Microsoft

Microsoft Visual C# Step by Step

Tenth Edition

Professional

John Sharp

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Microsoft

Microsoft Visual C# Step by Step

Tenth Edition

John Sharp

MICROSOFT VISUAL C# STEP BY STEP, TENTH EDITION
Published with the authorization of Microsoft Corporation by:
Pearson Education, Inc.

Copyright © 2022 by Agylia Group Ltd.

All rights reserved. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms, and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearson.com/permissions.

No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-13-761983-2
ISBN-10: 0-13-761983-9

Library of Congress Control Number: 2022930224

ScoutAutomatedPrintCode

TRADEMARKS
Microsoft and the trademarks listed at http://www.microsoft.com on the "Trademarks" webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

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

SPECIAL SALES
For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries,
please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S.,
please contact intlcs@pearson.com.

Editor-in-Chief: Brett Bartow

Executive Editor: Loretta Yates

Sponsoring Editor: Charvi Arora

Development Editor: Kate Shoup

Managing Editor: Sandra Schroeder

Senior Project Editor: Tracey Croom

Production Editor: Dan Foster

Copy Editor: Dan Foster

Indexer: Valerie Haynes Perry

Proofreader: Kim Wimpsett

Technical Editor: David Franson

Editorial Assistant: Cindy Teeters

Cover Designer: Twist Creative, Seattle

Compositor: Danielle Foster

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

Please contact us with concerns about any potential bias at *https://www.pearson.com/report-bias.html*.

Contents at a Glance

Contents

Acknowledgments

Hoo boy! Welcome to the 10th edition. In the acknowledgments to previous editions, I have made references to painting the Forth Railway Bridge and Sisyphus pushing the rock as never-ending tasks. In the future, maybe the role of updating *Microsoft C# Step By Step* will be added to this legendary list. That said, writing and updating books is far more rewarding than wielding a brush or rolling a stone up a hill forever and a day, with the added bonus that I can retire at some point.

Despite the fact that my name is on the cover, authoring a book such as this is far from a one-man project. I'd like to thank the following people who have provided unstinting support and assistance throughout this endeavor.

First, Loretta Yates at Pearson Education, who took on the role of prodding me into action and ever-so-gently tying me down to well-defined deliverables and hand-off dates. Without her initial impetus and cajoling, this project would not have gotten off the ground.

Next, Charvi Arora and her tireless team of editors, especially Kate Shoup and Dan Foster, who ensured that my grammar remained at least semi-acceptable and picked up on the missing words and nonsense phrases in the text. Also, David Fransen, who had the unenviable task of reviewing and testing the code and exercises. I know from experience that this can be a thankless and frustrating task, but the hours spent and the resulting feedback can only make for a better book. Of course, any errors that remain are entirely my responsibility, and I am happy to listen to feedback from any reader.

As ever, I must also thank Diana, my better half, who keeps me sane, fed, and watered. During Covid-19 lockdown, she felt that our house wasn't crowded enough, so she brought two rather manic kittens into the family. The dogs are now terrified, but we have endless hours of fun putting the curtains back up and playing "hunt the mouse/frog/spider or whatever they have captured and brought indoors." I wouldn't have home-life any other way.

And lastly, to James and Frankie, who have both now flown the nest. James has spent the last couple of years working for the British government in Manila (he says). Judging by the photos, it seems more like he has been on a touring holiday of the beaches of Southeast Asia. Frankie has remained closer to home so she can pop in and catch the mice/frogs/spiders from time to time. By the way, to those developers she manages at her place of work, it's time for you to make her a cup of tea!

About the author

JOHN SHARP is a principal technologist for CM Group Ltd, part of the Civica Group, a software development and consultancy company in the United Kingdom. He is well versed as a software consultant, developer, author, and trainer, with more than 35 years of experience, ranging from Pascal programming on CP/M and C/Oracle application development on various flavors of UNIX to the design of C# and JavaScript distributed applications and development on Windows 11 and Microsoft Azure. He also spends much of his time writing courseware for Microsoft, focusing on areas such as data science using R and Python, big data processing with Spark and CosmosDB, SQL Server, NoSQL, web services, Blazor, cross-platform development with frameworks such as Xamarin and MAUI, and scalable application architectures with Azure.

Introduction

A lot has changed in the last 20 years. For a laugh, I sometimes retrieve my copy of *Microsoft C# Step By Step,* first edition, released in 2001, and wonder at my naive innocence back in those days. Surely, C# was the peak of programming language perfection at that time. C# and the .NET Framework hit the world of development with a bang, and the reverberations continue to this day. However, rather than dying away, they rumble through software development with increased significance. Rather than being a single-platform approach as the naysayers of 2001 originally screamed, C# and .NET have shown themselves to be a complete multiplatform solution, whether you're building applications for Windows, macOS, Linux, or Android. Additionally, C# and .NET have proved themselves the runtime of choice for many cloud-based systems. Where would Azure be without them?

In the past, most common programming languages went through occasional updates, often spread several years apart. For example, if you look at Fortran, you will see standards named Fortran 66, Fortran 77, Fortran 90, Fortran 95, Fortran 2003, Fortran 2008, and Fortran 2018. That's seven updates in the last 55 years. While this relatively slow cycle of change promotes stability, it can also lead to stagnation. The issue is that the nature of problems that developers must address changes rapidly, and the tools they depend on should ideally keep pace so that they can develop effective solutions. Microsoft .NET provides a continually evolving framework, and C# undergoes frequent updates to make the best use of the platform. So, in contrast to Fortran, C# has undergone a rapid evolution since it was first released—six versions in the last five years alone, with another update due in 2022. The C# language still supports code written 20+ years ago, but these days the additions and enhancements to the language enable you to create solutions using more elegant code and concise constructs. For this reason, I make periodic updates to this book; this is now the 10th edition!

If you're interested, the following list contains a brief history of C#:

- C# 1.0 made its public debut in 2001.
- C# 2.0, with Visual Studio 2005, provided several important new features, including generics, iterators, and anonymous methods.
- C# 3.0, which was released with Visual Studio 2008, added extension methods, lambda expressions, and, most famously of all, the Language-Integrated Query (LINQ) facility.

- C# 4.0, released in 2010, provided further enhancements that improved its interoperability with other languages and technologies. These features included support for named and optional arguments and the dynamic type, which indicates that the language runtime should implement late binding for an object. Important additions to the .NET Framework, released concurrently with C# 4.0, were the classes and types that constitute the Task Parallel Library (TPL). Using the TPL, you can build highly scalable applications that can take full advantage of multicore processors.

- C# 5.0 added native support for asynchronous task-based processing through the `async` method modifier and the `await` operator.

- C# 6.0 was an incremental upgrade with features designed to make life simpler for developers. These features included items such as string interpolation (you need never use `String.Format` again!), enhancements to the ways in which properties are implemented, expression-bodied methods, and others.

- C# 7.0 through 7.3 added further enhancements to aid productivity and remove some of the minor anachronisms of C#. For example, these versions enabled you to implement property accessors as expression-bodied members, methods can return multiple values in the form of tuples, the use of `out` parameters was simplified, and `switch` statements were extended to support pattern- and type-matching. These versions of the language also included many other smaller tweaks to address concerns that many developers had, such as allowing the `Main` method to be asynchronous.

- C# 8.0, C# 9.0, and C# 10.0 continue this theme of enhancing the language to improve readability and aid developer productivity. Some major additions included records, which you can use to build immutable reference types; extensions to pattern matching, enabling you to use this feature throughout the language and not just in `switch` statements; top-level statements, which enable you to use C# as a scripting language (you don't always need to write a `Main` method); default interface methods; static local functions; asynchronous disposable types; and many other features, which are covered in this book.

It goes without saying that Microsoft Windows is an important platform for running C# applications, but now you can also run code developed by using C# on other operating systems, such as Linux, through the .NET runtime. This opens up possibilities for writing code that can run in multiple environments. Additionally, Windows supports highly interactive applications that can share data and collaborate as well as connect to services running in the cloud. The key notion in Windows is Universal Windows Platform (UWP) apps—applications designed to run on any Windows 10 or Windows 11 device, whether a full-fledged desktop system, a laptop, a tablet, or even an Internet of Things (IoT) device with limited resources. Once you've mastered the core features of C#, gaining the skills to build applications that can run on all these platforms is critical.

The cloud has become such an important element in the architecture of many systems—ranging from large-scale enterprise applications to mobile apps running on portable devices—that I decided to focus on this aspect of development in the final chapter of the book.

The development environment provided by Visual Studio makes these features easy to use, and the many new wizards and enhancements included in the latest version of Visual Studio can greatly improve your productivity as a developer. I hope you have as much fun working through this book as I had writing it!

Who should read this book

This book assumes that you are a developer who wants to learn the fundamentals of programming with C# by using Visual Studio and the .NET version 6 or later. By the time you complete this book, you will have a thorough understanding of C# and will have used it to build responsive and scalable applications that can run on the Windows operating system.

Who should not read this book

This book is aimed at developers new to C# but not completely new to programming. As such, it concentrates primarily on the C# language. This book is not intended to provide detailed coverage of the multitude of technologies available for building enterprise-level and global applications for Windows, such as ADO.NET, ASP.NET, Azure, or Windows Communication Foundation. If you require more information on any of these items, you might consider reading some of the other titles available from Microsoft Press.

Finding your best starting point in this book

This book is designed to help you build skills in several essential areas. You can use this book if you're new to programming or if you're switching from another programming language such as C, C++, Java, or Visual Basic. Use the following table to find your best starting point.

If you are	Follow these steps
New to object-oriented programming	**1.** Install the practice files as described in the upcoming section, "Code samples." **2.** Work through Chapters 1 to 22 sequentially. **3.** Complete Chapters 23 to 27 as your level of experience and interest dictates.
Familiar with procedural programming languages, such as C, but new to C#	**1.** Install the practice files as described in the upcoming section, "Code samples." **2.** Skim the first five chapters to get an overview of C# and Visual Studio 2022, and then concentrate on Chapters 6 through 22. **3.** Complete Chapters 23 to 27 as your level of experience and interest dictates.
Migrating from an object-oriented language such as C++ or Java	**1.** Install the practice files as described in the upcoming section, "Code samples." **2.** Skim the first seven chapters to get an overview of C# and Visual Studio 2022, and then concentrate on Chapters 8 through 22. **3.** For information about building Universal Windows Platform applications, read Chapters 23 to 27.
Switching from Visual Basic to C#	**1.** Install the practice files as described in the upcoming section, "Code samples." **2.** Work through Chapters 1 to 22 sequentially. **3.** For information about building Universal Windows Platform applications, read Chapters 23 to 27. **4.** Read the "Quick reference" sections at the end of the chapters for information about specific C# and Visual Studio 2022 constructs.
Referencing the book after working through the exercises	**1.** Use the index or the table of contents to find information about particular subjects. **2.** Read the "Quick reference" sections at the end of each chapter to find a brief review of the syntax and techniques presented in the chapter.

Most of the book's chapters include hands-on samples that let you try out the concepts you just learned. No matter which sections you choose to focus on, be sure to download and install the sample applications on your system.

Conventions and features in this book

This book presents information by using conventions designed to make the information readable and easy to follow.

- Each exercise consists of a series of tasks, presented as numbered steps (1, 2, and so on) listing each action you must take to complete the exercise.
- Boxed elements with labels such as "Note," "Tip," "Important," and "More Info" provide additional information or alternative methods for completing a step successfully.

- Text that you type (apart from code blocks) and screen elements you select appear in bold.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key while you press the Tab key.

System requirements

You will need the following hardware and software to complete the practice exercises in this book:

- Windows 10 (Home, Professional, Education, or Enterprise) or Windows 11 (Home, Professional, Education, or Enterprise).
- The most recent build of Visual Studio Community 2022, Visual Studio Professional 2022, or Visual Studio Enterprise 2022. (Make sure that you have installed any updates.) As a minimum, you should select the following workloads when installing Visual Studio 2022:
  - Universal Windows Platform development
  - .NET desktop development
  - ASP.NET and web development
  - Azure development
  - Data storage and processing
  - .NET Core cross-platform development

Note All the exercises and code samples in this book have been developed and tested using Visual Studio Community 2022. They should all work, unchanged, in Visual Studio Professional 2022 and Visual Studio Enterprise 2022.

- 1.8 GHz or faster 64-bit processor; quad-core or better recommended. ARM processors are not supported.
- 4 GB of RAM.
- Hard disk space: minimum of 850 MB up to 210 GB of available space, depending on features installed; typical installations require 20 to 50 GB of free space.

- Video card that supports a minimum display resolution of 720p (1280 by 720); Visual Studio will work best at a resolution of WXGA (1366 by 768) or higher.
- Internet connection to download software or chapter examples.

Depending on your Windows configuration, you might require local administrator rights to install or configure Visual Studio.

You also need to enable developer mode on your computer to be able to create and run UWP apps. For details on how to do this, see "Enable Your Device for Development," at *https://msdn.microsoft.com/library/windows/apps/dn706236.aspx*.

Code samples

Most of the chapters in this book include exercises with which you can interactively try out new material learned in the main text. You can download all the sample projects, in both their pre-exercise and post-exercise formats, from the following page:

MicrosoftPressStore.com/VisualCsharp10e/downloads

Installing the code samples

Follow these steps to install the code samples on your computer so that you can use them with the exercises in this book:

1. Unzip the CSharpSBS.zip file that you downloaded from the book's website, extracting the files into your Documents folder.
2. If prompted, review the end-user license agreement. If you accept the terms, select the Accept option and then click Next.

Note If the license agreement doesn't appear, you can access it from the same webpage where you downloaded the CSharpSBS.zip file.

Using the code samples

Each chapter in this book explains when and how to use the code samples for that chapter. When it's time to use a code sample, the book will list the instructions for how to open the files.

Important Many of the code samples depend on NuGet packages that are not included with the code. These packages are downloaded automatically the first time you build a project. As a result, if you open a project and examine the code before doing a build, Visual Studio might report a large number of errors for unresolved references. Building the project will resolve these references, and the errors should disappear.

If you'd like to know all the details, here's a list of the sample Visual Studio projects and solutions, grouped by the folders in which you can find them. In many cases, the exercises provide starter files and completed versions of the same projects that you can use as a reference. The completed projects for each chapter are stored in folders with the suffix "- Complete."

Project/Solution	Description
Chapter 1	
HelloWorld	This project gets you started. It steps through the creation of a simple program using a text editor. The program displays a text-based greeting.
HelloWorld2	This project demonstrates how to use the .NET Command Level Interface (CLI) to build and run a simple C# application.
TestHello	This is a Visual Studio project that displays a greeting.
HelloUWP	This project opens a window that prompts the user for his or her name and then displays a greeting.
Chapter 2	
PrimitiveDataTypes	This project demonstrates how to declare variables by using each of the primitive types, how to assign values to these variables, and how to display their values in a window.
MathOperators	This program introduces the arithmetic operators (+ – * / %).
Chapter 3	
Methods	In this project, you'll reexamine the code in the MathOperators project and investigate how it uses methods to structure the code.
DailyRate	This project walks you through writing your own methods, running the methods, and stepping through the method calls by using the Visual Studio 2015 debugger.
DailyRate Using Optional Parameters	This project shows you how to define a method that takes optional parameters and call the method by using named arguments.
Factorial	This project demonstrates a recursive method that calculates the factorial of a number.

Project/Solution	Description
Chapter 4	
Selection	This project shows you how to use a cascading if statement to implement complex logic, such as comparing the equivalence of two dates.
SwitchStatement	This simple program uses a switch statement to convert characters into their XML representations.
SwitchStatement using Pattern Matching	This is an amended version of the SwitchStatement project that uses pattern matching to simplify the logic in the switch statement.
Chapter 5	
WhileStatement	This project demonstrates a while statement that reads the contents of a source file one line at a time and displays each line in a text box on a form.
DoStatement	This project uses a do statement to convert a decimal number to its octal representation.
Chapter 6	
MathOperators	This project revisits the MathOperators project from Chapter 2 and shows how various unhandled exceptions can make the program fail. The try and catch keywords then make the application more robust so that it no longer fails.
Chapter 7	
Classes	This project covers the basics of defining your own classes, complete with public constructors, methods, and private fields. It also shows how to create class instances by using the new keyword and how to define static methods and fields.
Chapter 8	
Parameters	This program investigates the difference between value parameters and reference parameters. It demonstrates how to use the ref and out keywords.
Chapter 9	
StructsAndEnums	This project defines a struct type to represent a calendar date.
Chapter 10	
Cards	This project shows how to use arrays to model hands of cards in a card game.
Chapter 11	
ParamsArray	This project demonstrates how to use the params keyword to create a single method that can accept any number of int arguments.
Chapter 12	
Vehicles	This project creates a simple hierarchy of vehicle classes by using inheritance. It also demonstrates how to define a virtual method.
ExtensionMethod	This project shows how to create an extension method for the int type, providing a method that converts an integer value from base 10 to a different number base.

Project/Solution	Description
Chapter 13	
Drawing	This project implements part of a graphical drawing package. The project uses interfaces to define the methods that drawing shapes expose and implement.
Chapter 14	
GarbageCollectionDemo	This project shows how to implement exception-safe disposal of resources by using the Dispose pattern.
Chapter 15	
Drawing Using Properties	This project extends the application in the Drawing project developed in Chapter 13 to encapsulate data in a class by using properties.
AutomaticProperties	This project shows how to create automatic properties for a class and use them to initialize instances of the class.
Student enrollment	This project demonstrates how to use records to model structured immutable types.
Chapter 16	
Indexers	This project uses two indexers: one to look up a person's phone number when given a name and the other to look up a person's name when given a phone number.
Chapter 17	
BinaryTree	This solution shows you how to use generics to build a type-safe structure that can contain elements of any type.
BuildTree	This project demonstrates how to use generics to implement a type-safe method that can take parameters of any type.
Chapter 18	
Cards	This project updates the code from Chapter 10 to show how to use collections to model hands of cards in a card game.
Chapter 19	
BinaryTree	This project shows you how to implement the generic IEnumerator<T> interface to create an enumerator for the generic Tree class.
IteratorBinaryTree	This solution uses an iterator to generate an enumerator for the generic Tree class.
Chapter 20	
Delegates	This project shows how to decouple a method from the application logic that invokes it by using a delegate. The project is then extended to show how to use an event to alert an object to a significant occurrence, and how to catch an event and perform any processing required.
Chapter 21	
QueryBinaryTree	This project shows how to use LINQ queries to retrieve data from a binary tree object.

Project/Solution	Description
Chapter 22	
ComplexNumbers	This project defines a new type that models complex numbers and implements common operators for this type.
Chapter 23	
GraphDemo	This project generates and displays a complex graph on a UWP form. It uses a single thread to perform the calculations.
Parallel GraphDemo	This version of the GraphDemo project uses the Parallel class to abstract out the process of creating and managing tasks.
GraphDemo with Cancellation	This project shows how to implement cancellation to halt tasks in a controlled manner before they have completed.
ParallelLoop	This application provides an example showing when you should not use the Parallel class to create and run tasks.
Chapter 24	
GraphDemo	This is a version of the GraphDemo project from Chapter 23 that uses the async keyword and the await operator to perform the calculations that generate the graph data asynchronously.
PLINQ	This project shows some examples of using PLINQ to query data by using parallel tasks.
CalculatePI	This project uses a statistical sampling algorithm to calculate an approximation for pi. It uses parallel tasks.
ParallelTest	This program illustrates the dangers of allowing uncontrolled data access to shared data by parallel threads.
Chapter 25	
Customers	This project implements a scalable user interface that can adapt to different device layouts and form factors. The user interface applies XAML styling to change the fonts and background image displayed by the application.
Chapter 26	
DataBinding	This is a version of the Customers project that uses data binding to display customer information retrieved from a data source in the user interface. It also shows how to implement the INotifyPropertyChanged interface so that the user interface can update customer information and send these changes back to the data source.
ViewModel	This version of the Customers project separates the user interface from the logic that accesses the data source by implementing the Model-View-ViewModel pattern.

Project/Solution	Description
Chapter 27	
Web Service	This solution includes a web application that provides a REST web service that the Customers application uses to retrieve customer information and modify data held in a SQL Server database. The web service uses the Entity Framework to access the database. The database and the web service run using Azure.
Customers with insert and update features	This solution contains an updated version of the Customers project that uses the REST web service to create new customers and modify the details of existing customers.

Errata and book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at:

MicrosoftPressStore.com/VisualCsharp10e/errata

If you discover an error that is not already listed, please submit it to us at the same page.

For additional book support and information, please visit:

MicrosoftPressStore.com/Support

Please note that product support for Microsoft software and hardware is not offered through the previous addresses. For help with Microsoft software or hardware, go to:

http://support.microsoft.com

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

CHAPTER 8

Understanding values and references

After completing this chapter, you will be able to:

- Explain the differences between a value type and a reference type.
- Understand `null` values and how nullable types work.
- Modify how arguments are passed as method parameters by using the `ref` and `out` keywords.
- Describe how computer memory is organized to support value types and reference types.
- Convert a value into a reference by using boxing.
- Convert a reference back to a value by using unboxing and casting.

Chapter 7, "Creating and managing classes and objects," demonstrated how to declare your own classes and how to create objects by using the `new` keyword. That chapter also showed you how to initialize an object by using a constructor. In this chapter, you'll learn how the characteristics of the primitive types such as `int`, `double`, and `char` differ from the characteristics of class types.

Copying value type variables and classes

Most of the primitive types built into C#, such as `int`, `float`, `double`, and `char` (but not `string`, for reasons that will be covered shortly), are collectively called *value types*. These types have a fixed size, and when you declare a variable as a value type, the compiler generates code that allocates a block of memory big enough to hold a corresponding value. For example, declaring an `int` variable causes the compiler to allocate 4 bytes of memory (32 bits) to hold the integer value. A statement that assigns a value (such as 42) to the `int` causes the value to be copied into this block of memory.

Class types such as `Circle` (described in Chapter 7) are handled differently. When you declare a `Circle` variable, the compiler *does not* generate code that allocates a block of memory big enough to hold a `Circle` object. All it does is allot a small piece of memory that can potentially hold the address of (or a reference to) another block of memory containing a `Circle` object. (An address specifies the location of an item in memory.) The memory for the actual `Circle` object is allocated only when the `new` keyword is used to create the object.

A class is an example of a *reference type*. Reference types hold references to blocks of memory. To write effective C# code, you must understand the difference between value types and reference types.

Note The `string` type in C# is actually a class. This is because there is no standard size for a string (different strings can contain different numbers of characters), and allocating memory for a string dynamically when the program runs is far more efficient than doing so statically at compile time. The description in this chapter of reference types such as classes applies to the `string` type as well. In fact, the `string` keyword in C# is just an alias for the `System.String` class.

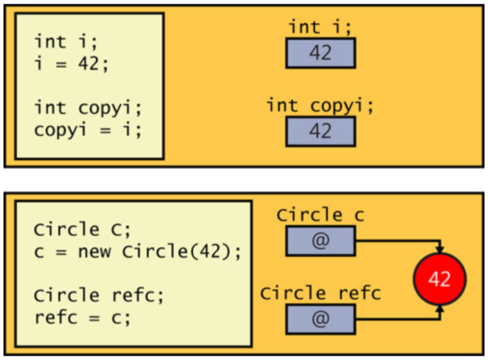
Consider a situation in which you declare a variable named `i` as an `int` and assign it the value 42. If you declare another variable called `copyi` as an `int` and then assign `i` to `copyi`, `copyi` will hold the same value as `i` (42). However, even though `copyi` and `i` happen to hold the same value, two blocks of memory contain the value 42: one block for `i` and the other block for `copyi`. If you modify the value of `i`, the value of `copyi` does not change. Let's see this in code:

```
int i = 42; // declare and initialize i
int copyi = i; /* copyi contains a copy of the data in i:
                            i and copyi both contain the value 42 */
i++;           /* incrementing i has no effect on copyi;
                       i now contains 43, but copyi still contains 42 */
```

The effect of declaring a variable `c` as a class type, such as `Circle`, is very different. When you declare `c` as a `Circle`, `c` can refer to a `Circle` object; the actual value held by `c` is the address of a `Circle` object in memory. If you declare an additional variable named `refc` (also as a `Circle` object) and you assign `c` to `refc`, `refc` will have a copy of the same address as `c`. In other words, there's only one `Circle` object, and both `refc` and `c` now refer to it. Here's an example in code:

```
var c = new Circle(42);
Circle refc = c;
```

The following illustration shows both examples. The at sign (@) in the `Circle` objects represents a reference holding an address in memory.



This difference is very important. It means that the behavior of method parameters depends on whether they are value types or reference types. You'll explore this difference in the next exercise.

Copying reference types and data privacy

If you actually want to copy the contents of a `Circle` object, `c`, into a different `Circle` object, `refc`, instead of just copying the reference, you must make `refc` refer to a new instance of the `Circle` class and then copy the data, field by field, from `c` into `refc`, like this:

```
var refc = new Circle();
refc.radius = c.radius; // Don't try this
```

However, if any members of the `Circle` class are private (like the `radius` field), you won't be able to copy this data. Instead, you can make the data in the private fields accessible by exposing them as properties and then use these properties to read the data from `c` and copy it into `refc`. You'll learn how to do this in Chapter 15, "Implementing properties to access fields."

Alternatively, a class could provide a `Clone` method that returns another instance of the same class but populated with the same data. The `Clone` method would have access to the private data in an object and could copy this data directly to another instance of the same class. For example, the `Clone` method for the `Circle` class could be defined as shown here:

```
class Circle
{
    private int radius;
    // Constructors and other methods omitted
    ...
    public Circle Clone()
    {
        // Create a new Circle object
        Circle clone = new Circle();

        // Copy private data from this to clone
        clone.radius = this.radius;

        // Return the new Circle object containing the copied data
        return clone;
    }
}
```

This approach is straightforward if all the private data consists of values, but if one or more fields are themselves reference types (for example, if the `Circle` class is extended to contain a `Point` object from Chapter 7, indicating the position of the `Circle` on a graph), these reference types also need to provide a `Clone` method; otherwise, the `Clone` method of the `Circle` class will simply copy a reference to these fields. This process is known as a *deep copy*. The alternative approach, wherein the `Clone` method simply copies references, is known as a *shallow copy*.

The preceding code example also poses an interesting question: How private is private data? Previously, you saw that the `private` keyword renders a field or method inaccessible from outside a class. However, this does not mean it can be accessed by only a single object. If you create two objects of the same class, they can each access the private data of the other within the code for that class.

This sounds curious, but in fact, methods such as `Clone` depend on this feature. For example, the statement `clone.radius = this.radius;` works only because the private `radius` field in the `clone` object is accessible from within the current instance of the `Circle` class. So, private actually means private to the class rather than private to an object. Don't confuse private with static, however. If you simply declare a field as private, each instance of the class gets its own data. If a field is declared as static, each instance of the class shares the same data.

To use value parameters and reference parameters

1. Start Microsoft Visual Studio 2022, if it is not already running.
2. Open the **Parameters** solution, which is located in the **\Microsoft Press\VCSBS\Chapter 8\Parameters** folder in your **Documents** folder.

   The project contains three C# code files: Pass.cs, Program.cs, and WrappedInt.cs.
3. Display the Pass.cs file in the Code and Text Editor window.

   This file defines a class called `Pass` that is currently empty apart from a `// TODO:` comment.

Tip You can use the Task List window to locate all `// TODO:` comments in a solution.

4. Add a public static method called `Value` to the `Pass` class, replacing the `// TODO:` comment. This method should accept a single `int` parameter (a value type) called `param` and have the return type `void`. The body of the `Value` method should simply assign the value 42 to `param`, as shown in bold type in the following code example:

```
namespace Parameters
{
    class Pass
    {
        public static void Value(int param)
        {
            param = 42;
        }
    }
}
```

Note You're defining this method using the `static` keyword to keep the exercise simple. You can call the `Value` method directly on the `Pass` class without first creating a new `Pass` object. The principles illustrated in this exercise apply in the same manner to instance methods.

5. Display the Program.cs file in the Code and Text Editor window and then locate the `doWork` method of the `Program` class.

   The `doWork` method is called by the `Main` method when the program starts running. As explained in Chapter 7, the method call is wrapped in a `try` block and followed by a `catch` handler.

6. Add four statements to the `doWork` method to perform the following tasks:

   - Declare a local `int` variable called `i` and initialize it to 0.
   - Write the value of `i` to the console by using `Console.WriteLine`.
   - Call `Pass.Value`, passing `i` as an argument.
   - Write the value of `i` to the console again.

   By running `Console.WriteLine` before and after the call to `Pass.Value`, you can see whether the `Pass.Value` method actually modifies the value of `i`. The completed `doWork` method should look exactly like this:

```
static void doWork()
{
    int i = 0;
    Console.WriteLine(i);
    Pass.Value(i);
    Console.WriteLine(i);
}
```

7. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

8. Confirm that the value 0 is written to the console window twice.

   The assignment statement inside the `Pass.Value` method that updates the parameter and sets it to 42 uses a copy of the argument passed in, and the original argument `i` is completely unaffected.

9. Press **Enter** to close the application.

   You'll now see what happens when you pass an `int` parameter that's wrapped within a class.

10. Display the WrappedInt.cs file in the Code and Text Editor window. This file contains the `WrappedInt` class, which is empty apart from a `// TODO:` comment.

11. Add a public instance field called `Number` of type `int` to the `WrappedInt` class, as shown in bold type in the following code:

```
namespace Parameters
{
    class WrappedInt
    {
        public int Number;
    }
}
```

12. Display the Pass.cs file in the Code and Text Editor window. Add a public static method called Reference to the Pass class. This method should accept a single WrappedInt parameter called param and have the return type void. The body of the Reference method should assign 42 to param.Number, as shown here:

```
public static void Reference(WrappedInt param)
{
    param.Number = 42;
}
```

13. Display the Program.cs file in the Code and Text Editor window. Comment out the existing code in the doWork method and add four more statements to perform the following tasks:

- Declare a local WrappedInt variable called wi and initialize it to a new WrappedInt object by calling the default constructor.
- Write the value of wi.Number to the console.
- Call the Pass.Reference method, passing wi as an argument.
- Write the value of wi.Number to the console again.

As before, with the calls to Console.WriteLine, you can see whether the call to Pass.Reference modifies the value of wi.Number. The doWork method should now look exactly like this (the new statements are in bold):

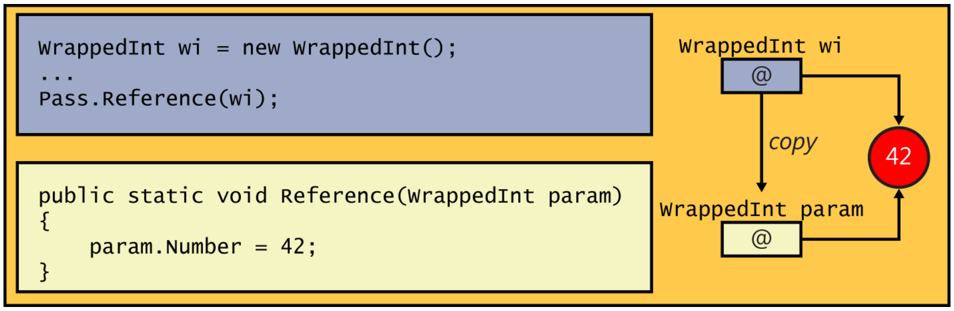
```
static void doWork()
{
    // int i = 0;
    // Console.WriteLine(i);
    // Pass.Value(i);
    // Console.WriteLine(i);
    var wi = new WrappedInt();
    Console.WriteLine(wi.Number);
    Pass.Reference(wi);
    Console.WriteLine(wi.Number);
}
```

14. On the **Debug** menu, select **Start Without Debugging** to build and run the application.

This time, the two values displayed in the console window correspond to the value of wi.Number before and after the call to the Pass.Reference method. You should see that the values 0 and 42 are displayed.

15. Press **Enter** to close the application and return to Visual Studio 2022.

To explain what the previous exercise shows, the value of wi.Number is initialized to 0 by the compiler-generated default constructor. The wi variable contains a reference to the newly created WrappedInt object (which contains an int). The wi variable is then copied as an argument to the Pass.Reference method. Because WrappedInt is a class (a reference type), wi and param both refer to the same WrappedInt object. Any changes made to the contents of the object through the param variable in the Pass.Reference method are visible by using the wi variable when the method completes. The following diagram illustrates what happens when a `WrappedInt` object is passed as an argument to the `Pass.Reference` method.


WrappedInt wi = new WrappedInt();
...
Pass.Reference(wi);
public static void Reference(WrappedInt param)
{
param.Number = 42;
}
WrappedInt wi
@
copy
42
WrappedInt param
@

Understanding null values and nullable types

When you declare a variable, it's always a good idea to initialize it. With value types, it's common to see code such as this:

```
int i = 0;
double d = 0.0;
```

To initialize a reference variable such as a class, you can create a new instance of the class and assign the reference variable to the new object, like this:

```
var c = new Circle(42);
```

This is all very well, but what if you don't actually want to create a new object? Perhaps the purpose of the variable is simply to store a reference to an existing object at some later point in your program. In the following code example, the `Circle` variable copy is initialized, but later it is assigned a reference to another instance of the `Circle` class:

```
var c = new Circle(42);
var copy = new Circle(99); // Some random value for initializing copy
...
copy = c;                  // copy and c refer to the same object
```

After assigning c to copy, what happens to the original `Circle` object with a radius of 99 that you used to initialize copy? Nothing refers to it anymore. In this situation, the runtime can reclaim the memory by performing an operation known as *garbage collection*, which you'll learn more about in Chapter 14, "Using garbage collection and resource management." The important thing to understand for now is that garbage collection is a potentially time-consuming operation, and that you should not create objects that are never used because doing so is a waste of time and resources.

You could argue that if a variable will be assigned a reference to another object at some point in a program, there's no point to initializing it. This is poor programming practice, however, and can lead to problems in your code. For example, you will inevitably find yourself in a situation in which you want to refer a variable to an object only if that variable does not already contain a reference, as shown in the following code example:

```
var c = new Circle(42);
Circle copy;                  // Uninitialized !!!
...
if (copy == // only assign to copy if it is uninitialized, but what goes here?)
{
    copy = c; ;                // copy and c refer to the same object
    ...
}
```

The purpose of the `if` statement is to test the `copy` variable to see whether it is initialized, but to which value should you compare this variable? The answer is to use a special value called `null`.

In C#, you can assign the `null` value to any reference variable. The `null` value simply means that the variable does not refer to an object in memory. You can use it like this:

```
Circle c = new Circle(42);
Circle copy = null; // Initialized
...
if (copy is null)
{
    copy = c; // copy and c refer to the same object
    ...
}
```

Note You can also use `== null` to check for a null reference. However, `is null` reads more naturally. Similarly, you can use `is not null` as well as `!= null` to check for a non-null reference.

The null-conditional and null-coalescing operators

The null-conditional operator enables you to test for `null` values very succinctly. To use the null-conditional operator, you append a question mark (?) to the name of your variable.

For example, suppose you attempt to call the `Area` method on a `Circle` object when the `Circle` object has a `null` value:

```
Circle c = null;
Console.WriteLine($"The area of circle c is {c.Area()}");
```

In this case, the `Circle.Area` method throws a `NullReferenceException`, which makes sense because you cannot calculate the area of a circle that does not exist.

To avoid this exception, you could test whether the `Circle` object is `null` before you attempt to call the `Circle.Area` method:

```
if (c is not null)
{
    Console.WriteLine($"The area of circle c is {c.Area()}");
}
```

In this case, if c is null, nothing is written to the command window. Alternatively, you could use the null-conditional operator on the Circle object before you attempt to call the Circle.Area method:

```
Console.WriteLine($"The area of circle c is {c?.Area()}");
```

The null-conditional operator tells the C# runtime to ignore the current statement if the variable you have applied the operator to is null. In this case, the command window would display the following text:

```
The area of circle c is
```

Both approaches are valid and might meet your needs in different scenarios. The null-conditional operator can help you keep your code concise, particularly when you deal with complex properties with nested reference types that could all be null valued.

Alongside the null-conditional operator, C# provides two null-coalescing operators. The first of these, ??, is a binary operator that returns the value of the operand on the left if it isn't null; otherwise, it returns the value of the operand on the right. In the following example, variable c2 is assigned a reference to c if c isn't null; otherwise, it is assigned a reference to a new Circle object:

```
Circle c = ...; // might be null, might be a new Circle object
...
var c2 = c ?? new Circle(42) ;
```

The null-coalescing assignment operator, ??=, assigns the value of the operand on the right to the operand on the left only if the left operand is null. If the left operand references some other value, it is unchanged.

```
Circle c = ...; // might be null, might be a new Circle object
Circle c3 = ...; // might be null, might be a new Circle object
...
var c3 ??= c; // Only assign c3 if it is null, otherwise leave unchanged;
```

Using nullable types

The null value is very useful for initializing reference types. Sometimes, though, you need an equivalent value for value types. null is itself a reference, so you cannot assign it to a value type. The following statement is therefore illegal in C#:

```
int i = null; // illegal
```

However, C# defines a modifier that you can use to declare that a variable is a *nullable* value type. A nullable value type behaves similarly to the original value type, but you can assign the null value to it. You use the question mark (?) to indicate that a value type is nullable, like this:

```
int? i = null; // legal
```

You can ascertain whether a nullable variable contains null by testing it in the same way as you test a reference type.

```
if (i is null)
      ...
```

You can assign an expression of the appropriate value type directly to a nullable variable. The following examples are all legal:

```
int? i = null;
int j = 99;
i = 100; // Copy a value type constant to a nullable type
i = j; // Copy a value type variable to a nullable type
```

You should note that the converse is not true. You cannot assign a nullable variable to an ordinary value type variable. So, given the definitions of variables `i` and `j` from the preceding example, the following statement is not allowed:

```
j = i; // illegal
```

This makes sense when you consider that the variable `i` might contain `null`, and `j` is a value type that cannot contain `null`. This also means that you cannot use a nullable variable as a parameter to a method that expects an ordinary value type. If you recall, the `Pass.Value` method from the preceding exercise expects an ordinary `int` parameter, so the following method call will not compile:

```
int? i = 99;
Pass.Value(i); // Compiler error
```

Note Take care not to confuse nullable types with the null-conditional operator. Nullable types are indicated by appending a question mark to the type name, whereas the null-conditional operator is appended to the variable name.

Understanding the properties of nullable types

A nullable type exposes a pair of properties that you can use to determine whether the type actually has a non-null value and what this value is. The `HasValue` property indicates whether a nullable type contains a value or is `null`. You can retrieve the value of a non-null nullable type by reading the `Value` property, like this:

```
int? i = null;
...
if (!i.HasValue)
{
    // If i is null, then assign it the value 99
    i = 99;
}
else
{
    // If i is not null, then display its value
    Console.WriteLine(i.Value);
}
```

In Chapter 4, "Using decision statements," you saw that the NOT operator (`!`) negates a Boolean value. The preceding code fragment tests the nullable variable `i`, and if it does not have a value (it is `null`), it assigns it the value 99; otherwise, it displays the value of the variable. In this example, using the `HasValue` property does not provide any benefit over testing for a `null` value directly. Additionally, reading the `Value` property is a long-winded way of reading the contents of the variable. However, these apparent shortcomings are caused by the fact that `int?` is a very simple nullable type. You can create more complex value types and use them to declare nullable variables where the advantages of using the `HasValue` and `Value` properties become more apparent. You'll see some examples in Chapter 9, "Creating value types with enumerations and structures."

Note The `Value` property of a nullable type is read-only. You can use this property to read the value of a variable but not to modify it. To update a nullable variable, use an ordinary assignment statement.

Using ref and out parameters

Ordinarily, when you pass an argument to a method, the corresponding parameter is initialized with a copy of the argument. This is true regardless of whether the parameter is a value type (such as an `int`), a nullable type (such as `int?`), or a reference type (such as a `WrappedInt`). This arrangement means that it's impossible for any change to the parameter to affect the value of the argument passed in. For example, in the following code, the value output to the console is 42, not 43. The `doIncrement` method increments a copy of the argument (`arg`) and *not* the original argument, as demonstrated here:

```
static void doIncrement(int param)
{
    param++;
}

static void Main()
{
    int arg = 42;
    doIncrement(arg);
    Console.WriteLine(arg); // writes 42, not 43
}
```

In the preceding exercise, you saw that if the parameter to a method is a reference type, any changes made by using that parameter change the data referenced by the argument passed in. The key point is this: although the data that was referenced changed, the argument passed in as the parameter did not. It still references the same object. In other words, although it's possible to modify the object that the argument refers to through the parameter, it's not possible to modify the argument itself—for example, to set it to refer to a completely different object. Most of the time, this guarantee is very useful and can help reduce the number of bugs in a program. Occasionally, however, you might want to write a method that actually needs to modify an argument. C# provides the `ref` and `out` keywords so that you can do this.

Creating ref parameters

If you prefix a parameter with the `ref` keyword, the C# compiler generates code that passes a reference to the actual argument rather than a copy of the argument. When using a `ref` parameter, anything you do to the parameter you also do to the original argument because the parameter and the argument both reference the same data.

When you pass an argument as a `ref` parameter, you must also prefix the argument with the `ref` keyword. This syntax provides a useful visual cue to the programmer that the argument might change. Here's the preceding example again, this time modified to use the `ref` keyword:

```
static void doIncrement(ref int param) // using ref
{
    param++;
}

static void Main()
{
    int arg = 42;
    doIncrement(ref arg); // using ref
    Console.WriteLine(arg); // writes 43
}
```

This time, the `doIncrement` method receives a reference to the original argument rather than a copy, so any changes the method makes by using this reference actually change the original value. That's why the value 43 is displayed on the console.

Remember that C# enforces the rule that you must assign a value to a variable before you can read it. This rule also applies to method arguments; you cannot pass an uninitialized value as an argument to a method even if an argument is defined as a `ref` argument. For example, in the following example, `arg` is not initialized, so this code will not compile. This failure occurs because the statement `param++;` within the `doIncrement` method is really an alias for the statement `arg++;` and this operation is allowed only if `arg` has a defined value:

```
static void doIncrement(ref int param)
{
    param++;
}

static void Main()
{
    int arg; // not initialized
    doIncrement(ref arg);
    Console.WriteLine(arg);
}
```

Creating out parameters

The compiler checks whether a `ref` parameter has been assigned a value before calling the method. However, there might be times when you want the method itself to initialize the parameter. You can do this with the `out` keyword.

The out keyword is syntactically similar to the ref keyword. You can prefix a parameter with the out keyword so that the parameter becomes an alias for the argument. As when using ref, anything you do to the parameter, you also do to the original argument. When you pass an argument to an out parameter, you must also prefix the argument with the out keyword.

The keyword out is short for *output*. When you pass an out parameter to a method, the method *must* assign a value to it before it finishes or returns, as shown in the following example:

```
static void doInitialize(out int param)
{
    param = 42; // Initialize param before finishing
}
```

The following example does not compile because doInitialize does not assign a value to param:

```
static void doInitialize(out int param)
{
    // Do nothing
}
```

Because an out parameter must be assigned a value by the method, you're allowed to call the method without initializing its argument. For example, the following code calls doInitialize to initialize the variable arg, which is then displayed on the console:

```
static void doInitialize(out int param)
{
    param = 42;
}

static void Main()
{
    int arg; // not initialized
    doInitialize(out arg); // legal
    Console.WriteLine(arg); // writes 42
}
```

Note You can combine the declaration of an out variable with its use as a parameter rather than performing these tasks separately. For example, you could replace the first two statements in the Main method in the previous example with this single line of code:

```
doInitialize(out int arg);
```

In the next exercise, you'll practice using ref parameters.

To use ref parameters

1. Return to the Parameters project in Visual Studio 2022.
2. Display the Pass.cs file in the Code and Text Editor window.

3. Edit the `Value` method to accept its parameter as a `ref` parameter.

   The `Value` method should look like this:

```
class Pass
{
    public static void Value(ref int param)
    {
        param = 42;
    }
    ...
}
```

4. Display the Program.cs file in the Code and Text Editor window.

5. Uncomment the first four statements.

   Notice that the third statement of the `doWork` method, `Pass.Value(i)`, indicates an error. The error occurs because the `Value` method now expects a `ref` parameter.

6. Edit this statement so that the `Pass.Value` method call passes its argument as a `ref` parameter.

Note Leave the four statements that create and test the WrappedInt object as they are.

   The `doWork` method should now look like this:

```
class Program
{
  static void doWork()
  {
    int i = 0;
    Console.WriteLine(i);
    Pass.Value(ref i);
    Console.WriteLine(i);
    ...
  }
}
```

7. On the **Debug** menu, select **Start Without Debugging** to build and run the program.

   This time, the first two values written to the console window are 0 and 42. This result shows that the call to the `Pass.Value` method has successfully modified the argument `i`.

8. Press **Enter** to close the application and return to Visual Studio 2022.

Note You can use the `ref` and `out` modifiers on reference type parameters as well as on value type parameters. The effect is the same: the parameter becomes an alias for the argument.

How computer memory is organized

Computers use memory to hold programs that are being executed and the data that those programs use. To understand the differences between value and reference types, it's helpful to understand how data is organized in memory.

Operating systems and language runtimes such as those used by C# frequently divide the memory used for holding data into two separate areas, each of which is managed in a distinct manner. These two areas of memory are traditionally called the *stack* and the *heap*. The stack and the heap serve different purposes:

- When you call a method, the memory required for its parameters and its local variables is acquired from the stack. When the method finishes (because it either returns or throws an exception), the memory acquired for the parameters and local variables is automatically released back to the stack to be made available again when another method is called. Method parameters and local variables on the stack have a well-defined lifespan: They come into existence when the method starts, and they disappear as soon as the method completes.

  The same lifespan applies to variables defined in any block of code enclosed by opening and closing braces. In the following code example, the variable `i` is created when the body of the `while` loop starts, but it disappears when the `while` loop finishes, and execution continues after the closing brace:

  ```
  while (...)
  {
      int i = …; // i is created on the stack here
      ...
  }
  // i disappears from the stack here
  ```

- When you create an object (an instance of a class) by using the new keyword, the memory required to build the object is acquired from the heap. You've seen that the same object can be referenced from several places by using reference variables. When the last reference to an object disappears, the memory used by the object becomes available again (although it might not be reclaimed immediately). Objects created on the heap therefore have a more indeterminate lifespan; an object is created by using the new keyword, but it disappears only sometime after the last reference to the object is removed. Chapter 14 includes a more detailed discussion of how heap memory is reclaimed.

Note All value types are created on the stack. By default, reference types (objects) are created on the heap, although the reference itself is on the stack. (There are some exceptions to this rule, which you'll learn about in later chapters.) Nullable objects are actually reference types, and they are created on the heap.

The names *stack* and *heap* come from the way in which the runtime manages the memory:

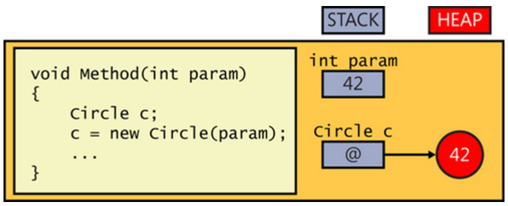
- Stack memory is organized like boxes stacked neatly on top of one another. When a method is called, each parameter is placed in a box that is added to the top of the stack. Each local variable is likewise assigned a box, which is placed on top of the boxes already on the stack. When a method finishes, think of it as being like a box being removed from the stack.
- Heap memory is like a large pile of boxes strewn around a room rather than stacked neatly on top of one another. Each box has a label indicating whether it is in use. When a new object is created, the runtime searches for an empty box and allocates it to the object. The reference to the object is stored in a local variable on the stack. The runtime keeps track of the number of references to each box. (Remember: two variables can refer to the same object.) When the last reference disappears, the runtime marks the box as not in use; at some point in the future, it empties the box and makes it available.

Using the stack and the heap

Now let's examine what happens when a method named `Method` is called:

```
void Method(int param)
{
    Circle c;
    c = new Circle(param);
    ...
}
```

Suppose the argument passed into `param` is the value 42. When the method is called, a block of memory (just enough for an `int`) is allocated from the stack and initialized with the value 42. As execution moves inside the method, another block of memory big enough to hold a reference (a memory address) is also allocated from the stack, but left uninitialized. This is for the `Circle` variable, `c`. Next, another piece of memory big enough for a `Circle` object is allocated from the heap. This is what the `new` keyword does. The `Circle` constructor runs to convert this raw heap memory to a `Circle` object. A reference to this `Circle` object is stored in the variable `c`. The following illustration shows this process:



```
void Method(int param)
{
    Circle c;
    c = new Circle(param);
    ...
}
```

At this point, you should note two things:

- Although the object is stored on the heap, the reference to the object (the variable `c`) is stored on the stack.
- Heap memory is not infinite. If heap memory is exhausted, the `new` operator will throw an `OutOfMemoryException` exception, and the object will not be created.

Note The `Circle` constructor could also throw an exception. If it does, the memory allocated to the `Circle` object will be reclaimed, and the value returned by the constructor will be `null`.

When the method ends, the parameters and local variables go out of scope. The memory acquired for `c` and `param` is automatically released back to the stack. The runtime notes that the `Circle` object is no longer referenced and at some point in the future will arrange for its memory to be reclaimed by the heap. (See Chapter 14.)

The System.Object class

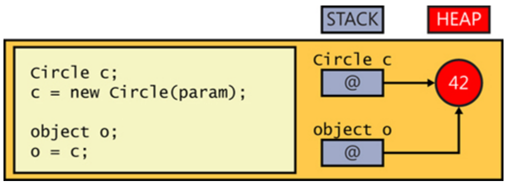
One of the most important reference types in .NET is the `Object` class in the `System` namespace. To fully appreciate the significance of the `System.Object` class, you must understand inheritance, which is described in Chapter 12, "Working with inheritance." For now, simply accept that all classes are specialized types of `System.Object` and that you can use `System.Object` to create a variable that can refer to any reference type. `System.Object` is such an important class that C# provides the `object` keyword as an alias for `System.Object`. In your code, you can use `object`, or you can write `System.Object`. They mean the same thing.

Tip Use the `object` keyword rather than `System.Object`. It's more direct, and it's consistent with other keywords that are synonyms for classes, such as `string` for `System.String` and others that are covered in Chapter 9.

In the following example, the variables `c` and `o` both refer to the same `Circle` object. The fact that the type of `c` is `Circle` and the type of `o` is `object` (the alias for `System.Object`) in effect provides two different views of the same item in memory.

```
Circle c;
c = new Circle(42);
object o;
o = c;
```

The following diagram illustrates how the variables `c` and `o` refer to the same item on the heap:

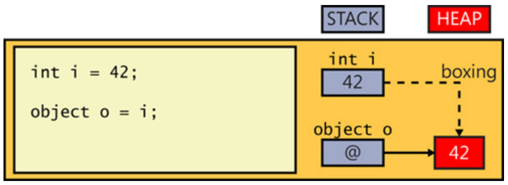


Boxing

As you have just seen, variables of type `object` can refer to any item of any reference type. However, variables of type `object` can also refer to a value type. For example, the following two statements initialize the variable `i` (of type `int`, a value type) to 42 and then initialize the variable o (of type `object`, a reference type) to `i`:

```
int i = 42;
object o = i;
```

The second statement requires a little explanation to appreciate what's actually happening. Remember that `i` is a value type and that it lives on the stack. If the reference inside o referred directly to `i`, the reference would refer to the stack. However, references should refer to objects on the heap. Creating uncontrolled references to items on the stack could seriously compromise the robustness of the runtime and potentially create a security flaw, so it is not allowed. Therefore, the runtime allocates a piece of memory from the heap, copies the value of integer `i` to this piece of memory, and then refers the object o to this copy. This automatic copying of an item from the stack to the heap is called *boxing*. The following diagram shows the result:



Important If you modify the original value of the variable `i`, the value on the heap referenced through o will not change. Likewise, if you modify the value on the heap, the original value of the variable will not change.

Unboxing

Because a variable of type `object` can refer to a boxed copy of a value, it's only reasonable to allow you to get at that boxed value through the variable. You might expect to be able to access the boxed `int` value that a variable o refers to by using a simple assignment statement such as this:

```
int i = o;
```

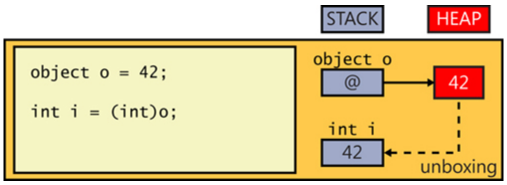
However, if you try this syntax, you'll get a compile-time error. If you think about it, it's fairly sensible that you can't use the `int i = o;` syntax. After all, o could be referencing absolutely anything and not just an `int`. Consider what would happen in the following code if this statement were allowed:

```
Circle c = new Circle();
int i = 42;
object o;
o = c; // o refers to a circle
i = o; // what is stored in i?
```

To obtain the value of the boxed copy, you must use what is known as a *cast*. This is an operation that checks whether converting an item of one type to another is safe before actually making the copy. You prefix the `object` variable with the name of the type in parentheses, as in this example:

```
int i = 42;
object o = i; // boxes
i = (int)o; // compiles okay
```

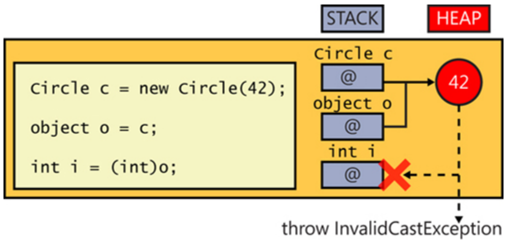
The effect of this cast is subtle. The compiler notices that you've specified the type `int` in the cast. Next, the compiler generates code to check what `o` actually refers to at runtime. It could be absolutely anything. Just because your cast says `o` refers to an `int`, that doesn't mean it actually does. If `o` really does refer to a boxed `int` and everything matches, the cast succeeds, and the compiler-generated code extracts the value from the boxed `int` and copies it to `i`. (In this example, the boxed value is then stored in `i`.) This is called *unboxing*. The following diagram shows what's happening:



On the other hand, if `o` does not refer to a boxed `int`, there is a type mismatch, causing the cast to fail. The compiler-generated code throws an `InvalidCastException` exception at runtime. Here's an example of an unboxing cast that fails:

```
Circle c = new Circle(42);
object o = c; // doesn't box because Circle is a reference variable
int i = (int)o; // compiles okay but throws an exception at runtime
```

The following diagram illustrates this case:



You'll use boxing and unboxing in later exercises. Keep in mind that boxing and unboxing are expensive operations because of the amount of checking required and the need to allocate additional heap memory. Boxing has its uses, but injudicious use can severely impair the performance of a program. You'll see an alternative to boxing in Chapter 17, "Introducing generics."

Casting data safely

By using a cast, you can specify that, in your opinion, the data referenced by an object has a specific type and that it's safe to reference the object by using that type. The key phrase here is "in your opinion." The C# compiler will not check that this is the case, but the runtime will. If the type of object in memory does not match the cast, the runtime will throw an `InvalidCastException`, as described in the preceding section. You should be prepared to catch this exception and handle it appropriately if it occurs.

However, catching an exception and attempting to recover if the type of an object is not what you expected it to be is a rather cumbersome approach. C# provides two more very useful operators that can help you perform casting in a much more elegant manner: the `is` and `as` operators.

The is operator

You've seen the `is` operator before, when checking for a `null` value, but it actually enables you to check for the type of any reference object. You can use the `is` operator to verify that the type of an object is what you expect it to be, like this:

```
var wi = new WrappedInt();
...
object o = wi;
if (o is WrappedInt)
{
    WrappedInt temp = (WrappedInt)o; // This is safe; o is a WrappedInt
    ...
}
```

The `is` operator takes two operands: a reference to an object on the left, and the name of a type (or `null`) on the right. If the type of the object referenced on the heap matches the type specified by the `is` operator, `is` evaluates to `true`; otherwise, `is` evaluates to `false`. The preceding code attempts to cast the reference to the `object` variable `o` only if it knows that the cast will succeed.

Another form of the `is` operator enables you to abbreviate this code by combining the type check and the assignment, like this:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
...
if (o is WrappedInt temp)
{
    ... // Use temp here
}
```

In this example, if the test for the `WrappedInt` type is successful, the `is` operator creates a new reference variable (called `temp`) and assigns it a reference to the `WrappedInt` object.

The as operator

The as operator fulfills a similar role to `is` but in a slightly truncated manner. You use the as operator like this:

```
WrappedInt wi = new WrappedInt();
...
object o = wi;
WrappedInt temp = o as WrappedInt;
if (temp is not null)
{
    ... // Cast was successful
}
```

Like the `is` operator, the as operator takes an object and a type as its operands. The runtime attempts to cast the object to the specified type. If the cast is successful, the result is returned and, in this example, is assigned to the `WrappedInt` variable `temp`. If the cast is unsuccessful, the as operator evaluates to the `null` value and assigns that to `temp` instead.

Note There's a little more to the `is` and as operators than is described here; Chapter 12 discusses them in greater detail.

The switch statement revisited

If you need to check a reference against several types, you can use a series of `if`...`else` statements in conjunction with the `is` operator. The following example assumes that you have defined the `Circle`, `Square`, and `Triangle` classes. The constructors take the radius (`radius`) or side length (`side`) of the geometric shape as the parameter:

```
var c = new Circle(42);       // Circle of radius 42
var s = new Square(55);       // Square of side 55
var t = new Triangle(33);     // Equilateral triangle of side 33
...
object o = s;
...
if (o is Circle myCircle)
{
    ... // o is a Circle, a reference is available in myCircle
}
else if (o is Square mySquare)
{
    ... // o is a Square, a reference is available in mySquare
}
else if (o is Triangle myTriangle)
{
    ... // o is a Triangle, a reference is available in myTriangle
}
```

As with any lengthy set of if...else statements, this approach can quickly become cumbersome and difficult to read. Fortunately, you can use the switch statement in this situation, as follows:

```
switch (o)
{
    case Circle myCircle:
        ... // o is a Circle, a reference is available in myCircle
        break;

    case Square mySquare:
        ... // o is a Square, a reference is available in mySquare
        break;

    case Triangle myTriangle:
        ... // o is a Triangle, a reference is available in myTriangle
        break;

        default:
            throw new ArgumentException("variable is not a recognized shape");
        break;
}
```

In both examples (using the is operator and the switch statement), the scope of the variables created (myCircle, mySquare, and myTriangle) is limited to the code inside the corresponding if block or case block.

case selectors in switch statements also support when expressions, which you can use to further qualify the situation under which the case is selected. For example, the following switch statement shows case selectors that match different sizes of geometric shapes:

```
switch (o)
{
    case Circle myCircle when myCircle.Radius > 10:
        ...
        break;
    case Square mySquare when mySquare.SideLength == 100:
        ...
    break;
        ...
}
```

Pointers and unsafe code

This sidebar is purely for your information and is aimed at developers who are familiar with C or C++. If you're new to programming, feel free to ignore this information.

If you have already written programs in languages such as C or C++, much of the discussion in this chapter concerning object references might be familiar in that both languages have a construct that provides similar functionality: a pointer. A *pointer* is a variable that holds the address of, or a reference to, an item in memory (on the heap or the stack).

A special syntax is used to identify a variable as a pointer. For example, the following statement declares the variable `pi` as a pointer to an integer:

```
int *pi;
```

Although the variable `pi` is declared as a pointer, it does not actually point anywhere until you initialize it. For example, to use `pi` to point to the integer variable `i`, you can use the following statements and the address-of operator (&), which returns the address of a variable:

```
int *pi;
int i = 99;
...
pi = &i;
```

You can access and modify the value held in the variable `i` through the pointer variable `pi` like this:

```
 *pi = 100;
```

This code updates the value of the variable `i` to `100` because `pi` points to the same memory location as the variable `i`.

One of the main problems that developers learning C and C++ encounter is understanding the syntax used by pointers. The * operator has at least two meanings (in addition to being the arithmetic multiplication operator), and there's often great confusion about when to use & rather than `*`.

The other issue with pointers is that it's easy to point somewhere invalid or to forget to point somewhere at all, and then try to reference the data pointed to. The result will be either garbage or a program that fails with an error because the operating system detects an attempt to access an illegal address in memory.

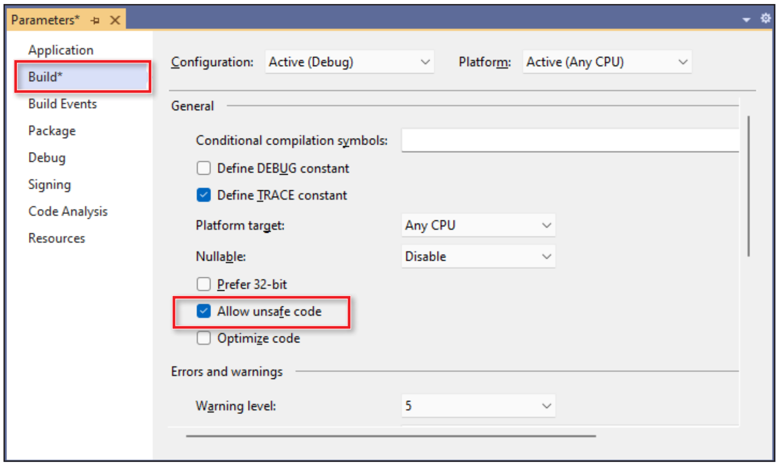
Finally, there are several security flaws in many existing systems resulting from the mismanagement of pointers. Some environments (not Windows) fail to enforce checks that a pointer does not refer to memory that belongs to another process, opening up the possibility that confidential data could be compromised.

Reference variables were added to C# to avoid all these problems. If you really want to, you can continue to use pointers in C#, but you must mark the code as `unsafe`. The `unsafe` keyword can be used to mark a block of code or an entire method, as shown here:

```
public static void Main(string [] args)
{
    int x = 99, y = 100;
    unsafe
    {
        swap (&x, &y);
    }
    Console.WriteLine($"x is now {x}, y is now {y}");
}

public static unsafe void swap(int *a, int *b)
{
    int temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

When you compile programs containing unsafe code, you must specify the Allow Unsafe Code option when building the project. To do this, right-click the project in Solution Explorer and then select Properties. In the Properties window, select the Build tab, select Allow Unsafe Code, and then, on the File menu, select Save All.



Configuration: Active (Debug)   Platform: Active (Any CPU)

General

Conditional compilation symbols:

☐ Define DEBUG constant

☑ Define TRACE constant

Platform target: Any CPU

Nullable: Disable

☐ Prefer 32-bit

☑ Allow unsafe code

☐ Optimize code

Errors and warnings

Warning level: 5

Unsafe code also affects how memory is managed. Objects created in unsafe code are said to be unmanaged. Although situations that require you to access memory in this way are not common, you might encounter some, especially if you're writing code that needs to perform some low-level Windows operations.

You'll learn about the implications of using code that accesses unmanaged memory in more detail in Chapter 14.

Summary

In this chapter, you learned about some important differences between value types that hold their value directly on the stack and reference types that refer indirectly to their objects on the heap. You also learned how to use the `ref` and `out` keywords on method parameters to gain access to the arguments. You saw how assigning a value (such as the `int` 42) to a variable of the `System.Object` class creates a boxed copy of the value on the heap and then causes the `System.Object` variable to refer to this boxed copy. You also saw how assigning a variable of a value type (such as an `int`) from a variable of the `System.Object` class copies (or unboxes) the value in the `System.Object` class to the memory used by the `int`.

- If you want to continue to the next chapter, keep Visual Studio 2022 running and turn to Chapter 9.
- If you want to exit Visual Studio 2022 now, on the File menu, select Exit. If you see a Save dialog, select Yes and save the project.

Quick reference

To	Do this

Copy a value type variable	Simply make the copy. Because the variable is a value type, you will have two copies of the same value. For example:
	`int i = 42;`
	`int copyi = i;`

Copy a reference type variable	Simply make the copy. Because the variable is a reference type, you will have two references to the same object. For example:
	`Circle c = new Circle(42);`
	`Circle refc = c;`

Declare a variable that can hold a value type or the `null` value	Declare the variable by using the ? modifier with the type. For example: `int? i = null;`

Pass an argument to a `ref` parameter	Prefix the argument with the `ref` keyword. This makes the parameter an alias for the actual argument rather than a copy of the argument. The method may change the value of the parameter, and this change is made to the actual argument rather than to a local copy. For example:
	`static void Main()`
	`{`
	`    int arg = 42;`
	`    doWork(ref arg);`
	`    Console.WriteLine(arg);`
	`}`

Pass an argument to an `out` parameter	Prefix the argument with the `out` keyword. This makes the parameter an alias for the actual argument rather than a copy of the argument. The method must assign a value to the parameter, and this value is made to the actual argument. For example:
	`static void Main()`
	`{`
	`    int arg;`
	`    doWork(out arg);`
	`    Console.WriteLine(arg);`
	`}`

To	Do this
Box a value	Initialize or assign a variable of type `object` with the value. For example: `object o = 42;`
Unbox a value	Cast the object reference that refers to the boxed value to the type of the value variable. For example: `int i = (int)o;`

Cast an object safely	Use the `is` operator to test whether the cast is valid. For example:
	`WrappedInt wi = new WrappedInt();`
	`...`
	`object o = wi;`
	`if (o is WrappedInt temp)`
	`{`
	`    ...`
	`}`
	Alternatively, use the `as` operator to perform the cast, and test whether the result is null. For example:
	`WrappedInt wi = new WrappedInt();`
	`...`
	`object o = wi;`
	`WrappedInt temp = o as WrappedInt;`
	`if (temp != null)`
	`  ...`

Index