Object-Oriented Programming with Visual Basic .NET

By J. P. Hamilton

Publisher : O'Reilly
Pub Date : October 2002
ISBN : 0-596-00146-0
Pages : 308

only for RuBoard

Object-Oriented Programming with Visual Basic .NET

By J. P. Hamilton

Publisher : O'Reilly
Pub Date : October 2002
ISBN : 0-596-00146-0
Pages : 308

Visual Basic .NET is a language that facilitates object-oriented programming, but does not guarantee good code. That's where Object-Oriented Programming with Visual Basic .NET comes in. It will show you how to think about similarities in your application logic and how to design and create objects that maximize the benefit and power of .NET. Packed with examples that will guide you through every step, Object-Oriented Programming with Visual Basic .NET is for those with some programming experience.

copyright

Printed in the United States of America.

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

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

Preface

This book is not a reference. That needs to be said right off the bat. It was written to be read cover to cover; it tells a story. It's an interwoven tale about object-oriented programming in the .NET world: building objects, moving them, and using them around the world. This is not just a how-to book; it's a why-to and a when-to book as well.

You will encounter many twists and turns ahead. Expect to learn the unexpected, but do not expect Visual Studio .NET. It didn't come out until a year after I got the .NET beta for the first time. Everything in this book is very hands-on, so if you're afraid you might chip a nail, turn back now. If you like being under the hood, though, you will feel right at home.

I started preparing for this book so long ago that it's not even funny. I actually have some old, crusty .doc files that refer to "Cool." That's what they were going to call C# before they called it C#. I'm not joking. This book began its life when most of the other .NET books began theirs shortly after the Microsoft Professional Developer's Conference in 2000. Now, two years later, someone is finally reading it. Hopefully, you will see that it wasn't rushed to market. I have thought about everything in this book very carefully and have spent about a year and a half of my free time putting it together.

Why? I'm on a mission. Several, in fact. My main purpose is to provide an alternative to the big, fat reference book (especially the ones written by more than one author). I love reading books about programming especially skinny books about computing that assume I am not an idiot. My goal is to write these kinds of books. I assume you know what HTTP, XML, and SOAP stand for. To me, that means something.

My second mission is to give all my readers a .NET epiphany. I remember talking to my editor Ron on the phone a couple of years ago. "Whaddya mean there's no COM?" I said. Shortly after the phone call, I received my first beta of .NET in the mail. The sheer size and depth of it stunned me. A super-secret, subterranean coding army must have been at work! I will never know how Microsoft managed to have something of this scope back then (that nobody knew about). My first goal was to "get it." I am still learning about .NET two years later, but now I "get it." I wasn't able to present everything under the sun in this book, but I think you'll also "get it" by the time you finish reading it.

My third mission is to make loads of cash. I remember a time when I sat around trying to think of how I could get a job that paid minimum wage in addition to all my contract work. I had too much of a social life back then, and all the time with my friends and family was really killing me. I found that I wasn't spending enough quality time in front of the computer. Then it hit me: "I'll write programming books!" I haven't slept since.

Audience

Anyone can read a reference manual, but the reference often neglects the fundamental principals. I remember when I first started trying to learn Windows in 1991. I always ended up having more questions than answers at the end of the day (thank God for Charles Petzold!). My inexperience was partly to blame, but I have never forgotten those days, and I hope that beginners who read my book will learn from the lessons that I've managed to learn only with great pain. However, I also wrote this book for professional VB gunslingers who have been using the language for years to bring home the bacon. In some ways, VB.NET is a familiar friend. In others, it's an entirely new beast.

If you are looking for a language reference, this ain't it. If you are new to programming, though, you will need a reference. I suggest O'Reilly's VB.NET Language in a Nutshell by Steven Roman, Ron Petrusha, and Paul Lomax, which is concise and user friendly.

If you are not new to programming, a week and a help file is all you need to learn the language's syntax (OK, I am exaggerating a bit). Although this book is not a language reference, you can learn a lot from the example code; much of the VB.NET syntax is used within context.

About This Book

This book primarily covers the topic of building objects. It discusses how they are designed, why they are designed that way, and how the design fits in with .NET. Here is a brief overview of what lies ahead.

Chapter 1, Introduction, is a high-level view of object-oriented programming and the key concepts of the .NET Framework. This chapter establishes key object-oriented terminology and shows how it applies to .NET.

Chapter 2, Object Fundamentals, discusses objects and the .NET world they live in. It includes discussions on compiling objects, namespaces, application domains, assemblies, intermediate language, and the .NET class library.

Chapter 3, Class Anatomy, shows how to build classes. Topics include member variables, methods, properties, access modifiers, and the use of access modifiers in class designs. The chapter also discusses passing parameters, the difference between reference types and value types, creating and destroying objects, the .NET garbage collector, events, and delegates.

Chapter 4, Object-Orientation, focuses on object-oriented programming (OOP). Topics include specialization and generalization, inheritance, and containment. The chapter also discusses polymorphism: substitution, method overloading and overriding, and shadowing. You will learn about using polymorphism, abstract base classes, and the Open-Closed Principle, which allows you to write flexible object hierarchies. Discussions of proper inheritance and the Liskov Substitution Principle are also included. The chapter ends with an in-depth look at interface-based programming and a few of the major interfaces you need to learn to make robust .NET objects.

Chapter 5, Interfacing .NET, discusses interface-based programming and how it fits into the world of OOP. The chapter also covers some of the most important .NET interfaces.

Chapter 6, Exceptional Objects, deals with exception handling within the .NET Framework. You will learn how and when to write your own exceptions, use the `AppDomain` unhandled exception handler, use a stack trace, resume and retry code, and use performance counters to profile application exceptions.

Chapter 7, Object Inspection, covers a powerful .NET technology called *reflection*, which allows you to query type information programmatically. The chapter discusses runtime type discovery, dynamic type inspection, and attributes. You will also learn how to build custom attributes by using them to provide behavior for VB.NET that mimics C# XML documentation comments.

Chapter 8, Object In, Object Out, deals with streams and serialization. Discussions include binary and XML serialization and the streams available in .NET, the Schema Definition Tool, and custom serialization. The chapter uses a TCP server and client to illustrate the use of network streams.

Chapter 9, *Object Remoting*, shows how to move objects into a distributed environment. It discusses channels, activation models, configuration, marshaling, lifetime leases, proxies, and other remoting fundamentals. The chapter uses a reusable Windows service to host remote objects and shows how to configure and use IIS to host remote objects. The chapter also demonstrates how to use object factories to build flexible, distributed systems.

Chapter 10, Web Services, describes how to write XML web services, host them from IIS, and make them available for .NET remoting. You will learn when to use web services and when to use remoting. The chapter also covers compatibility issues that affect consumption.

only for RuBoard

Assumptions This Book Makes

At times, this book is ideal for beginners. At other times, it is for intermediate programmers. And sometimes, it is for advanced coders. Regardless of your skills, this book assumes three things: the .NET Framework is installed on a machine and under your control, you have access to a web server, and, when you get curious, you know how to look up whatever interests you in the documentation. .NET is massive. However, I wasn't able to cover every subject; I wanted the book to be manageable for the readers, fun, and informative.

This book was intended to leave you with some unanswered questions. By the end of it, you should have a better idea about which questions you need to ask. A good companion to this text, Dave Grundgeiger's Programming Visual Basic .NET, fills in a few gaps that I left out on purpose, particularly ADO.NET, Windows Forms, and ASP.NET. In this book, objects are objects. Whether they make a window on the screen

or update a record in a database, you should follow fundamental rules when using them. That is one of the most important messages you'll learn from this book.

Conventions Used in This Book

I use the following font conventions in this book:

Italic is used for:

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

Boldface is used for:

- Names of GUI items: window names, buttons, and menu choices

`Constant width` is used for:

- Command lines and options that should be typed verbatim
- Names and keywords in programs, including method names, variable names, and class names
- XML element tags

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
1-800-998-9938 (in the United States or Canada)
1-707-829-0515 (international or local)
1-707-829-0104 (fax)

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

http://www.oreilly.com/catalog/objectvbnet

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

bookquestions@oreilly.com

For more information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

only for RuBoard

Acknowledgments

First, I would like to thank my editor, Ron Petrusha, for all his help. Let it be known that I am probably not the easiest writer to work with, so his guidance was immensely appreciated. Why he decided to do a second book with me, I will never know. Of course, I must extend this gratitude to the rest of the O'Reilly gang: Tatiana Diaz, John Osborn, Glen Gillmore, Brian Sawyer, and Claire Cloutier.

Next, I'd like to thank my technical reviewers. I owe Robert C. Martin a very special thank you for reviewing the major OO portions of the book (Chapter 4 and Chapter 5). Having his input was an awesome experience because much of what I have learned about OOP comes from his writings. I would also like to thank Ingo Rammer for answering many of my remoting questions. Buy his book, Advanced .NET Remoting (APress). It's the best. And last but not least, thanks to Daniel Creeron for his thorough review of the entire book.

On a personal note, I don't think this book would have been possible without the many people who helped me this past year more than they will ever know: Natasha Deveraux, Kristen Guggenheim, Melinda Parmley, Marty Kelly, Michael Anderson, Joby Erickson, Kelly Christopher, Jon Polley, Joe Boley, Steve Myers, David Braddy, Robert Smith, and Ogre. God bless you all.

only for RuBoard

Chapter 1. Introduction

To understand the world of object-oriented programming, look at the world around you for a moment. You might see vacuum cleaners, coffee makers, ceiling fans, and a host of other objects. Everywhere you look, objects surround you.

Some of these objects, such as cameras, operate independently. Some, such as telephones and answering machines, interact with one another. Some objects contain data that persists between uses, like the address book in a cell phone. Some objects contain other objects, like an icemaker inside of the freezer.

Many objects are similar in function but different in purpose. Bathtubs and kitchen sinks, for example, both provide water and are used for cleaning. But it is a rare occasion when you will take a bath in the kitchen sink or wash your dishes in the tub. However, the bathtub and the kitchen sink in your house probably share the same plumbing. Certainly, they share a common interface: hot and cold water knobs, a faucet, and a drain.

When you think about it, what is the difference between a sink and a bathtub? The location? The size of the basin? Their heights off the ground? How many more similarities are there than differences?

Sometimes the same action causes an object to do different things depending on the context of the situation. When you press Play on the remote, the DVD might play a movie on the television. But if a CD is in the player, it plays music out of the speakers. Same button, same action different results. When you flip the switch on the back porch, the light comes on. But the switch in the kitchen turns on the garbage disposal. You use the same kind of switch, but obtain different results.

You can think about many objects around you in terms of black boxes. You comprehend the fundamentals of these objects and possess a basic understanding of what makes them work, but the specifics of their operation are unknown to you. And you like it that way. Do you really want to have to know the inner mechanisms of every object in your house in order to use it?

Consider that light bulb on the back porch. The filament in the bulb is nothing more than a simple resistor. When the 100-watt bulb is "on," the filament's temperature is about 2550 degrees Celsius. The resulting thermal radiation, which is proportional to the length of the filament (but not the diameter), produces about 1750 lumens worth of visible light at a wavelength of about 555 nanometers. And by the way, the filament is made out of tungsten.

Do you really want to know these minute details, or do you just want the light to come on when you flick the switch?

Any object has two inherent properties: state and behavior. The light bulb on the back porch has state. It can be on or off. It has a brand name and a life expectancy. It has been in use for a certain number of hours. It has a specified number of hours left before the irregular evaporation of its tungsten filament causes it to burn out. Behaviorally, it provides light; it shines.

But an object is rarely an island unto itself.

Many objects participate collectively in a system. The television and surrounding sound speakers are a part of a system called a home theater. The refrigerator and oven belong to a system called a kitchen. These systems, in turn, are a part of a larger system that is called an apartment. Collections of apartments make up a system known as a complex. Apartments and houses belong to neighborhoods and so on, ad infinitum.

In essence, this book discusses systems. Building and designing objects is one aspect of this process of building a system. Determining how these objects interact with one another is another. Understanding both phases of development is crucial when building any system that has more than a modicum of complexity.

Generally, you can think of this process of developing a system as object-oriented programming and object-oriented design. Specifically, though, you are really working toward an understanding of the objects you build and the system in which they participate. Component-based programming forms the basis of this system.

Programming objects in software doesn't require an object-oriented language, and just because you use an object-oriented programming language doesn't mean that your code is object-oriented. Languages can only assist the process; they can't make any guarantees. The ability to write object-oriented software was always available with VB. Writing it just hasn't always been easy because the language wasn't always oriented in that direction. Developing binary reusable components in VB has been possible for some time now, but using these components across languages used to be considered somewhat of a black art until now.

Today, Visual Basic .NET is a cutting-edge, object-oriented language that runs inside of a state-of-the-art environment. It is feature-rich and designed to take advantage of the latest developments in object-oriented programming. Writing software and building components has never been easier.

only for RuBoard

1.1 Visual Basic .NET and Object-Oriented Programming

Visual Basic .NET is a fully object-oriented programming language, which means it supports the four basic tenets of object-oriented programming: abstraction, encapsulation, inheritance, and polymorphism.

We have already conceptualized many of these object-oriented concepts by just looking at the objects that surround us in our everyday lives. Let's look more closely at these terms and see what they actually mean and what they do for developers of object-oriented software.

1.1.1 Abstraction

A radio has a tuner, an antenna, a volume control, and an on/off switch. To use it, you don't need to know that the antenna captures radio frequency signals, converts them to electrical signals, and then boosts their strength via a high-frequency amplification circuit. Nor do you need to know how the resulting current is filtered, boosted, and finally converted into sound. You merely turn on the radio, tune in the desired station, and listen. The intrinsic details are invisible. This feature is great because now everyone can use a radio, not just people with technical know-how. Hiring a consultant to come to your home every time you wanted to listen to the radio would become awfully expensive. In other words, you can say that the radio is an object that was designed to hide its complexity.

If you write a piece of software to track payroll information, you would probably want to create an Employee object. People come in all shapes, sizes, and colors. They have different backgrounds, enjoy different hobbies, and have a multitude of beliefs. But perhaps, in terms of the payroll application, an employee is just a name, a rank, and a serial number, while the other qualities are not relevant to the application. Determining what something is, in terms of software, is abstraction.

In object-oriented software, complexity is managed by using abstraction. Abstraction is a process that involves identifying the crucial behavior of an object and eliminating irrelevant and tedious details. A well thought-out abstraction is usually simple, slanted toward the perspective of the user (the developer using your objects), and has probably gone through several iterations. Rarely is the initial attempt at an abstraction the best choice.

Remember that the abstraction process is context sensitive. In an application that will play music, the radio abstraction will be completely different from the radio abstraction in a program designed to teach basic electronics. The internal details of the latter would be much more important than the former.

1.1.2 Encapsulation

Programming languages like C and Pascal can both produce object-like constructs. In C, this feature is called a struct; in Pascal, it is referred to as a record. Both are user-defined data types. In both languages, a function can operate on more than one data type. The inverse is also true: more than one function can operate on a single data type. The data is fully exposed and vulnerable to the whims of anyone who has an instance of the type because these languages do not explicitly tie together data and the functions that operate on that data.

In contrast, object-oriented programming is based on encapsulation. When an object's state and behavior are kept together, they are encapsulated. That is, the data that represents the state of the object and the methods (Functions and Subs) that manipulate that data are stored together as a cohesive unit.

Encapsulation is often referred to as information hiding. But although the two terms are often used interchangeably, information hiding is really the result of encapsulation, not a synonym for it. They are distinct concepts. Encapsulation makes it possible to separate an object's implementation from its behavior to restrict access to its internal data. This restriction allows certain details of an object's behavior to be hidden. It allows us to create a "black box" and protects an object's internal state from corruption by its clients.

Encapsulation is also frequently confused with abstraction. Though the two concepts are closely related, they represent different ideas. Abstraction is a process. It is the act of identifying the relevant qualities and behaviors an object should possess. Encapsulation is the mechanism by which the abstraction is implemented. It is the result. The radio, for instance, is an object that encapsulates many technologies that might not be understood clearly by most people who benefit from it.

In Visual Basic .NET, the construct used to define an abstraction is called a *class*. The terms class and object are often used interchangeably, but an object is actually an instance of a class. A component is a collection of

one or more object definitions, like a class library in a DLL.

1.1.3 Inheritance

Inheritance is the ability to define a new class that inherits the behaviors (and code) of an existing class. The new class is called a child or derived class, while the original class is often referred to as the parent or base class.

Inheritance is used to express "is-a" or "kind-of" relationships. A car is a vehicle. A boat is a vehicle. A submarine is a vehicle. In OOP, the `Vehicle` base class would provide the common behaviors of all types of vehicles and perhaps delineate behaviors all vehicles must support. The particular subclasses (i.e., derived classes) of vehicles would implement behaviors specific to that type of vehicle. The main concepts behind inheritance are extensibility and code reuse.

In contrast to inheritance, there is also the notion of a "has-a" relationship. This relationship is created by using composition. Composition, which is sometimes referred to as aggregation, means that one object contains another object, rather than inheriting an object's attributes and behaviors. Naturally, a car has an engine, but it is not a kind of engine.

C++ supports a type of reuse called multiple inheritance. In this scenario, one class inherits from more than one base class. But many C++ programmers will tell you that using multiple inheritance can be tricky. Base classes with identical function names or common base classes can create nightmares for even the most experienced programmers.

VB.NET, like Java, avoids this problem altogether by providing support only for single inheritance. But don't worry, you aren't missing out on anything. Situations that seem ideal for multiple inheritance can usually be solved with composition or by rethinking the design.

When it comes to proper object-oriented design, a deep understanding of inheritance and its effects is crucial. Deriving new classes from existing classes is not always as straightforward as it might initially appear. Is a circle a kind of ellipse? Is a square a kind of rectangle? Mistakes in an inheritance hierarchy can cripple an object model.

1.1.4 Polymorphism

Polymorphism refers to the ability to assume different forms. In OOP, it indicates a language's ability to handle objects differently based on their runtime type.

When objects communicate with one another, we say that they send and receive messages. The advantage of polymorphism is that the sender of a message doesn't need to know which class the receiver is a member of. It can be any arbitrary class. The sending object only needs to be aware that the receiving object can perform a particular behavior.

A classic example of polymorphism can be demonstrated with geometric shapes. Suppose we have a `Triangle`, a `Square`, and a `Circle`. Each class is a `Shape` and each has a method named `Draw` that is responsible for rendering the `Shape` to the screen.

With polymorphism, you can write a method that takes a `Shape` object or an array of `Shape` objects as a parameter (as opposed to a specific kind of `Shape`). We can pass `Triangles`, `Circles`, and `Squares` to these methods without any problems, because referring to a class through its parent is perfectly legal. In this instance, the receiver is only aware that it is getting a `Shape` that has a method named `Draw`, but it is ignorant of the specific kind of `Shape`. If the `Shape` were a `Triangle`, then `Triangle`'s version of `Draw`

would be called. If it were a `Square`, then `Square`'s version would be called, and so on.

We can illustrate this concept with a simple example. Suppose we are working on a small graphics package and we need to draw several shapes on the screen at one time. To implement this functionality, we create a class called `Scene`. `Scene` has a method named `Render` that takes an array of `Shape` objects as a parameter. We can now create an array of different kinds of shapes and pass it to the `Render` method. `Render` can iterate through the array and call `Draw` for each element of the array, and the appropriate version of `Draw` will be called. `Render` has no idea what specific kind of `Shape` it is dealing with.

The big advantage to this implementation of the `Scene` class and its `Render` method is that two months from now, when you want to add an `Ellipse` class to your graphics package, you don't have to touch one line of code in the `Scene` class. The `Render` method can draw an `Ellipse` just like any other `Shape` because it deals with them generically. In this way, the `Shape` and `Scene` classes are *loosely coupled*, which is something you should strive for in a good object-oriented design.

This type of polymorphism is called parametric polymorphism, or generics. Another type of polymorphism is called overloading. Overloading occurs when an object has two or more behaviors that have the same name. The methods are distinguished only by the messages they receive (that is, by the parameters of the method).

Polymorphism is a very powerful concept that allows the design of amazingly flexible applications. Chapter 4 discusses polymorphism in more depth.

only for RuBoard

1.2 The .NET Framework

The objects you construct with VB.NET will live out their lives within the .NET Framework, which is a platform used to develop applications. The platform was designed from the ground up by using open standards and protocols like XML, HTTP, and SOAP. It contains a rich standard library that provides services available to any language running under its protection.

The impetus behind its creation was the desire to develop a platform for building, deploying, and running web-based services. In spite of this goal, the framework is ideal for developing all types of applications, regardless of the design. The .NET Framework makes child's play of some of programming's most sophisticated concepts, giving you the ability to take advantage of today's cutting-edge architectures:

- Distributed computing using open Internet standards and protocols such as HTTP, XML, and SOAP
- Enterprise services such as object pooling, messaging, security, and transactions
- An infrastructure that simplifies the development of reusable cross-language compatible components that can be deployed over the Internet
- Simplified web development using open standards
- Full language integration that make it possible to inherit from classes, catch exceptions, and debug across different languages

Deployment is made simpler because settings are stored in XML-based configuration files that reside in the application directory; there is no need to go to the registry. Shared DLLs must have a unique hash value, locale, and version, so physical filenames are no longer important once these considerations are met. Not having physical filenames makes it possible to have several different versions of the same DLL in use at the same time, which is known as *side-by-side* execution. All dependencies and references are stored within the executable in a section called the *manifest*. In a sense, we're back to the days of DOS because to deploy an application, you only need to *xcopy* it from one directory to another.

This book explores many aspects of .NET in order to gain a complete understanding of the components you

write and the world in which they live. Doing it any other way is impossible. The .NET Framework provides so many services your components will use that discussing one without referring to the other is literally impossible they are that closely tied together.

Two major elements of the .NET Framework will be addressed repeatedly throughout this book. The first is the Common Language Runtime (CLR), which provides runtime services for components running under .NET.

The second element is the .NET class library, a vast toolbox containing classes for everything from data access, GUI design, and security to multithreading, networking, and messaging. The library also contains definitions for all primary data types, such as bytes, integers, and strings. All of these types are inherently derived from a base class called `System.Object`, which you can think of as a "universal" data type; there is no distinction between the types defined by the system and the types you create by writing classes or structures. Everything is an object!

The term .NET means many things to many different people. When the term is used in this book, it always refers to the .NET Framework the Common Language Runtime and the .NET class library.

In the past, passing a string from a component written in VB to one written in C++ (or vice versa) could be frustrating. Strings in VB weren't the same as the strings in C++. In fact, under some circumstances, using a component written in C++ from VB was downright impossible because of issues involving data types. VB just doesn't know what to do with an `LPSTR`! Every language under .NET uses the same data types defined in the base class library, so interoperability problems of the past are no longer an issue.

This book touches on several major areas of the library and focuses on the development of components using VB.NET. However, if you follow the examples, you might be surprised at just how much you know.

1.2.1 The Common Language Runtime

The CLR is the execution engine for the .NET Framework. This runtime manages all code compiled with VB.NET. In fact, code compiled to run under .NET is called *managed* code to distinguish it from code running outside of the framework.

Besides being responsible for application loading and execution, the CLR provides services that will benefit component developers:

- Invocation and termination of threads and processes
- Object lifetime and memory management
- Cross-language integration
- Code access and role-based security
- Exception handling (even across languages)
- Deployment and versioning
- Interoperation between managed and unmanaged code
- Debugging and profiling support (even across languages)

Runtimes are nothing new. Visual Basic has always had some form of a runtime. Visual C++ has a runtime called *MSVCRT.DLL*. Perl, Python, and SmallTalk also use runtimes. The difference between these runtimes and the CLR is that the CLR is designed to work with multiple programming languages. Every language whose compiler targets the .NET Framework benefits from the services of the CLR as much as any other language.

.NET is also similar to Java. Java uses a runtime called the Java Virtual Machine. It can run only with Java code, so it has the same limitations as the other languages mentioned previously. Another distinction is that

the JVM is an interpreter. Although all languages in the .NET environment are initially compiled to a CPU-independent language called Intermediate Language (which is analogous to Java byte code), IL is not interpreted at runtime like Java. When code is initially executed, one of several just-in-time (JIT) compilers translate the IL to native code on a method-by-method basis.

Cross-language integration is one of the major benefits provided by the CLR. If a colleague has written a base class in C#, you can define a class in VB.NET that derives from it. This is known as cross-language inheritance. Also, objects written in different languages can easily interoperate. The two parts of the CLR that make this interoperation possible are the Common Type System and the Common Language Specification.

1.2.1.1 Common Type System

The Common Type System (CTS) defines rules that a language must adhere to in order to participate in the .NET Framework. It also defines a set of common types and operations that exist across most programming languages and specifies how these types are used and managed within the CLR, how objects expose their functionality, and how they interoperate. The CTS forms the foundation that enables cross-language integration within .NET.

1.2.1.2 Common Language Specification

The Common Language Specification (CLS) is a subset of the CTS that describes the basic qualities used by a wide variety of languages. Components that use only the features of the CLS are said to be CLS-compliant. As a result, these components are guaranteed to be accessible from any other programming language that targets .NET. Because VB.NET is a CLS-compliant language, any class, object, or component that you build will be available from any other CLS-compliant programming language in .NET.

1.2.2 A First-Class Citizen

VB has always been easy to learn, but the power of simplicity came with a price. The language itself has never gotten the respect it deserves because it always hid so much from the developer; getting under the hood required a sledgehammer. This is no longer true. While VB is still a great language and is relatively painless to learn and use, you are no longer restricted in how "low you can go."

One of the most important concepts behind .NET is that all languages are on a level playing field; the choice of language should be determined more by your style than anything else. This is probably the reason why you prefer VB over other languages: you like the syntax of Visual Basic and appreciate its simplicity. No longer is choice of language a concern, because VB.NET is just as fast as C# and it does a few things, such as event declaration and conditional exception handling, better. But for the most part, any language that runs under .NET will provide you with the tools to develop cutting edge software. Thus, it truly is a matter of style. VB.NET is no more or no less of a language than any other in the .NET Framework.

only for RuBoard

Chapter 2. Object Fundamentals

Before designing and building objects for the .NET environment, understanding the environment itself is important. In this respect, a little bit of code goes a long way. This chapter deviates from the standard "Hello, world" application in favor of a "Hello, world" component. Then it builds a small client that uses the component to display a message to the console window.

only for RuBoard

2.1 Creating and Compiling the Component

Example 2-1 contains the listing for our "Hello, world" component. It contains a single class named `Hello` with a single method named `Write`. Save the listing to a file named hello.vb. The rest of the chapter will use this listing as a foundation of discussion.

All Visual Basic source code should be saved to files with a .vb extension. One file can contain one class or several classes. How you organize the code is up to you.

Example 2-1. The "Hello, world" component

```
Option Strict On

Imports System

Namespace Greeting

Public Class Hello
    Public Sub Write(ByVal value As String)
        Console.WriteLine("Hello, {0}!", value)
    End Sub
End Class

End Namespace
```

The Visual Basic .NET command-line compiler is a program called vbc.exe that should be in your path once the .NET Framework is installed. All examples in this book assume that the example code exists in the root directory of your hard drive. This assumption is made to improve readability. If the code is not in your hard drive's root directory, you need to specify a fully qualified pathname to the compiled file or compile from the directory where the source code is located. With this in mind, you should be able to compile Example 2-1 to a dynamic link library (DLL) as follows:

```
C:\>vbc /t:library hello.vb
```

The `/t:` option is short for `target`, which can be one of the following values:

exe
: A console application. If the `/t` switch is omitted, this is the default value.

winexe
: A Windows executable.

library
: A DLL.

module
: A module. This value is similar to a *.lib* file in C++. It contains objects but is not an executable.

As a default, the compiler gives the output file the same name as the file being compiled. Here, a file named hello.dll is produced.

All examples in this book assume that `Option Strict` is turned on. `Option Strict` prevents the VB compiler from making implicit narrowing type conversions, which is often a source of errors. Implicit narrowing type conversion involves using one type where another type with a smaller range is expected. For example, the following code is illegal when `Option Strict` is turned on:

```
Dim x As Short = 5
```

```
Dim b As Byte = x
```

The code is illegal because `x` is a `Short`, a type whose range is `-32,768` through `32,767`. The code here assigns the value of this type to a `Byte`, an unsigned numeric type whose range is `0` to `255`. Since `Short` values from `-32,768` to `-1` and from `256` to `32,767` don't "fit" into a `Byte` variable, an implicit conversion could result in loss of data. Therefore, `Option Strict On` forbids the conversion; if you want to assign `x` to a `Byte` variable, you have to make the conversion explicit, as in the following code:

```
Dim x As Short = 5
Dim b As Byte = System.Convert.ToByte(x)
```

Or as in the following code, which uses an intrinsic Visual Basic function:

```
Dim x As Short = 5
Dim b As Byte = CByte(x)
```

You can put `Option Strict On` at the top of your source file to turn it on or specify `/optionstrict+` from the command line when you compile. This is the recommended setting, but unfortunately, the default is `Off`.

Many examples in this book run in a console window. To compile them using Visual Studio .NET, change the Startup Object in the project properties dialog to "Sub Main."

only for RuBoard

2.2 Namespaces

All classes are members of some namespace. You can think of a namespace as a user-defined scope an organizational construct that allows you to group your classes in a meaningful way and uniquely identify your classes and their members in case of naming conflicts. The `Hello` class from Example 2-1 is a member of a namespace called `Greeting` denoted by the `Namespace` block surrounding its definition. Even if the namespace block were removed, the `Hello` class would still be considered a member of the *root namespace*, which is scoped to the executable.

2.2.1 Imports

Every time anything is compiled with VB, two class libraries mscorlib.dll and Microsoft.VisualBasic.dll are referenced implicitly. The latter contains classes that provide backward compatibility to earlier versions of Visual Basic, while the former contains portions of the `System` and several other namespaces. Notice the second line of code in Example 2-1:

```
Imports System
```

This line brings the `System` namespace into the scope of the current file, hello.vb. This is done for the benefit of the call to `Console.WriteLine`, which writes a message to the console window. Without the `Imports` directive, the `Console` class could be referred to only through its namespace, which means the call would look like this:

```
Public Class Hello
    Public Sub Write(ByVal value As String)
        System.Console.WriteLine("Hello, {0}!", value)
    End Sub
End Class
```

This particular situation is not too bad, but if the file contained several calls to `Console.WriteLine`, things could get ugly. As most of the .NET class library is contained within the `System` namespace, importing it is usually your best option.

Framework Organization

The `System` namespace is defined in mscorlib.dll and System.dll. Although *mscorlib.dll* is referenced by the compiler automatically, *System.dll* is not.

If you need to use a class from a namespace in System.dll (or any other portion of the .NET class library), you have to add a reference to it when you compile with the `/r` compiler option. For example:

```
vbc /t:exe /r:system.dll code.vb
```

Other major portions of .NET include:

- ASP.NET, which is mostly contained in System.Web.dll
- Windows Forms, which is mostly contained in System.Windows.Forms.dll and System.Windows.Drawing.dll
- Data and XML, which are contained in System.Data.dll and System.XML.dll

only for RuBoard

2.3 Using a Component

Now that you have a component, you need a way to use it. Example 2-2 contains a listing for a simple client. Save it to a file named hello-client.vb. Let's see how everything fits together, and then you can compile it.

Example 2-2. "Hello, world" client

```
Imports System
Imports Greeting

Public Class Application

  Public Shared Sub Main( )
    Dim hw As New Hello( )
    hw.Write("World")
    Console.ReadLine( )
  End Sub

End Class
```

The `Greeting` namespace defined in Example 2-1 was imported. Without it, every class in the `Greeting` namespace would have to be referenced directly, as in the following code:

```
Public Shared Sub Main( )
  Dim hw As New Greeting.Hello( )
  hw.Write("World")
End Sub
```

All standalone executables require an entry point with this signature:

```
Public Shared Sub Main( )
```

This is where everything begins, but as you can see, not much is going on. The only thing the client does is declare an instance of the `Hello` class (which is defined in the component) and call its `Write` method.

When compiled, Example 2-2 must explicitly reference hello.dll for everything to compile. This can be accomplished with the `/r` compiler option. Assuming that the DLL lives in the same directory as the client code, this executable can be compiled as follows:

```
C:\>vbc /t:exe /r:hello.dll hello-client.vb
```

This compilation produces an executable named hello-client.exe. You can change the name of the output file by using the `/out` compiler option:

```
C:\>vbc /t:exe /r:hello.dll /out:hello.exe hello-client.vb
```

When the executable runs, the following code is dumped to the console:

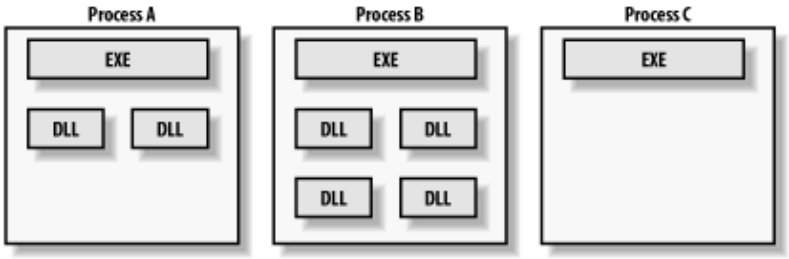
```
Hello, World!
```


only for RuBoard

2.4 Application Domains

In an unmanaged Windows environment, applications are isolated from one another by process boundaries. As shown in Figure 2-1, each Win32 program is given its own process and a 4 GB virtual address space to go along with it. Additional libraries or components share this address space with their client. The operating system handles all the work associated with mapping the virtual address space to an actual address in memory. The advantage of this isolation is that if a program crashes, it won't take the entire system down just the current process.

Figure 2-1. The Win32 process boundary



Under .NET, the CLR manages the memory; Windows doesn't give it to you directly. One reason for this is garbage collection. The CLR needs to know where memory is located to free it or determine if it is in use at all, which is why no pointers are allowed in managed code. This level of indirection, in regard to memory, allows the CLR to provide application isolation with more granularity. In .NET, an entity called an *application domain* determines isolation.

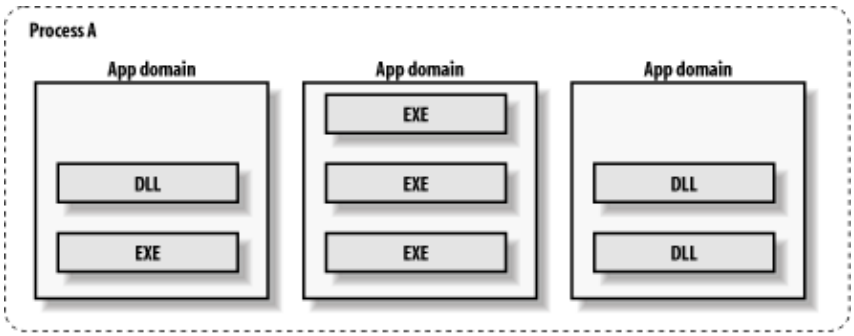
Managed Versus Unmanaged Code

The term managed code refers to code compiled under the .NET Framework. The framework automatically manages the lifetime of an object, hence the term. Unmanaged code is simply all other code running outside of the bounds of .NET.

As shown in Figure 2-2, several application domains can exist within the same process and still remain isolated from one another. They can also be loaded and unloaded independently of the process, which means

that if a crash occurs, the offending application domain can be unloaded without affecting the entire process. This is highly efficient, considering the amount of overhead involved in creating a process.

Figure 2-2. Application domains provide isolation for .NET applications within a process.



Application domains do not share the same restrictions as a process. They can contain any number of EXEs or DLLs in several combinations. Typically, though, an application is loaded into one application domain. This is the case with hello-client.exe and hello.dll. In fact, you can verify this yourself with the CLR shell debugger. In a console window, run hello-client.exe. Spawn a second console and start up the debugger from the command line like this:

```
C:>cordbg
```

Once the debugger is running, you can issue a `pro` command to get information on all managed .NET applications that are running on the system. It will look similar to this:

```
C:\>cordbg
Microsoft (R) Common Language Runtime Test Debugger Shell
Version 1.0.3705.0 Copyright (C) Microsoft Corporation 1998-2001.
All rights reserved.

(cordbg) pro

PID=0x818 (2072)  Name=C:\hello-client.exe
        ID=1  AppDomainName= hello-client.exe

PID=0x6c0 (1728)  Name=C:\WINNT\Microsoft.NET\Framework\v1.0.3705\aspnet_wp.exe
        ID=8  AppDomainName=/LM/w3svc/1/root/eyeofnet-7-126592112951200512
        ID=7  AppDomainName=/LM/W3SVC/1/ROOT-6-126592112840541392
        ID=6  AppDomainName=/LM/W3SVC/1/Root/oowd-5-126591711176576976
        ID=5  AppDomainName=/LM/w3svc/3/root/eyeofnet-4-126591677652271392
        ID=4  AppDomainName=/LM/W3SVC/3/Root-3-126591677517878144
        ID=1  AppDomainName=DefaultDomain
```

Loading hello-client.exe and hello.dll into two different application domains is possible. But exotic configurations such as this require additional code. The hello client would have to create the assembly dynamically and load the hello component into it. Calls between the two executables would require remoting across the application domain boundaries, which is very similar to the behavior of an out-of-process COM server.

only for RuBoard

2.5 Contexts

Application domains are subdivided further into *contexts*. Think of a context as a group of objects that share the same rules of use. These rules include such things as just-in-time activation, security, synchronization, thread affinity, transactions, and security. Under ordinary circumstances, application domains contain only

one context: the *default context*. However, in some situations, the application domain contains additional contexts. Objects that support transactions (provided by COM+), for instance, would be contained in a separate context. These objects are known as *context-bound* objects.

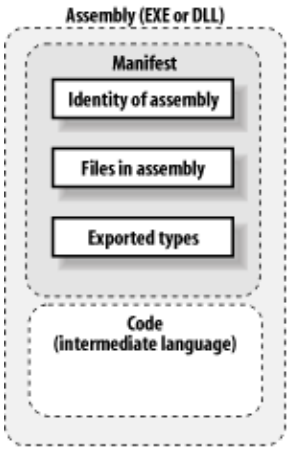
Now that you better understand the execution environment of a .NET executable, let's look at the executable itself.

2.6 Assemblies

When you have an instance of a class, you are said to have an object. A collection of object definitions comprises a component. For instance, *msado15.dll* is a COM component that contains the `Connection`, `Command`, and `Recordset` objects (among others) found in ADO. In .NET, the concept of an *assembly* is roughly analogous to that of a component, but you can think of it as more of a "super component."

In addition to your program's code, assemblies contain a *manifest*, which is a block of metadata that describes everything in the assembly and how it relates to everything else. As shown in Figure 2-3, it contains references to other assemblies that the current assembly might need, as well as a description of the types contained within the assembly. These references make the assembly self-describing, alleviating the need for type libraries and IDL files. In Visual Basic, assemblies can be a single executable with a `Sub Main` entry point or a class library in a DLL.

Figure 2-3. Structure of an assembly



Assemblies can also contain nonexecutable files of any type, similar to a resource file in a traditional Windows executable. The difference is that these additional files do not have to exist as binary information that is part of the executable. Every file that comprises an assembly can retain its individual identity within the filesystem. However, as far as the runtime is concerned, they are a single, cohesive unit. Multimodule assemblies, which contain resources in addition to code, are built using the Assembly Linker utility (al.exe) that is part of the .NET Framework SDK. The VB compiler only emits single module assemblies containing code.

Assemblies are the fundamental units in which code is deployed, version information is specified, and security permissions are defined. An assembly also represents a boundary for the identity of a type. If two different assemblies contain the same type definition, the runtime considers each a different type. This happens irrespective of the namespace the two types are defined within. Remember that namespaces are just mechanisms for organizing type information; the CLR resolves type names through namespaces.

2.6.1 Modules

hello-client.exe and hello.dll are two distinct assemblies. They can be deployed and versioned independently of each other, and each can maintain a different level of security. Instead of compiling hello.vb to a DLL, you can compile it into a *module* as follows:

```
C:\>vbc /t:module hello.vb
```

This compilation produces a file named hello.netmodule.

A module is similar to an assembly, but it is nonexecutable and does not have any of the attributes associated with an executable. It does not maintain a version, for instance. You can recompile the hello client and link the module by using the /addmodule compiler option:

```
C:\>vbc /addmodule:hello.netmodule hello-client.vb
```

One advantage to using modules is that they are compiled and can be distributed in place of source code.

only for RuBoard

2.7 Intermediate Language

VB is not compiled directly into machine code. It is first compiled to a CPU-independent language called Microsoft Intermediate Language (MSIL, or simply IL). You might think this compilation is a throwback to VB's early years as an interpreted language, but the situation is not so grim. The code is not interpreted; eventually, it is converted to machine code at runtime by a just-in-time (JIT) compiler. This happens during execution, as code is needed. Then it is cached as machine code until the process terminates.

The .NET Framework SDK ships with an IL disassembler called ILDASM, which allows you to view the IL produced by the VB compiler (or any .NET compiler, for that matter). This feature can be very useful if you want to see how something in the .NET class library was implemented or to determine what classes are available in a particular library. The hello.dll assembly can be examined by running the IL Disassembler (*ildasm.exe*) from the command line:

```
C:\>ildasm hello.dll
```

From the ILDASM dialog, you can view the manifest and navigate every namespace within the given assembly. As shown in Figure 2-4, ILDASM presents a tree view that allows inspection of the manifest, the various namespaces, classes, and methods contained within the assembly. Example 2-3 contains the entire IL listing for hello.dll, which was produced by selecting File/Dump from the menu.

Figure 2-4. The ILDASM dialog

Example 2-3. The IL dump of hello.dll

```
//  Microsoft (R) .NET Framework IL Disassembler.  Version 1.0.3705.0
//  Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 1:0:3300:0
}
.assembly extern Microsoft.VisualBasic
{
  .publickeytoken = (B0 3F 5F 7F 11 D5 0A 3A )
  .ver 7:0:3300:0
}
.assembly hello
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module hello.dll
// MVID: {8A2071A6-F906-43C1-B6DB-CA5058F54BC4}
.imagebase 0x00400000
.subsystem 0x00000002
.file alignment 512
.corflags 0x00000001
// Image base: 0x03090000
//
// ============== CLASS STRUCTURE DECLARATION ==================
//
.namespace Greeting
{
  .class public auto ansi Hello
         extends [mscorlib]System.Object
  {
  } // end of class Hello

} // end of namespace Greeting

// =============================================================

// =============== GLOBAL FIELDS AND METHODS ===================

// =============================================================

// =============== CLASS MEMBERS DECLARATION ===================
//   note that class flags, 'extends' and 'implements' clauses
//          are provided here for information only

.namespace Greeting
{
  .class public auto ansi Hello
         extends [mscorlib]System.Object
  {
    .method public specialname rtspecialname
            instance void  .ctor( ) cil managed
    {
      // Code size       7 (0x7)
      .maxstack  8
      IL_0000:  ldarg.0
      IL_0001:  call       instance void [mscorlib]System.Object::.ctor( )
      IL_0006:  ret
    } // end of method Hello::.ctor

    .method public instance void  Write(string 'value') cil managed
```