2	Chang	1	1	24 - 12 oz bottles	19	17	40
3	Aniseed Syrup	1	2	12 - 550 ml bottles	10	13	70
4	Chef Anton's Cajun Seasoning	2	2	48 - 6 oz jars	22	53	0
5	Chef Anton's Gumbo Mix	2	2	36 boxes	21.35	0	0
6	Grandma's Boysenberry Spread	3	2	12 - 8 oz jars	25	120	0
7	Uncle Bob's Organic Dried Pears	3	7	12 - 1 lb pkgs.	30	15	0
8	Northwoods Cranberry Sauce	3	2	12 - 12 oz jars	40	6	0
9	Mishi Kobe Niku	4	6	18 - 500 g pkgs.	97	29	0
10	Ikura	4	8	12 - 200 ml jars	31	31	0

Figure 7.8

The DataGrid displays the contents of the Products table. You have opened a database connection, executed a SQL statement, and filled a grid with data, with only two lines of code.

Handling Re-entrant Pages

You might have noticed some code in the DataTest1_Load event handler. That code was

```
If Not IsPostBack Then ' Evaluates true first time browser hits the page
```

The IsPostBack check is used to determine whether the user has called this form before. Think about the stateless nature of HTTP: Any time someone calls a form, it looks like a brand-new user to the Web server. ASP.NET, however, writes some ViewState information into the page when it is rendered and sent to the client. That way, when the client submits the page back to the server, ASP.NET can see that the page has been sent back to the user before. Therefore, the code inside this If statement runs only once for each user who requests this page. This can be important. For example, if you want to know which users visit which pages, you can record that information just once instead of each time that an event is passed back to the server.

ASP.NET server controls work on a re-entrant page model. In other words, when a control's events cause the form in the page to be submitted, the page calls itself; in other words, the Action attribute in the Form tag points to the same page you are already viewing. This means that many forms call themselves over and over, and the IsPostBack is important to improve the scalability and performance of the application.

Therefore, unlike a Windows application, the Load and Unload events can fire many times while using the same page. IsPostBack provides an easy way to see whether this is truly the first load for a particular user.

Summary

As you can see, Microsoft has added some powerful features to VB.NET. No longer do you have to go into another tool to create your Web applications. In Visual Studio.NET, you have a powerful HTML editor and a host of controls to handle everything from static HTML to form validation to data binding.

Microsoft has removed much of the programming challenge from handling events over the Web. By and large, you code your applications just as you would any Windows application, and they will work fine.

However, there are some issues that are clearly different when building Web applications. For example, state issues still require some planning and forethought. Many of the controls you use are still not as powerful as the native Windows controls. And, accessing resources on the user's machine, such as reading or writing to files, is not always permitted.

Still, Visual Studio.NET represents a huge leap forward in the ability to easily create event-driven Web applications. The next chapter carries that improvement a step further.



CHAPTER 8

Building Web Services with VB.NET

Microsoft likes to point out that .NET acts like a huge operating system. In effect, the entire Internet becomes your operating system. This means that pieces of your applications can be distributed over the Internet but the applications run as if the pieces were all on your local machine.

Imagine if you had told someone back in the early days of Visual Basic that someday they'd be writing their applications in a number of separate components and putting those parts on different machines. The application sitting on the user's desktop would call these components on other machines, and those components would access the data on still other machines. The data would be returned to these components and finally flow back to the client application.

Naturally, this sounds quite normal today. However, now consider taking those components, and even the database, and removing them from your internal network. Spread them out all over the Internet, so that the only way with which you can communicate with them is HTTP. This is precisely what a Web service is all about.

The idea behind a Web service is to create a reusable component that can be called over standard HTTP, but has the full power of a .NET language application. These components are *discoverable*, which means that you can locate and call available components. The format for calling particular methods is exposed as well, so anyone can determine what methods are available and how to call them.

Web services, like COM components, can be called by any front-end application. Therefore, both Windows forms and Web forms can call the same Web services. Web services are free to call other Web services.

To learn more about Web services, you will dive in and create your first Web service. After that, you'll go back and learn more about how Web services work.

Creating Your First Web Service

Currency conversion is a common activity needed by Web applications. Because of the global nature of the Web, you can browse stores anywhere in the world. Normally, of course, those stores show prices in their local currency. You might need to convert these values into your local currency. In this section, you'll build the basis for a currency conversion Web service.

Open Visual Studio.NET. Create a new Visual Basic project of the type Web Service. Name the project `CurrencyConverter` and make sure that the server on which you create the project is a Web server running IIS 4.0 or higher and the .NET Framework.

After the project is loaded, you have a blank designer, much like you have seen before. If you look in the Solution Explorer window, you'll see that the current page is called `Service1.aspx`. `.ASMX` is the extension for a Web service. You'll also notice a file named `CurrencyConverter.vsdisco`. A `.VSDISCO` file is a discovery file, and the discovery process will be discussed later in this chapter.

Right-click on the `Service1.aspx` file in the Solution Explorer and choose **Rename**. Name the file `CurrConvert.aspx`. Notice that this just changes the filename; the service is still named `Service1`. Double-click on the designer of the `CurrConvert.aspx` file, and the code-behind file named `CurrConvert.aspx.vb` will open.

If you look at the code, you'll notice that a function is commented out. The function is a public function named `HelloWorld()`, but there is a strange element before the name of the function: `<WebMethod()>`. This marks the function as a method that is callable over the Web. Using the example as a template, add the following code to the file:

```
<WebMethod()> Public Function ConvertCurrency(ByVal convertAmount As Decimal, _  
    ByVal convertFrom As String, ByVal convertTo As String) As Decimal  
    Select Case convertFrom  
        Case "British Pounds"  
            Return CDec(convertAmount * 1.44333)  
        Case "Japanese Yen"  
            Return CDec(convertAmount * 0.00859358)  
    End Select  
End Function
```

This code creates a method called `ConvertCurrency`. The method accepts three parameters: the amount of the currency to convert, the type of currency to convert from,

and the type of currency to convert to. Inside the procedure is a simple `Select Case` statement that determines whether you are converting from British pounds or Japanese yen. Inside each case, you convert the amount into United States dollars.

In the real world, of course, the money could be converted to one of many countries. More importantly, you could query a database or some other resource to determine the current exchange rate, instead of hard-coding it as is done here. Again, the purpose of this exercise is merely to show you how Web services work.

Now, click on the Build menu and choose Build. This will build the Web service on the Web server. The service is now ready for testing.

Testing the Web Service

Visual Studio.NET provides a very simple way to test a Web service. Right-click on `CurrConvert.asmx` in the Solution Explorer window and choose View in Browser.

VS.NET creates a default page that allows you to work with methods in your service whose parameters can be input via HTTP POST or GET. The only option on the page is the one method you created: `ConvertCurrency`. Click on the `ConvertCurrency` link, and a new page appears. The new page allows you to enter values for any parameters in the method you created, and also contains a button to submit those values. Figure 8.1 shows what this form looks like.

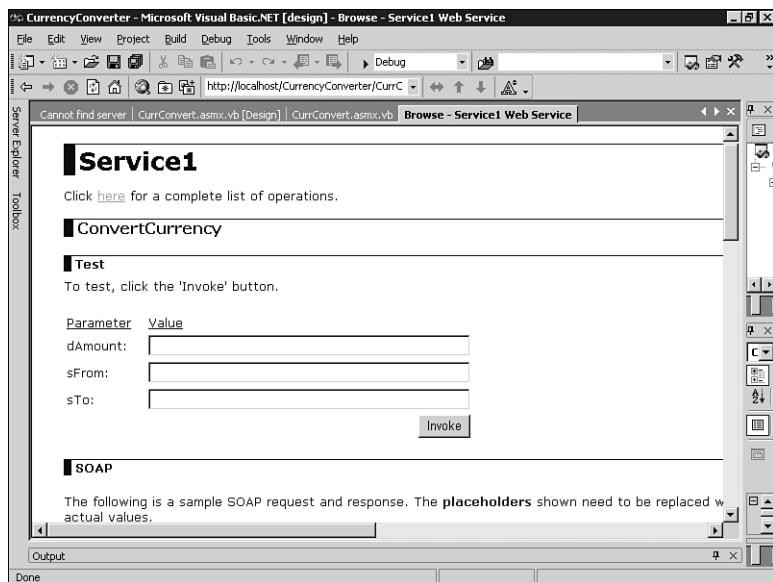


Figure 8.1

Visual Studio.NET automatically generates a page that allows you to test your Web services.

You can see three text boxes in the middle of the page that allow you to enter values and then test them using the Invoke button. Enter 100 in the convertAmount box, British Pounds in the convertFrom box, and US Dollars in the convertTo box. Technically, in this example, it doesn't matter what you enter in the convertTo box because your code never checks it. However, you can't leave the convertTo box blank because the parameter is not optional. It helps to go ahead and put in an actual value for the "convert to" field.

When you click the Invoke button, the page passes the data to the Web service. The method runs and the data is returned in an XML stream that looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<decimal xmlns="http://tempuri.org/">144.333</decimal>
```

You can see that the return is a value, of the decimal data type, of 144.333. This means that 100 British pounds buys 144.333 U.S. dollars. If you run the form again but put Japanese Yen in the convertFrom field, you will see that 100 yen buys approximately .86 U.S. dollars.

Immediately, you can see that the return from the Web service is an XML stream. What you can't see is that the call to the service is formatted as an XML stream as well. You'll learn more about this later.

Creating a Web Service Client

Now that you have created and tested the Web service, it is time to create a client that can access it. The client can be almost any kind of application, including a Windows application, Web application, or Web service. For this example, you'll create a Web Application project to test the service.

Create a new Web Application in Visual Studio.NET and name it ConversionClient. Make sure that you have it set to close the existing solution because you do not want this project to be added to the Web service solution.

You will wind up with a blank designer. Add the following Web Forms controls to the designer: four labels, one text box, two drop-down list boxes, and one button. Lay them out so that they look something like Figure 8.2.

NOTE

I had you drag four labels from the Web Forms tab of the Toolbox. Unless you're going to change them via code, you're actually much better off using the labels from the HTML tab. This HTML label just inserts a lightweight <DIV> tag, instead of creating a server control.

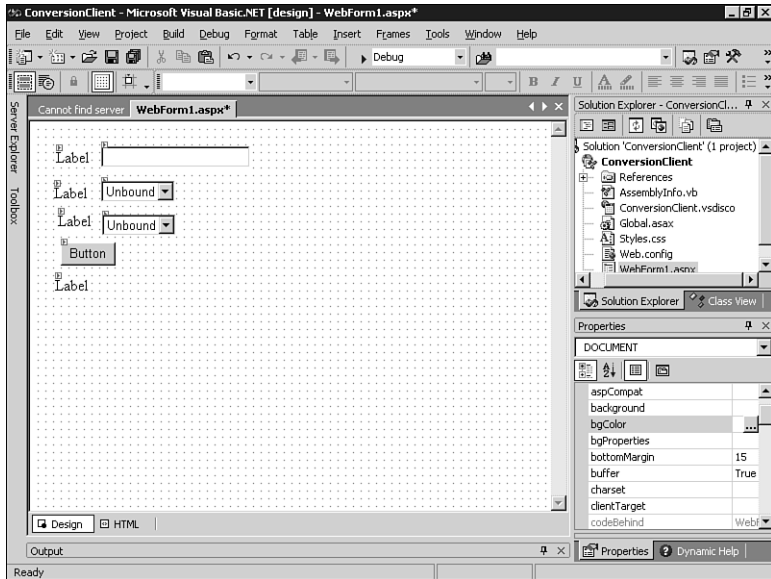


Figure 8.2

The construction begins on a client that will use your Web service.

Now, modify the first three labels to Amount:, From:, and To:. Change the text on the button to Convert. Clear out the value in the last label. Change the name of the last label to `resultValue` by changing its ID property.

Click on the drop-down list box next to the From label and name it `fromValue`. Now, locate the Items property under the Misc section and click the ellipses. This opens the ListItem Collection Editor, as shown in Figure 8.3. Click the Add button, and in the ListItem Properties panel, set the text to Japanese Yen and change the Selected attribute to True, as shown in Figure 8.4. Click the Add button again, but this time, enter British Pounds into the Text attribute. Do not set Selected to True for this one. Click the OK button.

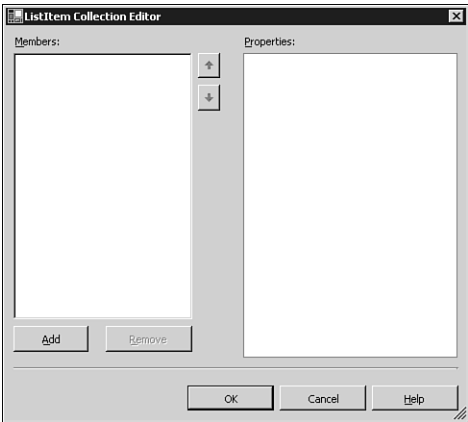


Figure 8.3
The ListItem Collection Editor.

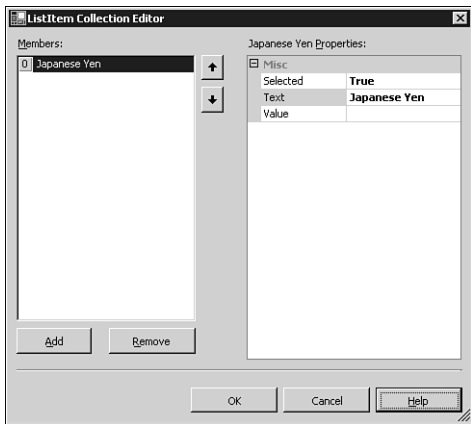


Figure 8.4
Adding items to the drop-down list box.

Now, click the drop-down list box next to the To: label. Change its name to toValue. Click the ellipses in the Items property box. Add one item, US Dollars, and mark it as Selected. Your final form should look like the one in Figure 8.5.

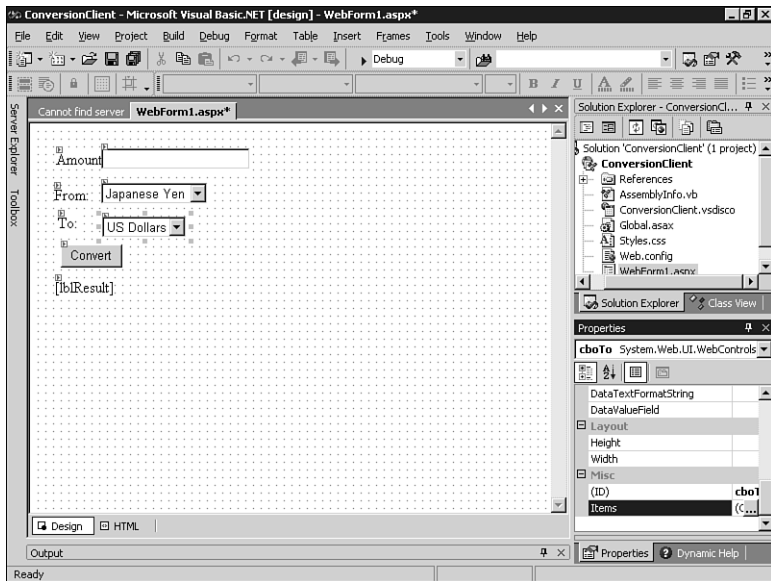


Figure 8.5

The final user interface for the form, showing the drop-down list boxes containing default values.

Next, double-click on the Convert button to get to the code window. Before you write the code for the click event, however, you need to add a reference to the Web service you created in the last section. Adding a reference is critical; doing so allows you to program against the Web service as if it were a local object.

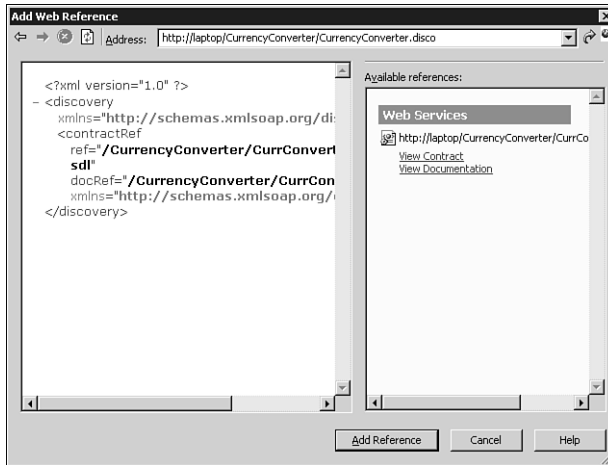
In the Solution Explorer, right-click on the ConversionClient project name and choose Add Web Reference. This is different from the Add Reference option, so make sure that you choose the correct one. The Add Web Reference dialog box will appear. In the Address field, you need to type the address of your Web service. This takes on the following format:

```
http://<servername>/<projectname>/<disco file>
```

Your server name might be different, but in my case, the address looks like the following (you could just use localhost as the server name if it is on the same machine):

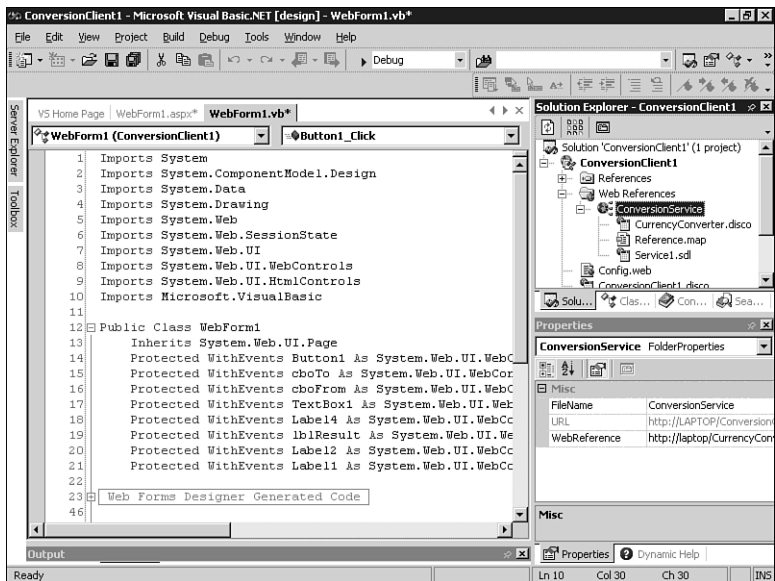
```
http://laptop/CurrencyConverter/CurrencyConverter.vsdico
```

After you have entered the URL, click the go arrow. The dialog box will make contact with the DISCO file and retrieve the necessary information from the Web service. Your Add Web Reference dialog box will look like the one in Figure 8.6. Click the Add Reference button to complete this process.

**Figure 8.6**

The Add Web Reference dialog box has retrieved information about your Web service.

After you add the reference, you'll see a Web References node appear in the Solution Explorer. If you expand the node, you'll see an item with the name of the server holding the component to which you just connected. You can rename this if you want, so right-click on the server name and change the name to *ConversionService*. Figure 8.7 shows what this should look like when you are done.

**Figure 8.7**

The Add Web Reference dialog box has retrieved information about your Web service.

Now, in the WebForm1.aspx.vb code window, you need to import the namespace for the Web service. The namespace is the name of your current application and the name of the server on which the Web service resides. However, you just changed the name to `ConversionService`, so add the following code just under the last Imports statement at the top of the code window:

```
Imports ConversionClient.ConversionService
```

`CoverersionClient` is the name of your client program, and `ConversionService` is how you reference the Web service.

In the `Button1_Click` procedure, you need to write the code to call the Web service, pass in the proper parameters, and then display the output. Type the following code:

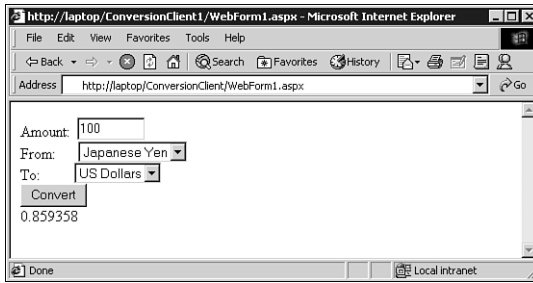
```
Public Sub Button1_Click(ByVal sender As Object, _
    ByVal e As System.EventArgs)
    Dim convertService As Service1
    convertService = New Service1()

    Dim convertAmount As Decimal
    Dim convertFrom As String
    Dim convertTo As String
    Dim convertAnswer As Decimal

    convertAmount = CType(TextBox1().Text, Decimal)
    convertFrom = fromValue().SelectedItem.ToString
    convertTo = toValue().SelectedItem.ToString
    convertAnswer = convertService.ConvertCurrency(convertAmount, convertFrom,
        convertTo)
    resultValue.Text = convertAnswer.ToString
End Sub
```

This code first creates an object of type `Service1`, which is the name of the service you created in the last section. You can call it this way because it resides inside the `ConversionService` namespace you imported at the top of the code window. Next, some variables are created and filled with values from the controls on the form. The code calls the `ConvertCurrency` method and passes in the parameters. The result is returned into a variable, which is then used to set the `resultValue.Text` property.

Go ahead and run the project. You'll see the Web form start up in IE. Put `100` in the `Amount:` box, and choose `Japanese Yen` for the `From:` box. Click the `Convert` button, and you should see the value `0.859358` appear below the button after the page makes a round trip to the server. The resulting page should appear like the one shown in Figure 8.8.

**Figure 8.8**

Your Web Application project has just received a result from your Web service.

One of the important things to understand about this process is that you never had to get your hands dirty handling the underlying XML. Instead, you programmed the service just like a Class Library application, with properties and methods, but you are now calling components that could be anywhere on the Internet.

Another thing to realize is that you have basically supplanted DCOM, which is specific to the Microsoft platform, and replaced it with the ubiquitous HTTP and XML. This means that the client does not have to be running the Windows operating system. Because the communication is all text, it flows freely through firewalls.

How Web Services Work

As you saw in the code, any method that you want to publicly expose is made public by adding the `<WebMethod(>)` attribute to the method declaration. This makes the method automatically discoverable by anyone accessing the project's URL. Any class that has one or more methods marked with `<WebMethod(>)` becomes a Web service. The Framework handles the task of setting up all the necessary hooks for the component to be callable via HTTP.

Most Web services will import the `System.Web.Services` namespace. This isn't necessary, but if you don't, you'll have to use the full `System.Web.Services.myService` syntax to call your service. Many services also inherit from the `System.Web.Services.WebService` base class because doing so gives them access to the intrinsic ASP objects, such as the `Application` and `Session` objects. This class contains the underlying methods to enable the Web service functionality. Web services also have a discovery file (the DISCO file) that later helps generate a WSDL (Service Description Language) file. You will see more on this later.

You saw earlier that you can test a Web service by choosing the View in Browser option from the ASMX file. The browser builds a page on-the-fly that contains text boxes that map to each parameter in the various methods in the service. This page

also contains some descriptive text and a link to view the SDL for the service. The SDL is quite lengthy even for the small, single method service you created earlier.

And You Thought Disco Was Dead

After you have created the service, it is time to deploy it to the Web server. Earlier in the chapter, deployment was easy because you just had to choose to build the project. However, if you now want to copy the service to another machine, you have to make sure that you place the proper discovery document on that machine. The DISCO file allows a user to determine what services are available on your server, and what methods these services support. When users point their browser to your VSDISCO file, they can discover the Web services you have to offer.

However, what if your user does not know the path to—or the name of—the discovery document? Any development machine with VS.NET installed has the dynamic discovery document, `default.vsdisco`. In general, ASP.NET servers rely on `default.disco` files for static discovery. Therefore, the Web service developer needs to think carefully about which services should be available to the outside world on the deployment machine, and must author the static disco file properly.

How you actually deploy the Web service depends on whether you are using Visual Studio.NET or performing a manual copy. Furthermore, it changes depending on whether you are distributing a binary file or the text-based ASMX file. This deployment discussion is beyond the scope of this book, but will be covered in my later version of this book, based on the final release of Visual Studio.NET.

Speaking of discovery, you should be aware of UDDI. UDDI stands for Universal Discovery, Description, and Integration. The UDDI Project is an attempt to form a registration service for Web services. This means that you could search for Web services within a particular industry or even a particular company. This initiative represents a real attempt to create a central repository for Web components, which means you could easily create Web applications using components that already exist and are accessible over the Web. You can learn more about the UDDI Project by visiting its site at <http://www.uddi.org>.

Accessing Web Services

ASP.NET currently supports three for accessing a Web service: HTTP-GET, HTTP-POST, and SOAP. SOAP, which stands for Simple Object Access Protocol, defines an XML format for calling methods and passing in parameters, all surrounded by the schema definition. In addition, the result comes back in an XML format. HTTP-GET and HTTP-POST can call Web services, but SOAP adds the ability to pass more complex data types, such as classes, structures, and datasets. Despite these advanced features, SOAP is still all text, which has the advantage of being portable and able to traverse firewalls, unlike DCOM and CORBA.

The Web services you create with VB.NET automatically support all three mechanisms. In fact, because your services support SOAP, they can be called not only with HTTP, but with other protocols as well, such as SMTP.

If you know the URL for your Web service, you can enter it in the browser and you will get the same type of page that was generated for you when you tested the service inside the IDE. For example, the following is a line that points to the service you built in this chapter. Don't forget to change the name of the server as necessary:

```
http://localhost/currencyconverter/currconvert.asmx
```

This creates a page that allows you to test the service. This is important because by knowing this address, anyone can connect to your Web service using any tool. There is also a link to the WSDL contract, which is used by your client application. This contract contains more detailed information about the service. More importantly, it is the file used to generate a Web service proxy on the client. You don't have to worry about doing this if your client is built in Visual Studio.NET and you add a Web reference, as you did in this chapter. VS.NET creates the proxy for you, which is just a VB.NET class file (or C#, depending on your language preference). If you want to create it manually, you merely save the SDL to your machine and run the `Webserviceutil.exe` program that comes with VS.NET.

Although it is beyond the scope of this book, security is a topic that often comes up when discussing Web services. You do have control over the security on these services, and you might not want everyone in the world to be able to access the service freely. Let's face it: Some companies actually want to *charge* people for the use of certain services, in a display of what our capitalist system is all about. Because you are building your Web services for .NET, you can take full advantage of the security features it offers. Because these services are being offered via HTTP, Internet Information Server is another possible point for adding security. As with deployment, security will be discussed in a full version of this book to be based on the final release of Visual Studio.NET.

Summary

Although this chapter is short, it shows you the basic building blocks for creating Web services. Web services are one of the most exciting technologies in the .NET Framework because they allow you to use the architecture of the Internet as a way to extend the distributed nature of applications. No longer are you tied to accessing components only within your organization. Now you can access components anywhere, via HTTP. And you can do it without worrying about firewalls, or what operating systems exist on the host or the client.



CHAPTER 9

Building Windows Services with VB.NET

Visual Basic.NET lets you build Windows Services natively in VB for the first time. Windows Services, formerly called NT Services, are potentially long-running executables that can start without a user logging in, can be paused and restarted, and can be configured to run under different security contexts. Windows Services had been the domain of C++ developers until now, when the .NET Framework opens the ability for any .NET language to create services.

Windows Services should not be confused with Web Services, that are implemented as a different project type. Services are only supported on Windows NT, Windows 2000, and Windows XP.

Services work a little differently from just about any other project type. You cannot start a project within the Visual Studio.NET environment and step through it for debugging purposes. You must install your application as a service with the operating system in order for it to run. Then, after it is running, you can attach a debugger to it. Although Windows Services do not have a user interface, they typically use the event log to communicate problems or messages to the user or administrator.

Services are useful in certain circumstances. For example, I worked on an application in which files could arrive at any time, via FTP. A service ran that monitored a directory, and received an event when a file was added to the directory. The

service then went about processing the file as necessary. This was a perfect use for a service: It needed to run all the time, regardless of whether anyone was logged in. The files could arrive at any time, so the service had to be running constantly. Finally, no user interface was necessary.

Once again, you will dive into a project and get it working, and then examine its various pieces. The service you will write checks the processor usage on a regular basis and logs the information to a file. This service introduces a couple of new items: working with text files and capturing performance counters. All this will be woven into your first service.

Creating Your First Windows Services Project

Open Visual Studio.NET and create a Windows Services project. Name the project LearningVBservice. You'll see that all you have in the Solution Explorer is a .VB file named Service1, along with the AssemblyInfo file.

Double-click on the designer to access the code page. You have the definition `Public Class Service1`. Change this line to `Public Class UsageMonitor`.

If you expand the code the IDE generated for you, you'll notice that you have a `Sub New`, which is not surprising. However, you also have a `Sub Main`, which is not something you have seen in most of your other applications. As you might suspect, the `Sub Main` is the first procedure that will run by default. Therefore, there is already some code in this routine to initialize the service. One line of code in `Sub Main` references `Service1`, but you've changed that name. Therefore, find the following line of code:

```
ServicesToRun = New System.ServiceProcess.ServiceBase() {New Service1() }
```

Change the code to this:

```
ServicesToRun = New System.ServiceProcess.ServiceBase() {New UsageMonitor() }
```

Next, find a line of code that starts with `Me.ServiceName` and change the line to this:

```
Me.ServiceName = "UsageMonitor"
```

Now, go back to the designer page. Open the Toolbox and in the Components tab, drag a `Timer` over and drop it on the designer. The entire Windows service is a component, which as you have seen, has a form-like designer. Because this designer is not a form, the `Timer` does not end up in the component tray; instead, VS.NET shows the control on the designer.

If you select the `Timer1` component and look at its properties, you'll see that by default it is disabled, and its `Interval` property is set to `100`. The `Interval` property is in milliseconds, and is of type `Double`. Therefore, to wait one second, you put in a value of `1000`. For this example, you want to capture statistics once every five

seconds, so enter **5000** and enable the timer by setting its `Enabled` property to `True`. In a real application, you might want to grab values only every minute or every five minutes, but you probably don't want to sit for 10 minutes just to make sure that this first Windows Services project works properly.

Next, drag over a `PerformanceCounter` control from the `Components` tab of the `Toolbox`. This allows you to access the same performance information you can get with the `Performance` tool (often called `Performance Monitor`, or `PerfMon`). Highlight the `performanceCounter1` control and modify the following properties:

- Set `CategoryName` to `"processor"`
- Set `CounterName` to `"% Processor Time"`
- Set `InstanceName` to `"_Total"`
- Leave `MachineName` set to a period, which is a shortcut for the local machine name

NOTE

An alternative approach is to drag over a `PerfCounter` object from the `Server Explorer`, and then tweak the properties. Don't forget about the `Server Explorer` when working with services.

Double-click on the `Timer1` component to open the `timer1_Tick` event procedure. Before entering anything for the `timer1_Tick` procedure, go to the very top of the code window and add the following line of code:

```
Imports System.IO
```

Now, in the `timer1_Tick` event procedure, enter the following code:

```
Protected Sub Timer1_Tick(ByVal sender As Object, _
    ByVal e As System.EventArgs)
    Dim file As StreamWriter = New StreamWriter("c:\output.txt", True)
    file.WriteLine("CPU Usage: " & _
        PerformanceCounter1.NextValue.ToString & " - " & Now)
    file.Close()
End Sub
```

This is the code that runs every five seconds. It opens a text file called `output.txt` in the root of the `C:` drive. The `True` argument appends to the existing file. The code writes a string that contains the CPU usage from the performance counter, and then appends the current date and time. Finally, it closes the file. Notice that to use the shortcut name `StreamWriter`, you had to add the `System.IO` namespace, which has classes that handle many different types of I/O operations. This keeps you from having to type `System.IO.StreamWriter` to reference the class.

If you look at the code further, you should see stubs for the `OnStart` and `OnStop` routines. Modify the stubs—and add an `OnContinue` routine—as shown in the following:

```
Protected Overrides Sub OnStart(ByVal args() As String)
    timer1.Enabled = False
    Dim file As TextWriter = New StreamWriter("c:\output.txt", True)
    file.WriteLine("Service Started")
    file.Close()
    timer1.Interval = 5000
    timer1.Enabled = True
End Sub

Protected Overrides Sub OnStop()
    Timer1.Enabled = False
    Dim file As TextWriter = New StreamWriter("c:\output.txt", True)
    file.WriteLine("Service Stopped")
    file.Close()
End Sub

Protected Overrides Sub OnContinue()
    Dim file As StreamWriter = New StreamWriter("c:\output.txt", True)
    file.WriteLine("Service Restarted")
    file.Close()
End Sub
```

As you can see, when the service starts, the code writes to the text file that the service is starting. Then the timer's interval is set and the timer is enabled, in case you failed to make the changes described in the last section. If the service is stopped, that fact is recorded in the text file as well. If someone pauses the service and later resumes it, the fact that the service was restarted is written in the text file.

This is all that needs to be written in the service. However, you have to do more to make the service usable: You have to add installers to your application.

Adding Installers to Your Service

You need to return to the designer for your service. Right-click on the designer and choose `Add Installer`. A new window is added to the designer area, labeled `ProjectInstaller.vb [Design]`. Two components will be added to the project. One is for installing your service, whereas the other is for installing the process that hosts your service while it is running. All services run in their own process so that they can run independently of any particular user.

Before building the project, you need to perform a couple of tasks because you changed the name of the class from `UserService1` to `UsageMonitor`. On the `ProjectInstaller.vb` designer, click on the `serviceInstaller1` control and change the

ServiceName property to UsageMonitor. Then, in the Solution Explorer window, right-click on the LearningVBservice project and choose Properties. In the property pages, change the Startup Object combo box to UsageMonitor.

You are now ready to build the service. Choose Build from the Build menu. Your service will now be built.

Configuring Your Service

Your service is compiled, but the system doesn't know anything about it yet. You have to install this service before it will be available for starting and stopping in the Service Control Manager. Open a command or console window (that's a DOS prompt), and go to the directory where the service was compiled. For most of you, this will be C:\Documents and Settings\<your username>\My Documents\Visual Studio Projects\LearningVBservice\bin. When you're there, type in the following command:

```
installutil LearningVBservice.exe
```

NOTE

InstallUtil.exe must be in the path in order to work in this example. If it is not in the path, you'll have to type the full pathname.

This will take a moment. A box pops up, asking you to enter the username and password under which you want this service to run. You can see this box in Figure 9.1. Notice that you need to type the name of the server or domain, depending on your setup. After that is done, you are ready to start your service and set its startup options.

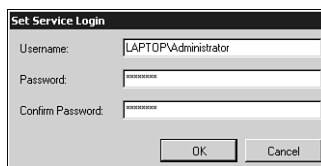
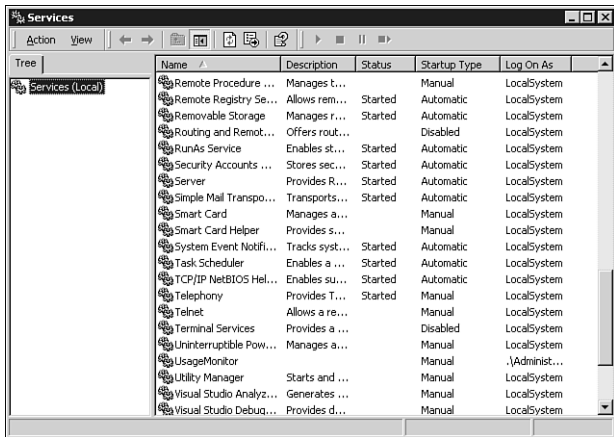


Figure 9.1

Installing your Windows service with the Services applet.

Open the Services management tool, which can be found in the Computer Management application in Windows 2000. Your service, UsageMonitor, will appear in the list, as you see in Figure 9.2. Double-click the UsageMonitor service to open the Properties dialog box. You can change the startup type from Manual to Automatic if you want the service to start every time Windows starts, and it will launch and begin operations even if no one logs on. At this point, you can just click the Start button and let the service run for a while. After it runs for a time, stop it. Now you can examine the log that it produced.

**Figure 9.2**

Your first Web service showing up in the Windows Services Control Manager.

The log file that is produced shows the CPU utilization every five seconds. The service can only be stopped and started. It cannot be paused because the `CanPauseAndContinue` property is set to `False` by default, and you did not change that before you built the service. Here is a sample of the file that was produced:

```
Service Started
CPU Usage: 0 - 2/24/2001 4:21:55 PM
CPU Usage: 9.504132 - 2/24/2001 4:21:57 PM
CPU Usage: 9.8 - 2/24/2001 4:22:02 PM
...
CPU Usage: 23 - 2/24/2001 4:24:07 PM
CPU Usage: 1.19760478 - 2/24/2001 4:24:12 PM
CPU Usage: 5.61122227 - 2/24/2001 4:24:17 PM
CPU Usage: 4.2 - 2/24/2001 4:24:22 PM
Service Stopped
```

The first value for the CPU usage is often a throwaway value. After that, all the numbers look legitimate.

Understanding Windows Services

As you know, Windows Services can run when Windows starts, even if no one logs in. These applications have no user interface and run in the background, independent of any user. They “log in” as the *service* account or as a specific user, which means they can run under the security context of a specific user, regardless of who is logged in.

Windows Services are unique in several ways. You cannot debug a service by starting it within the IDE because you must install it first so that Windows can manage it. You will see how to debug it later.

You had to add installers to this project, something you have not had to do in any other project. This, again, is because the service must be installed so that Windows can handle starting and stopping it.

Because a service does not have a user interface, you should plan to write any messages from the service into the Windows event log. This is the key to a long-running service; nothing blocks execution, such as waiting for a user to click the OK button to clear a message box. You want to have exemplary error handling in your service, capturing errors so that they can be written to the event log instead of causing a message to be displayed to the user, indicating that an error was encountered. If you try to pop up a message box, it will not be visible to the user, and the program might hang, waiting for the dialog box to be cleared.

Service Lifetime and Events

A service has a number of events to which it can respond, and these mirror the stages of a service's lifetime. For example, you've already seen the `OnStart` event handler. Here are the events to which the service can respond:

- **OnStart**—`OnStart` fires when the service starts. As you saw in the code in your service, this event writes to the log file the fact that the service is starting, and the date and time that the start occurred.
- **OnPause**—If the `CanPauseAndContinue` property is set to `True` for your service, the service can be paused from the Service Control Manager. If the service is paused, this event fires and can perform actions before the processing is actually paused.
- **OnContinue**—If the `CanPauseAndContinue` property is set to `True` for your service, this event fires when the service is continued after being paused.
- **OnStop**—When the service is stopped, this event is fired before the service stops. There is a `CanStop` property that when is set to `True` by default. If the property is set to `False`, the service does not receive the stop event (but it is still stopped).
- **OnShutdown**—If the when service is running and the machine is shut down, this event is fired. This is different from an `OnStop` in that this fires only when Windows is shutting down. There is a `CanShutdown` property that is set to `False` by default; this property must be set to `True` for your service to receive this event.

Your projects also have an `AutoLog` property, which by default is set to `True`. This allows your service to log certain events automatically. For example, if you run the service you created earlier, you can check the Event Viewer and, in the application log, you will see information messages about your service starting and stopping. These are created for you, but if you choose to turn them off, simply set `AutoLog` to `False`.

After you have installed the service and the Service Control Manager is handling the service, you can set the security context for the service. However, you can also set this inside the project. If you click on the `ServiceProcessorInstaller1` control, you'll see an `Account` property. By default, this is set to `User`. You can specify the `Username` and `Password` properties for the account under which you want the service to run.

Debugging Your Service

Your service cannot be debugged in the usual sense because it has to be installed into the Service Control Manager first. Because this is a Windows service, the IDE cannot start it for you; you need the Service Control Manager to start it for you. Therefore, to debug your Windows Services application, you'll have to build it, install it, and start it. This means that you actually are debugging a running application, which had to be installed before you started debugging. The IDE cannot start a service for you.

Place a breakpoint on one of the lines inside the `timer1_Tick` event handler. Next, choose `Processes` from the `Debug` menu. After the `Processes` dialog box is open, check the `Show system processes` box. In the list, your service will be listed as the name of the EXE, not the name of the service. Choose `LearningVBservice.exe` as shown in Figure 9.3. Click the `Attach` button, and you will see the screen shown in Figure 9.4. Choose `Common Language Runtime` and click the `OK` button. Now, click `Close` on the `Processes` dialog box.

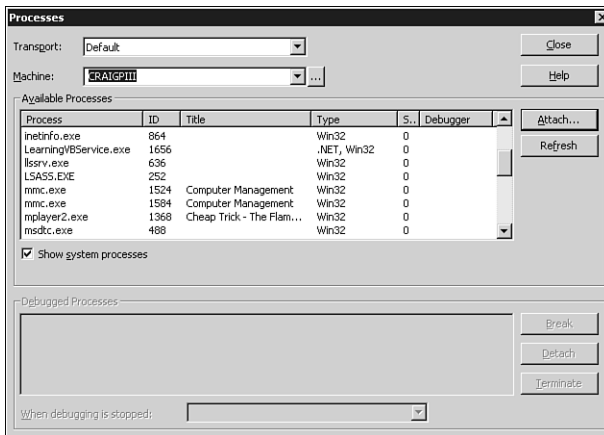


Figure 9.3

Attaching the Visual Studio.NET debugger to a running process.

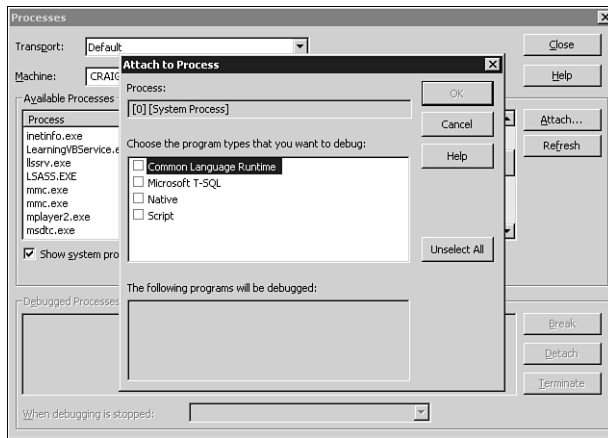


Figure 9.4

The form that allows you to choose what types of applications you want to debug.

The breakpoint in your program will be hit, and you can now interactively walk through the program as you would any other. Your running service is now in debug mode, so it is no longer running in the normal fashion; instead, it is responding only to your debugging commands. To stop the debugging process, just choose Stop Debugging from the Debug menu.

Before leaving this chapter, you might want to remove your service. To do this, go back to a command prompt, go to the directory holding the EXE for your service, and type the following line:

```
installutil /u LearningVBservice.exe
```

Summary

This chapter introduced you to Windows Services. This is a powerful class of applications that can be written in VB for the first time. The example in this chapter was rather simplistic; you might want the data logged to a database instead of a flat file, for example.

The programming model for services is quite simple, but there are a number of caveats when working with Windows Services. Debugging is one of the major differences between these services and most other projects, of course. In addition, don't forget to avoid any kind of UI.



CHAPTER 10

Upgrading VB6 Projects to VB.NET

So far, this book has focused on learning Visual Basic.NET, and how it differs from VB6. However, most readers have many projects written in VB6, and want to port those applications to VB.NET. Recoding every VB6 application you have is not something most people want to do, so it is helpful to see how VB.NET handles upgrading your applications.

Upgrading applications is a two-part process: First, you use the Visual Basic Upgrade Wizard (sometimes called the *Migration Wizard*) to convert your VB6 application to a VB.NET application. Second, you will probably need to make some modifications to complete the upgrade process.

You can improve the conversion and minimize the number of changes you need to make by making a few modifications to your VB6 code. For example, avoiding the use of late-bound variables in your VB6 code is helpful to the wizard when it comes time to migrate. There are other suggestions that will be examined later.

Upgrading Your First VB6 Application

To get a feel for what the Migration Wizard does for you and what modifications you'll need to make, you will create a new, simple VB6 application and then run it through the Migration Wizard.

Start VB6 and create a new Standard EXE project. Put a text box, a list box, and two buttons on the form. Using the property browser, change the Sorted property of the list box to True.

Double-click on the first button to open the code window. Enter the following code in the Command1_Click event procedure:

```
List1.AddItem Text1  
List1.ListIndex = List1.NewIndex
```

Press F5 to run the application, and notice that each time you click the command button, the contents of the text box are added to the list box. The list box displays the text in alphabetical order, and selects each line of text as it is added. Now, add a second form to your application. Add a label to the form and resize it to make it larger than normal. Double-click on the form and add the following code in the Form_Load event handler:

```
Label1 = "The text entered on Form1 is: " & Form1.Text1
```

Go back to Form1, and add the following code to the Command2_Click event procedure:

```
Form2.Show
```

You now have an application with two forms. One button adds the values entered in the text box to a list box on Form1. The second button opens the second form, which has a reference back to the first form. You can run the application to make sure that it works. Save the project as VB6upgrade.vbp and save the forms with any name you choose; I left them as Form1.frm and Form2.frm.

Save the application with the name VB6upgrade. Close VB6 and open VB.NET. Choose to open an existing project and open the VB6upgrade.vbp file you just created. Opening a VB6 project in VB.NET automatically starts the Visual Basic Upgrade Wizard.

The Visual Basic Upgrade Wizard

The Visual Basic Upgrade Wizard starts automatically because you opened a VB6 project in VB.NET. The first screen just displays some general information, so click the Next button to move into the wizard.

Step 2 of the wizard is shown in Figure 10.1. In most cases, you will leave the options at their default settings, but we will examine the options here for completeness. First, the wizard asks you what type of project you are upgrading. In this case, your only choice is EXE, and that is correct. (If you were upgrading an ActiveX EXE

server, you would have the choice to upgrade it to an EXE or DLL.) Below that is one check box; this option tells the wizard to generate default interfaces for any public classes you created. This is useful if the project is a DLL that exposes base classes that are implemented by other applications. Because you do not have any public classes, you can ignore this option. Leave the page as you see it in Figure 10.1 and click the Next button.

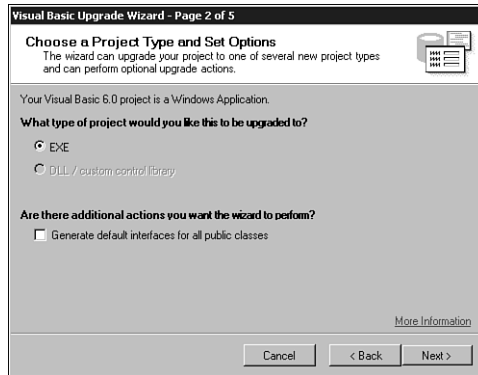


Figure 10.1

Step 2 of the Visual Basic Upgrade Wizard.

Step 3 of the wizard asks for the location of the resulting .NET project. By default, the wizard places the VB.NET project in a new directory inside the VB6 project directory called `<projectname>.NET`. If you want, you can change the path to another name and click the Next button.

Step 4 of the wizard begins the actual conversion, after which the upgraded project opens in Visual Basic.NET. The wizard always creates a new project; the old VB6 project is left unchanged.

Remember I said that you would have to make some modifications to your project after it is upgraded? Well, the Upgrade Wizard creates an upgrade report in HTML format that lists those modifications. This report shows up in the Solution Explorer, so you'll need to double-click on it to open it. Even though this was a very simple application, one error is reported, as you can see in Figure 10.2.

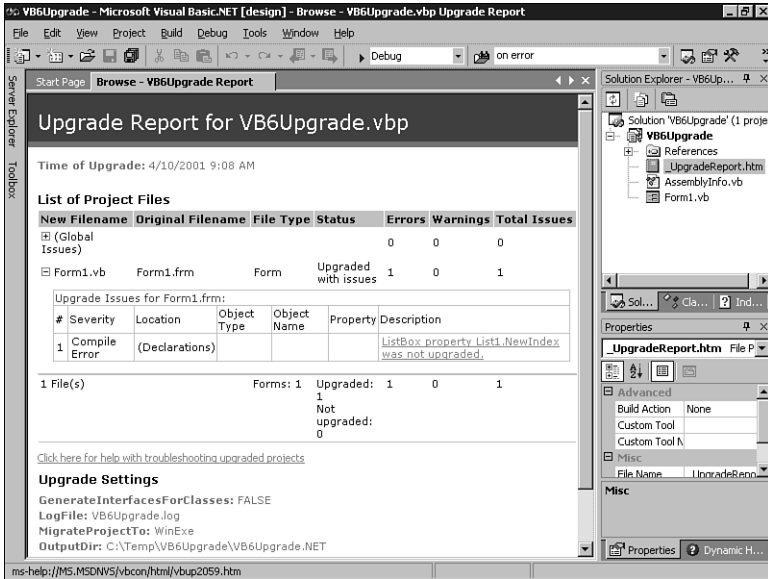


Figure 10.2

The upgrade report is created by the Upgrade Wizard as it upgrades your VB6 projects to VB.NET.

Examining the Upgraded Forms and Code

Now, double-click on `Form1.vb` to open it in the designer. Notice first that a `ToolTip1` control shows up in the component tray at the bottom of the form. By default, controls in VB.NET do not have a `ToolTip` property. Therefore, you must add a `ToolTip` control to a form for your controls on that form to have a `ToolTip` property.

Double-click on `Command1` to get to the code view. Notice that the line of code in the event handler has changed. The line now reads:

```
List1.Items.Add(Text1.Text)
```

First, you now have parentheses around the argument. You saw this in Chapter 2, “Your First VB.NET Application.” Also, in VB6, you referred to `Text1`, but in VB.NET, this has changed to `Text1.Text`. This should not be surprising because you have learned that VB.NET does not support default properties unless they accept parameters. You’ll also notice that the method has changed from `List1.AddItem` to `List1.Items.Add`.

Also notice that an upgrade issue was added that alerts you to a problem with the following line:

```
'UPGRADE_ISSUE: ListBox property List1.NewIndex was not upgraded.
Click for more: ms-help://MS.MSDNVS/vbcon/html/vbup2059.htm
List1.SelectedIndex = List1.NewIndex
```

If you click on the Click for more hyperlinks, you are taken to a help topic that explains the issue. In VB6, the `ListBox.NewIndex` property returned the index of the most recently added item. In VB.NET, the property doesn't exist; the index of the newly added item is returned from the `Items.Add` method.

To fix the problem, change the body of the event to this:

```
Dim NewIndex As Integer
NewIndex = List1.Items.Add(Text1.Text)
List1.SelectedIndex = NewIndex
```

Running this project reveals that all the code runs fine, and the project works as it did before. Not all projects will transition as smoothly, of course.

Modifications

Why do you need to make modifications yourself? Why can't the upgrade tool do the entire upgrade for you? The answer to this is twofold. In some cases (as in the preceding example), there is not an exact one-to-one correlation between the way code is written in VB6 and the equivalent in VB.NET. So, the best option for the upgrade tool is to alert you to the difference and tell you what you should change yourself. The second reason is that VB6 code that uses late binding is not fully interpreted until it is run. Because the upgrade tool examines the design-time representation of code, it can't perform default property resolutions or name changes for late-bound objects. Let's look at an example.

In VB6, the default property of a label control is `Caption`. In VB.NET, the default property is `Text`, and the property has to be fully qualified. The upgrade tool knows this, so the following VB6 code

```
Dim l As Label
Set l = Me.Label1
l = "Hello World"
```

upgrades perfectly to

```
Dim l As System.Windows.Forms.Label
l = Me.Label1
l.Text = "Hello World"
```

However, if you wrote the code using late binding, the upgrade tool could not perfectly upgrade the code because `o` is late-bound, so it cannot be resolved at design time:

```
Dim o As Object
Set o = Me.Label1
o = "Hello World"
```

upgrades to

```
Dim o As Object
Set o = Me.Label1
'UPGRADE_WARNING: Cannot resolve default property of object o
o = "Hello World"
```

In this case, you would need to resolve the default property yourself or change the code in VB6 and upgrade it again.

The upgrade tool alerts you to the changes you need to make by listing them in the upgrade report, and by putting to-do comments in code. There are four types of to-do comments:

- **UPGRADE_ISSUE**—Compile errors; these are things that must be fixed before the code compiles
- **UPGRADE_WARNING**—Differences in behavior; these are things that might cause a problem, and you certainly should look at before running your application
- **UPGRADE_TODO**—Code that was partially upgraded, but that you need to complete yourself
- **UPGRADE_NOTE**—Code that was significantly changed; you do not have to anything here, the message is purely informational

Differences in Form Code

If you examine the code in the `Command2_Click` event handler, you'll notice that the call has changed to this:

```
Form2.DefInstance.Show()
```

This points out one of the biggest differences between VB and VB.NET: Forms are not automatically created and ready for you to call. In other words, you couldn't use the following line in your code on `Form1`:

```
Form2.Show
```

Instead, forms are just another type of class, so you have to create the class first and then call a `Show` method. The Upgrade Wizard's approach to this is to create a new public property called `DefInstance`, and this property returns an instance of `Form2`. Therefore, you call this property, and then the `Show` method on the form that is exposed through the property. There is an alternative way to do this. Instead of the code generated for you by the wizard, you could have shown `Form2` using this code:

```
Dim frm2 as New Form2()  
frm2.Show()
```

As you add new forms to your project, the best way to show forms is to use code like the following:

```
Dim MyNewForm as New NewForm()  
MyNewForm.Show()
```

Another thing the tool changes: At the top of the form file, the following two lines are added:

```
Option Strict Off  
Option Explicit On
```

`Option Explicit` is turned on, even though you didn't explicitly turn it on in the VB6 project you just created. `Option Strict` is also turned off here, which means you can do some implicit conversions.

In the middle of your code is a collapsed block labeled `Windows Form Designer generated code`. If you expand this, you will see quite a bit of code that creates the form and the controls on the form, and sets up the size, location, and other properties of the form and controls. This code was always created for you in previous versions of VB, but it was not revealed to you. Now, you can see it and you can even modify it. Modifications, however, should be left to the designer, which means you go to the designer and make changes and the code is written for you.

An interesting effect of upgrading is that code is always upgraded to `Option Explicit On`—so if you have variables that are implicitly created, they will be explicitly created in the upgraded code.

The Visual Basic Compatibility Library

If you look in the Solution Explorer window and expand the References node, you'll see a reference to the `Microsoft.VisualBasic.Compatibility` library. This library is provided to make it easier to move from VB6 to VB.NET. In a normal VB.NET project, you would have to add a reference to it yourself. Any project that is upgraded, however, has the library added automatically.

The compatibility library contains classes and helper methods that allow the upgrade tool to upgrade your application more smoothly. For example, it includes `TwipsPerPixelX` and `TwipsToPixelsX` functions for working with screen resolutions, and control array extenders that give control array functionality to Windows forms.

As a rule of thumb, although it's fine to use the compatibility library functions, you'll find the new .NET Framework classes and methods offer richer and more powerful

functionality. Where possible, the upgrade tool upgrades applications to use these objects instead of those in the compatibility library.

The Upgrade Process

When upgrading your applications, there is a process you should consider following. It is

1. Learn VB.NET
2. Pick a small project and make sure that it works
3. Upgrade the project and examine the upgrade report
4. Fix any outstanding items in VB.NET

Learn VB.NET

One reason this section is at the end of the book is because you have to learn VB.NET before you can successfully upgrade a project. The reason for this is that the Upgrade Wizard can't do it all. There are too many programming styles and too many ambiguities in VB6 (especially with late-bound objects), and the Upgrade Wizard can't handle every situation. Therefore, you have to make sure that you can look at your upgraded code and not only fix any outstanding problems, but see where inefficiencies might lie and fix them also.

Learning VB.NET is as much learning the .NET Framework and what it offers as it is learning the new features of VB.NET. Make sure that you become familiar with the System namespaces and what they can do for you.

Pick a Small Project and Make Sure That It Works

In this chapter, you created a very small project and upgraded it. You'll want to start with your smaller projects, too. This way, you might just have one or two errors to fix and the task will not seem difficult. If you try to upgrade an application with 100 forms, 200 class modules, and API calls spread throughout, the task might be daunting if you see a large number of errors.

You should have both VB6 and VB.NET installed on the same machine on which you perform your upgrades. Why? You want to make sure that the project compiles and runs on that machine. For example, if you have dependencies on particular objects or controls and you do not have those installed on the machine performing the upgrades, there is no way for the Upgrade Wizard to verify the code, and your upgrade will fail. Therefore, make sure that the VB6 project compiles and runs on the machine performing the upgrade before you try to upgrade that project.

In addition, if you start with small projects, you might find the Upgrade Wizard repeatedly tripping over something you do often in your code. This would allow you to fix the repetitious problem in the VB6 project before you upgrade, allowing for a smoother conversion.

Upgrade the Project and Examine the Upgrade Report

Perform the upgrade on your project and then examine the upgrade report. The upgrade report shows you warnings and errors, and documents where they occurred. You can use this report to begin examining the code and making changes. In addition, the Upgrade Wizard adds comments into your code to identify any problems it had while converting, or to point out things you must do to ensure a smooth transition.

Fix Any Outstanding Items in VB.NET

Now that you have performed the upgrade and examined the upgrade report, it is time to fix any outstanding issues. This is when you have to use your knowledge of VB.NET to correct any problems that the Upgrade Wizard could not. You now see why the Upgrade Wizard is not a solution unto itself, but the first step in the process.

Helping Your VB6 Applications Upgrade

There are a number of things you can do in your VB6 application to make the upgrade easier. This is not a comprehensive list, but it covers some of the most important aspects.

Note that I refer to upgrading VB6 projects—what about VB3, VB4, and VB5 projects? Although the upgrade tool recognizes the format of VB5 and VB6 project types, some VB5 ActiveX references have problems after they are upgraded. For this reason, its better to upgrade your VB5 applications to VB6—and upgrade the controls to VB6 controls—before upgrading the application to VB.NET. The upgrade tool cannot upgrade projects saved in VB4 and earlier.

Do Not Use Late Binding

Late binding can present some problems because properties and methods cannot be verified during the upgrade process. One easy way to see this is to examine an object that is bound to a label control. Examine the following code:

```
Dim oLbl As Object
Set oLbl = Label1
oLbl.Caption = "Hello World"
```

This code works fine in VB6. However, the `Caption` property has been changed in VB.NET to a `Text` property. Because this is late-bound, the Upgrade Wizard would have no way to catch this. However, if you used early binding and specified `oLb1` to be of type `Label`, the Upgrade Wizard would see the object type and make the change for you.

Specify Default Properties

In the previous example, imagine if you had typed the last line of code this way:

```
oLb1="Hello World"
```

The Upgrade Wizard has no idea how to handle this. In fact, if you try to upgrade this code, the Upgrade Wizard will include a comment that says it cannot resolve the default property of `oLb1`.

You should specify the default properties on all your objects. If you don't, the Upgrade Wizard attempts to determine default properties wherever it can. However, default properties on late-bound objects are impossible for the wizard to resolve. You should avoid late binding and also specify any default properties.

Use Zero-Bound Arrays

In VB6, you could create an array with a lower boundary (`LBound`) of any number you wanted, using this syntax:

```
Dim MyArray(1 to 5) As Long
```

VB.NET does not support an `LBound` of anything but `0`, so your code here will end up looking like this in an upgraded project:

```
'UPGRADE_WARNING: LBound was changed from 1 to 0  
Dim MyArray( 5 ) As Integer
```

Examine API Calls

Most API calls work just fine from VB.NET. After all, the underlying operating system is still there and available. Some data types need to be changed because the `Long` data type in VB.NET is different from the `Long` data type in VB6. However, the Upgrade Wizard handles most of these for you.

There are some cases in which the Upgrade Wizard can't make the change, however. VB.NET doesn't natively support fixed-length strings, but some APIs expect a structure with a fixed-length string as one of the members. You can use the compatibility layer to get fixed-length strings, but you might have to add a `MarshalAs` attribute to the fixed-length string.

Also, any API calls that perform thread creation, Windows subclassing, or similar functions can cause errors in VB.NET. Most of these now have VB.NET equivalents, as you saw earlier when you created a multithreaded application.

Form and Control Changes

The OLE Container control is gone. If you have used it, you'll have to code a different way. The Line and Shape controls are gone as well, so lines, squares, and rectangles are replaced by labels. Ovals and circles cannot be upgraded by the wizard, so you'll have to recode those using the GDI+ library.

The Timer control must be disabled to stop it; the interval cannot be set to 0.

The Drag and Drop properties have changed so much from VB6 to VB.NET that they are simply not upgraded by the wizard. You'll have to rewrite that code.

These are not the only changes, of course, but they are some of the most common ones you will see as you upgrade your applications.

Summary

Microsoft has attempted to give you a good upgrade path for your Visual Basic 6 projects. The Upgrade Wizard is just one step, albeit an important one, in that process. You must first learn VB.NET and understand the underlying .NET Framework before you can hope to successfully port your applications to VB.NET.

Even before you move your applications, however, there are modifications you can make to your VB6 applications to ease the migration path. Keep these modifications in mind as you develop new VB6 applications between now and the final release of Visual Studio.NET.



APPENDIX A

The Common Language Specification

What Is the Common Language Specification?

The *Common Language Specification* (CLS) is a set of the language features directly supported by the Common Language Runtime. The runtime specifies the CLS to guarantee cross-language interoperability. By defining types and rules that all CLS-compliant code must use, you are guaranteed that a class written in C# can be inherited by VB.NET. You can rest assured that the data types of parameters are compatible across languages.

By using only types that are included in the CLS, your component will be accessible to clients written in any other language that supports the CLS. Basically, the CLS specifies a set of rules to which all languages must adhere in order to have full cross-language interoperability under the .NET Framework. Realize that only what is exposed by your component needs to be CLS-compliant. Inside a method, for example, you can use private variables that are not CLS-compliant, but the method can still be CLS-compliant because all the exposed items are part of the CLS.

The CLS specifies not just data types, but also a number of rules. Right now, there are 41 rules that specify the CLS. For example, one rule specifies that the type of an enum must be

one of the four CLS integer data types. Most of these rules are transparent to you as a developer; indeed, the Visual Studio.NET environment handles them for you.

VB.NET Data Types and the CLS

Rather than focus on the actual rules of the CLS, it is more useful to examine the underlying data types and how they map to VB.NET data types. Table A.1 shows some of the VB.NET data types and their equivalent CLS types.

Table A.1 *VB.NET and CLS Data Types*

Data Type	Range	Size	CLS Type
Byte	0 to 255 unsigned	1 byte	System.Byte
Short	−32,768 to 32,767	2 bytes	System.Int16
Integer	−2,147,438,648 to 2,147,438,647	4 bytes	System.Int32
Single	−3.402823E38 to −1.401298E45 for negative numbers 1.401298E−45 to 3.402823E38 for positive numbers	4 bytes	System.Single
Long	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	8 bytes	System.Int64
Double	−1.79769313486231E308 to −4.94065645841247E−324 for negative numbers 4.94065645841247E−324 to 1.79769313486232E308 for positive numbers	8 bytes	System.Double
Decimal	+/− 79,228,162,514,264,337,593,543,950,335 for whole numbers +/−7.9228162514264337593543950335 with 28 places to the right of the decimal	12 bytes	System.Decimal
Boolean	True or False	4 bytes	System.Boolean
Char	0 to 65535 unsigned	2 bytes	System.Char

Table A.1 *VB.NET and CLS Data Types*

Data Type	Range	Size	CLS Type
String	0 to 2 billion characters (Unicode)	10 bytes + 2 bytes per character	System.String (class)
Date	January 1, 1 CE to December 31, 9999	8 bytes	System.DateTime
Object	Any type	4 bytes	System.Object (class)
User-defined type	Determined by the members of the UDT	Determined by the sizes of the members	Inherited from System. Valuetype

This table covers only the primitive types. There are other rules as well:

- Arrays must be of a CLS type and have an LBound of 0.
- Exceptions can be thrown or caught, and they must inherit from System.Exception.
- Enumerations must inherit from System.Enum and must be of type Byte, Int16, Int32, or Int64.



INDEX

SYMBOLS

& (ampersand) symbol, 41
() (parentheses) characters, 50
> symbol, 120
>> symbol, 120
++ operator, 52
< symbol, 120
<< symbol, 120

A

abstract classes, 13
accessing
 databases, 105-110
 Web services, 163
Active Server Pages (ASP), 133
Add Reference dialog boxes, 78
Add Web Reference dialog box, 159
AddHandler statement, 83
adding Windows forms to projects, 116
ADO
 Connection objects, 110
 disconnected recordsets, 115
 XML, 123
ADO.NET, 122
 data, 115
 database accessing, 105-110
 DataSets, 122, 126
 creating, 126-127
 XML, 123

- disconnected architecture, 122
- objects, 123
 - DataReader, 125
 - DataRow, 127
 - OleDbCommand, 124
 - oleDbConnection, 124
 - SQLCommand object, 124
 - SQLConnection, 124
- overview, 122

ampersand (&) symbol, 41**Anchor property, 39****anchoring controls, 38-39****And operator, 51****API calls, 184****APIs (operating system specific), 9****AppBase, 89****<AppDomain> tag, 89****applications**

- ASP, 133
- ASP.NET, 134
- controls
 - anchoring, 38-39
 - sizing, 36-38
- database access, 105-110
- debugging, 5
- forms
 - opacity, 45
 - tab order, 42-44
- free-threaded, 70-72
- Hello World example, 34
- n-tier, 73
- .NET framework, 4
- reusing assemblies, 87
- services, 165
- VB.NET, 31, 34
- Web, 4
- Web services
 - accessing, 163
 - clients, creating, 156-161
 - deploying, 163
 - how they work, 162
 - testing, 155-156

architecture (ADO.NET), 122**arguments, optional, 54****arrays**

- changes between VB and VB.NET
 - assignment, 58
 - lower boundaries, 58
 - size, 57
- CLS, 189
- zero-bound, 184

ASP (Active Server Pages), 133**<asp:Calendar> tag, 142****ASP.NET**

- accessing Web services, 163
- advantages over ASP, 133
- applications, 134
 - data binding, 149-150
 - Design view, 135
 - functionality, 137
 - layout modes, 135
 - re-entrant pages, 151
 - rendering in browser, 136
 - requirements, 134
 - server controls, 138-141
 - validation controls, 142-148
- automatic state management, 137
- building pages, 5
- buttons, 139
- click events, 138
- code-behind pages, 135
- compiling pages, 138
- server controls
 - Calendar, 142
 - re-entrant page model, 151

.aspx extension, 135**assemblies, 10, 73**

- compiling, 86
- GAC, 11, 89
- locating, 88-89
- manifests, 10
- probing, 89
- reusing in other applications, 87
- sample Healthcare project, 84

assigning arrays, 58**assignment operators, 52****automatic state management (ASP.NET), 137****AutoSize option, 36**

B

base classes, 91

- inheriting from, 32
- polymorphism, 99-100

binding data, 116-119, 149-150

BindingManager code, 121

block-level scope, 53

Boolean operators, 51

borders, 38

bound controls, 112

building

- object models, 102
- Windows Services, 165-166, 168
 - adding installers to services, 168
 - configuring, 169
 - debugging, 172
 - installing, 169
 - lifetime and events, 171
 - PerformanceCounter control, 167
 - Timer component, 166

buttons

- ASP.NET and HTML, 139
- HTML, 141
- tab order, 43

ByRef keyword

- array assignment, 58
- modifying properties, 56

ByVal keyword

- changes between VB and VB.NET, 53
- ParamArray variables, 56

C

C#, 7

C++.NET, 4

caches (CLR), 11

Calendar ASP.NET server control, 142

calling

- operating system specific APIs, 9
- Web services, 163

CAS (Code Access Security), 18

Catch statements, 62

cboCountry combo box, 118

class libraries, 73

- classes
 - adding properties, 75
 - testing, 76
- creating, 74

Class Library projects, 24

Class View tab (Server Explorer), 26

Class View window, 27

classes, 12-13

- abstract, 13
- adding properties, 75
- base, polymorphism, 99-100
- changing names, 74
- class modules, 92
- compiling assemblies, 86
- constructors, 81
- default properties, 80
- derived, 69, 91
- events, 82-84
- fields, 13
- implementation inheritance, 93
- inheritance, 5
- methods, 82
- overriding, 69
- page, 138
- parameterized properties, 79
- Read-only/Write-only properties, 79
- reusing assemblies, 87
- shared members, 95-96
- testing, 76
- without constructors, 81
- XML, 128

Clear method, 118

click events (ASP.NET), 138

client-side data validation, 142

clients

- validation, 143
- Web services, 156-161

CLR (Common Language Runtime), 6-8

- caches, 11
- Common Type System, 12
- compiling code for, 7
- components, declaring dependencies, 11
- garbage collection, 7
- language interoperability, 17

- language-neutrality, 8
- metadata, 16
- security, 18

CLS (Common Language Specification), 17, 187

- arrays, 189
- CLS-compliant components, 17
- enumerations, 189
- exceptions, 189

.CLS extension, 74**code**

- ASP.NET, 137
- assemblies, 10
- BindingManager, 121
- blocks
- scope, 53
- code-behind page, 135
- collapsing/expanding, 33
- compiling (JIT), 9
- executing, 9
- inheritance, 92
- managed, 7-8
 - executing, 10
- IL, 9
- JIT compiler, 9
- portable executable, 9
- spaghetti, 133
- upgrading VB6 projects to VB.NET, 178
- Web pages, 138

Code Access Security (CAS), 18**code-behind page, 135****CodeBase values, 89****collapsing blocks of code, 33****COM**

- component requests, 11
- DLL hell, 11
- interoperability, 16

COM components, 73**Common Language Runtime. *See* CLR****Common Language Specification. *See* CLS****Common Type System, 12****CompareValidator control, 144****compatibility (DLLs), 11****compilers**

- exposing runtime, 8
- JIT, 9

compiling

- assemblies, 86
- class libraries, 73
- for the CLR, 7
- methods (JIT), 9
- pages in ASP.NET, 138

Component Services, 4**components**

- calling over HTTP, 4
- CLS-compliant, 17
- COM, 73
- Common Language Specification, 8
- compatibility, 187
- declaring dependencies, 11
- discoverable, 153
- multiple applications, 12
- .NET assemblies, 73
- OleDbDataAdapter controls, 117
- self-describing, 15
- side-by-side instancing/execution, 11
- VB, security, 18

configuring

- services, 169
- VS.NET IDE, 21

connected architectures, 122**connections**

- data sources, 124
- database, 105, 110, 124

Consol Application projects, 25**constructors, 81**

- changes between VB and VB.NET, 66
- classes without, 81

Container control, 185**controls**

- anchoring, 38-39
- bound, 112
- data binding, 149
- DataGrid, 149
- line, 44
- shape, 44
- sizing automatically, 36-38
- tying to DataSets, 118
- validators, 144-145

converting numbers to strings, 59**Create New Project link (VS.NET Start Page), 23**

creating object models, 102
 cross-language operability, 16
 currency converter, 157
 Currency data type, 60
 CustomValidator control, 144

D

data

- binding, 116-119, 149-150
- client-side validation, 142
- disconnected, 122
- displaying
 - with DataForm Wizard, 114
 - unbound controls, 119

data-driven VB.NET forms, 114

Data Link Properties dialog box (DataAdapter Configuration Wizard), 107

data types

- changes between VB and VB.NET, 59
 - Currency data type, 60
 - fixed-length strings, 60
 - Short, Integer and Long, 59
 - string/number conversions, 59
 - True value, 60
 - variables, 59
 - Variant data type, 61
- VB.NET and CLS, 188

DataAdapter Configuration Wizard, 107

databases

- accessing, 105-107, 110
- DataSets, 123
- making connections, 110
- Northwind, 107
- queries, parameterized, 117

DataForm Wizard, 112

- binding data without, 116-119
- displaying records, 113

DataGrid controls, 115, 149

DataReader objects, 125

DataRow objects, 127

DataSetAdapter objects, 126-127

DataSets

- adding to forms, 118
- ADO.NET, 122, 126
- binding fields manually, 116
- changing data, 115
- clearing, 118
- creating, 126-127
- loading, 111
- selecting in DataForm Wizard, 112
- XML, 123
- XML Document Object Model, 128

debugging, 5, 172

declarations

- changes between VB and VB.NET, 52
- delegates, 14
- variables, multiple, 52

default properties, 49-50, 80

DefInstance property (Visual Basic Upgrade Wizard), 180

delegates, 12, 14

deploying Web services, 163

derived classes, 69, 91

designer

- upgrading VB6 projects to VB.NET, 178
- Window Services projects, 166

destroying objects, 72

destructors, 66

DHTML (Dynamic HTML), 141

DISCO file, 163

disconnected architectures, 122

disconnected recordsets, 115

discoverable components, 153

displaying data in unbound controls, 119

Dispose subclass, 32

DLL hell, 11

.DLL extension, 73

documents (XML), 128

DTDs, 128

dynamic assemblies, 10

Dynamic Help feature, 28

Dynamic HTML (DHTML), 141

Dynamic value, 146

E

elements, converting to HTML server controls, 141
encapsulation, 102
End While loop, 54
Enroll method, 95
enumerations, 14, 189
Err.Description, 63
errors, 17. *See also* exceptions
 changes in handling between VB and VB.NET, 62-63
 User-defined type not defined, 78
events
 adding to classes, 82
 delegates, 14
 handling in the clients, 83-84
 Windows services, 171
exceptions, 17, 189
executables, 9
executing (side-by-side), 11
Exit statements, 127
expanding blocks of code, 33

F

fields, 13, 146
fixed-length strings, 60, 184
FlowLayout mode, 135
forcing inheritance, 96
Form Designer, 25, 33
form designer tab, 31
Form1.vb tab, 39
forms
 adding DataSets, 118
 controls
 anchoring, 38-39
 sizing, 36-38
 data-driven, 114
 dragging elements onto, 138
 FlowLayout mode, 135
 GridLayout mode, 135
 Inherits statements, 32
 opacity, 45

 tab order, 42-44
 unbound controls, 121
 upgrading VB6 projects to VB.NET, 180
 VS.NET functionality, 36

free threading, 70-72

functions, 50

G

GAC (Global Assembly Cache), 11-12, 89
garbage collection, 7, 72
Generate DataSet option (oleDBDataAdapter1 control options), 110
Get Started section (VS.NET Start Page), 21
GridLayout mode, 135

H

Headlines section (VS.NET Start Page), 22
Hello World example, 34
Help (VS.NET), 23
HTML (Hypertext Markup Language)
 buttons, 139
 comparing to ASP.NET server controls, 139
 dragging elements into forms, 138
 sending data to servers, 138
 server controls, 140-141
 tags, 142
HTTP (Hypertext Transfer Protocol)
 calling components over, 4
 XML, 123

I

IDEs

 Dynamic Help feature, 28
 VB.NET, 21, 25
 changes between VB and VB.NET, 66
 Server Explorer, 29

- Solution Explorer, 25
- Toolbox feature, 30
- VS.NET, 21
- IDL (Interface Definition Language), 9**
- IE (Internet Explorer), 144**
- IIS (Internet Information Server), 134**
- IL (intermediate language), 9**
- implementation, 69**
- implementation inheritance, 5, 91-95**
- importing namespaces, 15**
- Include Update Method box, 113**
- inheritance, 91, 103**
 - changes between VB and VB.NET, 69
 - CLS-compliant code, 187
 - forcing/preventing, 96
 - implementation, 5, 91-95
 - interface, 92-93
 - keywords, 96
 - namespaces, 15
 - overriding properties and methods, 97-99
 - polymorphism, 99-100
 - Visual Basic, 5
 - when to use, 102
- Inherits statements, 32, 95**
- InitializeComponent routine, 34**
- initializing variables, 52**
- <INPUT> tag, 141**
- installing Windows services, 169**
- InstallUtil.exe, 169**
- instances, 11**
- instantiating objects, 78**
- Integer data type, 59**
- Interface Definition Language (IDL), 9**
- interfaces, 12-13**
 - creating, 92
 - implementation, 102
 - inheritance, 5, 91-93
 - polymorphism, 101
- intermediate language (IL), 9**
- interoperability (CLS), 17**

J - K - L

JIT (Just-In-Time) compiler, 5, 9

keywords, 96-99

labels (Web services), 156

languages

- C#, 7
- compatibility, 187
- cross-language interoperability, 16
- language-neutral environments, 8
- unsupported data types/structures, 15

late binding, upgrading projects to VB.NET, 183

layout (Web Application projects), 135

LearningVB project, 25-26

LearningVbservice.exe, 172

libraries

assemblies

GAC, 89

locating, 88-89

classes, 73

adding properties, 75

compiling assemblies, 86

constructors, 81

default properties, 80

event handling, 83-84

events, 82

methods, 82

parameterized properties, 79

Read-only/Write-only properties, 79

reusing assemblies, 87

testing, 76

creating, 74

line controls, 44

listings

VB.NET sample program, 46

Windows Form Designer, code generated by, 33

ListItem Collection Editor, 158

loading DataSet, 111

Long data type, 59

M

MainMenu control (Toolbox feature), 41

managed code, 7-8

- executing, 10
- IL, 9
- JIT compiler, 9
- metadata, 8

manifests, 10

masks, 144

Menu Editor, 41

menus

- creating, 41-42
- Menu Editor, 41

metadata, 7

- managed code, 8
- use of, 15

methods

- adding to classes, 82
- callable over the Web, 154
- compiling (JIT), 9
- Object Browser, 27
- overriding, 97-99
- shared, 96

Microsoft Intermediate Language (MSIL), 4, 9

Microsoft news server, 22

Migration Wizard. *See* Visual Basic Upgrade Wizard

modifying

- code, 179
- properties (ByRef keyword), 56

modules (class), 74

MSIL (Microsoft Intermediate Language), 4, 9

Multiline property (TextBox control), 38

MustInherit keyword, 96

MustOverride keyword, 97

My Profile section (VS.NET Start Page), 22

N

n-tier applications, 73

n-tier model, 4

namespaces

- changes between VB and VB.NET, 67-68
- changes with VB.NET, 14
- importing, 15
- LearningVB project, 26
- nesting, 68

nesting namespaces, 68

.NET

- assemblies, 73
- compiling code, 9
- database access, 110
- debugging tools, 5
- discovery mechanism, 4
- framework, 4-5
- GAC (Global Assembly Cache), 11
- justification, 3
- locating assemblies, 88-89
- security, 18
- value types, 14
- XML, 128

.NET Framework

- assemblies, 11
- class library, 14
- CLR, 6-7
- JIT, 9
- metadata, 15
- namespaces, 15
- principals, 18
- role-based security, 18

New keyword, 78

New Project dialog box (VS.NET), 24

Northwind database, 107

Not operator, 51

Nothing value, 72

NotInheritable keyword, 96

NotOverridable keyword, 97

O

Object Browser, 27

object models, building, 102

objects

ADO Command, 110

ADO Connection, 110

ADO.NET, 105, 122-123

 DataReader, 125

 DataRow, 127

 OleDbCommand, 124

 oleDbConnection, 124

 SQLCommand object, 124

 SQLConnection, 124

creating, 78

DataSets, 126

encapsulation, 102

instantiate, 78

namespaces, 67

Nothing value, 72

XML, 128

OLE DB, 124

OleDbCommand objects, 124

oleDBConnection control, 149

oleDbConnection object, 124

OleDbDataAdapter controls, 107, 117, 149

OnContinue event (Windows services), 171

Online Community section (VS.NET Start Page), 22

OnPause event (Windows services), 171

OnShutdown event (Windows services), 171

OnStart event (Windows services), 171

OnStop event (Windows services), 171

opacity (forms), 45

operating system specific APIs, 9

operators

 ++, 52

 assignment, 52

 Boolean, 51

Option Strict statements, 58

optional arguments, 54

Optional Keyword, 54

Or operator, 51

overloading, 69

Overridable keyword, 97

overriding

 classes, 69

 properties and methods, 97-99

P

page classes, 138

pages

 re-entrant, 151

 Web

 code, 138

 validation controls, 143

ParamArray variables, 56

parameterized properties, 79

parameterized queries, 117

parameters

 default properties, 50

 passing by reference, 53

parentheses, 50

passing by reference, 53

PE (portable/physical executables), 9-10

PerformanceCounter control, 167

polymorphism

 base classes, 99-100

 interfaces, 101

portable executables (PE), 9-10

preventing inheritance, 96

principals, 18

probing assemblies, 89

procedures, 54

programming,

 cross-language interoperability, 16

 language-neutral environments, 8

projects

 adding Windows forms, 116

 ASP.NET, 134

 controls, unbound, 119

 creating VB.NET applications, 31, 34

 LearningVB, 25

 saving, 25

 services. *See* services

 testing, 77

- upgrading from VB6 to VB.NET, 175-176
 - API calls, 184
 - default properties, 184
 - form and control changes, 185
 - form code differences, 180
 - forms and code, 178
 - late binding, 184
 - modifying code, 179
 - process of, 182
 - tips, 183
 - to-do comments, 180
 - upgrade report, 177
 - Visual Basic Compatibility Library, 181
 - Visual Basic Upgrade Wizard, 177

- VB, 24
- VS.NET, 23
- Web Application, 161
- Web Service, 24, 154
- Web services

properties

- adding to classes, 75
- changes between VB and VB.NET, 56
- default, 80
- overriding, 97-99
- parameterized, 79

Properties window, 27-28

Q - R

queries, 117

RangeValidator control, 144

re-entrant pages, 151

Read-only properties, 79

recordsets

- ADO, 123
- disconnected, 115

referencing namespaces, 15

RegularExpressionValidator control, 144-145

rendering ASP.NET pages in browsers, 136

RequiredFieldValidator control, 143-145

resizing controls, 36-38

Return statements, 55

role-based security, 18

runtime

- metadata, 16
- security, 18

S

Save method, specifying XML format, 123

saving project information, 25

schemas (XML), 128-129

Search Online section (VS.NET Start Page), 22

searching (Server Explorer), 29

Security

- CAS (Code Access Security), 18
- role-based, 18
- VB components, 18

SelectedIndexChanged event, 118

self-describing components, 15

server controls

- ASP.NET, 138-141
 - Calendar, 142
 - re-entrant pages, 151
 - TextBox, 146
- HTML, 140-141

Server Explorer, 26, 29

Service Control Manager, 169

services

- support, 165
- use of, 166
- Window Services projects
 - adding installers, 168
 - configuring, 169
 - creating, 166-168
 - debugging, 172
 - installing, 169
 - lifetime and events, 171
 - overview, 170

Services applet, 169

Set keyword, 50

setting tab order, 42

shape controls, 44

shared members (classes), 95-96

Short data type, 59

ShowShortcut property, 42

side-by-side instancing/execution, 11

Simple Object Access Protocol (SOAP), 4, 163

simple types, 129

size

arrays, 57

controls, 36, 38

SOAP (Simple Object Access Protocol), 4, 163

Solution Explorer, 25, 77, 135

creating Web services, 154

database access, 111

Startup Object box, 115

Solutions Configuration option, 86

spaghetti code, 133

SQL Server's Query Analyzer, 115

SQL statements, 109

SQLCommand objects, 124

SQLConnection object, 124

SqlDataAdapter control, 107

Start Page (VS.NET), 21-23

Startup Object box (Solution Explorer), 115

static assemblies, 10

Static keyword, 55

stored procedures, 109

storing

disconnected data, 122

value types, 14

String Collection Editor, 116

strings

converting from numbers, 59

fixed-length (VB.NET), 60

structures, 64-65

Sub Destruct, 66, 72

Sub Main procedures, 166

subs, 50

synchronization (free-threaded applications), 72

System namespace, 14

System.IO.CreateDirectory() method, 96

System.Threading.Thread class, 71

T

tab order (forms), 42, 44

tags (HTML), 142

testing

classes, 76

Web services, 155-156

text

converting to HTML server controls, 141

XML, 123

TextBox control, 38

TextBox server controls, 146

Timer components, 166

timer1_Tick event procedure, 167

to-do comments, 180

Toolbox feature, 30

HTML tab, 138

MainMenu control, 41

Web Forms tab

RequiredFieldValidator control, 143

validators, 144

threads, 71

Type Here boxes, 41

type libraries (VB), 15

types (Common Type System), 12

U

UDTs, 64-65

UpdateDataSource method, 113

upgrade reports, 180

UPGRADE_ISSUE comment, 180

UPGRADE_NOTE comment, 180

UPGRADE_TODO comment, 180

UPGRADE_WARNING comment, 180

upgrading VB6 projects to VB.NET, 175-176

API calls, 184

default properties, 184

form and control changes, 185

forms and code, 178-180

late binding, 184

- modifying code, 179
- process of, 182
- tips, 183
- to-do comments, 180
- upgrade report, 177
- Visual Basic Compatibility Library, 181
- Visual Basic Upgrade Wizard, 177

UsageMonitor service, 169

User-defined type not defined error message, 78

V

validating XML documents, 128

validation controls

- ASP.NET, 142-144
- applying multiple, 145
 - modifying, 145
 - types, 144
- validation summaries, 146-148

ValidationSummary control, 145

validators

- applying multiple to the same field, 145
- types of, 144

values

- passing by reference, 53
- types, 12-14

variables

- block-level scope, 53
- declaring multiple, 52
- initializing, 52

Variant data type, 61

VB.NET

- abstract classes, 13
- applications, 31, 34
- assemblies, 10
- changes from VB
 - arrays, 57-58
 - block-level scope, 53
 - Boolean operators, 51
 - ByVal keyword, 53
 - constructors/destructors, 66
 - data types, 59-61
 - declarations, 52

- default properties, 49-50
- error handling, 62-63
- free threading, 70-72
- garbage collection, 72
- IDE, 66
- inheritance, 69
- namespaces, 67-68
- new assignment operators, 52
- Option Strict statements, 58
- optional arguments, 54
- overloading, 69
- ParamArray variables, 56
- properties, 56
- Return statements, 55
- Static keyword, 55
- structures/UDTs, 64-65
- subs and functions, 50
- While loops, 54

CLS, 188

code, 34

components, 73

data-driven forms, 114

database access, 105-110

DataGrid control, 115

fixed-length strings, 184

IDE, 21, 25

overriding methods, 97

Server Explorer, 29

Solution Explorer, 25

Toolbox feature, 30

implementation inheritance, 93-95

interfaces, 13

justification of, 3-5

learning, 182

managed code, 8

executing, 10

IL, 9

JIT compiler, 9

menus, 41-42

migrating to, 3

n-tier model, 4

.NET Framework, 6

polymorphism, 101

projects, creating, 23

shared members (classes), 95-96

- upgrading VB6 projects, 175-176
 - API calls, 184
 - default properties, 184
 - form and control changes, 185
 - forms and code, 178-180
 - late binding, 184
 - modifying code, 179
 - process of, 182
 - tips, 183
 - to-do comments, 180
 - upgrade report, 177
 - Visual Basic Compatibility Library, 181
 - Visual Basic Upgrade Wizard, 177
- versions, 6
- Web services, 153, 164
 - accessing, 163
 - clients, creating, 156-161
 - creating, 154
 - deploying, 163
 - how they work, 162
 - testing, 155-156
- Windows Services, 165
 - adding installers, 168
 - configuring, 169
 - creating projects, 166-168
 - debugging, 172
 - installing, 169
 - lifetime and events, 171
 - overview, 170
 - PerformanceCounter control, 167
 - Timer component, 166
 - use of, 166
- VB.NET.namespaces, 14**
- VB6 (Visual Basic 6), 13**
 - classes, 13
 - Component Services, 4
 - components
 - security, 18
 - type libraries, 15
 - inheritance, 5
 - interfaces, 13, 92-93
 - migrating to VB.NET, 3
 - New keyword, 78
 - polymorphism, 100
 - project types, 24
 - upgrading projects to VB.NET, 175-176
 - API calls, 184
 - default properties, 184
 - form and control changes, 185
 - forms and code, 178-180
 - late binding, 184
 - modifying code, 179
 - process of, 182
 - tips, 183
 - to-do comments, 180
 - upgrade report, 177
 - Visual Basic Compatibility Library, 181
 - Visual Basic Upgrade Wizard, 177
 - versions, 6
- versions (VB/VB.NET), 6**
- Visual Basic 6. *See* VB6**
- Visual Basic Upgrade Wizard, 175-176**
 - DefInstance property, 180
 - Option Explicit, 181
 - upgrade report, 177
- Visual InterDev, 5**
- Visual Studio.NET. *See* VS.NET**
- Visual Studio.NET debugger, 173**
- VS.NET (Visual Studio.NET), 21**
 - class libraries
 - adding events to classes, 82
 - adding methods to classes, 82
 - adding properties to classes, 75
 - compiling assemblies, 86
 - constructors, 81
 - creating, 74
 - default properties, 80
 - event handling, 83-84
 - parameterized properties, 79
 - Read-only/Write-only properties, 79
 - reusing assemblies, 87
 - testing classes, 76
 - without constructors, 81
 - Dynamic Help feature, 28
 - Form Designer, 25
 - IDE, configuring, 21
 - Properties window, 27

- Start Page, 21
- Toolbox, HTML tab, 138
- Web Application projects, 134
- Web services, testing, 156
- XSD files, 129
- Windows forms, 36

VSDISCO file, 163

W

Web Application projects, 4, 24, 134, 161

- data binding, 149-150
- Design view, 135
- functionality, 137
- layout modes, 135
- re-entrant pages, 151
- rendering in browser, 136
- requirements, 134
- server controls, 138-139, 141
- validation controls, 142-144
 - applying multiple, 145
 - modifying, 145
 - types, 144
 - validation summaries, 146-148

Web Control Library projects, 25

Web Forms tab, 143

Web pages

- code, 138
- validation controls, 143

Web servers, 134

Web Service projects, 24

Web services, 153

- accessing, 163
- clients, creating, 156-161
- creating, 154
- deploying, 163
- DISCO/VDISCO files, 163
- how they work, 162
- testing, 155-156

Web sites, Microsoft's news server, 22

<WebMethod(> element, 154

What's New section (VS.NET Start Page), 22

While loops, 54

Windows forms, adding to projects, 116

windows (Solution Explorer), 25

Windows 2000, 134

Windows Application projects, 24-25

Windows applications, database access, 106-110

Windows Control Library projects, 24

Windows Form Designer, 33

Windows Forms, .NET Framework, 6

Windows Service projects, 25

Windows Services, 165-166

- configuring, 169
- creating projects, 166-168
 - adding installers to services, 168
 - PerformanceCounter control, 167
 - Timer component, 166
- debugging, 172
- installing, 169
- lifetime and events, 171
- overview, 170
- Sub Main procedures, 166

WithEvents keyword, 83

wizards (DataForm), 112

Write-only properties, 79

writing CLS-compliant components, 17

X - Y - Z

XML

- ADO, 123
- classes, 128
- DataSets, 123
- integration, 128
- schemas, 128-129
- simple types, 129
- validating against an XML schema, 128

XML Designer, 129-131

XmlDataDocument object, 128

XSC, 128

XSD schema, 129

zero-bound arrays, 184

Coming Soon from Sams Publishing

ASP.NET Tips, Tutorials, and Code

Author: Scott Mitchell, et al.

ISBN: 0672321432

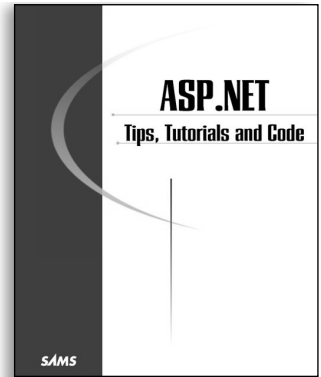
Price: \$49.99

Available: Summer 2001

With a foreword by Microsoft's Mark Anders, *ASP.NET Tips, Tutorials, and Code* consists of 19 chapters written by seven of today's leading experts in ASP.NET. These authors are professional developers who create ASP.NET applications, teach, and run well-known ASP.NET Web sites, either within or outside Microsoft. The tutorial framework for each chapter includes:

- A brief introduction, explaining the concept
- A code example, illustrating the concept
- A piece-by-piece explanation of the code

Most examples employ VB.NET, but there are also additional C# examples within each chapter, and all of the example programs will be available at the book's Web site in both VB.NET and C#. The code examples in this book are based upon the ASP.NET Beta2 specifications, a functionally complete version of the software.



Programming Data-Driven Web Applications with ASP.NET

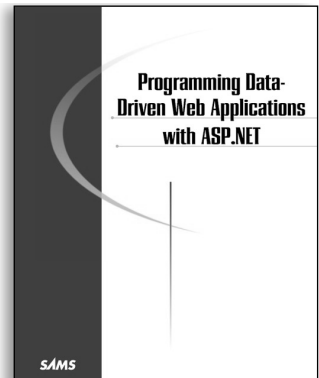
Author: Don Wolthuis and Doug Seven

ISBN: 0672321068

Price: \$39.99

Available: Summer 2001

Programming Data-Driven Web Applications with ASP.NET provides readers with a solid understanding of ASP.NET and how to effectively integrate databases with their Web sites. The key to making information instantly available on the Web is integrating the Web site and the database to work as one piece. The authors teach this using ASP.NET, server-side controls, ADO+, XML, and SOAP. Readers learn how to manage data by using ASP.NET forms, exposing data through ASP+ Web Services, working with BLOBs, and using cookies and other features to secure their data.



C# and the .NET Framework

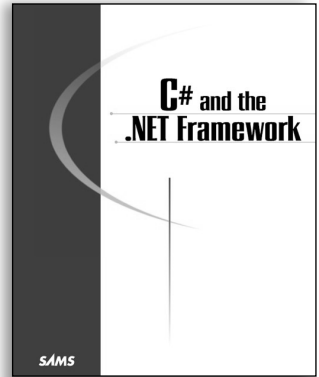
Author: Richard Weeks and Robert Powell

ISBN: 067232153X

Price: \$39.99

Available: Summer 2001

This book covers topics ranging from the general principles of .NET through the C# language and how it is used in ASP.NET and Windows Forms. Written by programmers for programmers, the content of the book is intended to get readers over the hump of the .NET learning curve and provide solid practical knowledge that will make developers productive from day one.



Building e-Commerce Sites in the .NET Framework

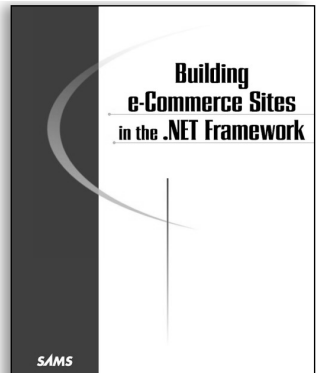
Author: Jason Bentrup

ISBN: 0672321696

Price: \$39.99

Available: Summer 2001

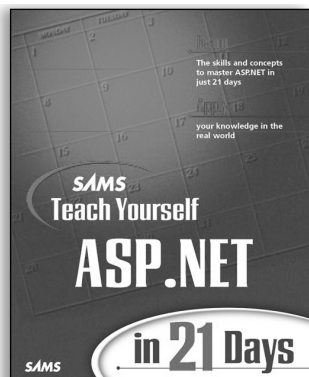
Building e-Commerce Sites in the .NET Framework describes the steps a developer will take to plan, develop and deploy an actual robust, scalable e-commerce application using the Microsoft Visual Studio .NET. There are detailed descriptions of design choices a developer makes and implementation details. The author's first-hand experience will save the reader time and effort. The development of a working, modern e-commerce site is provided in a case study approach along with clear and simple explanations, screenshots, and step-by-step code excerpts.



Sams Teach Yourself ASP.NET in 21 Days

Author: Chris Payne
ISBN: 0672321688
Price: \$39.99
Available: Summer 2001

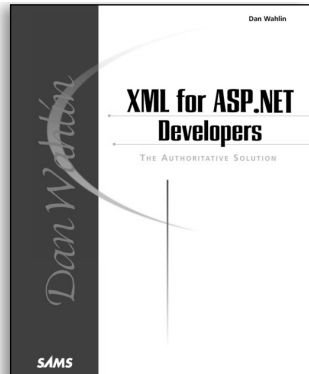
Sams Teach Yourself ASP.NET in 21 Days is the perfect book to introduce beginner and intermediate readers to the new technologies and frameworks presented by ASP.NET. By guiding readers through short but increasingly complex lessons, this book gives readers a strong foundation in ASP.NET, and the knowledge to develop their own creative solutions. Readers delve into the new framework, the C# and Visual Basic programming languages, and techniques to approach difficult problems.



XML for ASP.NET Developers

Author: Dan Wahlin
ISBN: 0672320398
Price: \$39.99
Available: Summer 2001

XML for ASP.NET Developers provides developers with detailed coverage of Microsoft XML technologies and their practical applications in developing .NET Web applications. XML expert Dan Wahlin first provides readers with a solid foundation in the basics of MSXML including XML Syntax, XML Schemas, Xpath, Xlink, Xpointer, and other concepts necessary to leverage the power of XML. After the building blocks of XML are thoroughly covered, Dan guides readers through manipulating XML documents using the Document Object Model (DOM) and XSL (Extensible Stylesheet Language) on both the client and the server. Detailed examples combined with easy-to-follow tutorials will have readers transforming XML documents into professional-looking applications quickly and easily.



Pure ASP.NET

A Code-Intensive Premium Reference

Author: Robert Lair and Jason Lefebvre

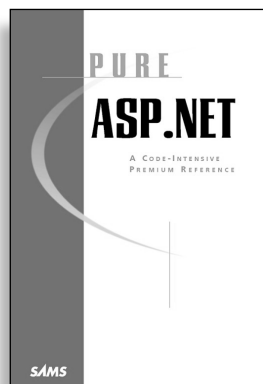
ISBN: 067232069X

Price: \$39.99

Available: Summer 2001

Pure ASP.NET is a premium reference for Active Server Pages development in the new Microsoft .NET Framework. *Pure ASP.NET* is comprised of three parts:

- **Part I Conceptual Reference** is a fast-paced primer that covers ASP.NET fundamentals and concepts.
- **Part II Techniques Reference** is full of well-commented, commercial-quality code that illustrates practical applications of ASP.NET concepts. Examples are presented in both Visual Basic and C# to appeal to a wide variety of programmers.
- **Part III Syntax and Object Reference** contains detailed coverage of .NET Namespaces such as System.Web and System.Data that are invaluable to ASP.NET developers, as well as Visual Basic and C# language references.



Sams Teach Yourself ASP.NET

in 24 Hours

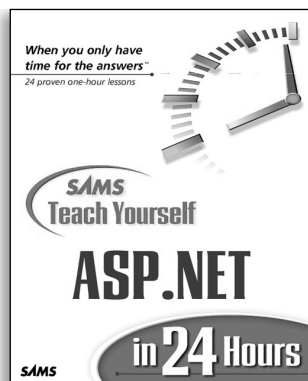
Author: Joe Martin and Brett Tomson

ISBN: 0672321262

Price: \$39.99

Available: Summer 2001

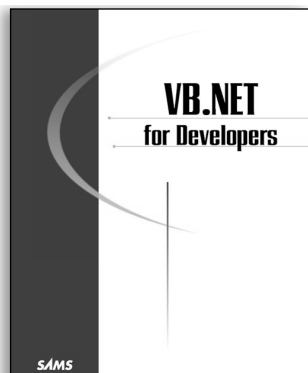
Building on an overview of the basic architecture of the .NET Framework, *Sams Teach Yourself ASP.NET in 24 Hours* guides the reader through ASP.NET's basic structure, function and working syntax (data types, operators, functions, Web forms, etc). The unique approach exposes and explains both VB.NET and C#, including examples for both. Then, the programmer is walked through the creation of a live ASP.NET application. Finally, concise explanations of data manipulation, security, deployment, and profiling/optimization are introduced.



VB.NET for Developers

Author: Keith Franklin
ISBN: 0672320894
Price: \$39.99
Available: Summer 2001

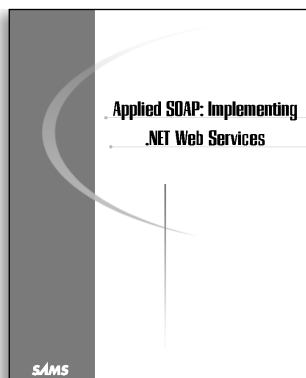
This book will smooth the transition to Visual Basic.NET and help developers understand the paradigm shift presented by the .NET Framework. Key differences between VB 6 and VB.NET will be highlighted in the code samples.



Applied SOAP: Implementing .NET Web Services

Author: Kennard Scribner and Mark Stiver
ISBN: 0672321114
Price: \$49.99
Available: Fall 2001

This book takes the reader from the architecture of .NET to real-world techniques he can use in his own Internet applications. The reader is introduced to .NET and Web Services and explores (in detail) issues surrounding the fielding of successful Web Services. Practical guidelines as well as solutions are provided that the reader may use in his own projects. Some of the issues involve lack of specific guidance in the SOAP specification, while others transcend SOAP and involve issues Internet developers have grappled with since the inception of the World Wide Web.



Want to learn more about

.net

Now that you've read the basics... get the in-depth training you will need to move to the next level.

.NET represents a significant shift in the way you will build applications. Get ahead of the learning curve with classroom training from **Volant Training!**

Volant Training is pleased to present .NET courseware aimed at making your transition to .NET a complete success.

● **COURSES AVAILABLE NOW:**

Moving from VB to VB.NET, and

Building ASP.NET Web Solutions with VB.NET

- Courseware available today based on BETA 2, and will be updated for the final version of VS.NET
- More courses coming soon

● **WHY VOLANT TRAINING?**

- Courseware focused on building scalable, enterprise-ready solutions
- Real-world experience with VB, ASP, .NET, and courseware authoring



FOR MORE INFORMATION, VISIT:

www.VolantTraining.com