Simple and Efficient Programming with C#

Skills to Build Applications with Visual Studio and .NET

Vaskaran Sarcar

Apress®

Simple and Efficient Programming with C#

Skills to Build Applications with Visual Studio and .NET

Vaskaran Sarcar

Apress®

Simple and Efficient Programming with C#: Skills to Build Applications with Visual Studio and .NET

Vaskaran Sarcar
Kolkata, West Bengal, India

ISBN-13 (pbk): 978-1-4842-7321-0 ISBN-13 (electronic): 978-1-4842-7322-7
https://doi.org/10.1007/978-1-4842-7322-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Smriti Srivastava
Development Editor: Laura Berendson
Coordinating Editor: Shrikant Vishwakarma

Cover designed by eStudioCalamar

Cover image designed by Pexels

Distributed to the book trade worldwide by Springer Science+Business Media LLC, 1 New York Plaza, Suite 4600, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, email orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC, and the sole member (owner) is Springer Science+Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com or visit http://www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-7321-0. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

Dear Reader,

You inspire me with your loving comments.
I get upset by your critical comments. But in every case,
you help me grow into a better person and a better author.
So, you are my teachers. I dedicate this work with love to you.
I also try my best to help you grow.

Table of Contents

About the Author

Vaskaran Sarcar obtained his Master of Engineering in software engineering from Jadavpur University, Kolkata, India, and an MCA from Vidyasagar University, Midnapore, India. He was a National Gate Scholar (2007–2009) and has more than 12 years of experience in education and the IT industry.

Vaskaran devoted his early years (2005–2007) to the teaching profession at various engineering colleges, and later he joined HP India PPS R&D Hub Bangalore. He worked there until August 2019. At the time of his retirement from HP, he was a senior software engineer and team lead. To follow his dream and passion, Vaskaran is now an independent, full-time author. His other Apress books include the following:

- *Design Patterns in C# Second Edition (Apress, 2020)*
- Getting Started with Advanced C# (Apress, 2020)
- *Interactive Object-Oriented Programming in Java Second Edition (Apress, 2019)*
- *Java Design Patterns Second Edition (Apress, 2019)*
- *Design Patterns in C# (Apress, 2018)*
- *Interactive C# (Apress, 2017)*
- *Interactive Object-Oriented Programming in Java (Apress, 2016)*
- *Java Design Patterns (Apress, 2016)*

The following list includes his non-Apress books:

- *Python Bookcamp (Amazon, 2021)*
- *Operating System: Computer Science Interview Series (Createspace, 2014)*

About the Technical Reviewer

Carsten Thomsen is primarily a back-end developer, but works with smaller front-end bits as well. He has authored and reviewed a number of books and created numerous Microsoft Learning courses, all to do with software development. He works as a freelancer/contractor in various countries in Europe; Azure, Visual Studio, Azure DevOps, and GitHub are some of the tools he works with. Being an exceptional troubleshooter—asking the right questions, including the less logical ones, in a most-logical to least-logical fashion—he also enjoys working with architecture, research, analysis, development, testing, and bug fixing. Carsten is a very good communicator with great mentoring and team-lead skills, and fantastic skills in researching and presenting new material.

Acknowledgments

First, I thank the Almighty. I sincerely believe that with HIS blessings only could I complete this book. I also extend my deepest gratitude and thanks to the following:

Ratanlal Sarkar and Manikuntala Sarkar: My dear parents, thanks for all your support towards me.

Indrani, my wife; **Ambika**, my daughter; **Aryaman**, my son: Sweethearts, I love you all.

Sambaran, my brother: Thank you for your constant encouragement toward me.

Carsten: You are a great technical advisor. Whenever I was in need, your support was there. Thank you one more time.

Celestin, Laura, and Smriti: Thanks for giving me another opportunity to work with you and Apress.

Shrikant, Nirmal, Sherly, Sankar and Mohan: Thank you for your exceptional support to finalize my work. Your efforts are extraordinary.

Introduction

Welcome to your journey through *Simple and Efficient Programming with C#: Skills to Build Applications with Visual Studio and .NET.* C# is an object-oriented programming (OOP) language. You may already know C# keywords, or even some interesting features. You may also know how to write simple programs in C#. You can learn these things from an introductory book or an online tutorial. These are useful things to know, but they are not sufficient to understand an enterprise codebase. This is why a novice programmer often finds it difficult to understand an expert's code. He or she wonders why an experienced programmer wrote the program differently. It may appear to the novice that the expert could have used an easier approach to solve the problem. But there are reasons why an experienced programmer might follow a different approach. The word "experienced" indicates that these programmers have more experience in programming and know the pros and cons of different approaches. They know how the C# features can be used in the best possible way to develop an application. So, the applications they make are usually powerful. What do I mean by a powerful application? For me, a powerful application is robust, extensible, and easily maintainable, but simple to use. This book is an introductory guide to develop such applications. This is the core aim of this book.

To write better quality programs, senior programmers follow in experts' footprints. They learn from collective wisdom and recorded experience from the past. So, instead of attempting an entirely new solution, you should first consider this knowledge base, which will help you produce better quality code. It is best to have some idea about why you should or shouldn't follow any specific guideline.

Malcolm Gladwell, in his book *Outliers* (Little, Brown and Company), discussed the 10,000-hour rule. This rule says that the key to achieving world-class expertise in any skill is, to a large extent, a matter of practicing the correct way, for a total of around 10,000 hours. I acknowledge that it is impossible to consider all experiences before you write a program. Also, sometimes it is OK to bend the rules if the return on investment (ROI) is nice. So, keep in mind the ***Pareto*** principle, or ***80-20 rule***. This rule simply states that 80% of outcomes come from 20% of all causes. This is useful in programming too. When you identify the most essential characteristics of top-quality programs and use

them in your applications, you also qualify yourself as an experienced programmer, and your application will be robust, flexible, and maintainable. In this book, I share with you these important principles, which will help you write better programs for case studies. Some of these principles you may know already, but when you see them in action and compare the case studies, you'll understand their importance.

How Is the Book Organized?

The book has four major parts, which are as follows:

- The first three chapters form **Part I**, in which there is a detailed discussion of polymorphism and the use of abstract classes and interfaces. Here, code comments will be examined, and you will learn when to use them effectively. These are the fundamental building blocks for the rest of the book.
- In the world of programming, there is no shortage of programming principles and design guidelines. Each of these suggestions has its own benefits. To become a professional programmer, you do not need to learn everything at the same time. So, in **Part II**, I discuss six design principles, which include SOLID principles and the DRY principle. These are the foundation of well-known design patterns. Once you understand them, you can consider yourself a better programmer.
- The best way of learning is by doing and analyzing case studies. So, in **Part III** of the book, you will see interesting applications that use some well-known patterns. This part gives you hints about how a professional coder develops an enterprise application.
- There is no end to learning. So, **Part IV** includes some interesting topics such as how to prevent memory leaks, how to choose between a static method and an instance method, and some common terms from software development that are not discussed in detail in this book. A quick overview of these topics will help you to be familiar with them when you see them in your future endeavors.

- You can download all the source code for the book from the publisher's website. I have a plan to maintain the "errata," and, if required, I can also make some updates/announcements there. So, I suggest that you visit those pages to receive any important corrections or updates.

Prerequisite Knowledge

This book is intended for those who are familiar with the basic language constructs of C# and have an idea about pure object-oriented concepts like polymorphism, inheritance, abstraction, encapsulation, and, most important, how to compile or run a C# application in Visual Studio. This book does not invest time in easily available topics, such as how to install Visual Studio on your system, or how to write a "Hello World" program in C#, or how you can use an if-else statement or a while loop, etc. This book is written using the most basic features of C# so that for most of the programs herein you do not need to be familiar with advanced topics in C#. The examples are simple and straightforward. I believe that they are written in such a way that even if you are familiar with another popular language such as Java, C++, and so on, you can still easily grasp the concepts in this book.

Who Is This Book For?

In short, you can pick up this book if the answer is "yes" to the following questions:

- Are you familiar with basic constructs in C# and object-oriented concepts like polymorphism, inheritance, abstraction, and encapsulation?
- Do you know how to set up your coding environment?
- Have you completed at least one basic course on C# and now are interested in writing better programs? Are you also interested to know how a professional programmer designs his or her applications?
- Are you interested in knowing how the core constructs of C# work behind standard design patterns?

You probably shouldn't pick this book if the answer is yes to any of the following questions:

- Are you absolutely new to C#?
- Are you looking for advanced concepts in C#, excluding the topics mentioned previously?
- Are you interested in exploring a book where the focus is not on standard design principles?
- "I do not like Windows, Visual Studio, and/or .NET. I want to learn and use C# without them only." —Is this statement true for you?

Guidelines for Using This Book

To use this book more effectively, consider the following:

- This book works best if you've gone through an introductory course on C# and are familiar with the common terms, such as polymorphism, and have heard about abstract classes and interfaces. If this is not the case, please read about these topics before you start reading this book.
- I suggest you go through the chapters sequentially. This is because some fundamental design techniques may have been discussed in a previous chapter and I have not repeated those techniques in later chapters.
- I started this book using ***Microsoft Visual Studio Community 2019 (Version 16.8.4) in a Windows 10 environment***. This community edition is free of charge. If you do not use the Windows operating system, you can use Visual Studio Code, which is also a source-code editor developed by Microsoft to support Windows, Linux, or Mac operating systems. This multi-platform IDE is also free. When I started the book, I started with the latest versions of C# that were available at that time. In this context, it is useful to know that

nowadays the C# language version is automatically selected based on your project's target framework(s), so you can always get the highest compatible version by default. In the latest versions, Visual Studio doesn't support changing the version value in the user interface, but you can change it by editing the `csproj` file.

- Later, I also used ***Microsoft Visual Studio Community 2019 Preview 4.0 and set my target framework to .NET 6.0***. As per the new rule, you can simply say that when your target framework is .NET 5.x (and later), you'll get C# 9.0 and later by default. If you are interested in the C# language versioning, you can go to this link: `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version`.
- Version updates will come continuously, but I strongly believe that these version details should not matter much to you because I have used the fundamental constructs of C#. So, the code in this book should execute smoothly in the upcoming versions of C#/Visual Studio as well. Though I also believe that the results should not vary in other environments, you know the nature of software—it is naughty. So, I recommend that if you want to see the exact same output, you mimic the same environment.
- You can download and install the Visual Studio IDE from `https://visualstudio.microsoft.com/downloads/`. You are expected to see the screen shown in Figure 1.

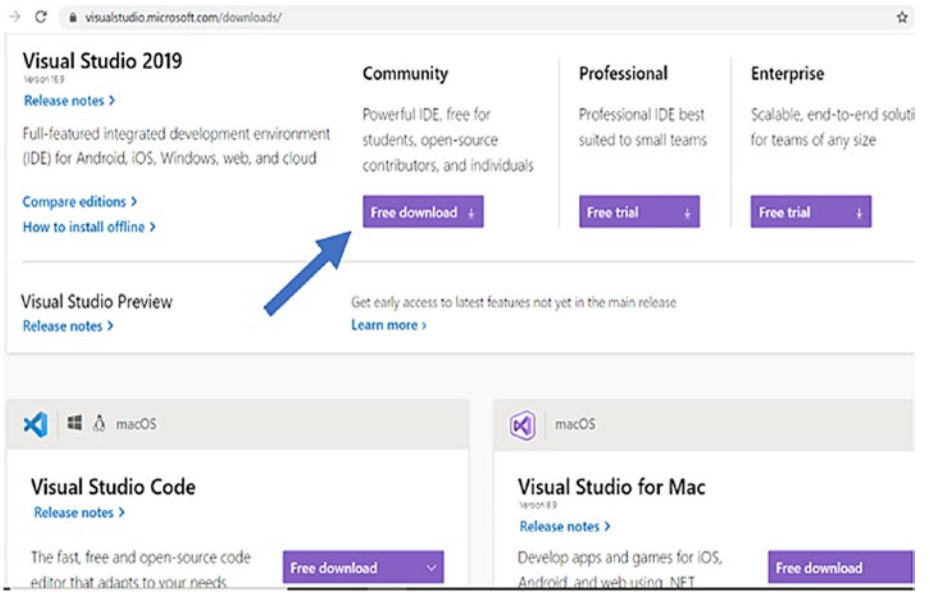

Figure 1. *Download link for Visual Studio 2019 and Visual Studio Code*

Note At the time of writing, this link works fine, and the information is correct. But the link and policies may change in the future.

- I have installed the class designer component in Visual Studio 2019 to draw class diagrams for my programs. But I needed to edit some of these diagrams for better readability. For example, I added some valuable notes in some diagrams, so that you can understand them easily.

Conventions Used in This Book

Here, I will mention only two points: In many places, to avoid more typing, I have used the word "his" only. Please treat it as "his" or "her," whichever is applicable for you.

Second, all the outputs and codes of the book follow the same font and structure. To draw your attention, in some places, I have made them bold. For example, consider the following code fragment (taken from Chapter 4 when I discuss LSP) and the lines in bold.

```
...
// Instantiating two registered users
RegisteredUser robin = new RegisteredUser("Robin");
RegisteredUser jack = new RegisteredUser("Jack");

// Adding the users to usermanager
helper.AddUser(robin);
helper.AddUser(jack);

GuestUser guestUser1 = new GuestUser();
helper.AddUser(guestUser1);

// Processing the payments using
// the helper class.
// You can see the problem now.
....
```

Final Words

I must say that you are an intelligent person. You have chosen a subject that can assist you throughout your career. If you are a developer/programmer, you need these concepts. If you are an architect of a software organization, you need these concepts. If you are a college student, you need these concepts, not only to score well on exams but also to enter the corporate world. Even if you are a tester who needs to take care of white-box testing or simply needs to know about the code paths of a product, these concepts will help you a lot.

Remember that you have just started on this journey. As you learn about these concepts, I suggest you write your own code; only then will you master this area. There is no shortcut for this. Do you remember Euclid's reply to the ruler? If not, let me remind you of his reply: ***There is no royal road to geometry.*** So, study and code; understand a new concept and code again. Do not give up when you face challenges. They are the indicators that you are growing better.

I believe that this book is designed for you in such a way that upon its completion, you will have developed an adequate knowledge of the topic, and, most important, you'll know how to go further.

Lastly, I hope that this book can provide help to you and that you will value the effort.

PART I

Fundamentals

Part I consists of three chapters, in which we will discuss the following questions:

- How can we use the power of polymorphism and why is it beneficial?
- How can we combine an abstract class and interfaces to make an efficient application?
- How can we use meaningful code comments and avoid unnecessary comments in a program?

Almost every C# application uses comments, the concept of polymorphism, and abstract classes and interfaces. When we implement these techniques in a better way, the program is better. I consider them the fundamental techniques for an efficient application.

CHAPTER 1

Flexible Code Using Polymorphism

Ask a developer, "What are the fundamental characteristics of object-oriented programming (OOP)?" and you will get an immediate reply saying, "Classes (and objects), inheritance, abstraction, encapsulation, and polymorphism are the most important characteristics in OOP". In addition, when you analyze enterprise code that is based on OOP, you'll find different forms of polymorphism. But the truth is, a novice programmer rarely uses the power of polymorphism. This chapter focuses on this topic. It shows you some simple but powerful code examples using this principle.

Recap

Polymorphism simply means there is *one name with many forms*. Consider the behavior of your pet dog. When it sees an unknown person, it starts barking. But when it sees you, it makes different noises and behaves differently. In both cases, this dog sees with his eyes but based on his observations he behaves differently. Polymorphic code can work in the same way. Consider a method that you might use to add some operands. If the operands were integers, you would expect to get a sum of the integers. But if you were to deal with string operands, you would expect to get a concatenated string.

Initial Program

Let's look at a program that compiles and runs successfully. In this program, there are three different types of animals: `tigers, dogs`, and `monkeys`. Each of them can produce a different sound. So, there are classes with these names, and in each class, there is a `Sound()` method. See whether you can improve this program.

V. Sarcar, *Simple and Efficient Programming with C#*, https://doi.org/10.1007/978-1-4842-7322-7_1

Demonstration 1

Here is a program that does not use the concept of polymorphism.

```
using System;

namespace DemoWithoutPolymorphism
{
    class Tiger
    {
        public void Sound()
        {
            Console.WriteLine("Tigers roar.");
        }
    }
    class Dog
    {
        public void Sound()
        {
            Console.WriteLine("Dogs bark.");
        }
    }
    class Monkey
    {
        public void Sound()
        {
            Console.WriteLine("Monkeys whoop.");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***Sounds of the different animals.***");
            Tiger tiger = new Tiger();
            tiger.Sound();
```

```
            Dog dog = new Dog();
            dog.Sound();
            Monkey monkey = new Monkey();
            monkey.Sound();
            Console.ReadKey();
        }
    }
}
```

Output

```
***Sounds of the different animals.***
Tigers roar.
Dogs bark.
Monkeys whoop.
```

Analysis

When you use `Tiger tiger = new Tiger();` the tiger is a reference to an object that is based on the `Tiger` class. This reference refers to the object but does not contain the object data itself. Even `Tiger tiger;` is also a valid line of code that tells you to create an object reference without creating the object.

Understand that when you use `Tiger tiger = new Tiger();` ***you are programming to an implementation***. Notice that in this case the reference and object are both of the same types. You can improve this program using the concept of polymorphism. In the upcoming implementation, I show you such an example. I use an interface in this example. I can achieve the same thing using an abstract class too. Before I show you the example, let me remind you of few important points:

- When you use an abstract class or an interface, the first thing that comes to mind is inheritance. How do you know whether you have correctly used inheritance? The simple answer is: you do an IS-A test. For example, a rectangle IS-A shape, but the reverse is not necessarily true. Take another example: a monkey IS-An animal but not all animals are monkeys. Notice that the IS-A test is unidirectional.

- In programming, if you inherit class B from class A, you say that B is the subclass and A is the parent class or base class. **But most important, you can say B is a type of A**. So, if you derive a `Tiger` class or a `Dog` class from a base class called `Animal` (or an interface, say, `IAnimal`), you can say that `Dog` IS-An `Animal` (or `IAnimal`) or `Tiger` IS-An `Animal` (or `IAnimal`).
- If you have an inheritance tree, this IS-A test can be applied anywhere in the tree. For example, a rectangle IS-A special type of shape. A square IS-A special type of rectangle. So, a square IS-A shape too.
- Let us say we represent rectangles and shapes using the `Rectangle` and `Shape` classes, respectively. Now when we say `Rectangle` IS-A `Shape`, programmatically we mean a `Rectangle` instance can invoke the methods that a `Shape` instance can invoke. If needed, a `Rectangle` instance can invoke some additional methods too. These additional methods can be defined in the `Rectangle` class.

You know that a superclass reference can refer to a subclass object. Here you see that each `tiger, dog,` or `monkey` is an animal. So, you can introduce a supertype and inherit all these concrete classes from it. Let's name the supertype `IAnimal`.

Here is a code fragment that shows the `IAnimal` interface. It also gives you the idea of how you can override its `Sound()` method in the `Tiger` class. The `Monkey` and `Dog` classes can do the same thing.

```
interface IAnimal
{
    void Sound();
}
class Tiger : IAnimal
{
    public void Sound()
    {
        Console.WriteLine("Tigers roar.");
    }
}
```

Programming to a supertype gives you more flexibility. It allows you to use a reference variable polymorphically. The following code segment demonstrates such a usage:

```
IAnimal animal = new Tiger();
animal.Sound();
animal = new Dog();
animal.Sound();
//remaining code skipped
```

Better Program

I have rewritten this program, which produces the same output. Let's have a look at the following demonstration.

Demonstration 2

This is a modified version of Demonstration 1.

```
using System;

namespace UsingPolymorphism
{
    interface IAnimal
    {
        void Sound();
    }
    class Tiger: IAnimal
    {
        public void Sound()
        {
            Console.WriteLine("Tigers roar.");
        }
    }
```

```
    class Dog: IAnimal
    {
        public void Sound()
        {
            Console.WriteLine("Dogs bark.");
        }
    }
    class Monkey: IAnimal
    {
        public void Sound()
        {
            Console.WriteLine("Monkeys whoop.");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***Sounds of the different animals.***");

            IAnimal animal = new Tiger();
            animal.Sound();
            animal = new Dog();
            animal.Sound();
            animal = new Monkey();
            animal.Sound();

            Console.ReadKey();
        }
    }
}
```

Analysis

Have you noticed the difference? Inside the `Main()` method, you use the superclass reference `animal` to refer to different derived class objects.

Now you not only type less, but you also use a program that is more flexible and easier to maintain. If you want, you could also iterate over a list. For example, you could replace the following code segment inside `Main()`:

```
IAnimal animal = new Tiger();
animal.Sound();
animal = new Dog();
animal.Sound();
animal = new Monkey();
animal.Sound();
```

with the following code:

```
List<IAnimal> animals = new List<IAnimal>
{
    new Tiger(),
    new Dog(),
    new Monkey()
};

foreach (IAnimal animal in animals)
animal.Sound();
```

If you run the program again with these changes, you will see the same output.

POINT TO REMEMBER

When you use `List<Animal>`, do not forget to include the following namespace at the beginning of the program:

```
using System.Collections.Generic;
```

This discussion is not over yet. Here, I have used one of the simplest forms of polymorphism. In this case, a thought may come into your mind: we know a supertype reference can refer to a subtype object in C#. So, when I use the following lines:

```
IAnimal animal = new Tiger();
animal.Sound();
```