CHAPTER 1

Overview of Object-Oriented Programming

To set the stage for your study of object-oriented programming (OOP) and C#, this chapter will look briefly 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 C# has evolved into one of the leading application programming languages.

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

- what object-oriented programming is
- 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 C#

What is OOP?

Object-oriented programming 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.

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 object 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.

Similarly, 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 needs to occur; you just wait for a response telling you if it printed. In the software program, the button object interacts with a printer object, which interacts with the actual printer to accomplish the task of printing the page.
The History of OOP

OOP concepts started surfacing in the mid-1960s with a programming language called Simula and further evolved in the 1970s 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-1980s there was a resurgence of interest in object-oriented methodologies. Specifically, OOP languages such as C++ and Eiffel became popular with mainstream computer programmers. OOP continued to grow in popularity in the 1990s, most notably with the advent of Java and the huge following it attracted. And in 2002, in conjunction with the release of the .NET Framework, Microsoft introduced a new OOP language, C# (pronounced C-sharp) and revamped their widely popular existing language, Visual Basic, so that it is now truly object-oriented. Today OOP languages continue to flourish and are a mainstay of modern programming.

Why Use OOP?

Why has OOP developed into such a widely used paradigm for solving business problems today? During the 1970s and 1980s, procedure-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 following shortcomings of structural programming 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.
- Structural programming 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, such as:

- Nonprogrammers demanded and got 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.