An Introduction to Object-Oriented Programming with Visual Basic .NET

DAN CLARK

Apress™

An Introduction to Object-Oriented Programming with Visual Basic .NET

ISBN (pbk): 1-59059-015-5

Printed and bound in the United States of America 12345678910

Technical Reviewer: Jon Box
Editorial Directors: Dan Appleman, Peter Blackburn, Gary Cornell, Jason Gilmore, Karen Watterson, John Zukowski
Managing Editor: Grace Wong
Project Manager: Alexa Stuart
Copy Editor: Kim Wimpsett
Production Editor: Kari Brooks
Composition: Impressions Book and Journal Services, Inc.
Indexer: Valerie Robbins
Cover Designer: Kurt Krames
Manufacturing Manager: Tom Debolski
Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.
In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com.
Outside the United States, fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section. You will need to answer questions pertaining to this book in order to successfully download the code.

Part One

Object-Oriented Programming and Design Fundamentals

CHAPTER 1

Overview of Object-Oriented Programming

To set the stage for your study of object-oriented programming and Visual Basic .NET, this chapter will briefly look at the history of object-oriented programming and the characteristics of an object-oriented programming language. You will look at why object-oriented programming has become so important in the development of industrial-strength distributed software systems. You will also examine how Visual Basic has evolved into one of the leading business application programming languages.

After reading this chapter you will be familiar with the following:

- The history of object-oriented programming
- Why object-oriented programming has become so important in the development of industrial-strength applications
- The characteristics that make a programming language object-oriented
- The history and evolution of Visual Basic

The History of OOP

Object-Oriented Programming (OOP) is an approach to software development in which the structure of the software is based on *objects* interacting with each other to accomplish a task. This interaction takes the form of messages passing back and forth between the objects. In response to a message, an object can perform an action, or *method*. If you look at how you accomplish tasks in the world around you, you can see that you interact in an object-oriented world. If you want to go to the store, for example, you interact with a car object. A car object consists of other objects that interact with each other to accomplish the task of

getting you to the store. You put the key in the ignition object and turn it. This in turn sends a message (through an electrical signal) to the starter object, which interacts with the engine object to start the car. As a driver, you are isolated from the logic of how the objects of the system work together to start the car. You just initiate the sequence of events by executing the start method of the ignition object with the key. You then wait for a response (message) of success or failure.

Object-oriented programs consist of objects that interact with each other to accomplish a task. Like the real world, users of software programs are isolated from the logic needed to accomplish a task. For example, when you print a page in your word processor, you initiate the action by clicking a print button. You are isolated from the internal processing that has to occur—you just wait for a response telling you if it printed. Internally, the button object interacts with a printer object, which interacts with the printer to accomplish the task of printing the page.

OOP concepts started surfacing in the mid-1960s with a programming language called Simula and further evolved in the 70s with advent of Smalltalk. Although software developers did not overwhelmingly embrace these early advances in OOP languages, object-oriented methodologies continued to evolve. In the mid-80s there was a resurgence of interest in object-oriented methodologies. Specifically, OOP languages such as C++ and Eifle became popular with mainstream computer programmers. OOP continued to grow in popularity in the 90s, most notably with the advent of Java and the huge following it attracted. And in 2002, with the latest version of Visual Studio, Microsoft introduced a new OOP language, C# (pronounced *C-sharp*) and revamped Visual Basic so that it is truly an OOP language.

Why Use OOP?

Why has OOP developed into such a widely used paradigm for solving business problems today? During the 70s and 80s, procedural-oriented programming languages such as C, Pascal, and Fortran were widely used to develop business-oriented software systems. Procedural languages organize the program in a linear fashion—they run from top to bottom. In other words, the program is a series of steps that run one after another. This type of programming worked fine for small programs that consisted of a few hundred code lines, but as programs became larger they became hard to manage and debug.

In an attempt to manage the ever-increasing size of the programs, structured programming was introduced to break down the code into manageable segments called *functions* or *procedures*. This was an improvement, but as programs

performed more complex business functionality and interacted with other systems, the shortcomings of structural programming methodology began to surface:

- Programs became harder to maintain.
- Existing functionality was hard to alter without adversely affecting all of the system's functionality.
- New programs were essentially built from scratch. Consequently, there was little return on the investment of previous efforts.
- Programming was not conducive to team development. Programmers had to know every aspect of how a program worked and could not isolate their efforts on one aspect of a system.
- It was hard to translate business models into programming models.
- It worked well in isolation but did not integrate well with other systems.

In addition to these shortcomings, some evolutions of computing systems caused further strain on the structural program approach:

- Nonprogrammers demanded and were given direct access to programs through the incorporation of graphical user interfaces and their desktop computers.
- Users demanded a more-intuitive, less-structured approach to interacting with programs.
- Computer systems evolved into a distributed model where the business logic, user interface, and backend database were loosely coupled and accessed over the Internet and intranets.

As a result, many business software developers turned to object-oriented methodologies and programming languages to solve these problems. The benefits included the following:

- A more intuitive transition from business analysis models to software implementation models
- The ability to maintain and implement changes in the programs more efficiently and rapidly

- The ability to more effectively create software systems using a team process, allowing specialists to work on parts of the system
- The ability to reuse code components in other programs and purchase components written by third-party developers to increase the functionality of their programs with little effort
- Better integration with loosely coupled distributed computing systems
- Improved integration with modern operating systems
- The ability to create a more intuitive graphical user interface for the users

The Characteristics of OOP

In this section you are going to look at the some fundamental concepts and terms common to all OOP languages. Do not worry about how these concepts get implemented in any particular programming language; that will come later. My goal is to merely familiarize you with the concepts and relate them to your everyday experiences in such a way that they make more sense later when you look at OOP design and implementation.

Objects

If you think about it, you live in an object-oriented world. You are an object. You interact with other objects. To write this book I am interacting with a computer object. When I woke up this morning I was responding to a message sent out by an alarm clock object. In fact, you are an object with data such as height and hair color. You also have methods that you perform or are performed on you—for example, eating and walking.

So what are objects? In OOP terms, an object is a structure for incorporating data and the procedures for working with that data. For example, if you were interested in tracking data associated with products in inventory, you would create a product object that is responsible for maintaining and working with the data pertaining to the products. If you wanted to have printing capabilities in your application, you would work with a printer object that is responsible for the data and methods used to interact with your printers.

Abstraction

When you interact with objects in the world, you are often only concerned with a subset of their properties. Without this ability to abstract or filter out the extraneous properties of objects, you would find it hard to process the plethora of information bombarding you and concentrate on the task at hand.

As a result of *abstraction*, when two different people interact with the same object, they often deal with a different subset of attributes. When I drive my car, for example, I need to know the speed of the car and the direction it is going. Because the car is an automatic, I do not need to know the RPMs of the engine, so I filter this information out. On the other hand, this information would be critical to a racecar driver, who would not filter it out.

When constructing objects in OOP applications, it is important to incorporate this concept of abstraction. If you were building a shipping application, you would construct a product object with attributes such as size and weight. The color of the item would be extraneous information and filtered out. On the other hand, when constructing an order-entry application, the color could be important and would be included as an attribute of the product object.

Encapsulation

Another important feature of OOP is *encapsulation*. Encapsulation is the process in which no direct access is granted to the data; instead, it is hidden. If you want to gain access to the data, you have to interact with the object responsible for the data. In the previous inventory example, if you wanted to view or update information on the products, you would have to work through the product object. To read the data, you would have sent the product object a message. The product object would then read the value and send back a message telling you what the value is. The product object defines what operations can be performed on the product data. If you send a message to modify the data and the product object determines it is a valid request, it will perform the operation for you and send a message back with the result.

You experience encapsulation in your daily life all the time. Think about a human resources department. They encapsulate (hide) the information about employees. They determine how this data can be used and manipulated. Any request for the employee data or request to update the data has to be routed through them. Another example is network security. Any request for the security information or a change to a security policy must be made through a network security administrator. The security data is encapsulated from the users of the network.

By encapsulating data you make the data of your system more secure and reliable. You know how the data is being accessed and what operations are being performed on the data. This makes program maintenance much easier and also greatly simplifies the debugging process. You can also modify the methods used to work on the data, and if you do not alter how the method is requested and the type of response sent back, then you do not have to alter the other objects using the method. Think about when you send a letter in the mail. You make a request to the post office to deliver the letter. How the post office accomplishes this is not exposed to you. If it changes the route it uses to mail the letter, it does not affect how you initiate the sending of the letter. You do not have to know the post office's internal procedures used to deliver the letter.

Polymorphism

Polymorphism is the ability of two different objects to respond to the same request message in their own unique way. For example, I could train my dog to respond to the command bark and my bird to respond to the command chirp. On the other hand, I could train them to both respond to the command speak. Through polymorphism I know that the dog will respond with a bark and the bird will respond with a chirp.

How does this relate to OOP? You can create objects that respond to the same message in their own unique implementations. For example, you could send a print message to a printer object that would print the text on a printer, and you could send the same message to a screen object that would print the text to a window on your computer screen.

Another good example of polymorphism is the use of words in the English language. Words have many different meanings, but through the context of the sentence you can deduce which meaning is intended. You know that someone who says, "Give me a break!" is not asking you to break his leg!

In OOP you implement this type of polymorphism through a process called *overloading*. You can implement different methods of an object that have the same name. The object can then tell which method to implement depending on the context (in other words, the number and type of arguments passed) of the message. For example, you could create two methods of an inventory object to look up the price of a product. Both these methods would be named `getPrice`. Another object could call this method and either pass the name of the product or the product ID. The inventory object could tell which `getPrice` method to run by whether a string value or an integer value was passed with the request.

Inheritance

Most objects are classified according to hierarchies. For example, you can classify all dogs together as having certain common characteristics, such as having four legs and fur. Their breeds further classify them into subgroups with common attributes, such as size and demeanor. You also classify objects according to their function. For example, there are commercial vehicles and recreational vehicles. There are trucks and passenger cars. You classify cars according to their make and model. To make sense of the world, you need to use object hierarchies and classifications.

You use *inheritance* in OOP to classify the objects in your programs according to common characteristics and function. This makes working with the objects easier and more intuitive. It also makes programming easier because it enables you to combine general characteristics into a parent object and inherit these characteristics in the child objects. For example, you can define an employee object that defines all the general characteristics of employees in your company. You can then define a manager object that inherits the characteristics of the employee object but also adds characteristics unique to managers in your company. The manager object will automatically reflect any changes in the implementation of the employee object.

Aggregation

Aggregation is when an object consists of a composite of other objects that work together. For example, your lawn mower object is a composite of the wheel objects, the engine object, the blade object, and so on. In fact, the engine object is a composite of many other objects. There are many examples of aggregation in the world around us. The ability to use aggregation in OOP is a powerful feature that enables you to accurately model and implement business processes in your programs.

The History of Visual Basic

By most accounts, you can trace the origins of Visual Basic to Alan Cooper, an independent software vender. In the late 1980s Cooper was developing a shell construction kit called Tripod. What made Tripod unique was it incorporated a visual design tool that enabled developers to design their Windows interfaces by dragging and dropping controls onto it. Using a visual design tool hid a lot of the complexity of the Windows Application Programming Interface (API) from the developer. The other innovation associated with Tripod was the extensible model it offered programmers. Programmers could develop custom controls and

incorporate them into the Tripod development environment. Up to this point, development tools were, for the most part, closed environments that could not be customized.

Microsoft paid Cooper for the development of Tripod and renamed it Ruby. Although Microsoft never released Ruby as a shell construction kit, it incorporated its form engine with the QuickBasic programming language and developed Thunder, one of the first Rapid Application Development (RAD) tools for Windows programs. Thunder was renamed to Visual Basic, and Visual Basic 1.0 was introduced in the spring of 1991. Visual Basic 1.0 became popular with business application developers because of its ease of use and its ability to rapidly develop prototype applications. Although Visual Basic 1.0 was an innovation in the design of Windows applications, it did not have built-in support for database interactivity. Microsoft realized this was a server limitation and introduced native support for data access in the form of Data Access Objects (DAO) in Visual Basic 3.0. After the inclusion of native data support, the popularity of Visual Basic swelled. It transitioned from being a prototyping tool to being a tool used to develop industrial-strength business applications.

Microsoft has always been committed to developing the Visual Basic language and the Visual Basic Integrated Development Environment (IDE). In fact, by many accounts, Bill Gates himself has taken an active interest in the development and growth of Visual Basic. At one point, the design team did not allow controls to be created and added to the Toolbox. When Bill Gates saw the product demo, he insisted that this extensibility be incorporated into the product. This extensibility brought on the growth of the custom control industry. Third-party vendors began to market controls that made programming an application even easier for Visual Basic developers. For example, a Resize control was marketed that encapsulated the code needed to resize a form and the controls the form contained. A developer could purchase this tool and add it to the Toolbox in the Visual Basic IDE. The developer could then drag the resize control onto the form, and without writing any code, the form and the controls it contained would resize proportionality.

By version 6.0, Visual Basic had evolved into a robust and industrial-strength programming language with an extremely large and dedicated developer base. But as strong as Visual Basic had become as a programming language, many programmers felt it had one major shortcoming. Visual Basic was considered by many to be an object-like programming language—not a true object-oriented programming language. Although Visual Basic 4.0 gave developers the ability to create classes and to package the classes in reusable components, Visual Basic did not incorporate such basic OOP features such as inheritance and method overloading. Without these features, developers were severely limited in their ability to construct complex distributed software systems. Microsoft has recognized these shortcomings and has changed Visual Basic into a true OOP language with the release of Visual Basic .NET.

Summary

In this chapter you became familiar with the following:

- The history of object-oriented programming
- Why object-oriented programming has become so important in the development of industrial-strength applications
- The characteristics that make a programming language object-oriented
- The history and evolution of Visual Basic

Now that you have an understanding of what constitutes an OOP language and why OOP languages are so important to enterprise-level application development, your next step is to become familiar with how OOP applications are designed. Successful applications must be carefully planned and developed before any meaningful coding takes place. The next chapter is the first in a series of three aimed at introducing you to some of the techniques used when designing object-oriented applications. You will look at the process of deciding what objects need to be included in an application and what attributes of these objects are important to the functionality of that application.