

Denis Panjuta · Jafar Jabbarzadeh

Learning C# Through Small Projects



Learning C# Through Small Projects

Denis Panjuta • Jafar Jabbarzadeh

Learning C# Through Small Projects



Springer

Denis Panjuta
Panjutorials GmbH
Köln, Germany

Jafar Jabbarzadeh
Panjutorials GmbH
Köln, Germany

ISBN 978-3-031-51913-0

ISBN 978-3-031-51914-7 (eBook)

<https://doi.org/10.1007/978-3-031-51914-7>

© The Editor(s) (if applicable) and The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
This work is subject to copyright. All rights are solely and exclusively licensed by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

The publisher, the authors, and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, expressed or implied, with respect to the material contained herein or for any errors or omissions that may have been made. The publisher remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

If disposing of this product, please recycle the paper.

Preface

In the ever-evolving world of software development, the ability to adapt, learn, and innovate is paramount. C# has emerged as a versatile and powerful language, enabling developers to craft everything from enterprise applications to captivating games. However, the journey to mastering C# can be daunting, especially when faced with dense textbooks and abstract concepts. *Learning C# by Small Projects* seeks to bridge this gap, offering a hands-on approach to understanding advanced C# concepts through engaging projects and minigames.

The rationale behind this book is simple: learning by doing. Instead of wading through pages of theory, you'll be diving straight into the action, building 11 distinct projects that range from an interactive storytelling program to a responsive Discord chatbot. Each project is meticulously designed to introduce and reinforce specific C# concepts, ensuring that you not only understand the theory but can also apply it in real-world scenarios.

The book is structured to provide a gradual learning curve. The initial chapters lay the foundation, introducing you to the basics of C# programming. As you progress, the projects become more intricate, delving into advanced topics such as asynchronous operations, data integrity, and API integration. By the end of your journey, you'll have a comprehensive understanding of C# and a portfolio of projects to showcase your skills.

As the authors of this book, we, Denis Panjuta and Jafar Jabbarzadeh, have the privilege of sharing our expertise with a combined student base of over 300,000. This book is more than just a guide—it's a mentor. Our extensive teaching experience is evident in every chapter, ensuring that complex topics are broken down into easily digestible segments. Moreover, our commitment to practical learning ensures that every concept is paired with a hands-on project, reinforcing your understanding and building your confidence.

In essence, *Learning C# by Small Projects* is more than just a book—it's a journey. A journey that takes you from the foundational concepts of C# to its advanced applications, all while building tangible projects that you can proudly showcase. Whether you're an aspiring game developer, an enterprise software engineer, or simply a coding enthusiast, this book promises to be an invaluable resource in your C# learning journey.

So, grab your computer, roll up your sleeves, and let's dive into the fascinating world of C# development. Your journey to mastering C# starts here.

Köln, Germany

Denis Panjuta
Jafar Jabbarzadeh

Contents

1	Hello World!	1
1.1	The Technology of Choice.....	2
1.2	What Can You Do With C#?	4
1.2.1	C# for Web Application Development.....	4
1.2.2	C# for Windows Applications	4
1.2.3	C# for Linux and macOS Applications	5
1.2.4	C# for Mobile App Development.....	5
1.2.5	C# for Video Game Development	6
1.3	Getting Started with C#	6
1.4	Hello World! A Simple Interactive Storytelling App.....	14
1.4.1	Our Project.....	14
1.4.2	Our Code	15
1.4.3	Source Code.....	20
1.5	Summary	20
	References.....	21
2	C# Data Types and Variables	23
2.1	Variables in C#.....	24
2.2	Variable Data Types	26
2.2.1	int (Integral Numeric Types)	27
2.2.2	float (Floating-Point Numeric Types)	27
2.2.3	char (Unicode UTF-16 Character).....	27
2.2.4	bool (Boolean Value)	28
2.2.5	The C# Built-in Value and Reference Types (Tables 2.1 and 2.2)	28
2.3	Variables Types in C#	29
2.3.1	Static Variables.....	29
2.3.2	Instance Variables	30
2.3.3	Array Variables	30
2.3.4	Value Parameters	30
2.3.5	Reference Parameters.....	31

2.3.6	Output Parameters	32
2.3.7	Local Variables.	33
2.4	The Random Class.	33
2.4.1	Generating a Random Number	34
2.5	A Random Number	35
2.5.1	Our Project.	35
2.5.2	Our Code	36
2.5.3	Source Code	42
2.6	Summary	42
	References.	42
3	C# Operators and Conditionals.	45
3.1	What Are Operators and Expressions	46
3.2	Types of Operators.	49
3.3	Operator Priority	50
3.3.1	Operator Precedence	51
3.3.2	Operator Associativity	52
3.3.3	Operand Evaluation	53
3.4	Conditional Statements	54
3.4.1	The if Statement.	55
3.4.2	The Ternary Operator “?:”	58
3.4.3	The Switch Statement	59
3.5	Rock-Paper-Scissors	61
3.5.1	The Project.	62
3.5.2	Our Code	62
3.6	Source	69
3.7	Summary	69
	References.	70
4	C# Classes, Methods, and User Input.	73
4.1	Classes in Object-Oriented C#.	74
4.1.1	Definition of Class	74
4.1.2	Object Creation	76
4.1.3	Class Inheritance	77
4.1.4	A Brief Introduction to Test-Driven Development	79
4.2	Methods in C#.	80
4.2.1	Defining a Method	81
4.2.2	Static and Non-static Methods.	82
4.2.3	Calling a Method	83
4.2.4	Method Return Types.	88
4.3	Expanding on the Console Class	90
4.3.1	The Console Class	90
4.3.2	Console Properties	91
4.3.3	Console Methods	93

4.4	The Convert Class	96
4.4.1	Definition of the Convert Class	96
4.4.2	Outcomes to Conversions	96
4.4.3	Convert Methods	99
4.5	Guess the Number	100
4.5.1	The Project.....	101
4.5.2	Our Code	101
4.5.3	Source Code.....	111
4.6	Summary	111
	References.....	112
5	C# Collections and Iterators	115
5.1	Arrays in C#.....	116
5.1.1	The Definition of an Array.....	116
5.1.2	Declaring Arrays	117
5.1.3	Types of Arrays	119
5.1.4	Array Properties and Methods.....	122
5.2	Collections in C#	123
5.2.1	The Definition of a Collection.....	123
5.2.2	System.Collections Classes	125
5.2.3	System.Collections.Concurrent Classes.....	126
5.2.4	System.Collections.Generic Classes	127
5.2.5	Collection Properties and Methods	129
5.2.6	Differences Between Array and Collection.....	131
5.2.7	The Shared System.Linq Namespace	132
5.3	Iteration Statements in C#	134
5.3.1	Loops and Iterators	135
5.3.2	Types of Iterators and Loops in C#	136
5.4	Tic Tac Toe.....	141
5.4.1	The Project.....	141
5.4.2	Our Code	142
5.4.3	Source Code.....	163
5.5	Summary	163
	References.....	164
6	C# File System	167
6.1	The System.IO Namespace	168
6.1.1	System.IO Definition.....	168
6.1.2	System.IO Classes	169
6.1.3	System.IO Structs, Enums, and Delegates.....	174
6.1.4	System.IO for Linux and macOS.....	175
6.2	String Manipulation	176
6.2.1	The Immutability of Strings.....	176
6.2.2	Verbatim and Quoted Strings.....	178

6.2.3	String Escape Sequences	178
6.2.4	Format Strings	179
6.2.5	Substrings	180
6.3	Guess the Word	181
6.3.1	The Project	182
6.3.2	Our Code	182
6.3.3	Source Code	196
6.4	Summary	196
	References	197
7	C# Async Operations	199
7.1	The Central Processing Unit	200
7.1.1	How Does Asynchrony Work on the CPU	200
7.1.2	Asynchrony Versus Multi-threading	200
7.2	Asynchronous Programming in C#	201
7.2.1	What Is Asynchronous Programming in C#	202
7.2.2	Common Use Cases and When to Use	202
7.2.3	Overview of the Asynchronous Model	207
7.3	Pattern Matching	211
7.3.1	What Is Pattern Matching	212
7.3.2	Pattern Matching in C#	212
7.3.3	Types of Patterns	215
7.4	Diffuse the Bomb	217
7.4.1	The Project	217
7.4.2	Our Algorithm	217
7.4.3	Source Code	233
7.5	Summary	233
	References	234
8	C# Hashing and Checksum	237
8.1	Data Integrity	238
8.1.1	What Is Data Integrity	238
8.1.2	Why Do We Care About Data Integrity	239
8.1.3	Types of Data Integrity	239
8.2	Checksum	241
8.2.1	What Is Checksum	241
8.2.2	Checksum Inconsistencies and Why We Shall Not Forget the Simple Things	243
8.3	Hashing	244
8.3.1	What Is Hashing	244
8.3.2	What Is the Difference Between Cryptographic Hashing and Non-cryptographic Hashing	246
8.3.3	Types of Cryptographic Hashing Algorithms	247

8.4	Checksum Checker	250
8.4.1	The Project	251
8.4.2	Our Code	251
8.4.3	Source Code	266
8.5	Summary	267
	References	267
9	C# Cryptography	269
9.1	Cybersecurity	270
9.1.1	What Is Cybersecurity and Why Should We Care	270
9.1.2	What Is the Difference Between Data Security and Cybersecurity	271
9.1.3	Cybersecurity Categories	272
9.2	Encryption in C#	273
9.2.1	What Is Encryption	274
9.2.2	Why Do We Encrypt?	274
9.2.3	Difference Between Hashing and Encryption	275
9.2.4	Types of Encryption	276
9.3	The Process of Encrypting and Decrypting Data	277
9.3.1	Advanced Encryption Standard	277
9.3.2	How Do We Encrypt Data	278
9.3.3	How Do We Decrypt Data	280
9.4	The Message Encryptor	282
9.4.1	The Project	282
9.4.2	Our Code	283
9.4.3	Source Code	302
9.5	Summary	302
	References	303
10	C# Simple Mail Transfer Protocol	305
10.1	Simple Mail Transfer Protocol or SMTP	306
10.1.1	How Does SMTP Work	306
10.1.2	How Do We Use SMTP	308
10.1.3	SMTP and Receiver Protocols	312
10.2	SOLID Programming	313
10.3	JavaScript Object Notation or JSON	318
10.3.1	How Do We Use a JSON File?	318
10.4	A Mass Email Sender App	321
10.4.1	The Project	322
10.4.2	Our Code	322
10.4.3	Source Code	339
10.5	Summary	339

11 C# Application Programming Interface	341
11.1 APIs	342
11.1.1 What Is an API.	343
11.1.2 Why Do We Use APIs	344
11.1.3 How APIs Work	345
11.1.4 API vs REST vs SDK	346
11.2 The HTTP Class.	347
11.3 Working with APIs.	349
11.3.1 What a Simple API Project Would Look Like.	349
11.4 A Discord E-Commerce Bot	355
11.4.1 The Project.	355
11.4.2 Our Code	356
11.4.3 Source Code.	390
11.5 Summary	390
References.	391

This Chapter Covers

- Using the C# programming language
- Creating a C# Console Application project
- Touring our template of choice
- Developing in Visual Studio Community Edition
- Building an interactive storytelling app

Learning a new skill can often be a daunting and scary task. While certainly not unobtainable, it feels far to reach and hard to grasp. One must think about finding a better way of accomplishing these goals without turning the journey into a regrettable memory. If we try to search for our favorite learning experiences, we tend to remember moments of accomplishment, self-improvement, and problem-solving. We tend to think of moments when we tried to use our abilities and got closer to mastering them every time. While learning theory is most important, it is practice that makes us experts.

This makes us consider why we cannot just turn the learning process into real-life problem-solving exercises. So it is this concept this book focuses on, learning to master a language in a dynamic yet meaningful way.

In this chapter, we will go over the steps to build a simple interactive storytelling application. Similar to Twine or Ren'Py, interactive storytelling applications allow us to write interactive stories with multiple choices and endings.

From a simple storytelling app to working efficiently with APIs,¹ we will try to learn the C# programming language through a small project in each chapter. To make that happen, we will start by getting to know our technology of choice, its relevance, what

¹Application Programming Interface. A set of programming code that enables data transmission between one software product and another.

makes this language stand out compared to other options, and where it truly exceeds. Furthermore, we will deep dive into C# development with our first project, learning its similarities with other programming languages we might know and familiarizing ourselves with the entire development process. Let us start this book with our first question: What is C#?

1.1 The Technology of Choice

If we come from a C, C++, Java, or Javascript background, we will already be familiarized with the C family of languages. Coming from the C family of languages, the C# object-oriented, type-safe² programming language is a general-purpose, multi-paradigm³ programming language.

C#'s transition to an open-source model has significantly broadened its scope and appeal, as it enables a collaborative environment where developers can contribute to its evolution and adapt it to emerging technologies and platforms. The modern design of C# facilitates the creation of secure and robust software, and when combined with the .NET Framework, which is also open-source, it empowers developers to build cross-platform, native multi-platform applications. At its core, C# is an object-oriented language enriched with component-oriented features. Its evolution, driven in part by its open-source nature, has incorporated language elements that cater to contemporary design paradigms, making it highly versatile for various applications, including Web apps, games, and mobile apps.

This blend of an open-source ecosystem and a continually evolving language design contributes to C#'s position as a staple programming language for many applications, like the following.

- **Static typing with support for dynamic typing.** Variable types are usually declared in advance, and by default, only parameters with the correct data type can be passed. This allows for compile-time checking of type safety, meaning that type errors can be found before the code is run, making the code more reliable. However, since C# 4, C# also supports dynamic typing, which enables variables to be typed at runtime, allowing for more flexibility at the cost of foregoing some compile-time type checking.
- **Lexically scoped.** Variables are only accessible from within the code block in which they are defined, helping to keep the code organized and easy to read.
- **Imperative and Declarative.** It covers both issuing commands to the computer and describing what the program should do, leaving no space for external complications to arise.

²C# is considered type-safe because it strictly enforces data types and prevents operations that could lead to unpredictable behavior or memory access violations by catching errors at compile time or runtime.

³A programming paradigm is the classification, style, or way of programming. It is an approach to solve problems by using programming languages.

- **Functional.** This programming paradigm treats computation as the evaluation of mathematical functions without changing state and mutable data. It makes it easier to reason and write correct and robust code. Although we will not go any deeper into this within this book, it is an important feature that further solidifies C# in the industry.
- **Generic.** Defines algorithms in terms of types that can be specified later and instantiated when needed. It allows for creating data types that are not specific to any one program. This allows for the reusability of code and the ability to create more versatile programs.
- **Object-oriented.** C# supports object-oriented programming (OOP), which primarily revolves around encapsulating data and functions into units called objects. While OOP is often linked with concepts like inheritance and polymorphism, these are not fundamental to all OOP languages. OOP can be handy for modeling real-world scenarios, but it is not a one-size-fits-all solution. For instance, capturing intricate relationships using OOP can sometimes be cumbersome or lead to design issues. The structure that OOP provides can result in cleaner and more maintainable code, but like any tool, it can be misused. Knowing when and how to use OOP effectively is key.
- **Language interoperability.** The ability of two or more languages to interact with each other. This is important because it allows developers to choose the best language for the task at hand rather than being forced to use a single language for everything.
- **And component-oriented.** The technique of developing software applications by combining pre-existing and new components. This means that developers can create libraries of code that can be used in multiple applications. This can save a lot of time and money and make it easier to develop large and complex applications.

All C# programs run on .NET, using a virtual system called Common Language Runtime (CLR)⁴ and some class libraries. C# code is compiled into an Intermediate Language⁵ compatible with the Common Language Interface (CLI), allowing languages and libraries to work together smoothly.

.NET provides libraries for various tasks, organized into namespaces (C# groups for related types and members). These libraries include features for file handling, string manipulation, XML parsing, Web frameworks, and Windows Forms controls. We'll explore namespaces more in a later chapter.

C# is one of the most well-known languages. Its community and range of abilities give it a place in almost every project.

Nonetheless, there is a question we want to cover to exemplify the current position of C# in the industry and, simultaneously, learn the different paths we, as developers, could go down in our C# careers.

⁴ Implementation by Microsoft of the Common Language Infrastructure (CLI).

⁵ Intermediate Language (IL), programming language designed to be used by compilers for the .NET Framework before static or dynamic compilation to machine code.

1.2 What Can You Do With C#?

Usually, the best way to know if a given technology is the right choice for our use case is to find out if similar projects are done with it. If we are building a Web application, make sure that Web applications were built before with that technology. If we are making games, we need to find games made with that language. Although this book's projects will stay within a console application, learning a language capable of accomplishing our favorite tasks is, naturally, more often than not the path we will want to take. So to ensure that C# is the right fit, we should learn about some known use cases for C# and its frameworks. Starting with an all-time favorite, Web application development.

1.2.1 C# for Web Application Development

C# has been popular for creating Web sites and Web applications, thanks to its ability to help build sites that can change and update in real time. To do this, C# can be used with the .NET Core platform, which is a collection of tools and libraries that makes it easier to develop Web applications.

One of the tools within .NET Core is called ASP.NET Core, which is an open-source Web application framework. ASP.NET Core is used for “server-side” Web development. This means it deals with the behind-the-scenes logic that happens on the Web server (a powerful computer that hosts Web sites) before a Web page is sent to your browser.

One particular part of ASP.NET Core is called Razor Pages. It makes it simpler to build web pages that primarily stand on their own and do not have overly complicated interaction with other pages or parts of the site. It is said to be more “accessible” for developers because it is simpler to understand and work with, especially for those new to Web development.

Another interesting tool is Blazor, which allows us to build Web pages where much of the action happens in our web browser and not on the Web server. This means that after the page has loaded, it can continue to update and change without having to reload the page.

The Xbox Web site, xbox.com, is a great example of a C#-based Web site. Trustpilot uses C# for web services and app development. Furthermore, StackOverflow, renowned among the programming community for its invaluable resources and discussions, also exemplifies a successful application of these technologies.

1.2.2 C# for Windows Applications

This example illustrates a clear application, taking into account that C# was developed both by and for Microsoft. This context streamlines the development process significantly, as every feature needed for Windows application development is seamlessly integrated into the C# language and its encompassing ecosystem.

To serve as examples, we can develop Windows applications through platforms such as WinForms, a platform to write client applications for desktop, laptop, and tablet PCs, aiming to simplify this development. Windows presentation foundation, or WPF for short, is a UI framework that creates desktop client applications. Or the MetroFramework, similar to WinForms, is a framework that helps to develop clean applications through a simplified interface.

Alternatively, we can also develop Windows applications through a console application, as done with the projects covered in this book.

Examples of applications built with C# include Visual Studio and Microsoft Office.

1.2.3 C# for Linux and macOS Applications

Although Linux and macOS applications are not the first things that come to mind when discussing a Microsoft-focused technology, any application can be optimized for Unix-based systems using .NET Core. Its cross-platform development capabilities allow it to develop code usable in most operating systems with little to no required modifications. These applications include, as an example, the previous two mentioned in the preceding subsection.

For current-gen ARM processors, like the M1 and M2 chip in the newest Mac computers, Microsoft has an SDK that will allow building and running .NET code on the newest ARM devices

(direct link <https://dotnet.microsoft.com/en-us/download/dotnet/thank-you/sdk-6.0.302-macos-arm64-installer/>).

Visual Studio can also be used for ARM processors, starting with Visual Studio 2022 for Mac version 17.4

(direct link <https://learn.microsoft.com/en-us/visualstudio/releases/2022/mac-release-notes-preview#17.0.0-pre.5>).

1.2.4 C# for Mobile App Development

Using C# Xamarin, mobile app development can be possible since Xamarin (direct link <https://dotnet.microsoft.com/en-us/apps/xamarin>) allows us to wrap native components and libraries into the .NET layer, without the need to rewrite almost any code. As of 2022, a new replacement for Xamarin has arrived with MAUI (direct link <https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-7.0>), a cross-platform framework for developing native mobile and desktop apps with C# and XAML. Although currently in its early days, it is expected to, once mass adoption takes place, serve as an evolution to Xamarin-based applications. Also, some apps can be directly developed in C# and built for mobile uses, like games.

We can include apps like the Slack mobile app built on top of Xamarin and various mobile games we will go over next.

1.2.5 C# for Video Game Development

If we include indie titles,⁶ games are mostly made in C# since it is the primary language in game engines like Unity. According to the Unity Gaming Report 2022, the number of games made on the Unity platform increased by 93% this last year (Unity Technologies, 2022) (direct link https://images.response.unity3d.com/Web/Unity/%7B10460b81-b6e7-4784-a735-e7347afdf06e%7D_Unity-Gaming-Report-2022.pdf?utm_source=demand-gen&utm_medium=ceros&utm_campaign=acquisition&utm_content=2022-gaming-report-ebook&elqTrackId=d813896edf9f48e6af45a569577d8845&elqaid=4085&elqat=2).

It is also an option in engines like CryEngine, Godot, and Stride. Furthermore, it is possible in the popular game engine Unreal Engine, although an initial setup is needed.

However, it's important to mention that while C# is versatile and beginner-friendly, some game developers have concerns regarding its performance characteristics, particularly due to its garbage-collected nature. Garbage collection can occasionally cause performance hiccups, which can be critical in games that require consistent frame rates. This is why several famous game developers prefer languages with manual memory management, such as C++.

Nevertheless, many successful games, including Cities Skylines, Escape from Tarkov, and mobile games like Genshin Impact, Honkai: Star Rail, and Hearthstone, have been developed using C#.

C# and .NET can also be run on a Raspberry Pi identically to any other platform. A .NET app can run as a self-contained app or in framework-dependent deployment modes. With the community behind this programming language and its wide variety of career paths, C# quickly becomes one of the essential programming languages that everyone must learn.

So let us start by learning a bit about our context and the environment we will be working in.

1.3 Getting Started with C#

Usually the best approach to learning a new language is to first learn everything that separates it from other languages through studying its context and preferred integrated development environment (IDE).

⁶An “indie title” refers to an independently developed video game, typically created by individuals or small teams without the financial backing or support of a major publisher.

We begin our C# journey with a simple “Hello World” project.

Nevertheless, do not worry. The projects will increase in difficulty exponentially, starting slow to familiarize us with the context and getting harder over time to challenge the C# process we are getting along the way.

Our first step is to learn about our integrated development environment (IDE), the tool with which we can write and test our code of choice.

Luckily, C# counts with a native IDE, Visual Studio. To be specific, **Microsoft Visual Studio Community 2022**. The community edition, also used with the popular game engine “Unity,” is free to use and can be downloaded through the official Web site (direct link <https://visualstudio.microsoft.com/downloads/>). Visual Studio Community is a fully featured, extensible, free IDE made for creating modern applications for Android, iOS, and Windows, Web applications, and cloud services, so chances are that we might have familiarized ourselves with it before.

After setting up and starting Visual Studio, a process discussed in the appendix in case we need it, we will find ourselves at the project creation window. Here is where we can begin our first traditional project.

We want to write some code that displays a “Hello World!” message, such as with HTML tags, labels, pop-ups, or other methods that we typically find in other programming languages and frameworks. In our case, we will display it using our Console, so a simple message will show when running the program.

Simple enough, but remember that our primary goal is to familiarize ourselves with the process. We will have to tackle asynchronous programming and application programming interfaces soon enough.

In the Microsoft Visual Studio Launcher, we will find a few options. One of them is to create a new project. That is what we want to do. Continue by clicking on “Create a new project” (Fig. 1.1).

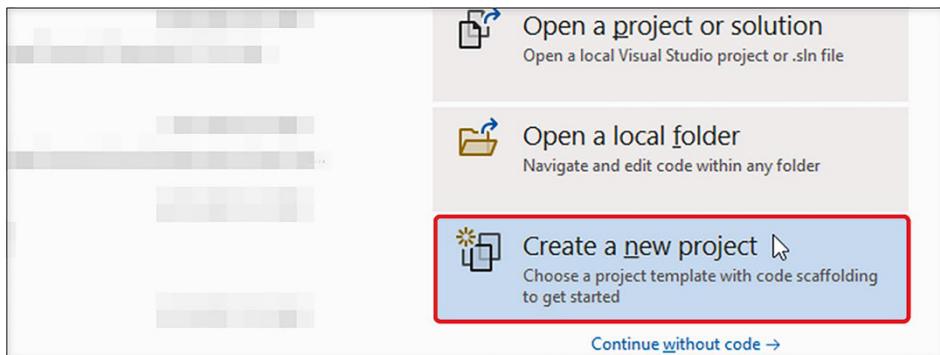


Fig. 1.1 We are starting by selecting the “Create a new project” option

In the next screen, select the template we will use for this project, a Console App.

We will also find the last template used in “Recent project templates” if we previously created a project on the left side. Since we are starting this book, assuming we have not yet created a project, we will have to select it from the list on the right.

We will find a list of templates currently installed on our device. This list varies depending on the packages we choose in the installation process.

Usually, the first template is the one we want here, specifically the C# Console App. We do need to make sure to select the correct one since there are several console app templates. We will only need the plain Console App (Fig. 1.2).

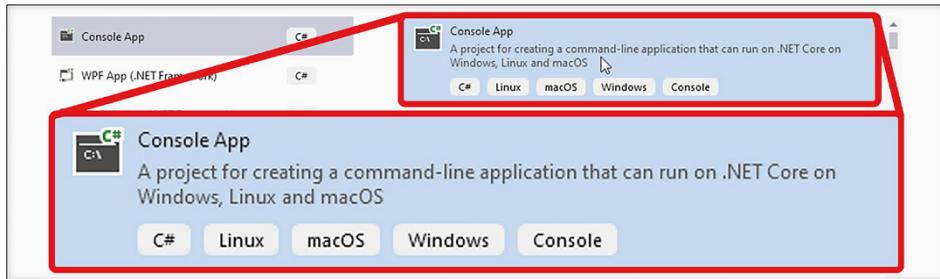


Fig. 1.2 We will mainly use the Console App project template for the projects in this book

This template will give us a preconfigured project to create a command-line application on .NET Core on Windows, Linux, and macOS devices.

After continuing by pressing the Next button, we will be prompted with a window that asks us for a project name, location, and Solution Name. We want to provide a descriptive name, like “Hello World,” and place it in our preferred directory. Also, there will be a checkbox to place the solution. A solution is a container for one or more projects. It allows us to manage dependencies and target different platforms with our code. We can see a solution as a box containing several projects, and since they are together, they can share code among them, simplifying development. This is generally done with, for example, Web development.

Although we can create a single solution for each of our projects, this time, for simplicity and to keep the projects separate, we will create a new solution for each one.

We can change the solution location individually by unchecking, but that will not be needed here; therefore, we will keep it checked. In case it does not default to a checked state, just make sure to check it.

Then continue by pressing the Next button again (Fig. 1.3).



Fig. 1.3 Name the project correctly and check the checkbox

In the next window, we can decide which framework to use for the project.

This option is the target framework we will be working on. Since we do not want to start a project entirely from scratch, we always want to start with something. In this example, we are building a .NET application, meaning we need to use a .NET Framework.

The available versions depend on the ones we installed before. In our case, we can find .NET 6 and .NET 5. We will be working with .NET 6 in this project, so just select “.NET 6.0 (Long term support)” from the dropdown menu and press “Create” to finally start up our first project.

In newer versions of Visual Studio we can also find a checkbox for “Do not use top-level statements.” Although we typically will leave that unchecked, it will not be relevant for the correct following of this book, as we will provide the base code we will start with in every chapter. However, in short, when we check that checkbox, we get the traditional structure with a “Program” class and Main() method as a result. If not, we will get our result we will see shortly (Fig. 1.4).

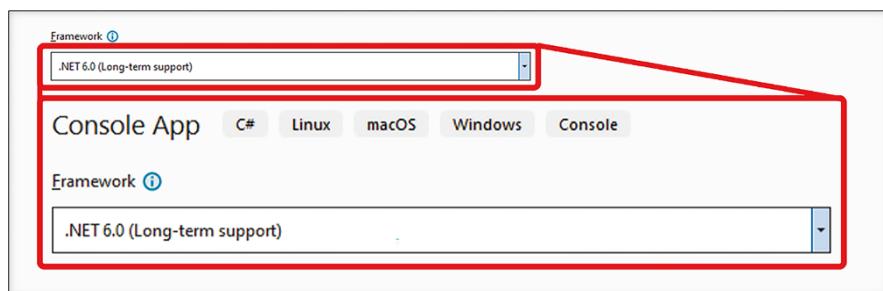


Fig. 1.4 We will be using .NET version 6.0 for our projects

A ready-to-go console app setup should greet us with the project configured to use .NET 6. Now, **what do we want to achieve, and how do we plan to do it?**

Our application will be a simple program to print a “Hello, World!” message in our Console. This goal should be, especially now with .NET 6, really straightforward.

Looking at the current project, we can see that the chosen template has already generated a Program.cs file, which includes a line of code with the exact message we want to display. But does that mean we are already done with our project? Is that it? Technically yes! However, we are not here just for that. We are here to understand how all of this works (Fig. 1.5).

A screenshot of the Visual Studio code editor showing the 'Program.cs' file. The code is as follows:

```
1 // See https://aka.ms/new-console-template for more information
2 Console.WriteLine("Hello, World!");
```

The code is highlighted with syntax coloring, and the entire file is enclosed in a red box.

Fig. 1.5 We can see the base template given by the chosen console app template

First, let us investigate what the project template includes to understand our working environment better.

On the right-hand side, we can see the Solution Explorer. That window will show our project file structure, including the Program.cs file we just discovered. If we right-click on the project name and select “Open Folder in the File Explorer,” we can see the entire folder structure that Visual Studio generated.

The .sln file represents the solution that Visual Studio uses. That is the solution we selected a location for in the project configuration and is what we will be opening with Visual Studio in the future.

Alongside the .sln file, we can find all the project contents. From the Program.cs file we saw open earlier to all the future files and folders we will be creating in future projects. This folder structure will change depending on the template, especially with more feature-rich templates. The number of folders and the depth of the layers can increase (Fig. 1.6).

Name	Date modified	Type	Size
.vs	6/24/2022 1:37 PM	File folder	
bin	6/24/2022 1:37 PM	File folder	
obj	6/24/2022 1:37 PM	File folder	
>HelloWorld.csproj	6/24/2022 1:37 PM	C# Project file	1 KB
HelloWorld.sln	6/24/2022 1:37 PM	Visual Studio Solu...	2 KB
Program.cs	6/24/2022 1:37 PM	C# Source File	1 KB

Fig. 1.6 The project structure within the Windows file explorer

Double-click on “Program.cs” to open directly in our IDE to get back to our project. We can open these files with other text editors we might have installed, but when installing Visual Studio, it sets itself as the default tool to open files with the .cs extension type.

The advantage of using Visual Studio instead of a standard text editor is mainly in the ease of use but it is not limited to that. One significant advantage is Visual Studio’s advanced error detection, IntelliSense. If we were to type out incorrect code, a regular text editor would not detect these issues, while our IDE notifies us and gives us possible solutions to that error. This advantage significantly eases the building and debugging of our applications.

Since .NET 6, the Visual Studio template has been simplified to the point where the files we work with do not have anything other than our custom code, with no need to add any boilerplate code. In the past, we usually found namespace and class declarations. Now, we only find the code that we want to run. That does not mean that namespace and class declarations are not needed anymore, but the structure changed such that we do not need to repeat that code in every program we create.

.NET 6 brought that requirement down by introducing top-level statements. This language feature allows us to implicitly set an entry point by writing statements outside a type declaration. By doing this, we no longer need to explicitly set a Main() method. Technically,

we do not need to write a single class in our project. Although we still will later exemplify other methods for C# development

(direct link <https://learn.microsoft.com/en-us/dotnet/core/tutorials/top-level-templates>).

As the included link in the code explains, “Starting with .NET 6, new C# projects using the console template generate different code than previous versions:” and “The new output uses recent C# features that simplify the code you need to write for a program. Traditionally, the console app template generated the following code:” with the relevant code included, respectively:

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

And:

```
using system;
```

```
namespace MyApp // Note: actual namespace depends on the project name.
{
    internal class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

However, as the official “.NET 6 template changes” documentation states, “These two forms represent the same program. Both are valid with C# 10.0. When we use the newer version, we only need to write the body of the Main method.”

Although we do not need to include the other program elements, we will include them later to ease our understanding of the processes and increase our programs’ transparency as soon as classes in C# are introduced.

For now, we will continue reviewing the current code we are presented with in our “Program.cs” file.

In the first line, we can find a text with the link we followed earlier at the very top. Comments are programmer-readable explanations or annotations in the source code of a computer program. They are written to make the source code easier to understand for humans. While compilers and interpreters generally ignore them, there are exceptions, such as C#’s XML documentation comments, which are meaningful to the IDE and can be used to generate documentation. Essentially, comments function in the same way across different programming languages but with some variations in how they can be utilized.

In C#, these are marked with two forward slashes “//” for single-line comments and an opening forward slash with an asterisk, closed by an asterisk forward slash “/* Comment */” for multi-line comments, as in the following example.

```
// I am a single-line comment
/*
I am a
multi-line comment
*/
```

In this case, it was added by default to include the link we followed earlier, which explained all the changes done by the template we used.

After that, we find the line at the beginning that included our message. That message is added inside of a method call. We will explain methods in further detail in the later sections of this book, but to understand what we are looking at, let us try to learn at least the basics of a method.

A method is code that performs a specific task. A program causes the code contained to be run by calling the method and specifying anything else needed for that method to run correctly.

The `Console.WriteLine` method specifically is a method that will write the data given within its argument list and move the cursor to the following line, meaning that if two or more `WriteLines` are added, the second one will be displayed on the following line, as in the following example.

```
Console.WriteLine("Hello,");
Console.WriteLine(" World!");
/*Output:
Hello,
World! */
```

Inside the parenthesis of `WriteLine`, we can specify strings, numbers, even operations on data types, and many more. We will also look over data types and operations in the following sections. Right now, what we wanted was to show a simple message.

Before going into much detail with this first purely introductory chapter, let us quickly wrap up this first project by seeing if we get the result we are after.

How we want to run our program highly depends on our situation. If we are already accustomed to working with Web apps or gaming software, we will know that the execution often happens outside of Visual Studio. However, since we only have a single line we want to display in our Console, the only thing needed in this specific case is to select the green Start arrow on the Visual Studio toolbar or by pressing F5. Additionally, to achieve the same from the command line, navigate to the project folder and use the “dotnet run” command (Fig. 1.7).

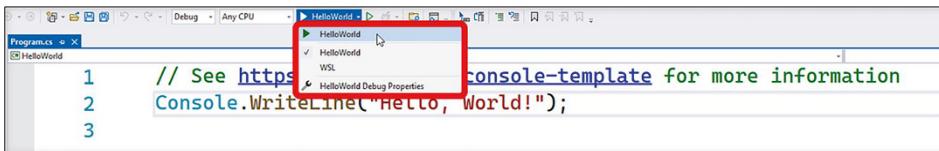


Fig. 1.7 Running any program using Visual Studio's built-in tool

This action will result in a window popping up, the Microsoft Visual Studio Debug Console. In this, we will see the result of our code. Then, as we can see, it successfully displays our “Hello, World!” message.

If the Console does not show up in our specific case, as is common on macOS devices, this can be found by pressing the Command key + F and searching for “terminal.” Or searching for the “command prompt” in the search window for Windows-based systems (Fig. 1.8).

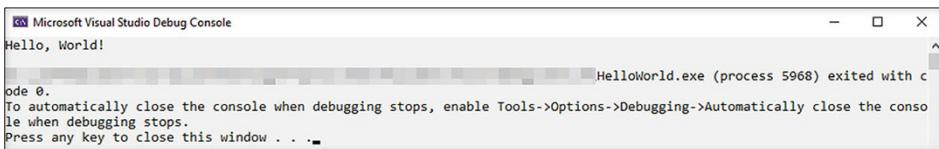


Fig. 1.8 The result of running our first project

Even if our system is identical to the one used for the figures in this book, our Console might look different from the one shown in Fig. 1.9. That is solely due to us using a simple console property to modify the background color and text color to improve the readability of the results. These are the following lines used at the very top of the code, so before everything else in our Program.cs file:

```
Console.BackgroundColor = ConsoleColor.White;
Console.Clear();
Console.ForegroundColor = ConsoleColor.Black;
```

Using these is not necessary unless we want to achieve the same look. If you are interested, we will go through console properties in a future chapter of this book. For now, let us continue with this chapter.

As a reminder of what we have learned so far, let us try adding another Console.WriteLine line to the code and write whatever message we want to display.

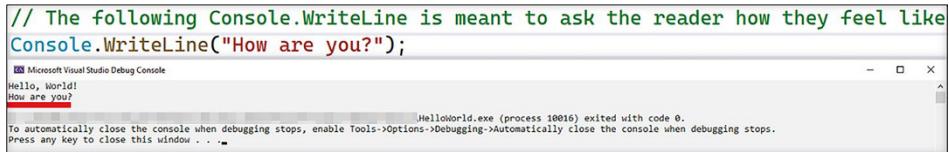
After a quick comment explaining this second Console.WriteLine, we will show the text “How are you?”

```
// The following Console.WriteLine is meant to
// ask the reader how they feel like
Console.WriteLine("How are you?");
```

This would be what we are left with.

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
// The following Console.WriteLine is meant to ask the reader
// how they feel like
Console.WriteLine("How are you?");
```

Then after rerunning it, we get the following result (Fig. 1.9).



```
// The following Console.WriteLine is meant to ask the reader how they feel like
Console.WriteLine("How are you?");
Microsoft Visual Studio Debug Console
Hello, World!
How are you?

HelloWorld.exe (process 10016) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Fig. 1.9 After some additions, we can see our new result

There we go. Naturally, that was quite a simple task. Practically, we only repeated the same existing lines and created a second comment and WriteLine combo. So, let us go deeper and try to make something fun that necessitates a tiny bit more complexity. If we come from a different programming language, we will surely recognize the code we will be developing now. However, if there is anything that might not be clear, rest assured that we will go over it during the following chapters. Now, we want to make something fun, so let us see what that will be.

NOTE The projects for this chapter and Chap. 2 will use concepts not directly covered in each chapter. However, these concepts should and will be familiar to anyone with intermediate C# knowledge or knowledge about any other object-oriented language. With that said, although this may sound like a call to skip the first two chapters, it is not recommended to do so, as these contain valuable information even for those well versed in C#. Also, the projects are pretty cool, I think.

1.4 Hello World! A Simple Interactive Storytelling App

Let us continue with the project we created and build our first proper app. Although we will keep the complexity low, we will try to make something that can already count as a finished application. After all, our goal is to make as many projects as possible so that you can then take them and extend them to the largest they can be. So, what is it, then?

1.4.1 Our Project

Let us analyze what we want to achieve here to determine the next step.

Our theory currently only includes the first steps and understanding “how to ‘hello, world!’” in the C# language. So for this project, we will want to include more than that.

You might know about conditional statements and simple input management if you are familiar with similar programming languages. If not, as said, follow along for now since we will learn more about these topics further down the line.

We will be using these to build a simple interactive storytelling application. If you have heard about Twine (direct link <https://twinery.org/>) or Ren'Py (direct link <https://www.renpy.org/>), you will know that interactive storytelling applications allow us to write interactive stories with multiple choices and endings. Although simple, we can use light apps like these to, for example, showcase a storyline we have for a project online for users to enjoy and give you feedback before any actual production begins. Furthermore, if this topic interests us, we can expand its capabilities and features with everything we learn down the line and even develop some serious competition for current options.

Naturally, this chapter's project will not yet be a competitor in the storytelling app market but will be an excellent introduction to learning about this language's possibilities.

1.4.2 Our Code

Starting our project, we could simply go ahead and erase anything we had set before and end up with a blank project. So make sure to do so next.

We will only work with what we have seen now and a few simple additions. These additions include the `ReadKey()` method, which simply waits for the user to press any key and keeps a record of it, and an `if` statement, which is the same as in any other programming language. Well, at least conceptually, that is.

This project will not include a story builder mechanic since that exceeds this chapter's scope, so we will simply build our story within the code as console messages using the `Console.WriteLine()` method primarily. Starting with our initial prompt.

```
Console.WriteLine("Hello, World!' - A tiny story by TutorialsEU");
Console.WriteLine(
    "Press any key to progress,
    and either 'a' or 'b' when prompted"
);
```

We are introducing our story and adding a small tutorial of sorts. This marks the next step as being a key press. As we just mentioned, we can use `ReadKey()` for that. Just like `Console.WriteLine()` add a `Console.ReadKey()` right after our current text.

```
Console.WriteLine("Hello, World!' - A tiny story by TutorialsEU");
Console.WriteLine(
    "Press any key to progress,
    and either 'a' or 'b' when prompted"
);
Console.ReadKey();
```

This line will stop the execution of the program and wait for any kind of key press. Once pressed, we can start our story right after.

```
Console.WriteLine("\n You are walking through a dense forest where  
visibility is limited to a mere few meters. Time is passing by,  
and it feels like hours have passed since you last saw  
anything that wasn't just trees.");
```

You might have noticed that there is an \n at the beginning of the text. That \n can simply be seen as if the “Enter” key was used. It will move the written text to the next line leaving a clear space between each text. If you come from Web development, you might remember the
 tag doing something similar. In concept, it would be the same.

Now simply keep adding ReadKey()s and more text until you want to get to a choice point, like we did here, for example.

```
Console.WriteLine("\n You are walking through a dense forest where  
visibility is limited to a mere few meters. Time is passing by,  
and it feels like hours have passed since you last saw  
anything that wasn't just trees.");  
Console.ReadKey();  
Console.WriteLine("\n What seems like a clearing opens up in front  
of you as you desperately try to approach it. Revealing a path  
that splits into two, which path may you choose?  
path 'a' or 'b'?");
```

As we can see, we are asking the user to press either “a” or “b” to choose from two different paths. We need to somehow get what key the user pressed and then continue the story accordingly. This can be done by adding the result of ReadKey() to a variable. Variables are a space to store data in, which is also similar to other languages.

The way we would do that is as follows.

```
Console.WriteLine("\n What seems like a clearing opens up in front of  
you as you desperately try to approach it. Revealing a path that splits  
into two, which path may you choose? path 'a' or 'b'?");  
var path = Console.ReadKey().Key;
```

We are adding Console.ReadKey() to a “var” or variable called “path.” Also, we are adding a “.Key” at the end of our ReadKey(). The latter is to get what key was pressed specifically. There we will get things like “Escape” if the user presses the Esc key, or “Space” if the user presses the space key. Really simple behavior that will serve us to get interaction from the user. So now that “path” has stored the exact key the user pressed, we can use that to branch the story.

For that, we use the if statement. In short, this will execute the code within the “if” if the condition is met. If not, we get sent to an optional else. For now, we will need something like this:

```
if (path == ConsoleKey.A)
{
    //If path is the A key then this is executed.
}
else
{
    //If path is NOT the A key then this is executed.
}
```

The condition to meet for the inside of the if statement to happen is “path == ConsoleKey.A.” This means that if the user presses the “a” key, we will run whatever code is contained within the curly braces. If it is not the “a” key, anything within the else is run. In this project, we will use the else to detect if the user pressed the “b” key. Not perfect since any other key will be detected as a “b” key as well, but it will work for what we need right now.

So, next, simply write your story depending on the choice made.

```
if (path == ConsoleKey.A)
{
    Console.WriteLine("\n You decide to follow path 'a'. The path
        leads you to a sign stating 'west'. If you are heading
        west, you hope to find a way out of the forest. Suddenly,
        a rumbling sound comes from the mountainside.");
    Console.ReadKey();
    Console.WriteLine("\n As it appears to be a landslide, you try
        to avoid it by going down the right side of the path
        towards what appears to be a safe zone.");
    Console.ReadKey();
    Console.WriteLine("\n It seems as you avoided the disaster, you
        look towards the path you are on now and see a sign, it
        says 'north'.");
    Console.ReadKey();
    Console.WriteLine("\n Seeing as you cannot go back anymore, you
        decide to continue this way.");
}

else
{
    Console.WriteLine("\n You decide to follow path 'b'. The path
        leads you to a sign stating 'north'. Meaning that you are
        heading north you hope to find a way out of the forest.
        Suddenly, you hear a distant rumble.");
    Console.ReadKey();
    Console.WriteLine("\n Further up dust clouds seem to form, being
        far enough from the danger, it seems like you are safe.
        You continue on your path");
}
```

To clarify, our story will return to the same storyline after branching into two. If you are interested, this writing style is called a foldback structure in narrative. So we want to add the shared storylines right after our if statement like this:

```
Console.WriteLine("\n Further up dust clouds seem to form, being  
    far enough from the danger, it seems like you are safe.  
    You continue on your path");  
}  
Console.ReadKey();  
Console.WriteLine("\n Following the path north, you encounter a young  
    person looking directly at you with a friendly face. Nonetheless,  
    you feel the need to be cautious.");  
Console.ReadKey();  
Console.WriteLine("\n 'Hello, traveler! Fret not, for you now save!'  
    Says the friendly figure. 'Should I trust him?' You ask yourself.  
    What do you do? Trust him 'a', or not 'b'?");
```

And after a few lines, we are back to another choice. Now, we would simply have to repeat the same block as before. With a tiny change, we do not have to write “var” again, just the variable name, like this:

```
path = Console.ReadKey().Key;
```

Then, again, branch out our story into two.

```
if (path == ConsoleKey.A)  
{  
    Console.WriteLine("\n You decide to trust him, following him to  
        an opening that reveals the escape you sought after for so  
        long. It was good to trust him! He lead you to freedom!");  
    Console.ReadKey();  
    Console.WriteLine("\n 'Hello, World!' He exclaims. Are you safe now?");  
    Console.ReadKey();  
    Console.WriteLine("\n You look back, and he is gone, together with  
        the forest. Only a vast open space is what is visible from  
        your point of view. Are you safe now? You do not know the answer.");  
}  
else  
{  
    Console.WriteLine("\n You decide to not trust him, going back the way  
        you came from. Although something is changed. It does not seem  
        like you truly are going back. The path is not the same anymore.  
        But a clearing appears, making you run towards it in hopes of  
        finding an escape.");  
    Console.ReadKey();
```

```
Console.WriteLine("\n You find yourself at the edge of the forest,\n to your left you can hear someone exclaim 'Hello, World!' .\n It seems to be the young person from earlier. How could he\n get here so fast? Are you safe now?");\nConsole.ReadKey();\nConsole.WriteLine("\n You look to the front, and you see what seem\n to be infinite plains with no tree or building in sight.\n You look back, and the person is gone, together with the forest.\n Only a vast open space is what is visible from your point of view.\n Are you safe now? You do not know the answer.");\n}
```

This time we even have multiple endings! Well, sort of. Nevertheless, that will conclude our tiny story! Now, simply add an “End screen,” a final line representing the end of our story.

```
}
```

```
Console.ReadKey();\n\nConsole.Clear();\nConsole.WriteLine("'Hello, World!' - A tiny story by TutorialsEU');
```

Now run this and follow along with this tiny adventure we just wrote (Fig. 1.10).

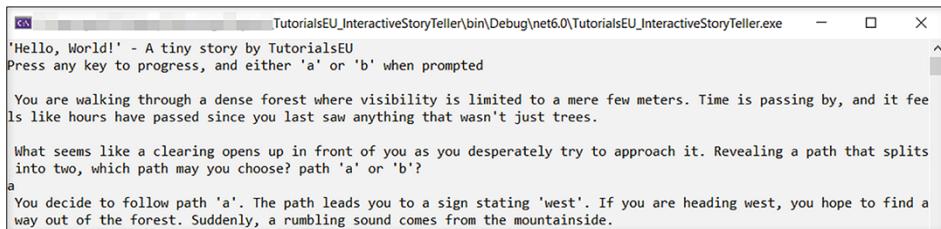


Fig. 1.10 Our final result after building our tiny interactive storyteller app

We are done! We hope you liked our tiny story! Make sure to make this your own story and see if you can make some additional variations to this project. If you liked this process and want to do more, make sure to return to this project once you learn a few more programming concepts. This can be vastly extended to be a proper product, and if you set up some sort of a builder and a way to visualize it outside of the Console, like a Web site, we can be the next Twine. That is for sure.

Obviously, from now on, the topics will become more complex. Starting with variables in the next chapter, all the way to APIs and cryptography in the later chapters. Gradually, we will become better and better developers through the usage of programming theory and the development of a bunch of small projects with each passing chapter.

We hope that with this approach, we will be able to deliver as much value as possible to you, teaching everything there is about C# with this practical project-based approach to make the process less tedious.

NOTE If anything is not working or you simply want to get our code implementation, we will link to our official GitHub page at the end of each chapter, where you can always find all of the projects from this book. The links should be working for the foreseeable future, and if not, our Web site Tutorials.eu, our official discord, or any of our YouTube videos are excellent places to reach out for corrected links.

So, this first chapter taught us everything about C# we need to know going forward. Then, in the end, we set up our first project. Although a relatively simple one, it introduced us to the basics of app development. Now, after this conclusion, let's get ready for the next chapter.

1.4.3 Source Code

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_SimpleInteractiveStorytellingApp

1.5 Summary

This chapter taught us:

- The C# object-oriented, type-safe programming language is a general-purpose, multi-paradigm programming language.
- The C# language allows developers to efficiently create safe and robust Microsoft .NET code.
- C# has widespread usage across Web, OS, and videogame applications development.
- Visual Studio is the native Integrated Development Environment for C# with other options like Visual Studio Code or Rider.
- The first steps for creating a C# console .NET application using .NET version 6.0.
- A simple console application contains a sample “Hello World!” written in a pre-structured environment.
- Comments are a programmer-readable explanation or annotation in the source code of a computer program.
- In C#, comments are marked with two forward slashes “//” for single-line comments and an opening forward slash with an asterisk, closed by an asterisk forward slash “/* Comment */” for multi-line comments.

- `Console.WriteLine` is a method that will write the data given within its parentheses and move the cursor to the following line, meaning that if two or more `WriteLines` are added, the second one will be displayed on the next line.
 - A simple Console project can be executed by pressing the F5 key or by pressing the green Start arrow on the Visual Studio toolbar.
-

References

- Microsoft Corporation. (2023). What is .NET MAUI?. Retrieved May 1, 2023 from learn.microsoft.com: https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-7.0
- Microsoft Corporation. (n.d., n.d. n.d.). Xamarin. Retrieved May 1, 2023 from [dotnet.microsoft.com: https://dotnet.microsoft.com/en-us/apps/xamarin](https://dotnet.microsoft.com/en-us/apps/xamarin)
- .NET Foundation and Contributors. (2022). .NET Micro Framework. Retrieved April 1, 2022 from [dotnetfoundation.org: https://old.dotnetfoundation.org/projects/dotnet-micro-framework](https://old.dotnetfoundation.org/projects/dotnet-micro-framework)
- codecademy. (n.d., n.d. n.d.). What is .NET? Retrieved 04 01, 2022 from www.codecademy.com: https://www.codecademy.com/article/what-is-net
- Indicative Inc. (n.d., n.d. n.d.). What Is A Programming Paradigm? Retrieved April 1, 2022, from www.indicative.com: https://www.indicative.com/resource/programming-paradigm/
- Lestiyo, I. (2021). 2021 GAMING REPORT. Unity Technologies. Retrieved April 1, 2022
- Lutkevich, B. (2021, December n.d.). runtime. Retrieved April 1, 2022 from www.techtarget.com: https://www.techtarget.com/searchsoftwarequality/definition/runtime
- Microsoft Corporation. (2022a). Microsoft technical documentation. Retrieved March 31, 2022, from docs.microsoft.com: https://docs.microsoft.com/en-us/
- Microsoft Corporation. (2022b, March 11). What is the .NET SDK? Retrieved April 1, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/core/sdk
- Unity Technologies. (2022). Unity Gaming Report 2022. Unity Technologies. Retrieved December 9, 2022 from https://images.response.unity3d.com/Web/Unity/%7B10460b81-b6e7-4784-a735-e7347afdf06e%7D_Unity-Gaming-Report-2022.pdf?utm_source=demand-gen&utm_medium=ceros&utm_campaign=acquisition&utm_content=2022-gaming-report-ebook&elqTrckId=d813896edf9f48e6af45a5695



C# Data Types and Variables

2

This Chapter Covers

- Using variables in C# as opposed to other languages
- Working with the different types and data types
- Using the Random Class in C#
- Concatenating strings in code
- Developing a dice-rolling application

In Chap. 1, we got our first presentation on the C# language, were introduced to the C# console app development process with our interactive storytelling app, and learned about what this book has prepared for us in future chapters. Hopefully, this has helped those intermediate in the C# programming language understand what to expect from this book, as well as introduced the process of C# console app development for those experienced in other object-oriented languages.

In this chapter, we will be working on some randomness. Specifically, we will work on a dice-rolling application, which simulates the act of rolling a dice, commonly used in board games. This digital version is perfect for board gaming nights when physical dice are unavailable or inconvenient. Similar to our previous project, this will serve as a great base to expand upon with everything we learn in future chapters. Apart from that, it will be perfect for introducing us to variables, data types, concatenation, and the generation of random numbers.

As you can see, during these first five chapters, we will cover everything we will need about the basics of C# to build our projects, from variables to collections, classes, and methods, with the intent of teaching us how our basic programming concepts work within C# and solidifying our bases on this language. As we all know, a house is only as strong as its foundation.

So then, let us start this chapter by discovering everything about how C# works with variables.

2.1 Variables in C#

Every value or thing we use through our code is, at least, temporarily stored somewhere within our machine. We can visualize this process by imagining a scenario where we must calculate the outcome of adding two numbers together. At first, receive the values that must be added and memorize them temporarily, like the numbers 2 and 3. This process of temporally memorizing is similar to our usage of variables, where we maybe decide on creating a variable that temporarily stores each number. Then we can use those memorized numbers, add them, and present the outcome.

Additionally, in compiled languages like C#, variable names are replaced by the data location itself once executed. This simply means that once a program is run, the variables cannot be renamed anymore simply because the names are not used anymore. In languages like Python, for example, you can change a variable's name, and it will still point to the same location. However, in languages like C#, that is not possible.

Let us use an analogy to understand the workings of a variable properly. We must imagine variable names like addresses. The people who live at that address can change, but the building will remain the same, so if we want to know who is currently living there, we can find that out. While if we were to change the address, we would not be able to find the building anymore, thus losing any reference to that original space.

Listing 2.1 Pseudocode to Visualize the Compilation Process of a Variable

```
#A
var beachHouse = "Jim";
var cityHouse = "Cindy";
#B
afternoonsunset street 123 = "Jim";
majormaincity street 456 = "Cindy";
```

#A Before Execution

#B After Execution

Now it makes sense why we would not be able to change the name since that would change the location of that variable and not point toward the correct value anymore.

Also, variables must be assigned before their value can be obtained unless they present default values. In our building analogy, we can take the situation where an address is given, but no person is living in it. Then, if you ask about the inhabitants, naturally, you will get either nothing or that it is empty. So someone must inhabit that building before we can report on who exactly is living there. Then also, some buildings may have inhabitants

by default, therefore representing default values. That is again in contrast to other programming languages like Python. Python is not “statically typed.” We do not need to declare variables or their type before using them. A variable is created as soon as we assign a value to it. In C#, even if using var to create a new variable but it doesn’t get initialized, we get the following error.

```
CS0818: Implicitly typed locals must be initialized
```

Listing 2.2 This Code Forces an Error for an Unassigned Local Variable, Since We Are Trying to Use a Variable That Has Not Been Initialized Yet

```
var beachHouse = "Jim"; // Correct
var cityHouse; // CS0818: Implicitly typed locals must be initialized
int mountainHouse; // Technically correct as long as its set before use
Console.WriteLine(mountainHouse); // CS0165 Use of unassigned local
// variable 'name'
```

Another thing to keep in mind is that to declare variables, it is considered good practice to declare them using camel case, meaning that the first letter must always be lowercase, and for each word that comes after that as part of the name of the variable, its first letter is capitalized. To expand on that, we can also find the snake case and Pascal case, which we will see used in other cases. We can see them represented in the following graphic.

camelCase

- First word lowercase, each consecutive work capitalized

snake_case

- All lower case, words separated by underscore

PascalCase

- All words capitalized

In Listing 2.2, we used “int” instead of “var” to create a variable. Besides storage and access of variables, the data type is also part of a variable, precisely the variables’ value, for example how in our analogy, the data type would be “person.”

NOTE It is worth noting that, although not used by convention in C#, all-uppercase snake case, also known as SCREAMING_SNAKE_CASE, is used for some identifiers like static constants in Java, for its naming, so to simplify its readability.

In C#, there are a multitude of types. So to be able to declare values in C# correctly, we must start by finding the answer to: **What is a data type?**

2.2 Variable Data Types

C# is a type-safe language, ensuring that variables of a specific type can only hold values of the same type. This prevents type-related mismatch errors and enables Visual Studio's IntelliSense to warn about potential type errors during code development, without requiring compilation.

A data type defines the size and type of variable values. Essentially, it specifies the kind of value a variable can hold. For example, a numeric variable can only store numbers, not words. The size refers to the maximum capacity of specific types, such as Integral numeric types.

The “var” keyword in C# allows you to define a variable without specifying its data type. Instead, the data type is inferred based on the value assigned to the variable. This can be useful in certain situations, but whether to use “var” or explicit data types depends on personal preference and coding style. The important thing is to ensure that the variable type can be identified, even if declared as “var”: a “var.”

```
var unclearVariable = otherVariable;
Program redundantVariable = new Program();
var intuitiveVariable = new Program();
```

As of .NET 6, you can use “new()” instead of specifying the class to initialize a new variable, but this only works if the variable is declared with a specific data type:

```
var intuitiveVariable = new Program();
Program alsoCorrectVariable = new();
```

Ultimately, the primary goal when declaring variables is to reduce verbosity without sacrificing clarity. In the following example, myLongVariable is unnecessarily declared as a long, while myVarVariableInt is declared as a var and becomes an integer. Meanwhile, myVarVariableLong is declared as a var and becomes a long:

```
long myLongVariable = 100;
var myVarVariableInt = 100;
var myVarVariableLong = 10000000000; // Acts as a long, since the
// number is too big for an integer.
```

Some developers prefer using “var” to avoid repeating data types, while others value the clarity provided by explicitly stating the data type. As of .NET 6, the target-typed syntax has become popular, reducing the need for “var” unless its adaptability is required.

Even if you choose not to explicitly declare data types, understanding them is essential, as using “var” doesn't negate their importance. Once initialized, a variable declared with “var” becomes a specific data type, and its rules still apply.

So then, let us go over those to know what we are dealing with.

2.2.1 int (Integral Numeric Types)

As previously stated, ints represent numbers. Specifically, whole numbers with no decimal points. So things like “1,” “2,” or “123456789.” The Int data type also has a specified maximum range of $-2,147,483,648$ to $2,147,483,647$ or -2^{31} to $2^{31} - 1$. Anything larger than that requires using the “long” data type, which, although the same in every other way, has a range of $-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$ or -2^{63} to $2^{63} - 1$.

To visualize where you might need to consider the use of a “long” data type, say we were building a simulation for a gravitational system and are using real-world mass values for our calculations. Taking our Earth’s gravity as a base for 1 g, we would need to be able to work with Earth’s total mass, calculated to be around 5.972×10^{24} kg, which is significantly more than what int can hold. This example would necessitate the use of long to store our mass. Now, sure we could use approximate data, like 1 being the base representing the mass of our Earth. However, if these calculations need to be precise and perfect, we will not have that option available to us.

2.2.2 float (Floating-Point Numeric Types)

Floating-point numbers are positive or negative whole numbers with a decimal point. As an example, “1.5” would be considered a float. “15” would not. Floating-point numbers get their name from how the decimal point can “float” to any necessary position. Similar to Integral numeric types, float has a maximum range, this one being $\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$, equivalent to seven decimal digits of precision, anything larger than that requiring the use of “double,” ranging from $\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$, equivalent to 15–16 decimal digits.

2.2.3 char (Unicode UTF-16 Character)

Char represents a literal character. So single letters, among other things. For example, the letter “a” would need to be stored in a char. Nevertheless, the limit of char stays at single characters, since for whole words, as mentioned before, we need to use “String,” another data type that, internally, is stored as a sequence of Char objects. Because of that, single letters can also be stored in strings since they will just be converted into a single-length sequence of chars; essentially, just a char.

2.2.4 bool (Boolean Value)

Boolean is simply the representation of true and false. Yes and no. “You shall pass” or “You shall not pass.” A bool expression can control the execution flow in the if, do, while, and for statements as well as the conditional operator ?. We will cover this further later in this book.

Generally, these data types are considered the most widely used data types. However, we can still find many more. Let us attempt to summarize their definition in the following tables.

2.2.5 The C# Built-in Value and Reference Types (Tables 2.1 and 2.2)

Table 2.1 The definitions for each C# value type

C# value type	Definition
bool	Either true or false. Either pass or not pass
byte	Numbers ranging from 0 to 255
sbyte	Number ranging from -128 to 127
char	A single character
decimal	A decimal number with up to 28–29 decimal digits
double	A decimal number with up to 15–16 decimal digits
float	A decimal number with up to seven decimal digits
int	Numbers ranging from -2,147,483,648 to 2,147,483,647
uint	Numbers ranging from 0 to 4,294,967,295
nint	Numbers with a range depending on platform (computed at runtime). Positive and negative numbers
nuint	Numbers with a range depending on platform (computed at runtime). Only positive numbers
long	Numbers ranging from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
ulong	Numbers ranging from 0 to 18,446,744,073,709,551,615
short	Numbers ranging from -32,768 to 32,767
ushort	Numbers ranging from 0 to 65,535

Table 2.2 The definitions for each C# reference type

C# reference type	Definition
Object	Holds addresses that refer to objects
String	A sequence of zero or more characters
Dynamic	Type that bypasses compile-time type checking

Besides their data type, variables can also vary in their type. In other words, not only what kind of value they can store but also how they store it. Let us learn more about that.

2.3 Variables Types in C#

NOTE These definitions assume a pre-existing knowledge about classes, instantiation, methods, and other advanced topics. Although the general idea should be understandable, it is advised to revisit this section of this chapter once these topics are fully covered. The intention behind this is purely organizational, with the idea that once we have completed this book, we can always go back to the individual chapters to get anything we need related to this topic. For example, in this case, we will find variable types within the chapter dedicated to variables instead of the chapter dedicated to, let's say, methods.

C# defines seven types of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables.

Generally, variable types define where in our code they can be seen, the number of variables they can hold, and where they are used.

To get more into specifics, let us go through each of them and learn how they would be correctly implemented.

2.3.1 Static Variables

Static variables, so those declared with the `static` modifier, are variables that share their value between all instances of their class. So in the following example, `staticVariable` will keep its value in every instance of `TwoVariables` while `nonStaticVariable` will be unique, or also described as instance variables, as we will see in Sect. 2.3.2.

Listing 2.3 The Variable “`staticVariable`” Represents a Static Variable, While the Variable “`nonStaticVariable`” Represents an Instanced Variable

```
class TwoVariables
{
    public static int staticVariable;
    public int nonStaticVariable;
}
```

As for use cases, static variables are commonly used if only one copy is required. This situation means that static variables are shared between all class instances. Generally, this type of variable should only be used if the code needs it, for example, the maximum number of students a school class can have. And instanced variables should be used whenever that is not specifically necessary, for example, the individual students of any given school class.

2.3.2 Instance Variables

Any non-static field would be an instance field.

A class instance variable is created when a new instance of that class is created. In other words, taking both static variables and instance variables into consideration, if we created more than one instance of a class, meaning that if we created more than one concurrent occurrence of a block of data, static variables would have a shared value between each one of them while instance variable would be separate and unique for each individual instance.

Listing 2.4 The Variable “nonInstanceVariable” Represents a Static Variable, “instanceVariable” Represents an Instanced Variable, and “alsoNonInstanceVariable” a Static Variable Again

```
class ThreeVariables
{
    public static int nonInstanceVariable;
    public int instanceVariable;

    static void Method() {
        int alsoNonInstanceVariable;
    }
}
```

2.3.3 Array Variables

Array variables are, essentially, elements of an array. So those variables contained within an array. We will learn more about arrays in Chap. 5.

Listing 2.5 A Variable of Type Array Containing Several String-Type variables

```
string[] array = new string[] { "arrayVar01", "arrayVar02", "arrayVar03"};
```

2.3.4 Value Parameters

Value parameter variables are those declared upon invocation of functions.

The creation of Value parameter variables occurs when a new instance of a parameter containing method, instance constructor, accessor, operator, and anonymous function to which the parameter belongs is called. This can be further explained through the use of the following example:

```
static void Method(string methodParameter)
{
    //methodParameter will act as a value parameter, not affecting
    // the original variable
    methodParameter = "Changed parameter";
    Console.WriteLine(methodParameter); //Will result in "Changed
    // parameter"
}

static void Main()
{
    string parameter = "Value Parameter"; //original variable
    Method(parameter); //This call sets methodParameter to "Value
    // Parameter"
    Console.WriteLine(parameter); //Will still result in "Value
    // Parameter"
}
```

Using the example above, only the value of `parameter` is passed, and a new variable is created, contained entirely within the method. So if displayed from within the method, the new value is shown, while outside, the original value remains.

2.3.5 Reference Parameters

Reference parameter variables are those declared with a `ref` upon writing of function members.

In contrast to value parameters, reference parameters are not created upon method invocation but are just references to existing variables. While usage is similar, any changes done within a method will affect the original variable since no new allocation was created, but the original one is used.

```
static void Method(ref string methodParameter)
{
    //parameter will act as a reference parameter, affecting the
    //original variable
    methodParameter = "Changed parameter";
}

static void Main()
{
    string parameter = "Value Parameter"; //original variable
    Method(ref parameter); //This call sets methodParameter to "Value
    //Parameter"
```

```
Console.WriteLine(parameter); //Will result in "Changed  
//Parameter"  
}
```

Since the parameter variable was passed as a reference, the method now contains a direct reference to the original variable declaration, making any changes within the method effective for the original. Also, both need `ref` in their declaration since the method call `Method(ref parameter)` must specify the variable as a `ref`, and the method itself `static void Method(ref string methodParameter)` needs to specify that it receives a reference to a variable.

It should be mentioned that in C#, parameters of a method are passed by value by default. The “`ref`” keyword is used to indicate that a parameter should be passed by reference. While in contrast, objects are passed as reference by default, important to note when handling passing objects.

2.3.6 Output Parameters

Output parameter variables are those declared with an `out` upon invocation of function members.

In contrast to reference parameters, output parameters do not require the variable to be initialized before it is passed. Still, output variables act as a reference parameter because they just reference existing variables, and any changes done within a method will affect the original variable since no new allocation was created, but the original one is used.

```
static void Method(out string methodParameter)  
{  
    //parameter will act as a reference parameter, affecting the  
    //original variable  
    methodParameter = "Changed parameter";  
}  
  
static void Main()  
{  
    string parameter; //original variable, now NOT initialized  
    Method(out parameter); //This call sets methodParameter to "Value  
    //Parameter"  
    Console.WriteLine(parameter); //Will result in "Changed  
    //Parameter"  
}
```

This time, similar to reference, the variable was passed in as a reference changing the original at method scope, with the difference that we could leave the original as it is this

time. In the same way as with reference parameters, `out` has to be used for the caller as well as the method.

2.3.7 Local Variables

Local variables are those declared within the local block where they are supposed to be used since they will stay within the scope of that block.

They can also be seen in a `for`-statement, a `switch`-statement, a `foreach` statement, a `using` statement, or a `specific-catch` statement.

A local variable is used where the variable's scope is within the method in which it is declared, meaning that only statements within that block can reference and use it. It can also be used as a constant whose value cannot be modified within the method or statement block in which it is declared.

```
static void Main()
{
    string parameter = "Local"; //local variable declaration
    Console.WriteLine(parameter); //Will result in "Local"
}
Console.WriteLine(parameter); //Will result in the following error:
//The name 'parameter' does not exist in the current context
```

Attempting to use a local variable outside of its scope will result in an error since anything outside that specific scope has no reference to the local parameter.

2.4 The Random Class

In this subsection, we'll begin our project for this chapter, focusing on generating a random whole number, specifically an integer. The purpose of generating a random number is to simulate a dice roll in our dice-rolling application, ensuring that each throw results in an unpredictable outcome, as one would expect from a physical die. Before diving into the implementation, we need to discuss the Random Class in C#, which will be used to generate these random integers.

The Random Class is a class responsible for the generation of pseudo-random numbers, specifically an algorithm that creates random numbers following a set of requirements given on its execution. See the following example as a simple Random Class implementation in code.

```
Random random = new Random();
```

Now, why did we mention pseudo-randomness specifically? Although we are dealing with randomness throughout this chapter, it is known that true randomness is just not achieved. That is not because we are learning things wrong, but simply because true randomness is not achievable via code, only pseudo-randomness.

To quote D. E. Knuth, “Pseudo-random numbers are chosen with equal probability from a finite set of numbers. The chosen numbers are not completely random because a mathematical algorithm is used to select them, but they are sufficiently random for practical purposes.” Pseudo-randomness is sufficient for our and practically any purposes, so while it is not an issue, it is worth acknowledging.

In fact, to quote Stephan M. Levy (2014), “There are many natural phenomena from which random numbers may be derived. However, for the purposes of economic or statistical studies, such naturally occurring random numbers are not convenient to use. Economic and statistical studies may require thousands or even millions of random numbers. The process of capturing and storing so many numbers from a natural phenomenon may be burdensome in terms of time and computing resources. Further, economic and statistical studies need to be reproducible, for purposes of debugging and replicating the analysis.” We can deduce that, in most cases, this pseudo-randomness is the desired behavior. Where true randomness is difficult to work with, pseudo-randomness gives us the advantage of predictability and reproducibility.

This fact might come in handy for future chapters, so even though we will not be using it during this chapter, we can keep it in mind when working on our later chapter projects, where this pseudo-randomness comes into play. Always good to know what is going on behind the curtains.

2.4.1 Generating a Random Number

The Random Class provides methods that can generate random Boolean values, random floating-point values with a range inside 0–1, random 64-bit integers, randomly selecting an element from an array or collection, and random bytes. But for this chapter, we are focusing on another method, the Next() method.

By using the Next() method, we can generate a single random whole number between two values defined within.

```
Random rnd = new Random();
int randomNum = rnd.Next(-5, 5);
Console.WriteLine(randomNum); //A random number between -5 and 5
//gets generated
```

In this example, using the Next() method, we can generate a value between -5 and $+5$ and store that value on randomNum. Then we display randomNum using the WriteLine() method.

Additionally, to generate a random floating-point number within a specific range, we can use the NextDouble() method:

```
Random rnd = new Random();
double randomFloat = rnd.NextDouble();
Console.WriteLine(randomFloat); // A random floating-point number
//between 0 and 1 gets generated
```

Conceptually, it seems simple enough, and if we come from other programming languages, this should be common knowledge. So then, let us use it right away with this chapter's project. We start with a simple test implementation and then turn our project into a roll the dice app, ready for our next game night.

2.5 A Random Number

After creating a new project, we will find ourselves in front of a simple, empty template that will display a “Hello World!” message on execution. This is the base we will be using for this second project (Fig. 2.1).



Fig. 2.1 We will be starting our second project with an empty project, as shown

2.5.1 Our Project

Let us analyze what we want to achieve here to determine the next step.

After the previous StoryTelling app, where we only made a simple sequence of `WriteLine()` methods interactive, we want this project to be useful. Although it is still simple, this should be of use for any tabletop game or as an addition to a larger project that needs a random dice throw. Then it is what we are building this time, a “RollTheDice” app.

We will make use of the `Random` Class and the `Next()` method to generate a random value from 1 to 6, like the faces on a traditional die. And for those who want to see something new applied again, we will have to use what is known as a loop. This app will stay ready to detect a single key press to generate a new number. That way, there is no need to restart the app, which would be pretty annoying.

As we have mentioned, the concept of a loop in C# has not yet been covered. However, it should be second nature to anyone intermediate in C# or knowledgeable in any other object-oriented language. Nonetheless, Chap. 5 will talk about iterators in C#, so rest assured that we will cover this topic.

2.5.2 Our Code

As mentioned, we will start by building a base to understand the usage of the Random Class by an implementation. Then we will modify our code to build the final "RollTheDice" app.

We can now start by creating a new variable that will hold our future random number in this new project. We want to generate a random whole number without any decimals, which means that we should store our number in an `int` data type.

To create a variable, type out the data type, a name of our choosing that ideally represents its use or contained value. In our case, it will be `randomNumber`. Then initialize it to 0. And the result should look something like this:

```
int randomNumber = 0;
```

NOTE Variables can stay un-initialized without throwing any error, as long as their first use is assigning it a value. We will test that out right after our next step.

For this next step, we will try to write to console our initial value. Since we already learned how to work with `console.WriteLine()`, we only need to add "Number start as" to our `randomNumber` variable. That should result in this:

```
int randomNumber = 0;
Console.WriteLine("Number start as " + randomNumber);
```

Let us see what console result this gives us after running it (Fig. 2.2).



The screenshot shows the Microsoft Visual Studio Debug Console window. The code in the editor is:

```
// See https://aka.ms/new-console-template for more information
int randomNumber = 0;
Console.WriteLine("Number start as " + randomNumber);
```

The output window shows the following text:

```
Microsoft Visual Studio Debug Console
Number start as 0
ARandomNumber.exe (process 18208) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Fig. 2.2 The result of writing the variable value to the console

This result is already the right step toward our goal. However, we have not yet addressed what concatenation is. So then, let us go over the **basics of concatenation**.

Concatenation is the procedure of appending one string to another string. As we just saw, this process is achieved by adding a `+ sign` in between our strings or variables.

By joining our string with our variable, we created a concatenated string. This is achieved by converting the variables to a string and adding the conversions together. This results in a new string with the result of the concatenation being assigned to it, while leaving the original strings unmodified.

In the case of “integer” values, this is done through a method called “`ToString()`” which we will cover in more detail in a later chapter. But just to note, variables added to this concatenation will always be converted to the string data type before any operations are done on them.

Let us look at string literals and, additionally, the `+=` operator.

For **strings**, we would see examples like ours, simply typing strings and adding a `+` sign in between to add them together, for example like this:

```
string text = "My First String" + "My Second String ";
```

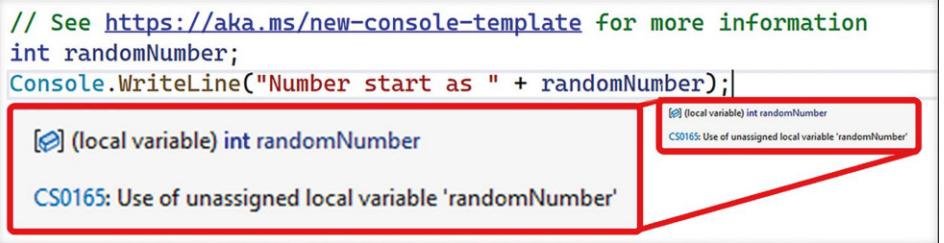
The **`+=` operator**, on the other hand, is to add string to the initial concatenation after.

That way, we can add additional values to a concatenation, for example, when asking the user for any input. An example of the `+=` operator would be:

```
string text = "My First String" + "My Second String ";
text += "My Third String";
```

With these concatenation systems, we would be covered for the vast majority of use cases, but like with everything, there are many other ways to accomplish this goal. For this project, there is no need to delve any deeper into this topic, but we will get back to it in a later chapter.

With that, we can now generate our random number, but just before that, let us check what happens if we did not initialize our `randomNumber` and then try to display it. So instead of `int randomNumber = 0`, we simply use `int randomNumber` (Fig. 2.3).



```
// See https://aka.ms/new-console-template for more information
int randomNumber;
Console.WriteLine("Number start as " + randomNumber);
[?] (local variable) int randomNumber
CS0165: Use of unassigned local variable 'randomNumber'
CS0165: Use of unassigned local variable 'randomNumber'
```

Fig. 2.3 We get a “Use of unassigned variable” error. We need to initialize our `randomNumber` variable if the first thing we do is display it

Immediately we see that we get an error, error code CS0165 “Use of unassigned local variable ‘randomNumber.’” As we discussed earlier in the chapter, most variables have to be initialized before they can be used. Unless we assign it, there is no use, so let us leave it as it was, so `int randomNumber = 0`.

Perfect, that answers our question. Now, finally, let us see how random numbers are generated. We will be using `Random.Next()` for that. We will first create a new variable to store our `System.Random` data.

We will need a new data type, the Random data type. Then, initialize with a new Random().

```
int randomNumber = 0;
Console.WriteLine("Number start as " + randomNumber);
Random random = new Random();
```

So with that random initialization, we are creating a generator that we can later use to generate the values needed. For example, we want to generate a number between 0 and 100, using the Next() method we talked about before. We can now actually generate one with the following code:

```
int randomNumber = 0;
Console.WriteLine("Number start as " + randomNumber);
Random random = new Random();
randomNumber = random.Next(0, 100);
```

To finally display that, just use console.WriteLine() as usual.

```
int randomNumber = 0;
Console.WriteLine("Number start as " + randomNumber);
Random random = new Random();
randomNumber = random.Next(0, 100);
Console.WriteLine("Our random number is " + randomNumber);
```

Let us check out the final result once we run this little program (Fig. 2.4).

```
// See https://aka.ms/new-console-template for more information
int randomNumber = 0;
Console.WriteLine("Number start as " + randomNumber);
Random random = new Random();
randomNumber = random.Next(0, 100);
Console.WriteLine("Our random number is " + randomNumber);
```

Microsoft Visual Studio Debug Console
Number start as 0
Our random number is 63
RandomNumber.exe (process 6204) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

Fig. 2.4 We see the result after generating a random number and displaying it

A final result with a random number is achieved. We can see it starting as 0 and, after generating, in our case, being assigned the number 63. However, note that your result may vary as this value is randomly generated.

NOTE In this method, the value 100 will never be generated, so only values from 0 to 99 will be generated when the values 0 to 100 are given. So this method is between the values of 0 and 100 excluding the latter. We will see this again later.

Alright, so getting it to work was easy enough. Now, how do we turn this into a usable app? We essentially need some interface so that the user knows how to work the app and functionality akin to a real dice-throw.

The latter should be easy enough now that we know how to generate random numbers. Let us simply rename our “randomNumber” variable to “randomDiceRoll” and remove the first `Console.WriteLine()` method. Also, remember to add our changed variable name to the second `Console.WriteLine()` method, like so:

```
int randomDiceRoll = 0;
Random random = new Random();
randomDiceRoll = random.Next(0, 100);
Console.WriteLine("Our random number is " + randomDiceRoll);
```

Then, we need to change the values we set to generate to represent the six sides of a die. For that, we will have to set the `random.Next()` method to go from 1 to 7. Why 7? In the `Next()` method, the `maxValue` for the upper limit is exclusive, meaning that the range includes from `minValue` up to `maxValue-1`. So if we want 1 to 6, we need to write 1 to 7.

```
int randomDiceRoll = 0;
Random random = new Random();
randomDiceRoll = random.Next(1, 7);
Console.WriteLine("Our random number is " + randomDiceRoll);
```

And technically, that would suffice. However, this does not yet resemble a finished app. So let us add some additional code to make it feel more real. At the very top, start by greeting the user and explaining the app’s functionality, like this:

```
Console.WriteLine("Roll the dice! A dice rolling application for
your game nights!");
Console.WriteLine("\nThe dice used for this application has
6 sides, with chances to roll results from 1 to 6.");
int randomDiceRoll = 0;
```

Now, with what we have seen in our previous project, let us use the “Press any key” method again so it feels more controlled.

```
Console.WriteLine("Press any key to roll the dice!
Good luck on your roll!");
Console.ReadKey();
int randomDiceRoll = 0;
```

Great, this already “pauses” the flow to wait for the user to start a dice roll. But if we run it now, we will see that it ends once the die is rolled. Having to rerun the app each time you want to roll the die is not user-friendly. We need a way to repeat the die-roll as many times as we want. The previous chapter showed that `ReadKey()` tells us what key was explicitly pressed. We could use that to check. For example, if the user presses the Esc key, we exit the app, but if not, we continue rerolling.

Then, let us do the same as previously done and store the specific key that was pressed here.

```
ConsoleKey keyPressed = Console.ReadKey().Key;
int randomDiceRoll = 0;
```

Now, we need to use a loop (a programming concept we will dive deep into in Chap. 5) to make the code repeat as long as Esc is not pressed. Specifically, we will use a while loop. We should know about while loops from languages like Javascript, Java, C++, and many others, so this should not be your first encounter. But if it is, for a quick rundown, a while loop is a statement that repeats whatever it contains only if its condition is true.

So if we check, for example, if our key press was not an Esc key, this is done with the `!=` operator, a concept we will cover in Chap. 3, then repeat the body of the while loop, and within, we ask for another key press, like this:

```
ConsoleKey keyPressed = Console.ReadKey().Key;
while (keyPressed != ConsoleKey.Escape)
{
    int randomDiceRoll = 0;
    Random random = new Random();
    randomDiceRoll = random.Next(1, 7);
    Console.WriteLine("Our random number is " + randomDiceRoll);
    keyPressed = Console.ReadKey().Key;
}
```

We essentially made it so that each time we press a key and enter the while condition, we continue looping through the random number generation code as long as we do not press the Esc key. Just remember to add the internal `ReadKey()` method so it does not continue looping without pause since this can cause what is known as *an infinite loop*.

Now, we only need to make the given result seem more akin to a dice-roll app and add some instructions to press any key to reroll, like this:

```
Console.WriteLine("Roll the dice! A dice rolling application for
your game nights!");
Console.WriteLine("\nThe dice used for this application has
6 sides, with chances to roll results from 1 to 6.");
Console.WriteLine("Press any key to roll the dice!
Good luck on your roll!");
ConsoleKey keyPressed = Console.ReadKey().Key;
```

```
while (keyPressed != ConsoleKey.Escape)
{
    int randomDiceRoll = 0;
    Random random = new Random();
    randomDiceRoll = random.Next(1, 7);
    Console.WriteLine("You rolled the number " + randomDiceRoll);
    Console.WriteLine("\nPress any key to reroll!
        Press ESC to stop the application");
    keyPressed = Console.ReadKey().Key;
}
```

Now, let us add a `Console.Clear()` method at the beginning of the while loop so it cleans our canvas each time it enters.

```
Console.WriteLine("Roll the dice! A dice rolling application
    for your game nights!");
Console.WriteLine("\nThe dice used for this application has
    6 sides, with chances to roll results from 1 to 6.");
Console.WriteLine("Press any key to roll the dice!
    Good luck on your roll!");
ConsoleKey keyPressed = Console.ReadKey().Key;
while (keyPressed != ConsoleKey.Escape)
{
    Console.Clear();
    int randomDiceRoll = 0;
    Random random = new Random();
    randomDiceRoll = random.Next(1, 7);
    Console.WriteLine("You rolled the number " + randomDiceRoll);
    Console.WriteLine("\nPress any key to reroll!
        Press ESC to stop the application");
    keyPressed = Console.ReadKey().Key;
}
```

And this is enough already. This works as it should and seems clear enough as the app it tries to be. Let's see what our final result looks like (Fig. 2.5).

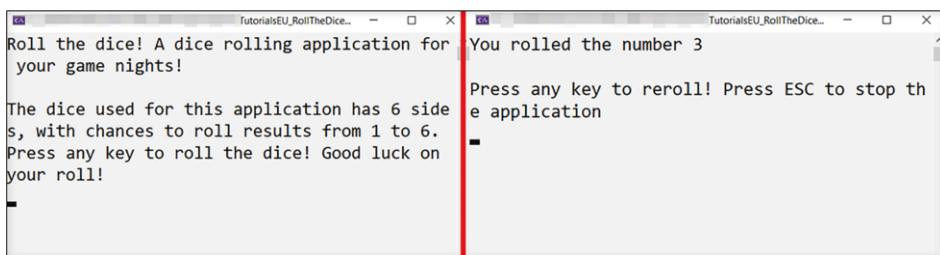


Fig. 2.5 A functioning dice-rolling app showing the initial prompt, pictured on the left, and the looped result, pictured on the right

We could now add a way to visually represent the rolled die and even make a simple animation with a rolling die, but let us not get too far ahead of ourselves. We already made use of while loops even though we start properly working with them in Chap. 5. Nevertheless, you can always return to this project after finishing future chapters and improving on this. Don't let this be the extent of your portfolio. With everything we cover in this book, we can get this app to be much more than its current stage.

Other than that, we now have a working app that should be ready for your next D&D night!... Wait, how many faces can a D&D die have...? And you have to use seven dice?!

Well, that could be your first step to extend this chapter's app. But for now, let us get ready for the next chapter.

2.5.3 Source Code

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_RollTheDice

2.6 Summary

This chapter taught us:

- Variables in programming are a way to store specific data until further use.
- C# defines seven types of variables: static variables, instance variables, array elements, value parameters, reference parameters, output parameters, and local variables.
- Most of the time, we will be using the types of int (Integral numeric types), float (Floating-point numeric types), char (Single character), and bool (Boolean value).
- The Random Class represents a pseudo-random number generator, an algorithm that generates numbers that meet specific requirements for randomness.
- Concatenation is the procedure of appending one string to another string.
- If variables are used, they need to be initialized first. Not doing so will result in a compilation error.
- With a few exceptions, we can use var to declare variables. However, it is mainly used to have variables that allow the compiler to infer their type.
- .Net 6 introduces target-typed syntax, which allows for cleaner code by not requiring type specification when the type is known.

References

Microsoft Corporation. (2021, September 15). Built-in types (C# reference). Retrieved April 12, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/built-in-types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/built-in-types)

- Microsoft Corporation. (2021, September 15). Compiler Error CS0165. Retrieved April 12, 2022 from [docs.microsoft.com](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/cs0165): <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/compiler-messages/cs0165>
- Microsoft Corporation. (2021, September 15). How to concatenate multiple strings (C# Guide). Retrieved April 6, 2022, from [docs.microsoft.com](https://docs.microsoft.com/en-us/dotnet/csharp/how-to/concatenate-multiple-strings): <https://docs.microsoft.com/en-us/dotnet/csharp/how-to/concatenate-multiple-strings>
- Microsoft Corporation. (2021, August 6). IntelliSense in Visual Studio. Retrieved April 12, 2022 from [docs.microsoft.com](https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022): <https://docs.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022>
- Microsoft Corporation. (n.d., n.d. n.d.). Random Class. Retrieved April 6, 2022 from [docs.microsoft.com](https://docs.microsoft.com/en-us/dotnet/api/system.random?view=net-6.0): <https://docs.microsoft.com/en-us/dotnet/api/system.random?view=net-6.0>
- Saluja, N. (2020, March 07). C# Constructor. Retrieved April 12, 2022 from [www.c-sharpcorner.com](https://www.c-sharpcorner.com/article/constructors-in-C-Sharp/): <https://www.c-sharpcorner.com/article/constructors-in-C-Sharp/>
- Stephan M. Levy. (2014, October 7). Fooled by Pseudo-Randomness. Retrieved April 12, 2022, from [papers.ssrn.com](https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2507276): https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2507276
- TechTarget. (2008, September n.d.). What is an instance? Retrieved April 12, 2022 from www.techtarget.com: <https://www.techtarget.com/whatis/definition/instance>
- TheDeveloperBlog. (n.d., n.d. n.d.). C# Byte Type. Retrieved April 12, 2022 from thedevoloperblog.com: <https://thedevoloperblog.com/byte>
- TutorialTeacher. (2021, August 19). Variable Scopes in C#. Retrieved April 12, 2022 from [www.tutorialteacher.com](https://www.tutorialteacher.com/articles/variable-scopes-in-csharp): <https://www.tutorialteacher.com/articles/variable-scopes-in-csharp>
- W3Schools. (n.d., n.d. n.d.). C# Arrays. Retrieved April 12, 2022 from [www.w3schools.com](https://www.w3schools.com/cs/cs_arrays.php): https://www.w3schools.com/cs/cs_arrays.php
- W3Schools. (n.d., n.d. n.d.). C# Method Parameters. Retrieved April 12, 2022 from [www.w3schools.com](https://www.w3schools.com/cs/cs_method_parameters.php): https://www.w3schools.com/cs/cs_method_parameters.php



C# Operators and Conditionals

3

This Chapter Covers

- Using operators and expressions in C#
- Correctly controlling Operator associativity and precedence
- Working with Conditional Selection statements in C#
- Developing a Rock-Paper-Scissors minigame

In the previous chapters, we have already set up two projects. We learned about variables, data types, and the Random Class. We also learned about concatenation and how we can get custom results using more exciting features like conditional statements and loops. These previous chapters were somewhat limited in terms of theory and had to up the complexity by adding potentially unfamiliar statements. However, this chapter will achieve the necessary theoretical complexity to develop a proper project. This is mainly because we will finally deep dive into conditional statements in C#, covering statements such as the switch statement and the ternary operator. It will also discuss everything commonly missed about operators, operator evaluation, operator precedence, and, additionally, learning about nullable and what constant variables have that switch statements want, among others.

Throughout this book, each chapter attempts to, even if simple for developers with intermediate experience, educate on details we might have missed during our development as programmers, including points commonly left behind and teachings that will polish our knowledge about the basics while, if our experience stems from outside C#, revealing some of the similarities and differences of C# relative to other coding languages.

So then, what will we be building in this book's third chapter? The question is, how can we make something fun and interesting that we can be proud of while simultaneously constraining ourselves to our set boundaries of this chapter's coverage? By making a

Rock-Paper-Scissors game, of course! Use switch cases to check the player's move, use random for the AI's move, then use operators to compare and choose the winner! A classic!

We still have some time before we get into the password encrypters and E-commerce Discord bots, so let us, for now, continue having a bit of fun with these minigames.

Let us start this chapter with a bit of theory, defining an operator in C#.

3.1 What Are Operators and Expressions

Programming operators do not stray very far from following the more common knowledge of mathematical operators, where we use addition, subtraction, and other signs to represent different calculations.

An operator is a sign or symbol telling the compiler to perform specific mathematical or logical manipulations, serving us the same way they do outside the programming scope.

Mostly directly mimicking their non-code counterparts, we can find the addition operator (+) as an example. As we have seen in the previous chapter, its behavior was adding together two strings to form a single, continuous string.

Listing 3.1 The Line of Code That Used an Operator in the Previous Chapter's Project

```
Console.WriteLine("You rolled the number " + randomDiceRoll);
// Output: You rolled the number 5
```

When operators are used, they form part of an “expression,” which is essentially a combination of operators and operands, the latter being the ones that are operated upon, like numbers in a mathematical operation (Fig. 3.1).

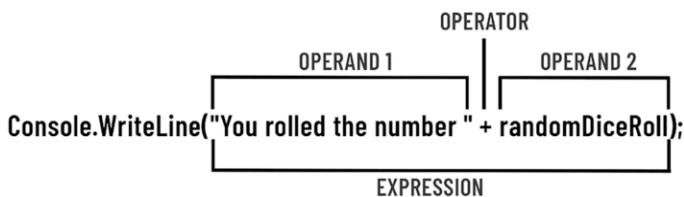


Fig. 3.1 The anatomy of an operator expression in C#

Let us take as an example the following simple calculator app, let us call it “Simple Calculator.” This code will prompt the user to enter two numbers and an operator. It then performs the specified operation and displays the result. If we want, we can follow along and create our own mini project before our main chapter project. If we do, we can get even more than 11 projects out of this book, so it is just winning for us. However, the intent behind this is to exemplify, so there is no need to do so.

Listing 3.2 A Simple Calculator Application Capable of Doing Addition and Subtraction

```
Console.WriteLine("Simple Calculator");
Console.WriteLine("-----");

Console.Write("Enter first number: ");
double num1 = double.Parse(Console.ReadLine());

Console.Write("Enter second number: ");
double num2 = double.Parse(Console.ReadLine());

Console.Write ("Enter an operator (+, -): ");
string op = Console.ReadLine();

double result;

switch (op)
{
    case "+":
        result = num1 + num2;
        break;
    case "-":
        result = num1 - num2;
        break;
    default:
        Console.WriteLine("Invalid operator");
        return;
}

Console.WriteLine("Result: " + result);
```

To develop this application, just a few steps were needed.

Start by adding the following code to display a header for the calculator:

```
Console.WriteLine("Simple Calculator");
Console.WriteLine("-----");
```

Next, prompt the user to enter the first number using the following code:

```
Console.Write("Enter first number: ");
double num1 = double.Parse(Console.ReadLine());
```

Prompt the user to enter the second number in a similar manner:

```
Console.Write("Enter second number: ");
double num2 = double.Parse(Console.ReadLine());
```

Prompt the user to enter an operator for the calculation (+ or -):

```
Console.Write ("Enter an operator (+, -): ");
string op = Console.ReadLine();
```

Create a variable named result to store the result of the calculation:

```
double result;
```

Use a switch statement to perform the calculation based on the operator entered by the user:

```
switch (op)
{
    case "+":
        result = num1 + num2;
        break;
    case "-":
        result = num1 - num2;
        break;
    default:
        Console.WriteLine("Invalid operator");
        return;
}
```

And finally, display the result to the user:

```
Console.WriteLine("Result: " + result);
```

And with that, we have already created a small and simple application that uses operators. However, just as with non-programming applications, operators present themselves in several ways, from the more common addition and subtraction operators to the lesser-known Assignment and Lambda declaration. Especially the Lambda operator is one that can vastly improve our applications but that even intermediate to advanced users still do not use.

So then, let us look at the primary groups that form the operators in the C# language.

3.2 Types of Operators

In C#, we can find many operators to work in our expressions. Many of those are supported by built-in types and allow us to perform basic operations. These operators are grouped into several operator types, as follows.

Arithmetic Operators

Arithmetic operators are operators that perform arithmetic¹ operations with operands. These are meant for performing mathematical operations on the given operands, such as addition, subtraction, multiplication, division, and modulus.

Listing 3.3 A List of Variables Utilizing the Various Arithmetic Operators

```
int arithmeticOperatorAdd = 5 + 5; // arithmeticOperator = 10
int arithmeticOperatorSubtract = 5 - 5; // arithmeticOperator = 0
int arithmeticOperatorMultiply = 5 * 5; // arithmeticOperator = 25
int arithmeticOperatorDivide = 5 / 5; // arithmeticOperator = 1
int arithmeticOperatorModulus = 5 % 5; // arithmeticOperator = 0
```

Comparison Operators

Comparison Operators are operators that compare operands. These are meant to compare different operands with each other, such as less than, greater than, less than or equal, and greater than or equal.

Listing 3.4 A List of Variables Utilizing the Various Comparison Operators

```
bool comparisonOperatorLessThan = 4 < 5;
// comparisonOperator = true;
bool comparisonOperatorGreaterThan = 4 > 5;
// comparisonOperator = false;
bool comparisonOperatorLessThanOrEqual = 4 <= 5;
// comparisonOperator = true;
bool comparisonOperatorGreaterThanOrEqual = 4 >= 5;
// comparisonOperator = false;
```

Equality Operators

Equality operators are operators that check if their operators are equal or not. Usually seen as part of the comparison operators, these are categorized in their own group of operators.

¹Arithmetic is the branch of mathematics that deals with the study of numbers using various operations on them. Basic operations of math are addition, subtraction, multiplication, and division (StudyPad, Inc, n.d.).

Listing 3.5 A List of Variables Utilizing the Various Equality Operators

```
bool equalityOperatorEqual = (6 == 5); // equalityOperator = false  
bool equalityOperatorNotEqual = (6 != 5); // equalityOperator = true  
// Both numbers do not match so Equal is false, while NotEqual is true
```

Conditional Boolean Operators

Boolean operators, also known as logical operators, are operators used to determine the logic between operands. These serve as a gate for logical operations, such as AND and OR.

Listing 3.6 A List of Variables Utilizing the Various Boolean Operators

```
bool booleanOperatorsAND = (5 > 3) && (5 > 10); // booleanOperators = false;  
bool booleanOperatorsOR = (5 > 3) || (5 > 10); // booleanOperators = true;  
// Left side is true so && is false, while || is true
```

In the operator priority table, we will go through all the different operators in C# and their priority in an expression.

In the context of operator priority, the sequence in which operations are executed in an expression containing multiple operations is important. While we can often apply the rules of mathematical operation priority to simple expressions, making them straightforward to understand, this approach does not always work for more complex expressions. This is because, in programming, operations that are similar to those in mathematics may have different levels of precedence.

In examples where addition and multiplication exist, the order is obvious enough, but in an expression where we include, for example, a division and a modulus operator, the order of operations is not as obvious. For that, let us go over the basics of operator priority in C#.

3.3 Operator Priority

We are used to execution priority in mathematical operations, with examples like parentheses prioritizing the operations within them and multiplications preceding additions. Nevertheless, when it comes to programming operations, these precedences are often not considered by many, thinking that order takes care of precedence.

In every programming language that includes operator calculations, so, virtually every programming language, operator precedence is considered. Luckily, most of the time, precedence mimics mathematical operation precedence. This mimicry means that usual rules like parentheses first and multiplication before addition still hold up.

However, programming operators are not limited to those found in mathematical operations, nor do all precedences mimic those, meaning that we must know what these C# code-unique priorities are.

3.3.1 Operator Precedence

When working with expressions containing multiple operations, we generally try to understand which one is handled first by using the deeply rooted rules we learned in math class. Although this is right in many cases, in programming, we often find ourselves in front of operators not used in mathematics, so we get to learn a few additional rules.

One such rule we use in mathematics is called PEMDAS, which stands for Parentheses, Exponents, Multiplication, and Division (from left to right), and Addition and Subtraction (from left to right). This acronym helps us remember the order of operations to solve mathematical expressions.

As an example, we can take a simple expression of addition, subtraction, and multiplication like the following. We might be tempted first to do addition and subtraction before multiplication. However, due to our previous mathematics knowledge, which includes PEMDAS, we should know how this expression will be handled.

```
int precedence = 10 + 2 * 6 - 3;  
int precedenceControlled = 10 + (2 * 6) - 3;  
// 2 * 6 = 12, then 10 + 12 = 22, then 22 - 3 = 19.
```

But as soon as we start working with programming-specific operators, PEMDAS will not be able to help us anymore since we have no more rules left for, let's say, equality or conditional AND operators.

As per the official Microsoft documentation, we get the following table which lists the C# operators starting with the highest precedence to the lowest. The operators within each row have the same precedence and will thus follow the regular left-to-right order of precedence unless specific exceptions such as those noted in Sect. 3.3.2 for operator associativity occur (Microsoft Corporation, 2022) (direct link <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/>) (Table 3.1).

Table 3.1 This table depicts some operator precedence rules, starting with the highest precedence to the lowest. The operators within each row have the same precedence and will thus follow the regular left-to-right order of precedence

Operators	Category or name
x.y, f(x), x[y], x? <y>, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y</y>	Primary
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false	Unary
x..y	Range
x * y, x / y, x % y	Multiplicative
x + y, x - y	Additive
x << y, x >> y	Shift
x < y, x > y, x <= y, x >= y, is, as	Relational and type-testing
x == y, x != y	Equality
x & y	Boolean logical AND or bitwise logical AND
x ^ y	Boolean logical XOR or bitwise logical XOR
x y	Boolean logical OR or bitwise logical OR
x && y	Conditional AND
x y	Conditional OR

3.3.2 Operator Associativity

As seen in the previous table for operator precedence, there are multiple cases for operators with the same precedence. The left-to-right rule usually resolves this uncertainty. However, this varies depending on the situation, and hence the determination of precedence through operator associativity.

Left-associative operators are evaluated in the left-to-right order. This evaluation, by far, is the most common solution for equal precedence resolution of expressions. All operators, except for assignment operators, null-coalescing operators, and the conditional operator ?:, are left-to-right associative, for example, in additive operators, as seen in the following example:

```
int result = 1 + 2 - 3; // Evaluated as (1 + 2) - 3 Resulting in 0
```

Right-associative operators are evaluated in the right-to-left order, as previously mentioned, for example, in assignment operators, as seen in the following example:

```
int finalResult = result = 2;
// Evaluated as finalresult = (result = 2) Resulting in 2
```

Outside of operator precedence, operands must also be evaluated in a specific order.

3.3.3 Operand Evaluation

Similar but unrelated to operator precedence and associativity, we can see that an expression's operands are also evaluated from left to right. We can see some solutions for operand evaluation orders using this short list of examples from the Microsoft official documentation. We can see that in an expression, the operands are evaluated first before the operators. In an example like the first, $a + b$, the operands a and b are evaluated before the operator (Microsoft Corporation, 2022) (direct link <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/>) (Table 3.2).

Table 3.2 This table shows the order of evaluation for a list of different expressions

Expression	Order of evaluation
$a + b$	$a, b, +$
$a + b * c$	$a, b, c, *, +$
$a / b + c * d$	$a, b, /, c, d, *, +$
$a / (b + c) * d$	$a, b, c, +, /, d, *$

This pre-evaluation of operands will result in this case in a being a and b being b since nothing additional has to be considered, but the importance of operand precedence becomes obvious with cases like expressions that include returning methods. Consider the following program.

Listing 3.7 This Simple Program Prints on the Console the Result of Adding the Return Values of Both Methods

```
int x = 0;
Main();

int f1()
{
    x = 5;
    return x;
}

int f2()
{
    x = 10;
    return x;
}

void Main()
{
    int p = f1() + f2();
    Console.WriteLine(p); // Output: 15
}
```

In the case of `int p = f1() + f2();` the order of execution is `f1()`, `f2()`, `+`. This precedence ensures that both functions are valued first to determine the method int return and then add them together to get the desired final int value. So essentially, the final expressions after evaluating the operands would result in the following:

```
int p = 5 + 10;
```

Additionally, although each operand is usually evaluated in an expression, some operators evaluate operands conditionally. This change means the left-hand side of conditional operators, such as conditional logical AND (`&&`) and OR (`||`) operators, is evaluated before deciding if further operand evaluation is necessary.

For example, in an expression where two relational operations are performed with a conditional AND in the middle, there is no need to evaluate the second operation. Since both need to return “true,” for the AND to return “true”, as soon as one results in “false,” we can assume that AND will result in “False,” as shown in the following example:

```
bool booleanOperatorsAND = (5 < 3) && (5 < 10);  
// (5 < 10) is never evaluated since (5 < 3) is already false,  
// clearly indicating that the AND operator will result in false
```

This conditional operand evaluation could be seen as memory and processing optimizing procedures by avoiding unnecessary calculations.

In an evaluation of `a && b`, the evaluation of `b` is unnecessary if `a` does not result in “true,” since the resulting expression will result in false no matter what the result of the evaluation of `b` might be. Knowing this can be of great value for the development of efficient code. Keeping in mind what is being considered by the processor and what is not can potentially lead to serious performance improvements in our applications, especially if we start to size them up significantly.

3.4 Conditional Statements

In C#, we can find many decision-making statements, known as conditional statements. These are logical blocks with the purpose of executing a block of code depending upon an available condition.

These available conditions are certain expressions that must result in a determined outcome. This condition can result in a Boolean deciding the flow of a program toward a true or a false outcome, but also in a more case-based conditional block, determining the flow depending on a specific outcome.

However, essentially, all conditional statements are based on true-or-false conditions. We can visualize the general conditional statement flow with the help of the following graphic (Fig. 3.2).

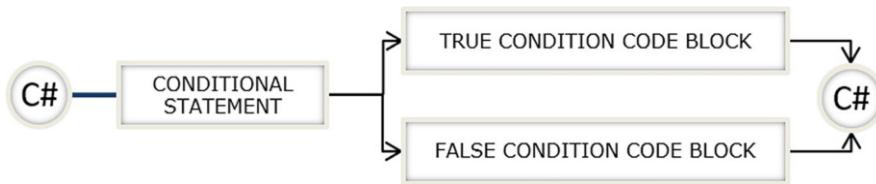


Fig. 3.2 A visual representation of a conditional statement flow assuming the use of an if conditional statement followed by an else statement. “C#” represents code before and after the conditional statement

In C#, we can find the following selection statements: the “if” statement and the “switch” statement.

3.4.1 The if Statement

The “if” statement is a conditional statement evaluating a Boolean condition that executes a containing code block depending on the result. The condition for the “if” statement needs to evaluate to a Boolean value, for it is that which decides on further execution. If the Boolean evaluates to `true`, the contained code block will be executed. If `false`, it will not.

The “if” statement syntax² is `if (condition) { //code block if true}` using the “if” keyword tailed by a condition inside of parentheses, followed by curly braces enclosing the code block to be executed if the condition returns true. If the Boolean expression results in `false`, the execution will continue beyond the conditional statement.

```

int number = 1;

if (number == 1)
{
    Console.WriteLine("If resulted in true!"); //This is the true one
}
  
```

The “else” statement is a fail catch for “if” statements, essentially executing what is contained within them if the previous “if” statement fails with a `false`.

The “else” statement can only be used directly after an “if” statement.

```
if (condition){ // code block for true} else { // code block for false}.
```

²When referring to a [programming language](#), the syntax is a set of rules for grammar and spelling. In other words, it means using [character](#) structures that a computer can interpret (Computer Hope, 2022).

The syntax is an “else” keyword followed by curly braces enclosing the code block to be executed if the “if” condition is `false`.

```
int number = 2;

if (number == 1)
{
    Console.WriteLine("If resulted in true!");
}
else
{
    Console.WriteLine("If resulted in false :("); //This is the true one
}
```

The else-if statement can be used to include additional conditions to be evaluated if the previous “if” or “else if” statement fails. The difference between simply creating another if statement instead of an “else if” is that if the previous if statement returns true, the “else if” statement will not be evaluated.

```
if (true)
{
    Console.WriteLine("This text will be written on console");
}
else if (true) // This will never be evaluated
{
    Console.WriteLine("This will not be written on console");
}
```

This condition is useful for handling various conditions by using a single “if” followed by multiple “else if” statements representing different conditions. Any number of “else if” after the “if” statement can be used to handle every condition.

The syntax is mostly the same as an if statement, with the only exception being the keyword itself. So starting with the `else-if` keyword, tailed by parentheses containing the Boolean expression, followed by curly braces enclosing the code block to be executed if the condition returns `true`.

```
int number = 2;

if (number == 1)
{
    Console.WriteLine("If resulted in true!");
}
```

```
else if (number == 2)
{
    Console.WriteLine("Else if resulted in true!");
    //This is the true one
}
else
{
    Console.WriteLine("If resulted in false :(");
}
```

Additionally, C# supports the use of if statements with or without curly braces; however, it is important to note that using braces is considered better practice. When braces are used, they create a compound statement, treating the code within as a single statement for the “if” or “else” path, which executes only the next statement. For example:

```
if (true)
{
    Console.WriteLine("This text will be written on console");
    // This is within the if's body
}

if(true) Console.WriteLine("This text will be written on console");
// This is within the if's body

if (true)
Console.WriteLine("This text will be written on console");
// This is within the if's body
Console.WriteLine("This text will be written on console");
// This is NOT within the if's body
```

Not using curly braces after an “if” statement can lead to bad practice and potential confusion, as demonstrated in the last line of the example code above. It is always recommended to use curly braces to ensure clear code structure and avoid misunderstandings.

Nested conditional statements can use “if,” “else if,” or else statement inside another “if,” “else if,” or “else” statement, and hence the “nested” naming. This feature allows the implementation of multiple conditions depending on the success of previous conditions. However, it is advised not to make overly extensive use of nested statements since it is generally a symptom of complex and unmaintainable code that should be improved and simplified. We will go over alternative solutions later in this chapter.

```
int number = 2;

if (number == 1)
{
    Console.WriteLine("If resulted in true!");
}
else if (number > 1)
{
    Console.WriteLine("Else if resulted in true!");
    //This is the true one
    if (number == 2)
    {
        Console.WriteLine("And nested if resulted in true!");
        //This is also the true one
    }
}
else
{
    Console.WriteLine("If resulted in false :(");
}
```

While on the topic of complex and unmaintainable code, we must emphasize a general rule that should always be followed when developing in any programming language to prevent this from happening, the importance of readable naming.

In programming, choosing descriptive and meaningful variable names is crucial, especially when working with conditional statements. For example, consider the following code snippet:

```
if (! IsNotWorking == true) {}
```

It might be confusing for a reader to understand the intended logic behind the code. On the other hand, a more clear and straightforward approach would be:

```
if (IsWorking) {}
```

The goal should be to write code that is as close to natural language as possible, making it easier for others to understand. This not only helps in producing clean code but also makes the code more maintainable and scalable. By following this recommendation, we can ensure that our code is readable and understandable, leading to a more efficient and productive development process.

3.4.2 The Ternary Operator “?:”

We can also find a short-hand if-else statement, the conditional operator ?:, also known as the ternary conditional operator. It is called the “ternary operator” because it consists of a condition, a `true` block, and a `false` block.

If the conditional statement only uses one condition followed by an “else” statement, this operator can replace multiple lines of code with a single line.

The syntax of the ternary conditional operator can be seen in the following example:

```
condition ? true block : false block
```

Like with the “if” and “else if” statements, the condition expression must evaluate as either `true` or `false`. If the expression results in `true`, the immediate to the interrogation sign “?” code block gets executed. If the result is `false`, the immediate code block to the colon symbol “:” gets executed.

The ternary conditional operator must have and will only have a single condition, one `true` block, and one `false` block, although within each block, nesting will be allowed.

Ternary conditional statements can also be used to set a variable’s value, including initialization.

```
int number = 2;
string result = (number == 1)
    ? "If resulted in true!": "If resulted in false :(";
Console.WriteLine(result);
```

We also have the `switch` statement as an alternative for conditional cases where more than one option is possible.

3.4.3 The Switch Statement

The C# “switch” statement executes the code of one of the specified condition result cases. This statement serves as an alternative specifically to situations that require the use of “else-if” statements.

In a “switch,” we can find a condition that pairs with one or more cases and a final default block, the latter serving a similar role as the “else” statement. The default option is executed if the switch condition does not match any case values.

```
switch (condition) //Pseudocode
{
    case 1:
        //Code
        break;
    case 2:
        //Code
        break;
    default:
        //Code
        break;
}
```

While similar to “if” statements, “switch” has some limitations and characteristics that have to be considered when using it. Readability and performance are usually the features we are after when choosing “switch” as our conditional statement, but its limitations should not be ignored. The following short list contains a few important notes to consider when working with “switch”:

- In C#, duplicate case values are not allowed, so each case must be unique.
- The variable’s data type in the “switch” and the case value must be the same.
- The value of a case must be a constant or a literal. Variables are not allowed.
- The default statement is optional and can be used anywhere inside the switch statement.
- Multiple default statements are not allowed.

These notes are reflected in the syntax of a “switch.” Starting with the “switch” keyword and an expression inside of parentheses, contained within curly braces, one or several cases with possible matches to the resulting evaluation of that condition followed by a colon symbol “:” and the code to be executed can be found.

Finally, a default is optionally used to catch the execution if no case matches.

```
int number = 2;
switch (numbser)
{
    case 1:
        Console.WriteLine("Number is 1");
        break;
    case 2:
        Console.WriteLine("Number is 2"); //This is the true one
        break;
    default:
        Console.WriteLine("Number is either 1 nor 2");
        break;
}
```

The switch expression is evaluated once. The expression’s resulting value is compared with each case’s values. If there is a match, the associated block of code is executed. In this case, case 2 is the correct one.

Also, we need to consider that further case evaluation after a successful case match is not allowed. To prevent this, we can use the `break` statement at the end of the code block to be executed. This statement will prevent continued sequential execution of the switch statement.

If a `return` is used, if a function contains this switch statement, it will cease to execute further. Let us exemplify `break` and `return` as well as the other “switch” scenarios.

Listing 3.8 An Example Code That Uses All of the Explained Switch Statement Scenarios

```
int number = 2;
switch (number)
{
    case 1:
        Console.WriteLine("Number is 1, which is unacceptable");
        return; //This would stop the entire execution
    case 2:
        Console.WriteLine("Number is 2"); //This is the true one
        break;
    default:
        Console.WriteLine("Number is less than 1");
        break;
}
```

To provide further detail, in the given C# switch statement, the difference between break and return is as follows:

- **Break:** When the break statement is encountered, it terminates the innermost enclosing switch or loop statement, and the program continues executing the statements following the terminated statement. In this case, when number is 2, the program prints “Number is 2” and exits the switch statement, resuming the execution of any subsequent code after the switch block.
- **Return:** The return statement is used to end the execution of the current function or method, and the control is returned to the calling function. In the given switch statement, if number were 1, the program would print “Number is 1, which is unacceptable” and then immediately stop executing the entire function containing the switch statement, returning control to the calling function.

This concludes the theory for this chapter! As we intend to keep these initial chapters short with only the necessary content to prepare us for the following big projects, we will now delve right into this chapter’s project, the Rock-Paper-Scissors minigame.

3.5 Rock-Paper-Scissors

We will start this third project like the previous ones by creating a new .NET Console App project. The intention is to always start with a clean and empty template. So Fig. 3.3 should be how our project starts.



A screenshot of a Microsoft Visual Studio code editor window titled "Program.cs". The code contains three lines of C# code:

```
1 // See https://aka.ms/new-console-template for more information
2 Console.WriteLine("Hello, World!");
3
```

Fig. 3.3 We will be starting our third project with an empty project as shown

3.5.1 The Project

Let us analyze what we want to achieve here to determine the next step.

Conceptually, this project is nothing more than a simple Rock-Paper-Scissors game. If we've ever played a round of this with someone, which we probably have, we know that the rules are simple.

Rock-Paper-Scissors, also known as RPS, Ro-Shambo, or Rochambeau, is a game where each side makes a gesture by using one hand to represent either Rock, Paper, or Scissors. The game's objective is to choose a gesture that will beat the opponent's gesture.

This winning condition relies on three simple rules:

- Rock beats Scissors
- Scissors beat Paper
- Paper beats Rock

If both parties choose the same gesture, the game is a draw.

The game ends when any of the aforementioned winning conditions are met.

Now, why are we explaining a game that probably everyone knows? Because even if it does not seem like it, breaking down the project, no matter how simple it is, before starting the development is arguably the most critical step in any software development.

This simple breakdown allows us to determine a small list of components that will be needed, giving us a path to develop for this project to become a reality. This not only helps us organize our idea, but also avoids any possible confusion during the development and sets us straight in line to develop this specific product instead of deviating by adding components unnecessary for our initial goal and essentially succumbing to feature creep.

This might seem obvious for any advanced developer out there, but for anyone still new, or often even intermediate, in the programming world, we cannot stress this enough.

Break your project idea down and set a clear path to follow before you start writing your first line of code. And yes, I am talking from experience.

That said! Now that we know what we need let us get to writing code!

3.5.2 Our Code

To start this project, we can begin by deleting the contents of our template, as we will not need any of that for our game. So we will be left with a clean and blank slate to work with.

Then, we will have to implement the initial introduction to the project, this being a simple greeting and explanation of the rules, just as with our other projects.

For that, we will need more `WriteLine()` methods.

```
Console.WriteLine("Welcome to Rock-Paper-Scissors!");
Console.WriteLine("The rules are simple: Rocks beats Scissor, Scissor
beats Paper, and Paper beats Rock.");
Console.WriteLine("Choose wisely...");
```

NOTE As a quick tip, to instantly type out the “`Console.WriteLine()`” sentence, we can write “cw” and then press the TAB key twice. This will automatically write out the complete line.

Next, it is time to prompt the user to choose one, so we can retrieve his input to start the main game. Like this.

```
Console.WriteLine("Choose your move: (1) Rock, (2) Paper, (3) Scissors");
```

And now, just retrieve his input, the same way we did in the first project, just that this time we need to convert the input to a number, as stated in the prompt.

```
int playerMove = Convert.ToInt32(Console.ReadLine());
```

By the way, if conversions are a topic you are still unfamiliar with, do not worry; we will cover data type conversion in the next chapter.

Let us try that out by doing a simple test to ensure we get the correct value. For example, let us throw our `playerMove` variable into a `Console.WriteLine()` to be able to see it.

Right after our previous line, let us add the following.

```
Console.WriteLine(playerMove);
```

Now, if we run the project and press, for example, “1” when prompted, we should get the following result (Fig. 3.4).

```
Microsoft Visual Studio Debug Console
Welcome to Rock-Paper-Scissors!
The rules are simple: Rocks beats Scissor, Scissor
beats Paper, and Paper beats Rock.
Choose wisely...
Choose your move: (1) Rock, (2) Paper, (3)
int playerMove = Convert.ToInt32(Console.ReadLine()); 1
Console.WriteLine(playerMove); 1 ————— STORED PLAYER INPUT
```

Fig. 3.4 The input we got from the player after he pressed the 1 key is correctly stored as a 1

Great, now that it is working, we can remove this last debugging line, leaving us with this base.

Listing 3.9 The Base for Our Rock-Paper-Scissors Game Before the Gameplay Mechanics Are Implemented

```
Console.WriteLine("Welcome to Rock-Paper-Scissors!");  
Console.WriteLine("The rules are simple: Rocks beats Scissor,  
    Scissor beats Paper, and Paper beats Rock.");  
Console.WriteLine("Choose wisely...");  
  
Console.WriteLine("Choose your move: (1) Rock, (2) Paper, (3) Scissors");  
int playerMove = Convert.ToInt32(Console.ReadLine());
```

Now we can create a competitor for our player. In this case, let us make a simple AI, or artificial intelligence. Mind you, artificial intelligence, in this case, solely refers to a random move generated by our code.

This means we will use our Random Class to generate a random choice for the competitor. As we have already seen previously, doing that is pretty simple. We will only need to instantiate a new Random Class object and then generate a number in a specific range. So first, let us create a new Random object.

```
Random random = new Random();
```

Now, we can use the `random` variable to create a random number that will serve as the choice for our AI competitor.

```
Random random = new Random();  
int computerMove = random.Next(1, 3 + 1);
```

As we can see, we must always range it from the lowest to the highest value, adding one because of the way the Random Class operates. So always keep that in mind to avoid unexpected behavior.

Great, now we have both the player's choice and the AI's choice. We need to take these "number" choices and turn them into a usable format that states the specific choice. In other words, turn "1" into "rock."

We will do this for both the player and the AI. That way, except for the random generation, everything stays the same between the two. So if we want to modify the project to use a second player instead, we should be able to do so easily.

Starting with the player, we will need a new variable that we can use to store his decision. We'll call it "playerMoveString."

```
string playerMoveString = "";
```

Next, we will be using a switch. If we remember from earlier in this chapter, using a switch necessitates a variable, a set of cases, and a result, so essentially, what we want to do with that information.

Knowing that we can use the `playerMove` variable as the variable for our switch, we want to know what the user typed in, like so:

```
switch (playerMove)
{ }
```

This we can now use to set a few cases. In this example, we will just cover all three selection options.

So these cases would be the following:

```
switch (playerMove)
{
    case 1:

    case 2:

    case 3:
}
```

If we leave that as is, we will notice a new error appearing, to be specific, “CS8070: Control cannot fall out of switch from final case label.” This is simply due to missing breaks, so always make sure to give it a break!

```
switch (playerMove)
{
    case 1:
        break;
    case 2:
        break;
    case 3:
        break;
}
```

Perfect. Now, what does this switch do? As we know from earlier, the switch uses the given variable, so `playerMove`, and checks its value. Then, in the cases, if the value of `playerMove` matches one of the cases, it will enter that as if it was an “if” statement. We could compare the previous code to this “if” equivalent.

```
if (playerMove == 1)
{
    //Do something
}
else if (playerMove == 2)
{
    //Do something
}
else if (playerMove == 3)
{
    //Do something
}
```

As we can see, both would do the same, but the switch statement is cleaner to read and is certainly more manageable than the row of ifs.

That is great, but we only have a switch statement that finds the correct case and does not yet do anything with it. We do have to change this, so let us do so.

This switch case intends to set the correct move to each contestant's hand depending on the number they have typed, so as we saw from our prompt at the beginning, that is "1" for "Rock," "2" for "Paper," and "3" for "Scissors." So in each case, we can populate the still empty playerMoveString variable with the corresponding move, like so:

```
switch (playerMove)
{
    case 1:
        playerMoveString = "Rock";
        break;
    case 2:
        playerMoveString = "Paper";
        break;
    case 3:
        playerMoveString = "Scissors";
        break;
}
```

Great, we have one move set done. Now we just need to repeat the same with the competitor, which would be our "AI." Like this:

```
string computerMoveString = "";
switch (computerMove)
{
    case 1:
        computerMoveString = "Rock";
        break;
```

```
        case 2:  
            computerMoveString = "Paper";  
            break;  
        case 3:  
            computerMoveString = "Scissors";  
            break;  
    }  
}
```

There we go. However, you may have noticed that there is the possibility that the player types in something that is not 1, 2, or 3. Furthermore, we do not have an option equivalent to an “else” statement. We do not have a case for when none of the previous cases match. Given how our minigame will be programmed in the next section, there technically does not need to be a fourth case. Nonetheless, we can still create one.

We have several options here. We can either let the user know and make him re-enter the correct option, give him a randomly selected choice, or implement a secret Easter egg that gives the player a special move whenever he types anything unexpected. Something fun and family-friendly. Like a Bucyrus 95-Ton Steam Rail Shovel.

```
switch (playerMove)  
{  
    case 1:  
        playerMoveString = "Rock";  
        break;  
    case 2:  
        playerMoveString = "Paper";  
        break;  
    case 3:  
        playerMoveString = "Scissors";  
        break;  
    default:  
        playerMoveString = "Bucyrus 95-Ton Steam Rail Shovel";  
        break;  
}
```

Jokes aside, this will always depend on your specific application and needs. We will probably keep it without a default case since we will have it programmed so the player would lose if he entered anything except for 1, 2, or 3. So, now that we have all our moves ready, we can focus on the main game logic, the comparison operation.

First, to set the stage, let us inform the player about the moves that have been chosen. We can achieve this by using the `WriteLine()` `Console` class method. A line like this would suffice:

```
Console.WriteLine($"You chose {playerMoveString}."  
    $"The computer chose {computerMoveString}.");
```

This has some advanced string concatenation, but do not worry. In short, the “\$” sign only prepares the string so we can easily add variables in between by placing them within “{}” curly braces.

Now if we ran the project and chose, for example, “Rock” and the AI chose “Paper,” we would get the following result (Fig. 3.5).

```
Console.WriteLine($"You chose {playerMoveString}. The computer chose {computerMoveString}.");
Microsoft Visual Studio Debug Console
Welcome to Rock-Paper-Scissors!
The rules are simple: Rocks beats Scissor, Scissor beats Paper, and Paper beats Rock.
Choose wisely...
Choose your move: (1) Rock, (2) Paper, (3) Scissors
1
You chose Rock. The computer chose Paper.
```

Fig. 3.5 The result if the player chooses “1” for Rock and the AI chooses “2” for Paper

Looks good so far. After this step, we can use operators to check for a winner. So how are we doing that? This time, with some if statements. Starting with the draw.

A draw would be when both the player and the AI choose the same move, so we can make that work by using the “==” operator. Like this:

```
if (playerMove == computerMove)
{
    Console.WriteLine("It's a draw!");
}
```

Then, followed by an “else if,” we can get the player’s winning conditions. As we already know, there are three situations where a player can win: Rock beats Scissors, Scissors beat Paper, and Paper beats Rock.

We could use all three of them in a single “else if” by using the “||” operator. Essentially, an “or” would make the condition true as long as one of the three winning conditions is met.

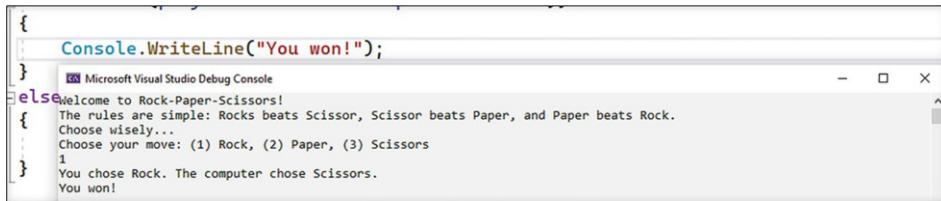
```
else if ((playerMove == 1 && computerMove == 3) ||
          (playerMove == 2 && computerMove == 1) ||
          (playerMove == 3 && computerMove == 2))
{
    Console.WriteLine("You won!");
}
```

Here, we are essentially checking `if (player == rock and AI == scissor)`, but for each condition. If any of these are met, we know that the player won.

And with that, we got both a draw and a win. We are now just missing a loss condition. Since there are only three outcomes to this game, losing is the only possible outcome left, so we could use a final “else” statement here, and we would be good to go, like this:

```
else
{
    Console.WriteLine("You lost!");
}
```

With that final code, we have our minigame done. Now the only thing left is to try out our final product to see if everything works correctly. So let us run the project and see how it looks. So if we, as the player, choose “1” for “Rock,” and the AI chooses “3” for “Scissors,” we would get the following result (Fig. 3.6).



The screenshot shows the Microsoft Visual Studio Debug Console window. The console output is as follows:

```
{  
    Console.WriteLine("You won!");  
}  
Microsoft Visual Studio Debug Console  
else  
Welcome to Rock-Paper-Scissors!  
{  
    The rules are simple: Rocks beats Scissor, Scissor beats Paper, and Paper beats Rock.  
    Choose wisely...  
    Choose your move: (1) Rock, (2) Paper, (3) Scissors  
    1  
    You chose Rock. The computer chose Scissors.  
    You won!
```

Fig. 3.6 The final result of our minigame if the player chooses “1” for Rock and the AI chooses “3” for Scissors

With that, our project ran successfully and is now another great addition to the list of smoothly running projects we can call ours.

During this chapter, we learned about operators and conditional statements. Now, it is time to continue to the next chapter, where we will tackle the development of a number-guessing minigame. Moreover, we will master our knowledge about classes, methods, and other essential components that C# offers. So let's get ready for the next chapter.

3.6 Source

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_RockPaperScissor

3.7 Summary

This chapter taught us:

- Operators are divided into six basic types: Arithmetic operators, Comparison Operators, Equality operators, Boolean logical operators, Bitwise operators, and shift operators.
- An operator is a sign or symbol telling the compiler to perform specific mathematical or logical manipulations, serving us the same way they do outside the programming scope.

- All operators, except for assignment operators, null-coalescing operators, and the conditional operator ?:, are left-to-right associative.
- Operands are evaluated before operators.
- Operand evaluation is obligatory unless the operator is of a conditional nature, in cases such as conditional logical AND (&&) and OR (||) operators, and the conditional operator ?:, where the operand evaluation becomes conditional.
- In C#, we can find the following selection statements: the “if” statement, which selects a statement to execute based on the value of a Boolean expression, and the “switch” statement, which selects a statement list to execute based on a pattern match with an expression.
- If statements are used through the “if” keyword for a single conditional statement, followed by the optional “else if” statements. If additional conditions are needed, finally, followed by the also optional “else” statement, without a condition, as a last resort if no condition returns `true`.
- We can also find a short-hand if-else statement, the conditional operator ?:, also known as the ternary conditional operator.
- The C# “switch” case statement executes the code of one of the specified condition result cases.
- The “switch” statement serves as an alternative specifically to situations that require the use of “else-if” statements.

References

- ankita_saini. (2020, April 22). Switch Statement in C#. Retrieved April 14, 2022 from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/switch-statement-in-c-sharp/](https://www.geeksforgeeks.org/switch-statement-in-c-sharp/)
- Chand, M. (2022, March 2). C# Switch. Retrieved April 14, 2022 from [www.c-sharpcorner.com: https://www.c-sharpcorner.com/article/c-sharp-switch-statement/](https://www.c-sharpcorner.com/article/c-sharp-switch-statement/)
- Computer Hope. (2022, February 7). Syntax. Retrieved April 28, 2022 from [www.computerhope.com: https://www.computerhope.com/jargon/s/syntax.htm](https://www.computerhope.com/jargon/s/syntax.htm)
- GeeksforGeeks. (2021, July 09). C# | Operators. Retrieved April 14, 2022 from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/c-sharp-operators/](https://www.geeksforgeeks.org/c-sharp-operators/)
- Microsoft Corporation. (2021, November 20). ref (C# Reference). Retrieved April 28, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/ref)
- Microsoft Corporation. (2022, April 9). 12 Statements. Retrieved April 14, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/statements](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/statements)
- Microsoft Corporation. (2022, January 25). C# operators and expressions (C# reference). Retrieved April 14, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/)
- Microsoft Corporation. (2022, January 28). Selection statements (C# reference). Retrieved April 14, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/selection-statements)

- Microsoft Corporation. (n.d., n.d. n.d.). Exception Class. Retrieved April 28, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.exception?view=net-6.0)
- SOFTWARETESTINGHELP. (2022, April 3). Tutorial On C# Conditional Statements. Retrieved April 14, 2022 from [www.softwaretestinghelp.com: https://www.softwaretestinghelp.com/c-sharp/csharp-conditional-and-decision-statements/](https://www.softwaretestinghelp.com/c-sharp/csharp-conditional-and-decision-statements/)
- StudyPad, Inc. (n.d., n.d. n.d.). Arithmetic - Definition with Examples. Retrieved April 28, 2022 from [www.splashlearn.com: https://www.splashlearn.com/math-vocabulary/addition/arithmetic](https://www.splashlearn.com/math-vocabulary/addition/arithmetic)
- Tutorials Point. (n.d., n.d. n.d.). C# - Operators. Retrieved April 14, 2022 from [www.tutorialspoint.com: https://www.tutorialspoint.com/csharp/csharp_operators.htm](https://www.tutorialspoint.com/csharp/csharp_operators.htm)
- tutorialsteacher. (2020, June 24). C# - if, else if, else Statements. Retrieved April 14, 2022 from [www.tutorialsteacher.com: https://www.tutorialsteacher.com/csharp/csharp-if-else](https://www.tutorialsteacher.com/csharp/csharp-if-else)
- TutorialsTeacher. (n.d., n.d. n.d.). C# Operators. Retrieved April 14, 2022 from [www.tutorialsteacher.com: https://www.tutorialsteacher.com/csharp/csharp-operators](https://www.tutorialsteacher.com/csharp/csharp-operators)
- W3Schools. (n.d., n.d. n.d.). C# If ... Else. Retrieved April 14, 2022 from [www.w3schools.com: https://www.w3schools.com/cs/cs_conditions.php](https://www.w3schools.com/cs/cs_conditions.php)
- W3Schools. (n.d., n.d. n.d.). C# Operators. Retrieved April 14, 2022, from [www.w3schools.com: https://www.w3schools.com/cs/cs_operators.php](https://www.w3schools.com/cs/cs_operators.php)
- W3Schools. (n.d., n.d. n.d.). C# Switch. Retrieved April 14, 2022 from [www.w3schools.com: https://www.w3schools.com/cs/cs_switch.php](https://www.w3schools.com/cs/cs_switch.php)



C# Classes, Methods, and User Input

4

This Chapter Covers

- Properly using classes in C#
- Working with methods to give our code functionality
- Using the console class, its methods, and properties
- Converting data with the Convert class and its methods
- Developing a number guesser minigame

Nearing the end of Part 1, Introduction to the C# Programming Language, we slowly become more familiar with the C# development process. We find new types and shapes of coding language with each chapter that we can use to build our programs. So far, we have learned about the fundamentals of this language. We have learned about variables, data types, randomness, operators, and conditionals and got pretty good at working in our development environment. Furthermore, we were able to build a complete, scalable project in each chapter, which will not change with this chapter.

This chapter's project, the “Guess the number” minigame, will consist of a random number that needs to be guessed by the user. Each time a user input is given, the code will compare the given number to the random number generated and tell the user if the guessed number is too high, too low, or matched. We can then keep track of the number of tries to compete with others.

This chapter's minigame project will teach us everything about user interactivity as well as introduce us in detail to classes, methods, the console class, and the Convert class. Furthermore, like always, it will provide us with a project ready to be vastly expanded upon.

So then, as this chapter will mainly focus on getting to know classes and methods on a deeper level, we should be able to start off by learning about Classes in the object-oriented language of C#.

4.1 Classes in Object-Oriented C#

We understand that in C#, everything is associated with classes, along with their properties and methods. We can use the typical car object example as a metaphorical similarity. When a car is considered a class, including its attributes such as weight and color and methods such as drive and brake, the car class can have individual instances considered objects. Such an object can be, for example, a luxury sedan. Let us put this practical example into code.

```
class Car
{
    string color = "red"; //class attribute
    static void Main(string[] args)
    {
        Car luxurySedan = new Car();
        //New "luxury sedan" object from the class "car"
        luxurySedan.color = "white"; //This object is white
        Console.WriteLine(luxurySedan.color);
        //Resulting in the color white
        Car sportsCar = new Car();
        //New "sports car" object from the class "car"
        Console.WriteLine(sportsCar.color);
        //No color change here, maintaining the color red
    }
    void drive() //class method
    {
        Console.WriteLine("Driving");
    }
}
```

As we can see, the overall place of a class within our programming flow is easy to comprehend.

4.1.1 Definition of Class

Classes and objects are the basic building blocks of object-oriented programming. A class can be seen as a user-defined blueprint from which objects are created. We can essentially understand classes as prepared blocks of features for use in multiple objects simultaneously, without rewriting on each object creation process.

We have probably seen classes in base C# applications where the class Program is created in our program.cs file, as we will be seeing later in this chapter done by us. Also, both

the Console and Convert classes are commonly used in C# development. Both contain their internal code accessible to use without the need for rewriting them.

Additionally, in C#, classes support polymorphism and inheritance. As this is an essential component in the C# classes development process, we will go into detail right after the class definition and object creation basics.

Syntax-wise, a class contains only a keyword, followed by the identifier of the class, this being the name we want to assign it. In its purest form, this would be the look:

```
class Car
{
    //Containing code
}
```

There is more to a class declaration to make the execution successful, but we will go over further details later in this chapter.

However, we can quickly review some class attributes that will be useful later. As seen in the following example, these include:

Listing 4.1 In This Example, Public Is the Modifier, Class Is the Keyword, MyClass Is the Class Identifier, BaseClass Is the Base Class, IMyInterface Is the Interface Implemented by the Class, and the Class Body Is Enclosed in the Curly Braces {}.
The Constructor Is Defined Inside the Class Body

```
public interface IMyInterface
{
    // Interface code here
}

public class BaseClass
{
    // BaseClass code here
}

public class MyClass : BaseClass, IMyInterface
{
    public int MyProperty { get; private set; }
    public MyClass(int initialValue)
    {
        // Constructor code: Initializing the
        // property with the given value
        MyProperty = initialValue;
    }

    // Class body here
}
```

Listing 4.1 shows some of the attributes a class can have. Let us take a look at what these are.

- **Modifiers.** A class can be public, internal, protected, protected internal, private protected, and private, among others. Each offers varying degrees of accessibility. Listing example: `public`.
- **Keyword class.** The `class` keyword is used to declare the type class. Listing example: `class`.
- **Class Identifier.** The identifier, or name of the class. This name should begin with an initial capitalized letter by convention. Listing example: `MyClass`.
- **Baseclass or Superclass.** An optional name of the class's parent. If existent, it must be preceded by the “`:`” colon. Listing example: `BaseClass`.
- **Interfaces.** An optional comma-separated list of interfaces implemented by the class. If existent, preceded by the “`:`” colon. Listing example: `IMyInterface`.
- **Body.** The body is surrounded by “`{}`” curly braces. Within the body of a class, we can find Fields, properties, methods, and events collectively referred to as class members. Listing example: `// Class body here`.
- **Constructor.** A class can have parameterized or parameterless constructors. The constructor will be called when we create an instance of a class. Constructors can be defined by using an access modifier and class name. Listing example: `// Constructor code:`

4.1.2 Object Creation

In C#, an object is a self-contained component containing properties and methods needed to make specific data useful. An object can be a piece of data, a function, or a combination of both. In C#, objects are created from classes, essentially templates that define the properties and methods of the object. Let us refer to an example of the creation process for an object.

```
Car sportsCar = new Car();
```

From the class `Car`, through the `new` keyword, we get the derived object referenced by the local variable `sportsCar`, written in camel case. These objects are often referred to as instances of classes. A reference to the object is passed when such an instance is created. This reference refers to the newly created object and can be used further to manipulate that specific instance of our class.

Object references can also be left without an assignment, meaning no instance is created. However, similarly to uninitialized variables, this can produce runtime errors if such a reference is read before any assignment takes place. Additionally, assigning object references with previously created object references is possible. Consider the following example:

```
class Car
{
    static void Main(string[] args)
    {
        Car sportsCar = new Car();
        Car theSameSportsCar = sportsCar;
    }
}
```

The reference `theSameSportsCar` was assigned `sportscar` correctly. While in this example:

```
class Car
{
    static void Main(string[] args)
    {
        Car sportsCar;
        Car theSameSportsCar = sportsCar;
        sportsCar = new Car();
    }
}
```

It would throw a `NullReferenceException`, an exception that is thrown when we try to access a member on a type whose value is null since `sportscar` was not yet assigned any value.

4.1.3 Class Inheritance

In C#, classes fully support class inheritance, an essential and commonly useful characteristic of object-oriented programming. Upon creation, classes can be inherited from any other deriving class.

Furthermore, one or more classes can be inherited simultaneously.

Classes can be inherited by declaring the base class preceding a “`:`” colon following the class name.

```
class Vehicle
{
    //Body of Vehicle
}
class Car : Vehicle
{
    //body of car inheriting everything from the vehicle superclass
}
```

The above example lets us quickly determine the significant use cases found in such baseclasses/superclasses. The derived class inherits all the inherited class members when such a superclass is declared, except for constructors.

This characteristic serves as a foundation of a tiered class system where, following the rehearsed example, a vehicle superclass serves as the groundwork for specific vehicle classes, like cars, planes, boats, and so forth.

While C# does not support multiple inheritances for classes, allowing only a single base class inheritance, it enables a workaround through hierarchical inheritance. In this approach, a derived class can inherit from a base class, which in turn inherits from another class, effectively allowing a single class to inherit properties and methods from multiple ancestors in the hierarchy.

```
class TransportationDevice
{
    //Body of TransportationDevice
}
class Vehicle : TransportationDevice
{
    //Body of Vehicle inheriting everything from TransportationDevice
}
class Car : Vehicle
{
    //body of car inheriting everything from the vehicle and the
    //TransportationDevice superclass
}
```

Let us try to use a new class to separate the execution of this method into its class. Keeping the same naming convention as our file name, we can call it “program,” populating its body with a console print.

```
public class Program
{
    Console.WriteLine("Hello!");
}
```

Upon intended execution, we are already given several errors, even though the `Console.WriteLine()` method once used to work before. There is something we need to keep in mind when working with classes. We can see that if we remove our `Console.WriteLine()` (Fig. 4.1), we get an error code, specifically, CS5001. The error reads: Program does not contain a static ‘Main’ method suitable for an entry point. Evidently, there is still something missing for the correct working of our class.

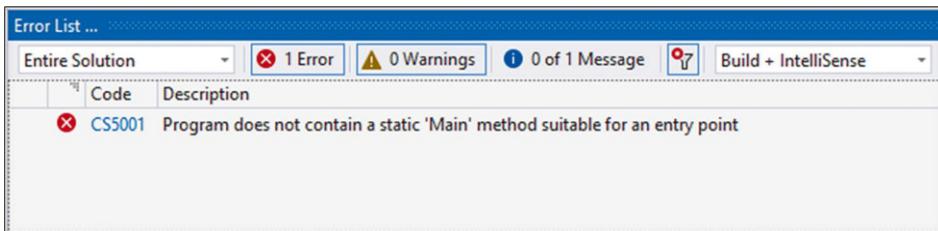


Fig. 4.1 Error code CS5001 is signaling that our program does not contain a static 'Main' method suitable for an entry point

To see how we get this to work, we can try to run this fixed code now.

```
public class Program
{
    static void Main()
    {
        Console.WriteLine("Hello!");
    }
}
```

Then if we do that, we see that the code runs perfectly. This is because the `Main()` method is the program's starting point, and not including it will result in the error we have received.

As a side note, although we will not need inheritance or static for our class within this chapter, we will go over this in addition to further class and object characteristics when future chapter projects necessitate them. Also, inheritance can be of great use in TDD, or Test-Driven Development.

We will not go too in-depth into the topic of TDD in this book; however, we can give a brief intermission to introduce us to the world of TDD. So, what is it?

4.1.4 A Brief Introduction to Test-Driven Development

As mentioned, this book will not entirely cover Test-Driven Development. Nonetheless, learning about TDD, even as a brief introduction, is something we can definitely achieve in this section.

Test-Driven Development (TDD) is a software development process where you write tests before writing the actual code. The tests define what the code should do, and the code is written to pass these tests.

Here's an overview of how TDD works:

- **Write a test:** Before writing any code, you write a test for a small piece of functionality. The test should define what you want the code to do and it should initially fail because you haven't written the code yet. The test is written using a testing framework, such as MSTest in C#.
- **Write the code:** Once you have written the test, you write the code to pass the test. You should write only enough code to pass the test and no more.
- **Run the tests:** After writing the code, you run all of your tests to ensure that the code you just wrote passes all of them and that your changes haven't broken any existing code.
- **Repeat:** Repeat this process (writing a test, writing code to pass the test, and running tests) for each new feature or bug fix.

TDD can aid us greatly in ensuring that our code is correct, robust, and meets the requirements defined in our tests. Additionally, TDD can help us catch bugs early in the development process before they become more complex and time-consuming to fix.

Although we could go more in-depth, like including one or more example projects, this would far exceed our scope. So, the best thing we can do right now is recommend one of Springer's TDD books if you are interested in diving deeper into this development style.

Now, in this chapter, we will continue learning about something we have seen being used before. To be exact, we will learn about how methods work in C#.

4.2 Methods in C#

Methods are a named block of code that holds a series of statements. The program causes the statements to be executed by calling the method and specifying any required arguments, if there are any. In C#, all executed instructions are performed within such a method.

NOTE Often, we hear methods being called functions. Generally, there is a difference between the two. To put it simply, if a function is within a class, it is called a method. However, as C# is an Object-oriented language, it does not allow functions to be created outside of a class, meaning that any function you see in C# is also a method, hence both terms being used interchangeably.

The Main method, which we will see more often now in the following projects, can be defined as the entry point for a C# application. When the program is started, it is called by the Common Language Runtime, or CLR, meaning that without us explicitly calling it, the method will run upon execution. We have heard the word "methods" multiple times during these first chapters. Together with classes and objects, methods are one of the essential building blocks for practically every C# program.

4.2.1 Defining a Method

Methods are defined using the following syntax:

Listing 4.2 PSEUDOCODE to Demonstrate the Anatomy of the Syntax for a C# Method

```
[access modifier] [return type] [method name] ([parameters])
{
    // method body
}
```

Here, the access modifier specifies the method's visibility, such as whether it can be accessed from other parts of the program. The return type specifies the data type of the value returned by the method. The method name is the name of the method, and the parameters are the input values that are passed to the method. The method body is the code that defines what the method does.

With that, we can find the following components to a method.

- **Access modifier.** Limits the visibility of a method from another class. For example, the `public` access modifier is used if it has to be accessed globally.
- **Return type.** A method can return a value, but does not have to. The return data type is the data type of the value the method returns. If the method does not return any values, the type is `void`.
- **Method name.** The method name is the identifier that is used to identify the method. It is a name that is used to identify the method within the class in which it is created. In C#, it is good practice to begin the method identifier with an initial capitalized letter by convention, also named PascalCasing. The method name is followed by a set of parentheses, which may contain parameters that are given to the method when it is called.
- **Parameter list.** The parameters are used to pass in and receive data from a method. This list refers to the type, order, and number of parameters of a method.
- **Method body.** This contains the instructions needed to complete the required activity.

Some examples of methods in C# are:

```
public string GetFullName(string firstName, string lastName)
{
    return firstName + " " + lastName;
}

public int AddNumbers(int a, int b)
{
    return a + b;
}
```

The GetFullName () method has the public access modifier, a string return type, and takes two string parameters: firstName and lastName. In the method body, the first and last names are concatenated with a space in between and returned as the result of the method.

The AddNumbers() method has the public access modifier, an int return type, and takes two int parameters: a and b. In the method body, the values of a and b are added together and returned as the result of the method.

We will now go in-depth with all the components of a method.

4.2.2 Static and Non-static Methods

C# supports two types of class methods, static and non-static methods.

In C#, a static method is a method linked with a class instead of an instance of that class. We can call a static method without having to create an object of the class first. To declare a static method, the static keyword is used before the method's return type, like so:

```
public static void PrintMessage(string message)
{
    Console.WriteLine(message);
}
```

You can then call this method using the name of the class and the method like this:

```
MyClass.PrintMessage("Hello, World!");
```

On the other hand, a non-static method, known as an instance method, belongs to an instance of a class. To call a non-static method, we must first create an object of the class. To declare a non-static method, the static keyword is not used like so:

```
public void PrintMessage(string message)
{
    Console.WriteLine(message);
}
```

This method can be called using an object of the class like this:

```
MyClass obj = new MyClass();
obj.PrintMessage("Hello, World!");
```

There are several differences between static and non-static methods. Static methods could be called without an object, while non-static methods must be called on an object. Static methods do not have access to the instance data of a class, while non-static methods have access to this data. Static methods can be used to provide utility functions or other functions that apply to a class as a whole, while non-static methods operate on the instance data of a class and are specific to a particular instance. Finally, non-static methods can be

overridden in derived classes using the `override` keyword, but static methods cannot be overridden and are not inherited by derived classes.

Let us now continue by delving a bit deeper into the process of calling a method and create a small list of ways we could do so. This next section will hopefully help us in future instances when we need to come back to this chapter to remember a specific piece of information.

4.2.3 Calling a Method

Calling a method entails several situations for accessibility, scope, and intended use. Concurrently, this process can become trivial once the necessary steps become clear.

For Static Methods

Calling a method only necessitates the method's name and parentheses. Arguments are listed within the parentheses and separated by commas.

```
public class Program
{
    static void MyMethod()
    {
        Console.WriteLine("I just got executed!");
    }
    static void Main(string[] args)
    {
        MyMethod();
    }
}
```

For Non-static Methods

The example above is only possible if the called method is set to static. If it is non-static, an object reference has to be instanced.

```
public class Program
{
    void MyMethod()
    {
        Console.WriteLine("I just got executed!");
    }
    static void Main(string[] args)
    {
        Program program = new Program();
        program.MyMethod();
    }
}
```

For a Non-static Outside of a Class Call

If not set to static, we can still call a method from outside its class if it is set as public, like the following example where static and non-static could be called.

```
public class Program
{
    // Static method
    static void MyMethod()
    {
        Console.WriteLine("I just got executed (static)!");
    }

    // Non-static method
    public void MySecondMethod()
    {
        Console.WriteLine("I also just got executed (non-static)!");
    }

    static void Main(string[] args)
    {
        // Calling the static method directly
        MyMethod();

        // Creating an instance of the Program class to call
        // the non-static method
        Program programInstance = new Program();
        programInstance.MySecondMethod();
    }
}
```

Calling Before and After Definition

Methods can be called before or after their definition. In the following example, both method calls will work.

```
public class Program
{
    static void MyMethod()
    {
        Console.WriteLine("I just got executed!");
    }

    static void Main(string[] args)
    {
        MyMethod();
        MySecondMethod();
    }
}
```

```
        static void MySecondMethod()
        {
            Console.WriteLine("I also just got executed!");
        }
    }
```

Recursion¹ Calling

A method can call itself. We see a single recursion occurring in the following example until “callAgain” gets set to true.

```
public class Program
{
    static void MyMethod()
    {
        bool callAgain = false;
        Console.WriteLine("I just got executed!");
        if (!callAgain)
        {
            callAgain = true;
            MyMethod();
        }
    }
    static void Main(string[] args)
    {
        MyMethod();
    }
}
```

If a recursive method doesn’t include an escape clause, it can cause the memory to be exhausted and throw an exception, which can be very harmful in a real-world scenario. It’s important to note that this can happen if the method is not properly implemented.

Calling with Parameters

In Chap. 2, we learned about variable parameters and saw their usage in methods. We just lightly went over the concept of calling a method with parameters. Information can be passed to methods as a parameter. Parameters act as variables for within the method.

¹A recursive method is a method which calls itself again and again on the basis of a few statements which need to be true (Dhar, 2020).

```
public class Program
{
    static void MyMethod(string param)
    {
        Console.WriteLine(param);
    }
    static void Main(string[] args)
    {
        MyMethod("String");
    }
}
```

We can add as many parameters as we want, just separating them with a comma.

```
public class Program
{
    static void MyMethod(string param1, string param2)
    {
        Console.WriteLine(param1, param2);
    }
    static void Main(string[] args)
    {
        MyMethod("string1", "string2");
    }
}
```

Default Parameters

You can also use the “=” equals sign to use a default parameter value and only in a right-to-left order. If we call the method with no argument, it uses the default value.

```
public class Program
{
    static void MyMethod(string param = "String")
    {
        Console.WriteLine(param);
    }
    static void Main(string[] args)
    {
        MyMethod();
    }
}
```

It is possible to specify default values for multiple parameters in a method. In the following example, the PrintMessages method has three parameters: message, count, and newLine. The count and newLine parameters both have default values, meaning that the method caller can omit these arguments if they want to use the default values.

```
public class Program
{
    public static void PrintMessages(string message, int count = 1,
bool newLine = true)
    {
        for (int i = 0; i < count; i++)
        {
            Console.Write(message);
            if (newLine)
            {
                Console.WriteLine();
            }
        }
    }
    static void Main(string[] args)
    {
        PrintMessages("string");
    }
}
```

Additionally, if we remember from, for example, Visual Basic, in C#, we cannot leave a blank space between commas to use the default value for a parameter. Instead, we use named arguments to specify values for specific parameters and let others use their default values. Here's an example:

```
public class Program
{
    static void Main(string[] args)
    {
        PrintMessages("Hello, World!", newLine: true);
    }

    static void PrintMessages(string message, int count = 1, bool
newLine = false)
    {
        for (int i = 0; i < count; i++)
        {
            Console.Write(message);

            if (newLine)
            {
                Console.WriteLine();
            }
        }
    }
}
```

When calling the PrintMessages method, we specify values for message and newLine while omitting the count parameter. The count parameter will use its default value of 1. To do this, we use the named argument syntax: newLine: true.

And what about returning a value? We have seen that methods can have a return value. One of them is void, for returning nothing, but what if we want to return something?

4.2.4 Method Return Types

Even though not obligatory, methods can return a value after calling. In the previous definition for methods, we mentioned the return type as part of the anatomy of a C# method. Let us dive a bit deeper into that here.

We often see a datatype before the method identifier on method construction. If not, it is usually void, as in the void keyword, meaning it does not return anything, just like we often use.

Generally, we can sum up method returns with the following:

```
public int AddToItself(int number)
{
    return number + number;
}

static void Main(string[] args)
{
    AddToItself(5);
}
```

The method AddToItself() has a return type of int, meaning that it will return an int when called. The returned value can then be stored for further use.

```
public int AddToItself(int number)
{
    return number + number;
}

static void Main(string[] args)
{
    int addedNumber = AddToItself(5);
}
```

In the previous example, the Main() method calls the AddToItself() method, passing the value “5” as the argument. The AddToItself() method is called with the argument “5,” so the value of number within the method is also “5.” The method returns the result of

“5 + 5,” which is “10.” The value returned by the `AddToItself()` method is then assigned to the variable `addedNumber`. Therefore, after the call to `AddToItself()`, the value of `addedNumber` will be “10.”

Notice that the `return` keyword had to be used for a return method to return a value successfully. A requirement for using a return type is to tell the method the final return value, which is done using the `return` keyword followed by the value. This return can be a variable, a fetch, an operation, really anything as long as it comes out to a final value of the method return type.

The return states `return number + number` in the previous example, which can be interpreted as `return 10`.

If we do not wish to return a value, the `void` keyword has to be used, as we can see used for the `Main()` method. If the `void` keyword is used as the return type, no `return` keyword is needed within the method body.

And why would we even want to use methods in the first place? Although seemingly obvious, let us briefly discuss the advantages of using methods in our code.

Methods can provide significant advantages to the execution of the code besides several others. In an effort to simplify, we can see three main advantages:

- **Code reuse** by defining the code once and using it many times
- **Time and therefore cost-saving** through the reduction of code to be written
- **Better readability** of the code, especially for large codebases

Minimal API

Something worth mentioning if we talk about classes, methods, and general syntax is that there is something called Minimal API. Although we won’t be covering this in this book, knowing this is still important for future reference. So let us attempt to carry out a short introduction to Minimal API before we continue with our two classes that we will be making use of in our chapters.

The Minimal API is a set of APIs in the .NET runtime that are designed to be as small as possible while still providing essential functionality for running .NET applications. This is intended to make it easier to create lightweight applications that can run on devices with limited resources, such as IoT devices or mobile devices.

One way the Minimal API can improve the experience of writing C# code is by providing a more streamlined development environment. Because the API is smaller, it can be easier to learn and use, making it faster and simpler to develop applications. Additionally, the smaller API size may result in faster startup times and reduced memory usage for applications, making it more efficient to run and debug code.

Another way the Minimal API can improve the experience of writing C# code is by providing a more targeted set of APIs specifically tailored to the needs of lightweight applications. By limiting the scope of the API to the essential functionality, it can be easier to find the specific APIs we need and to understand how they work. This can also make

writing code optimized for performance easier, as we can focus on using the most efficient APIs for our specific needs.

Overall, the Minimal API is intended to make it easier and more efficient to develop lightweight applications using C# and the .NET runtime by providing a streamlined development environment and a targeted set of APIs.

Now, although this chapter is mainly about classes and methods, we are due to get a proper explanation of what the Console and Convert classes are. So before we get into the meat of this chapter, let us get to know them a bit better, as we will be using them quite a bit.

4.3 Expanding on the Console Class

We have already started working with the Console class in the previous three chapters through the `WriteLine()` method, which served us to display text in our Console. Nevertheless, `WriteLine()` is just one of the many methods within the Console class.

Although this chapter's main focus referring to user Input will be a specific Console method, that being the `ReadLine` method, we will have to get a deeper understanding of the Console class as a whole to make the best use of its strengths.

Let us start with a simple view of the Console class itself.

4.3.1 The Console Class

The console class represents console applications' standard input, output, and errors, or in other words the input from the user, the result shown on the screen, and the display of errors that may occur. Using an operating system window, or console terminal, the console class allows users to interact with our text-based console applications entering text input through, for example, a keyboard and reading text output from the terminal.

Listing 4.3 The Line of Code That Used an Operator in the Previous Chapter's Project

```
Console.BackgroundColor = ConsoleColor.White;
Console.Clear();
Console.ForegroundColor = ConsoleColor.Black;
Console.WriteLine("Welcome!");
Console.WriteLine("Enter your name: ");
string name = Console.ReadLine();
Console.Write("Nice to meet you, ");
Console.Write(name + "!"');
```

As shown in the previous example, we can use the console class for user interface implementation. The console class, available in the “System” namespace,² provides a host of methods and properties which we can implement in our projects to expand upon the user interactivity of our app. All the methods and properties within the console class are accessible from anywhere simply by using the console class name.

Let us start with console properties, where, for those who noticed, we will finally go over the three console lines we have been using since Chap. 1 to invert our console windows coloring.

4.3.2 Console Properties

The C# console class provides many properties³ for manipulation and customization of our development environment. To limit the scope of this book to a certain degree, we will not go over every single property, nor will we do that for any other of the C# features, but we will learn about some of the more common ones known to be used in the majority of applications.

To start, we can take a look at the three-line combo we learned about before. Specifically, the `BackgroundColor` and `ForegroundColor` properties.

Listing 4.4 The Line of Code That Used an Operator in the Previous Chapter’s Project

```
Console.BackgroundColor = ConsoleColor.White;
Console.Clear(); // <- Console Method
Console.ForegroundColor = ConsoleColor.Black;
```

This three-line combo of console class components turns the background of our console window to white, clears the window, used to cover the entirety with the new color, and changes the text color to black. With that, we can deduce already that:

- **Console.BackgroundColor.** This property specifies the background color of the text. If we use it, we will notice that only the background of the text will be set to white. We naturally do not want that behavior, so that is what the `Clear()` method is for. By clearing the `Console`, we reset the formatting of our console window, applying the background color to the entire `Console`. Nevertheless, beware, since the `Clear()` method, as

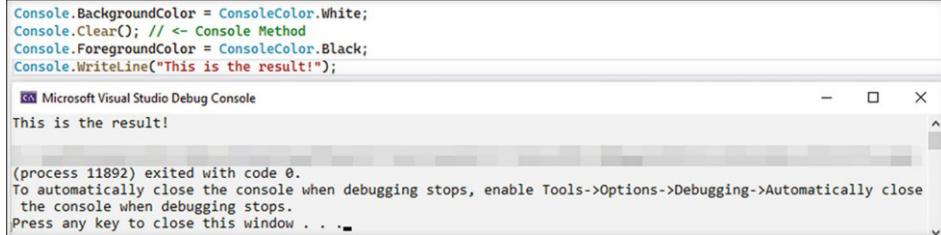
²The namespace keyword is used to declare a scope that contains a set of related objects (Microsoft Corporation, 2022).

³A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field.

it states, will clear the Console of all the previous entries, so make sure to execute the property change at the beginning of the program.

- **Console.ForegroundColor.** This property simply changes the text to the specified color.

Combining these with the previous console method will achieve our desired result of a white background with black text. At least it can be said that that is our desired result for legibility purposes (Fig. 4.2).



The screenshot shows the Microsoft Visual Studio Debug Console window. The code in the editor is:

```
Console.BackgroundColor = ConsoleColor.White;
Console.Clear(); // <- Console Method
Console.ForegroundColor = ConsoleColor.Black;
Console.WriteLine("This is the result!");
```

The console output shows:

```
This is the result!
```

(process 11892) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .

Fig. 4.2 The result after implementing our trio of Console properties and methods. BackgroundColor sets the color of our Console, Clear() resets the formatting, and ForegroundColor sets the color of our text

Another two worthy mentions are title and cursor size, commonly used to customize the look of a fairly simple console application. These can be summed up in the following descriptions.

- **Console.Title.** This property specifies the title of the console application.
- **Console.CursorSize.** This property specifies the height of the cursor in the console window, ranging from 1 to 100.

However, these are not the only ones. Using the following table, we will go through an additional seven examples of commonly used console properties. Covering all of them would exceed this chapter's scope, so we recommend visiting the official documentation page (Microsoft Corporation, n.d.) (direct link <https://docs.microsoft.com/en-us/dotnet/api/system.console?view=net-6.0>).

Continuing with our selection of additional properties, we can find modifiers for error reading, input and output stream detection, and window sizing control. Although we might not need them during this chapter, it is worth noting their use (Table 4.1).

Table 4.1 This table describes a selected list of properties for the console class

Property	Description
Error	Gets the standard error output stream
In	Gets the standard input stream
Out	Gets the standard output stream
WindowHeight	Gets or sets the height of the console window area
WindowLeft	Gets or sets the leftmost position of the console window area relative to the screen buffer
WindowTop	Gets or sets the top position of the console window area relative to the screen buffer
WindowWidth	Gets or sets the width of the console window

Other than properties, the console class also counts with methods, like the `Console` method we just saw, the `Clear` method, or one we have been using a fair bit, the `WriteLine` method. Let us better understand what they are and how we could use them for our code.

4.3.3 Console Methods

Console properties allow the user to adjust and customize the console environment itself, but these do not modify the execution of the process itself or permit the reading or writing of user input or code-generated data to the `Console`. For these uses, we use console methods. Even though we will go more in-depth in this chapter on the definition of a method as a concept, we can already clarify its position as action handlers in the context of the `Console` class.

From a simplified point of view, the `Console` class methods perform certain specified and highly specialized actions. Take as an example our already well-known `Console.WriteLine()` method. We can identify it as a method able to write any given data type such as `string`, `int`, or `bool` as a `string` onto the `Console`.

To better understand, let us look at these commonly used examples for console methods:

Console.WriteLine("string")

Displays the specified message on the console window within the same line (Fig. 4.3).



Fig. 4.3 The `Console.WriteLine("string")` method successfully displaying the word “string” to the `Console`

Console.WriteLine("string")

Similar as the Write() method but automatically moves the cursor to the next line after writing the message (Fig. 4.4).

```
Console.WriteLine("string1");
Console.WriteLine("string2");
Console.WriteLine("string3");
Microsoft Visual Studio Debug Console
string1string2 TEXT MOVED TO NEXT LINE AFTER WRITELINE METHOD
string3
34012) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Fig. 4.4 The Console.WriteLine("string2") method successfully displaying the word “string2” to the Console and moving the cursor to the next line after

As seen in Fig. 4.4, “string2” was printed within the same line since “string” was printed with a Write() method, but “string3” was printed on the following line simply because of the previous WriteLine() method, which after printing moved the cursor to the next line.

We can cover most method use cases with these two methods alone. These, combined with other systems, have the potential to display data in any way required for our software.

Then also, reading user input is a significant use case for console methods.

Console.Read()

Reads a single character from the keyboard then returns its ASCII⁴ value. The data type has to be int since it returns the ASCII value. As in the following example, the input “a” confirms the return of a correct ASCII value of “97” (Fig. 4.5).

```
int input = Console.Read();
Console.WriteLine(input);
Microsoft Visual Studio Debug Console
a USER INPUTS "A"
97 OUTPUT OF READ() IN ASCII
29174) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Fig. 4.5 The Console.Read() method reading the user’s input “a” and outputting it as an ASCII value

Console.ReadLine()

Reads a string value entered with the keyboard written until the ENTER key is pressed, then returns the entered value. As it returns the entered string value, the datatype will be a

⁴ASCII stands for American Standard Code for Information Interchange (Injosoft AB, n.d.).

string. As seen in the example figure, the user input “a” outputs the expected string, “a” (Fig. 4.6).

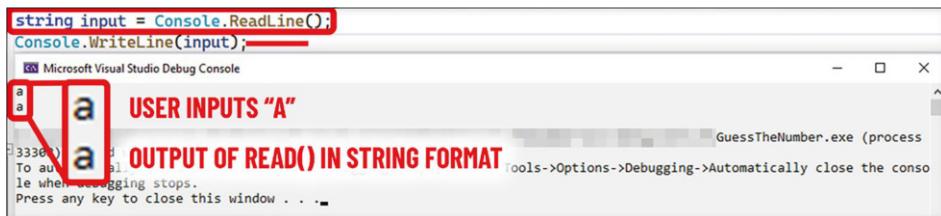


Fig. 4.6 The `Console.ReadLine()` reads the user’s input “a” and outputs it as a string value

Like with properties, `Console` methods are not limited to the aforementioned few methods but, as of C# 10 with Net 6, a list of 58 methods. We can find the complete list on the official Microsoft documentation page (Microsoft Corporation, n.d.) (direct link <https://docs.microsoft.com/en-us/dotnet/api/system.console?view=net-6.0>).

As with properties, we will limit this list to seven console methods (Table 4.2).

Table 4.2 This table describes a selected list of methods for the console class

Methods	Description
Beep	Plays the sound of a beep through the console speaker
Clear	Clears the console buffer and corresponding console window of display information
GetCursorPosition	Gets the position of the cursor
ReadKey	Obtains the next character or function key pressed by the user. The pressed key is displayed in the console window
ResetColor	Sets the foreground and background console colors to default
SetWindowPosition	Sets the position of the console window relative to the screen buffer
SetWindowSize	Sets the height and width of the console window to the specified values

So with this list, we have seen most of the available methods for the `console` class. We are looking for a specific method to “interrupt” the execution flow to collect user input. Here interrupt refers to the fact that some methods will stop further execution until a respective action is given. We will further understand this when we attempt to use it in our projects.

From what we have seen, our best bet would be the `ReadLine` method. The `ReadLine` method will capture a string value from the keyboard. However, since our program needs an integer, and even when typing in the number 1 from the keyboard, we get a string value “1” and not an integer value of 1, a common issue with capturing input from the keyboard, we will need to make use of the `Convert` class in the next section to prevent this type mismatch.

4.4 The Convert Class

For situations where a specific data type is required but not provided by the code that feeds it to us, like in the `ReadLine()` method providing us a string where `int` is wanted, the conversion class proves beneficial.

This class can be used to convert between non-compatible types, or example as a string to a `DateTime`.

The `Convert` class is the way to go if we need to convert between types in C#. Let us continue with the definition of the `Convert` class.

4.4.1 Definition of the Convert Class

The `Convert` class provides methods to convert most data types to different data types. The base types supported by the `Convert` class are `Boolean`, `char`, `sbyte`, `byte`, `int16`, `int32`, `int64`, `uint16`, `uint32`, `uint64`, `single`, `double`, `decimal`, `dateTime`, and `string`.

Adding to the base characteristics of the `Convert` class is the ability to convert custom objects to any base type and support for `base64` encoding.

If a narrowing conversion results in data loss, an `OverflowException` can occur. We will go over conversion outcomes later in this chapter.

In terms of the `Convert` class syntax, the following can be observed.

```
Convert.ToInt32(Console.ReadLine());
```

Knowing about the type mismatch where `ReadLine()` will return a string when we need an `int`, the `Convert.ToInt32` will do just that for us: take the output as a string and convert it to an `int` type. If we were to take the result of this method now, when the user inputs 1, we would get an `int` of value 1.

This result leaves us to learn what other outcomes this class will generate.

4.4.2 Outcomes to Conversions

A conversion method exists to convert every type to another type. However, the call to a particular conversion method could produce one of five outcomes, depending on the type's value at runtime and the target type. These five outcomes are `No conversion`, `InvalidOperationException`, `FormatException`, `OverflowException`, and, naturally, a successful conversion.

As we can see, most of these are simply exceptions, meaning that something went wrong during the conversion process. To be more specific about what exceptions represent in programming, we should know that Exceptions in C# are runtime errors or unexpected events that occur during the execution of a program. They represent exceptional conditions

that can interrupt the normal flow of program execution. Specifically, conversion exceptions occur when an attempt is made to convert one data type to another, but the conversion is not possible. For example, trying to convert a string of text to a number when the text cannot be parsed as a valid number will result in a FormatException, as seen in the following detailed outcomes of the use of conversions:

No Conversion

This outcome results when an attempt is made to convert from and to the same type, for example, when calling Convert.ToInt32(Int32) with an argument of type Int32. The method returns the original type without creating a new “version” in this case, meaning that the same argument we provide gets returned to us.

Listing 4.5 The Convert.ToInt32() Method Attempts to Convert the int “10” Into an Integer. Since They Are Both the Same Data Type, the Same Variable Will Be Returned

```
int number = 10;
int conversion = Convert.ToInt32(number);
// Will simply return back the number 10
```

An InvalidCastException

This outcome occurs when we try to convert a value from one datatype to another, but the value cannot be converted to the target datatype.

Listing 4.6 The Convert.ToInt32() Method Attempts to Convert the String “hello” Into an Integer. Since the String “hello” Cannot Be Parsed as a Valid Integer, an InvalidCastException Is Thrown and Caught by the Catch Block

```
string input = "hello";
int number;

try
{
    // Attempt to convert the string to an integer
    number = Convert.ToInt32(input);
}

catch (InvalidCastException e)
{
    // Handle the invalid cast exception by displaying an error message
    Console.WriteLine("Unable to convert '{0}' to an integer.", input);
}
```

A FormatException

This outcome occurs when we try to convert a string value into a numeric datatype, but the string is not in the correct format for the desired datatype.

Listing 4.7 The Convert.ToString() Method Attempts to Convert the String “3.14hello” Into a Float. Since the String “3.14hello” Is Not in the Correct Format for a Float (It Contains Both a Decimal Point and Letters, Which Are Not Allowed in a Float), a FormatException Is Thrown and Caught by the Catch Block

```
string input = "3.14hello";
float number;

try
{
    // Attempt to convert the string to a float
    number = Convert.ToString(input);
}

catch (FormatException e)
{
    // Handle the format exception by displaying an error message
    Console.WriteLine("Unable to convert '{0}' to a float.", input);
}
```

NOTE Although similar, **InvalidOperationException** and **FormatException** differ in some ways. In both examples, the try block attempts converting a value from one datatype to another, but the conversion fails. In listing 4.4, the value being converted (the string “hello”) cannot be converted to the target datatype (an integer) because it is not in a format that can be parsed as an integer. In listing 4.5, the value being converted (the string “3.14hello”) is in the wrong format for the target datatype (a float) because it contains letters, which are not allowed in a float.

An OverflowException

This outcome occurs when a numeric value is too large or too small to be used by the target datatype.

Listing 4.8 The Convert.ToInt16() Method Attempts to Convert the String “1234567890” Into a Short Integer. Since the Number 1234567890 is Too Large to Be Represented By a Short (Which Has a Maximum Value of 32767), an OverflowException is Thrown and Caught by the Catch Block

```
string input = "1234567890";
short number;

try
{
    // Attempt to convert the string to a short
    number = Convert.ToInt16(input);
}
catch (OverflowException e)
{
    // Handle the overflow exception by displaying an error message
    Console.WriteLine("The value '{0}' is too large
        to be represented by a short.", input);
}
```

A Successful Conversion

These outcomes result in a correct and successful conversion from the input data type to the final desired type, applicable to any conversion not mentioned in the previous points in this list.

Listing 4.9 The Convert.ToInt32() Method Attempts to Convert the String “10” Into an Integer. This Conversion Will Successfully Return the Given String in an int Format

```
string text = "10";
int conversion = Convert.ToInt32(text);
// Will successfully return the given string in an int format.
```

To get these outcomes, we must now dive deeper into the various Convert class methods.

4.4.3 Convert Methods

This finally leaves us with the question of our options for successful conversions. Even though we have confirmed the Convertible types, those being Boolean, char, sbyte, byte, int16, int32, int64, uint16, uint32, uint64, single, double, decimal, dateTime, and string, we have yet to see proper syntaxes for any method except for Int32.

Let us look at ten common methods for the Convert class that we can find in most projects (Table 4.3).

Table 4.3 This table describes a selected list of methods for the Convert class

Method	Description
GetTypeCode	Returns the TypeCode for the specified object
ToInt32	Converts a specified value to a 32-bit signed integer
ToDouble	Converts the specified value to an equivalent double-precision value
ToDecimal	Converts the specified value to an equivalent decimal value
ToDateTime	Converts the specified value to an equivalent date-time value
ToChar	Converts the specified value to its equivalent Unicode character
ToByte	Converts the specified value to an equivalent 8-bit unsigned integer
ToBoolean	Converts the specified value to an equivalent Boolean value
ToSingle	Converts the specified value to an equivalent single-precision floating-point number
ChangeType	Returns an object of the specified type whose value is equivalent to the specified object

As always, there are way too many examples to list them all, let alone define them in detail. However, we can always visit the official documentation (Microsoft Corporation, n.d.) (direct link <https://docs.microsoft.com/en-us/dotnet/api/system.convert?view=net-6.0#methods>) for a complete list.

Since we have already seen some examples of the use of some of the most common Convert classes, we should probably continue to the next section to finally cover one of the most important topics in programming, classes.

We are ready for this chapter's project, the "Guess the number" game.

Let us get started.

4.5 Guess the Number

As in the previous chapters, we must start with a base by creating a new .NET Console App project. The intention is to always start with a clean and empty template (Fig. 4.7).



```
Program.cs  X
TutorialsEU_GuessTheNumber
1 // See https://aka.ms/new-console-template for more information
2 Console.WriteLine("Hello, World!");
3
```

Fig. 4.7 A new empty project, the beginning of our fourth project

Let us analyze what we want to achieve here to determine the next step.

4.5.1 The Project

Guess the Number is a game where we must use our logic to guess a secret number selected by the computer at the beginning of the game. The number is formed with a range between 1 and 10.

A player then guesses this number via multiple attempts. An attempt consists of a proposed number entered by the player and the computer's reply. In its reply, the computer must tell us if the secret number is less than or greater than the guess.

Using information from the computer's reply, we must guess the number using as few guesses as possible.

So what we need is, first, to generate a random number and save it to an accessible variable. We will achieve that in our program entry point, the Main() method. Then, we will ask the user for a number. This will be compared to the random value, and a response will be formulated. The user input number is less than, greater than, or equal to the random value.

Depending on that result, we will either let the user know about the difference in the random value and prompt another attempt or cue the win condition, ending the game. We will increment the counter on each try to further assist the achievable goal of guessing the correct number in the least number of tries possible.

Without further ado, let us get started with the development.

4.5.2 Our Code

Currently, we are in front of the basic template project provided by Visual Studio. We need to start the project by creating a class with its entry point. Since we already learned how to accomplish this, let us simply clean the project and add our class and static method.

Our current starting point should look something like this (Fig. 4.8):

```
0 references
public class Program
{
    static void Main()
}
```

Fig. 4.8 We have our class and entry point set up

Now we can start implementing the logic that our project requires. As we already stated, we will need a randomly generated number.

The first step is to create a new “Random” class variable. Then initialize it with a new `Random()`.

```
public class Program
{
    static void Main()
    {
        // Initialise an instance of the Random Class
        Random random = new Random();
    }
}
```

Then, get a new random number set to a variable. Now, last time we simply created a new variable and initialized it with a random number. While we could still do that, this would turn our variable into a local variable, meaning it is only accessible within our current method. In this project, the guessing code will be located outside the `Main()` method, meaning our random number variable will not be accessible. Then again, we already saw that calling a method with parameters is an option, removing the need for global variables, but these cases are the most ideal for global variables.

The design for this chapter’s project will be to isolate the guessing code within a different method since it is a block that may need execution more than once.

So it is correct to guess that this value must be accessible from everywhere in the code for this to function correctly.

Luckily, doing so is pretty easy, so let us do so.

Create a variable on top of the `Main()` method.

```
public class Program
{
    // Initialise the secretNumber to zero
    int secretNumber = 0;
    static void Main()
    {
        // Initialise an instance of the Random Class
        Random random = new Random();
    }
}
```

Now, in our `Main()` method, set the variable with a random number.

```
public class Program
{
    // Initialise the secretNumber to zero
    int secretNumber = 0;
```

```
static void Main()
{
    // Initialise an instance of the Random Class
    Random random = new Random();

    // Generate a random (secret) number between 1 and 10
    secretNumber = random.Next(1, 10);
}
```

This will throw an error, again something that used to work but now does not (Fig. 4.9).

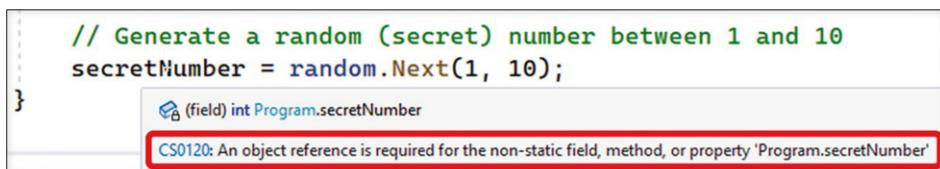


Fig. 4.9 We get a new error, error code CS0120, referring to an object reference needed for the non-static field

While these minor variations in functionality can sometimes be frustrating, we will notice that once we learn to handle them correctly, they stop being an inconvenience. Here, we need to use a small feature we learned previously in this chapter, object creation.

We must remember that static methods, in contrast to non-static methods, are not inherently instanced each time but run as one shared instance. This creates the problem where we must specify the affected instance if we try to modify a variable or call a method in a specific code.

To accomplish this, we must first do the following: create an instance reference to our program class within the Main method.

```
//Create an instance reference of our Program class
Program program = new Program();
```

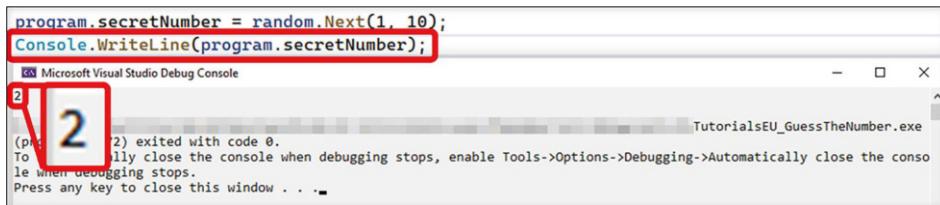
Then, we can use it to specify which object's variable we want to set, leaving us with this result:

```
//Create an instance reference of our Program class
Program program = new Program();
// Generate a random (secret) number between 1 and 10
program.secretNumber = random.Next(1, 10);
```

Finally, the error is gone. We can test it by displaying our random value with a call to the `Console.WriteLine` method.

```
Console.WriteLine(program.secretNumber);
```

This current code will now result in the following (Fig. 4.10).



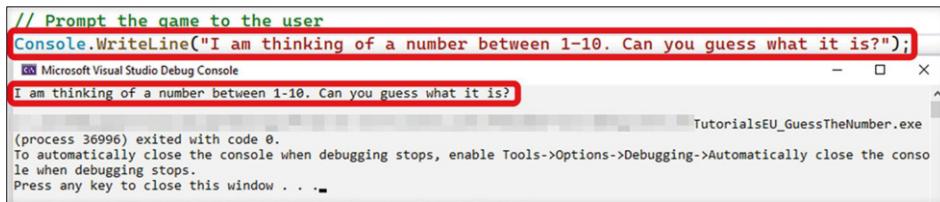
A screenshot of the Microsoft Visual Studio Debug Console window. The console output shows the line "Console.WriteLine(program.secretNumber);". Below it, the number "2" is displayed in a large blue font, indicating the randomly generated secret number. The console also includes standard debugging information like exit code and options for closing the window.

Fig. 4.10 Our first result with a randomly generated number

Great, we will not need that `Console.WriteLine()` anymore, but we can use it to prompt the user with the game, so the user will know how to play it. Something like “I am thinking of a number between 1 and 10. Can you guess what it is?” would do.

```
program.secretNumber = random.Next(1, 10);
// Prompt the game to the user
Console.WriteLine("I am thinking of a number between 1-10.
Can you guess what it is?");
```

This will display our title at the beginning of the first execution (Fig. 4.11).



A screenshot of the Microsoft Visual Studio Debug Console window. The console output shows the line "Console.WriteLine("I am thinking of a number between 1-10. Can you guess what it is?");". Below it, the text "I am thinking of a number between 1-10. Can you guess what it is?" is displayed in a large blue font, indicating the game's prompt to the user. The console also includes standard debugging information like exit code and options for closing the window.

Fig. 4.11 Our game was successfully prompted to the user

Now that we have the game prompted and the random number ready to be guessed, we need to start working on the guessing logic.

Let us start by creating a new void method called `Guess`.

```
void Guess()
{
}
```

We need to figure out our first step in entering this `guess` method. Before any comparison or any kind of logic can happen, we need to get the user’s input. Previously in this chapter, we learned how to read a user input from the Console, which was through the use of `Console.ReadLine()`. This `Console` class method will return whatever the user types in as a string, which we then take and store into a variable.

Let us then first create a global variable for the user's guess.

```
public class Program
{
    // Initialise the secretNumber to zero
    int secretNumber = 0;
    // Initialise the guess to zero
    int guess = 0;
    static void Main()
    {
```

This should give us a variable ready to be populated with the return value of our `Console.ReadLine()` method. So again, right after our `Main()` method, in our `guess()` method, let us do just that.

```
void Guess()
{
    guess = Console.ReadLine();
}
```

And straight away, we get an error, to be specific, error code CS0029 stating “Cannot implicitly Convert type ‘string’ to ‘int.’”. This does make sense as it seems that we got ourselves into a type mismatch. This is due to the `guess` variable being an “int” while the return type of `Console.ReadLine()` is “string” (Fig. 4.12).

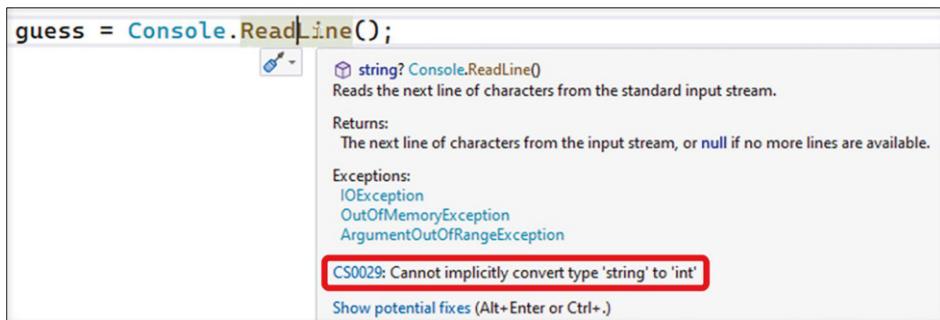


Fig. 4.12 We got an error, CS0029, cannot implicitly Convert type ‘string’ to ‘int’

A simple solution would be to just change our `guess` variable to “string” instead of “int,” but this would generate issues further on in our code when we try to compare the user guessed number with our randomly generated number, which is, if we remember correctly, an “int.”

For situations like that, we make use of the `Convert` class. Recalling the table of possible `Convert` class methods, we can find a specific method that converts any specified string

representation of a number to an equivalent 32-bit signed integer, the Convert.ToInt32() method.

Using it is as simple as the following:

```
void Guess()
{
    guess = Convert.ToInt32(Console.ReadLine());
}
```

This will already convert our returned string to a usable int. Let us make sure everything is working as expected by displaying the user guessed number once read.

```
void Guess()
{
    guess = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(guess);
}
```

And once run nothing happens. We never called the method. Since no parameters need to be passed in, doing so is as simple as writing the method at the end of our Main() method specifying our current object.

```
// Prompt the game to the user
Console.WriteLine("I am thinking of a number between 1-10.
    Can you guess what it is?");

// Call the instanced Guess method
program.Guess();

}

void Guess()
{
```

Great, run again, and now we can see something happen when we type and enter (Fig. 4.13).



Fig. 4.13 Matching user input with the read line

As we can see, the user input value matched the print from our code perfectly. We are almost done with the main bulk of the project! Let us set up our game logic! Here we need to compare the user input value with the randomly generated one. Let us do what we already know how to do. And that is set up a simple “if-elseif-else” conditional statement.

The basic logic behind this is, if the user input number is less than the randomly generated number, do this. Else if, if it is higher than the randomly generated number, do that. Else, if both match, cue the win condition.

The resulting conditional statements should look something like this:

```
void Guess()
{
    guess = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(guess);
    if (guess < secretNumber)
    {
    }
    else if (guess > secretNumber)
    {
    }
    else
    {
    }
}
```

Perfect, let us determine the content of our conditional statements: what needs to happen on each match of a specific condition.

The concept is to prompt the user with a hint if the random number is higher or lower and then start the guess again, calling the guess() method once again. At least that is the case for our “if” and our “else if” condition. Let us display a text like this: “No, the number I am thinking of is higher than ‘ + guess + .’ Can you guess what it is?”

```
if (guess < secretNumber)
{
    Console.WriteLine("No, the number I am thinking of is
                      higher than " + guess + ". Can you guess what it is?");
}
```

Then, call the guess method again for another try. Since we already know that recursive method calling is possible in C#, the only thing needed is the following:

```
if (guess < secretNumber)
{
    // If the player's guess is less than the secret number
    // Give the player a hint and call guess again
    Console.WriteLine("No, the number I am thinking of is
                      higher than " + guess + ". Can you guess what it is?");
    Guess();
}
```

This time, we can call the method without the “Program.” simply because we no longer call a non-static method from a static method.

Now, do the same for the “else if” condition.

```
else if (guess > secretNumber)
{
    // If the player's guess number is greater than the secret number
    // Give the player a hint and call guess again
    Console.WriteLine("No, the number I am thinking of is lower
                      than " + guess + ". Can you guess what it is?");
    Guess();
}
```

NOTE We are using recursive method calling in this chapter’s project. However, we must say that this is only done here since we assume there will not be a need for many recursions. While this is not incorrect in this specific scenario, we do have to warn that if this is done in a code where the recursion would need to execute thousands or even only hundreds of times, it is advised not to use recursion but rather a loop, which we will cover in the next chapter of this book.

And finally, set a win scenario inside the else condition since the only condition that will match with “else” is both the user input number and the randomly generated number to be the same, which happens to be our game’s win condition.

Let us call a new method, a Win() method, where we will handle our win scenario.

```
else
{
    // Any case other than the previous two, is a match and
    // therefore, a win condition. So call Win method with
    // the current guess attached.
    Win();
}
```

We naturally need a Win() method, which we can create after our current Guess() method.

```
void Win()
{
}
```

Now, let us formulate a win prompt, something like “Well done! Your guess of 5 matched with the secret number of 5!”

```
void Win()
{
    Console.WriteLine(
        "Well done! Your guess of " + guess +
        "matched with the secret number of " + secretNumber + "!");
}
```

And with that, let's try it out (Fig. 4.14).

```
void Win()
{
    Console.WriteLine("Well done! Your guess of " + guess + " matched with the secret number of " + secretNumber);
}

Microsoft Visual Studio Debug Console
I am thinking of a number between 1-10. Can you guess what it is?
5
5
No, the number I am thinking of is higher than 5. Can you guess what it is?
8
8
No, the number I am thinking of is lower than 8. Can you guess what it is?
7
7
Well done! Your guess of 7 matched with the secret number of 7! Only took you 3 tries!
```

Fig. 4.14 Our win condition triggered and correctly showed the values given

After two tries, we managed to guess the correct number! Let us find a simple way to show how many tries were needed.

For that, just simply create a new “int” type variable called “tries” right after our “guess” variable.

```
// Initialize the tries to zero
int tries = 0;
```

Each time `guess ()` is executed, add a try to it with the “`++`” operator. With this, we essentially add one to the `tries` variable, similar to doing `tries + 1`.

```
guess = Convert.ToInt32(Console.ReadLine());
tries++;
```

And within the win condition, modify the `Console.WriteLine()` method to include the `tries` in the winning prompt.

```

void Win()
{
    Console.WriteLine(
        "Well done! Your guess of " + guess +
        "matched with the secret number of " +
        secretNumber + "! Only took you " + tries +
        "tries!");
}

```

Use this opportunity to clean up our code for debugging `Console.WriteLine()`s also, like the one we used to check if we read the user input correctly. Keeping the importance of clean code in mind, especially now that we are starting to increase complexity, is important. A famous quote about this convention is, “*Write code as if the next person reading it knows where you live.*” Although amusing at first, it shows the significance of caring for not only us but also those who work with us (Fig. 4.15).

```

void Guess()
{
    guess = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(guess); REMOVE THIS LINE
    tries++;

    if (guess < secretNumber)
    {
}

```

Fig. 4.15 Keep our code clean and readable without unnecessary lines

And now, since we are done, we can finally proceed to run our finished project (Fig. 4.16).

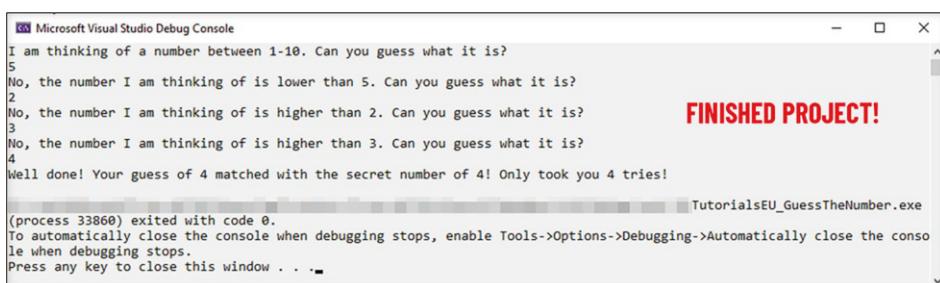


Fig. 4.16 Everything displays correctly, marking the end of this project

NOTE To simplify the scope and complexity of the projects, we have not yet covered each possible user input scenario. This means there is no specific solution to inputs that do not match the expected input type.

With this fourth chapter finished, we have not yet seen the full potential of C#, its classes, methods, or any of the fundamentals for that matter. We will get into much more detail in future chapters, however. For one, classes were just slightly used during this chapter's project. However, in Chap. 10, where we will attempt to build an SMTP mailing service, and we will get introduced to the Single Responsibility principle, where each class should be responsible for only one specific task. So we will delve much deeper into every topic at hand over time.

For now, though, we want to wrap up the fundamentals of C# with our next chapter, covering arrays, collections, loops, and the System.Linq namespace to build a fully functional Tic Tac Toe game for both playing against an AI and playing against another user.

Much to cover and much to learn, so let's get ready for the next chapter.

4.5.3 Source Code

Link to the project on [Github.com](https://github.com/tutorialseu/TutorialsEU_GuessTheNumber):

https://github.com/tutorialseu/TutorialsEU_GuessTheNumber

4.6 Summary

This chapter taught us:

- The console class represents the standard input, output, and error streams for console applications.
- The C# console class provides many properties for manipulation and customization of our terminal environment.
- We use console methods to permit the reading or writing of user input or code-generated data to the Console.
- We use the Convert class for situations where a specific data type is required but not provided by the code that feeds it to us.
- The Convert class provides methods to convert most data types to a different data type, requiring the input data type to match the conversion base and outputting the desired data type.
- The base types supported by the Convert class are Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime, and String.
- Classes and objects are the basic building blocks of object-oriented programming. A class can be seen as a user-defined blueprint from which objects are created.
- Generally, syntax-wise, a class contains only a keyword, followed by the identifier of the class, this being the name we want to assign it.
- Although sometimes used interchangeably, a class and an object are distinct concepts in C#. A class serves as a template for creating objects during compile time, defining

their structure, properties, and methods. In contrast, an object is an instance of a class created during runtime, representing data in memory that can be manipulated through methods defined by the class.

- In C#, classes fully support inheritance, an essential and commonly useful characteristic of object-oriented programming.
 - A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required arguments. In C#, every executed instruction is performed in the context of a method.
 - Methods are declared in a class, struct, or interface by specifying the access level, optional modifiers, the return value, the name of the method, and any method parameters.
 - Even though not obligatory, methods can return a value or object after calling.
-

References

- Bodnar, J. (2022, January 6). Methods in C#. Retrieved May 19, 2022 from [zetcode.com: https://zetcode.com/lang/csharp/methods/#:~:text=C%23%20method%20signature,not%20include%20the%20return%20type](https://zetcode.com/lang/csharp/methods/#:~:text=C%23%20method%20signature,not%20include%20the%20return%20type)
- Dhar, S. (2020, December 9). What is Recursion in C# | C# Tutorials. Retrieved May 19, 2022 from [tutorialslink.com: https://tutorialslink.com/Articles/What-is-Recursion-in-Csharp-Csharp-Tutorials/234#:~:text=C%23%20%7C%20C%23%20Tutorials-,The%20recursive%20function%20or%20method%20is%20a%20very%20strong%20functionality,known%20as%20a%20recursive%20function](https://tutorialslink.com/Articles/What-is-Recursion-in-Csharp-Csharp-Tutorials/234#:~:text=C%23%20%7C%20C%23%20Tutorials-,The%20recursive%20function%20or%20method%20is%20a%20very%20strong%20functionality,known%20as%20a%20recursive%20function)
- Fedewa, J. (2021, February 25). What Is a Human Interface Device (HID)? Retrieved May 19, 2022 from [www.howtogeek.com: https://www.howtogeek.com/713565/what-is-a-human-interface-device-hid/](https://www.howtogeek.com/713565/what-is-a-human-interface-device-hid/)
- GeeksforGeeks. (2019, April 3). C# | Convert Class. Retrieved May 2, 2022 from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/c-sharp-convert-class/](https://www.geeksforgeeks.org/c-sharp-convert-class/)
- GeeksForGeeks. (2021, August 16). C# | Methods. Retrieved May 2, 2022 from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/c-sharp-methods/](https://www.geeksforgeeks.org/c-sharp-methods/)
- GeeksForGeeks. (2022, February 23). C# | Class and Object. Retrieved May 2, 2022 from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/c-sharp-class-and-object/](https://www.geeksforgeeks.org/c-sharp-class-and-object/)
- Injisoft AB. (n.d., n.d. n.d.). ASCII Code - The extended ASCII table. Retrieved May 19, 2022 from www.ascii-code.com: https://www.ascii-code.com/
- Microsoft Corporation. (2021, September 15). Introduction to classes. Retrieved May 2, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/classes](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/classes)
- Microsoft Corporation. (2021, December 9). Methods (C# Programming Guide). Retrieved May 2, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/methods)
- Microsoft Corporation. (2021, September 15). sealed (C# Reference). Retrieved May 19, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/sealed)
- Microsoft Corporation. (2022, August 12). C# Coding Conventions. Retrieved August 16, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions)

- Microsoft Corporation. (2022, January 12). Delegates (C# Programming Guide). Retrieved May 19, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/)
- Microsoft Corporation. (2022, January 12). namespace. Retrieved May 19, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/namespace](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/namespace)
- Microsoft Corporation. (n.d., n.d. n.d.). Console Class. Retrieved April 27, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.console?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.console?view=net-6.0)
- Microsoft Corporation. (n.d., n.d. n.d.). Console.CancelKeyPress Event. Retrieved May 18, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.console.cancelkeypress?redirectedfrom=MSDN&view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.console.cancelkeypress?redirectedfrom=MSDN&view=net-6.0)
- Microsoft Corporation. (n.d., n.d. n.d.). Convert Class. Retrieved May 2, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.convert?view=net-6.0#NonDecimal](https://docs.microsoft.com/en-us/dotnet/api/system.convert?view=net-6.0#NonDecimal)
- Microsoft Corporation. (n.d., n.d. n.d.). Convert class - Methods. Retrieved May 2, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.convert?view=net-6.0#methods](https://docs.microsoft.com/en-us/dotnet/api/system.convert?view=net-6.0#methods)
- Parewa Labs. (n.d., n.d. n.d.). C# Method. Retrieved May 2, 2022 from [www.programiz.com: https://www.programiz.com/csharp-programming/methods](https://www.programiz.com/csharp-programming/methods)
- Parewa Labs Pvt. Ltd. (n.d., n.d. n.d.). C# Method Overloading. Retrieved May 19, 2022 from [www.programiz.com: https://www.programiz.com/csharp-programming/method-overloading](https://www.programiz.com/csharp-programming/method-overloading)
- Tutorials Point. (n.d., n.d. n.d.). C# - Methods. Retrieved May 2, 2022 from [www.tutorialspoint.com: https://www.tutorialspoint.com/csharp/csharp_methods.htm](https://www.tutorialspoint.com/csharp/csharp_methods.htm)
- TutorialsTeacher. (n.d., n.d. n.d.). C# Class. Retrieved May 2, 2022 from [www.tutorialsteacher.com: https://www.tutorialsteacher.com/csharp/csharp-class](https://www.tutorialsteacher.com/csharp/csharp-class)
- Vatsa, A. K. (2019, February 24). Static Method In C#. Retrieved May 2, 2022 from [www.c-sharpcorner.com: https://www.c-sharpcorner.com/UploadFile/abhikumarvatsa/static-methods-in-C-Sharp/](https://www.c-sharpcorner.com/UploadFile/abhikumarvatsa/static-methods-in-C-Sharp/)
- W3Schools. (n.d., n.d. n.d.). C# Classes and Objects. Retrieved May 2, 2022 from [www.w3schools.com: https://www.w3schools.com/cs/cs_classes.php](https://www.w3schools.com/cs/cs_classes.php)
- W3Schools. (n.d., n.d. n.d.). C# Method Parameters. Retrieved May 2, 2022 from [www.w3schools.com: https://www.w3schools.com/cs/cs_method_parameters.php](https://www.w3schools.com/cs/cs_method_parameters.php)
- W3Schools. (n.d., n.d. n.d.). C# Methods. Retrieved May 2, 2022 from [www.w3schools.com: https://www.w3schools.com/cs/cs_methods.php](https://www.w3schools.com/cs/cs_methods.php)



C# Collections and Iterators

5

This Chapter Covers

- Using Array in C#
- Working with Collections
- Making use of the shared System.Linq namespace
- Improving our code with loops and iteration
- Developing a Tic Tac Toe minigame

With this fifth chapter, we have finally reached the end of the introductory portion of this volume. So far, we have learned everything about C# as a utility, medium, and tool, from the technology stack behind to the individual components that make this language. We learned programming basics and built a project on each step throughout our journey.

With this final step, we will get comfortable with the last programming basics essential to any application and set up the most extensive and feature-rich project so far.

In this chapter, we will create an application famous for its seemingly simple appearance but surprisingly complicated code, the well-known Tic Tac Toe minigame.

This project entails the need to master much more than what we have worked on until now. This time, apart from variables, conditionals, console methods, and operators, we also need to move toward arrays, collections, and iterators. Even a simple AI is needed not to obligate the user to include a second player.

As we can see, this fifth project will be the first to encompass everything introductory a programming language has to offer, aiding us to master the language by itself before we embark on this book's advanced sections.

5.1 Arrays in C#

To introduce ourselves to the final basics of software development, we will start with the simplest form of object collectors, arrays. We might have noticed if we peeked at the continuing subchapters of this fifth chapter that we made a differentiation between both arrays and collections. That differentiation is not arbitrary but intentional, and we will reiterate the reasoning after going over arrays and collections as individual concepts stating core differences between one and the other.

5.1.1 The Definition of an Array

It is a structure representing a fixed-length ordered collection of values or objects of the same Type. That is, by definition, an array. However, this does not clarify why we should care about arrays.

Arrays make it easy to organize and operate large amounts of data. Take, for example, a situation where we need tens or hundreds of variables of the same Type. Declaring them one by one would be tedious and generally unusable. Using arrays, however, we can easily store each of those integers without needing to address them individually.

Listing 5.1 A Simple Example of the Use of an Array to Create 100 Variables

```
int[] numbers = new int[100];

for (int i = 0; i < numbers.Length; i++)
{
    numbers[i] = i;
}

Console.WriteLine("The elements of the array are:");
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

Console.ReadLine();
```

In an array, each containing variable item is called an element of the array, and their datatypes may be any valid datatype like char, int, or float, among others, including another array type. The variables in the array are ordered, each with an index beginning from 0.

Array lengths have to be specified when the array instance is created, making that array of a set size that cannot be changed.

Let us go through some important points about arrays in C#.

In C#, all arrays are dynamically allocated. This means that memory space is assigned during the execution or runtime.

To determine the length of an array, differing from C and C++, where this is achieved using the sizeof operator, in C#, since arrays are objects of base type System.Array, we can use its Length property.

Default values of numeric array and reference type elements are set to zero and null, respectively.

All arrays in C# implement the IList¹ and IEnumerable² interface. With this, we can use the foreach statement to iterate through an array. Single-dimensional arrays also implement IList<T> and IEnumerable<T>. More on these later.

Arrays and specialized collections are statically typed, which means that the element restriction to just being a single one is strict, but also that when an element is fetched, no recasting is needed.

NOTE Reference type arrays are only mostly safe when storing values because of array covariance, the implicit reference conversion for array types, delegate types, and generic type arguments. This is seen as an early design mistake for which we won't be able to expand since it would go far beyond the scope of this book. However, if we want to learn more about that, we can find more information in the official documentation (Microsoft Corporation, 2021) (direct link <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/>).

5.1.2 Declaring Arrays

A C# array variable is declared similarly to a non-array variable, adding square brackets ([])) after the type specifier to denote it as an array, as seen in this code snippet.

```
string[] path = new[] { "UK", "US", "UAE" };
```

When instantiating a new array, a new keyword is used. Also, within the square brackets, a length for the array has to be specified. If the final values are known at initialization, we also have the option to initialize it using the elements in a comma-separated list surrounded by curly braces ({}). However, note that the number of elements given on initialization will be used to set the array's length.

In the following example, we can see the declaration of an array using both methods.

```
// Declare an array of length 8 without setting the values.  
string[] stringArray = new string[8];
```

¹ Represents a non-generic collection of objects that can be individually accessed by index.

² Exposes an enumerator, which supports a simple iteration over a non-generic collection.

```
// Declare array and set its values to 3, 4, 5.  
int[] intArray = new int[] { 3, 4, 5 };
```

In C#, the elements of an array are numbered incrementally, starting at 0 for the first element. This is a characteristic commonly challenging to get used to. For example, the third element of an array would be indexed at 2, and the sixth element of an array would be indexed at 5. Knowing this, a specific element can be accessed using the square bracket operator, surrounding the index with square brackets.

In the following example, we can see element index-specific operations.

```
// Initialize an array with 6 values.  
int[] numbers = { 3, 14, 59, 26, 53, 0 };  
  
// Assign the last element, the 6th number in the array, to 58.  
numbers[5] = 58;  
  
// Store the first element, 3, in the variable `first`.  
int first = numbers[0];
```

As we probably noticed in this previous example, we can avoid the new expression and the array type when we initialize an array upon declaration, as shown in the following code. This is called an implicitly typed array:

```
int[] array2 = { 1, 3, 5, 7, 9 };  
string[] weekDays2 = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

As previously mentioned, the Length property of a C# array can be used to get the number of elements in a particular array. We use the following example to visualize this procedure.

```
int[] someArray = { 3, 4, 1, 6 };  
Console.WriteLine(someArray.Length); // Prints 4  
  
string[] otherArray = { "foo", "bar", "baz" };  
Console.WriteLine(otherArray.Length); // Prints 3
```

Definition and declaration are cleared up with these previous two subsections, but we have not yet touched on the various types of arrays.

5.1.3 Types of Arrays

Even though already mentioned, until now, we already have seen single-dimensional arrays, being those that see the most use, but these are only one of the three types available. In C#, we can find single-dimensional arrays, multi-dimensional arrays, and jagged arrays. Let us continue with a quick overview of how these can be described and declared.

Single-Dimensional Arrays

Single-dimensional arrays are simple arrays with a set of elements in order. As we have seen in previous examples, we can create single-dimensional arrays using the new operator specifying the array element type and length. The following example teaches the declaration of such an array.

```
int[] array = new int[5];
```

This array is declared with a total of five elements ranging from index array[0] to array[4], initializing, being of type integer, with the number 0.

NOTE We can declare an array variable without creating it, but to compile, we must use the new expression once we wish to populate it.

```
int[] array3;  
array3 = new int[] { 1, 3, 5, 7, 9 }; // OK  
//array3 = {1, 3, 5, 7, 9}; // Error
```

Multi-dimensional Arrays

Commonly, arrays can have more than one dimension. A multi-dimensional array in C# is an array that contains more than one dimension to store the data, up to 32. The multi-dimensional array can be declared by adding commas in the square brackets.

For example, the following declaration creates a two-dimensional array of four rows and two columns. And additionally, an array of three dimensions, 4, 2, and 3.

```
int[,] array = new int[4, 2]; //2 Dimensions  
int[, ,] array1 = new int[4, 2, 3]; //3 Dimensions
```

As with single-dimensional arrays, we can initialize multi-dimensional arrays upon declaration. In the following example, we populate a two-dimensional and three-dimensional array.

```
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };  
// Two-dimensional array.
```

```
int[,] array3D = new int[,] { { { 1, 2, 3 }, { 4, 5, 6 } }, { { 7, 8, 9 }, { 10, 11, 12 } } }; // Three-dimensional array.
```

As long as we add the necessary commas between the square brackets, we can also declare multi-dimensional arrays without setting the number of rows.

```
int[,] array;  
array = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

Furthermore, individually setting an array element to a new value can be done like single-dimensional arrays by using the square brackets and specifying the position to be filled.

```
//array before: { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };  
array[2, 1] = 25;  
//array after: { { 1, 2 }, { 3, 4 }, { 5, 25 }, { 7, 8 } };
```

Jagged Arrays

Jagged arrays differentiate themselves from the two previous arrays by being arrays whose elements are arrays. Jagged arrays are sometimes called “array of arrays.” Essentially, jagged arrays are just single-dimensional or multi-dimensional arrays capable of referencing other arrays within.

This array type is represented by the use of two sets of square brackets instead of one. The following example shows a simple declaration of a single-dimensional array filled with three int-type arrays.

```
int[][] jaggedArray = new int[3][];
```

Note that before we can make use of a jagged array, its elements must be initialized. You can initialize the elements like this:

```
jaggedArray[0] = new int[3];  
jaggedArray[1] = new int[4];  
jaggedArray[2] = new int[5];
```

In the previous example, we implemented three single-dimensional arrays of integer type. And as we already know, initializing them at declaration is an option. This can still be said for single-dimensional arrays within jagged arrays, achieved in two ways.

Listing 5.2 The Jagged Arrays Are Either Initialized With a Single-Dimensional Array on Each Position or Using the Array Initializer Syntax. Initialized to Contain Three Elements, Each of Which Is an Array of Integers

```
// Method one.  
jaggedArray[0] = new int[] { 1, 2, 3 };  
jaggedArray[1] = new int[] { 4, 5, 6 };  
jaggedArray[2] = new int[] { 7, 8, 9 };  
  
//Method two  
int[][] jaggedArray2 = new int[][]  
{  
    new int[] { 1, 2, 3 },  
    new int[] { 4, 5, 6 },  
    new int[] { 7, 8, 9 }  
};  
  
//Shorthand method  
int[][] jaggedArray3 =  
{  
    new int[] { 1, 2, 3 },  
    new int[] { 4, 5, 6 },  
    new int[] { 7, 8, 9 }  
};
```

One interesting feature is the inclusion of multi-dimensional arrays within jagged arrays. Next, we can see an example of a single-dimensional jagged array containing two multi-dimensional arrays.

```
int[,][] jaggedArray4 = new int[2][,]  
{  
    new int[,] { {1,2}, {3,4} },  
    new int[,] { {5,6}, {7,8}, {9,10} },  
};
```

Additionally, we know that in C#, arrays are objects. This fact differentiates them from arrays in C and C++. Because of that, we can use properties and other class members for arrays.

5.1.4 Array Properties and Methods

The `Array` class is the base class for all arrays in C#, defined in the `System` namespace. It provides many practical methods and properties. An example can be seen by the well-known `Length` property, which returns the size of the attached array.

```
int[] numbers = { 1, 2, 3, 4, 5 };
int lengthOfNumbers = numbers.Length;
//Output: 5
```

Besides that, we also have many more, like those intended for sorting, searching, and copying. The following example uses the `Rank` property to display the number of dimensions of an array.

```
class TestArraysClass
{
    static void Main()
    {
        int[,] theArray = new int[5, 10];
        Console.WriteLine(theArray.Rank);
    }
}
// Output: 2
```

As expected, those two are not the only ones in the `array` class, so we will list them in one of those tables we like to use every now and then. The following table describes some of the most commonly used properties of the `Array` class (Table 5.1).

NOTE In the following table, we will find a property called `IsFixedSize` which indicates if an array is of fixed size or not. As we know, arrays will always be of fixed size in C#. This leads us to question the reason why this property exists to begin with. In short, the `IsFixedSize` property will indeed always return true for all arrays. `Array` implements the `IsFixedSize` property simply because it is required by the `System.Collections.IList` interface.

Table 5.1 This table describes a selected list of properties of the `Array` class

Property	Description
<code>IsFixedSize</code>	Gets a value indicating whether the array has a fixed size
<code>IsReadOnly</code>	Gets a value indicating whether the array is read-only
<code>IsSynchronized</code>	Gets a value indicating whether access to the array is synchronized (thread safe)
<code>LongLength</code>	Gets a 64-bit integer that represents the total number of elements in all the dimensions of the array
<code>MaxLength</code>	Gets the maximum number of elements that may be contained in an array

Let us continue with another table containing some of the methods available in the array class (Table 5.2).

Table 5.2 This table describes a selected list of methods of the Array class

Methods	Description
Clear()	Clears the contents of an array
Clone()	Creates a shallow copy of the array. A copy with references to the same elements as the original array, rather than copies of the elements themselves
Copy()	Copies a range of elements from an array starting at the first element and pastes them into another array starting at the first element. The length is specified as a 32-bit integer
Equals()	Determines whether the specified object is equal to the current object
GetType()	Gets the Type of the current instance

As we noted in previous chapters, we cannot provide the full lists due to their size and the planned scope of this chapter. For a complete list, consider visiting the official documentation (Microsoft Corporation, n.d.) (direct link <https://docs.microsoft.com/en-us/dotnet/api/system.array?view=net-6.0>).

Arrays are the object collectors we will be using the most throughout this book, especially during this chapter's project, but it is not the only one we will need over the following chapters. Arrays help deal with a collection of a known size. However, frequently, we do not know the number of elements we need to gather and operate on. For these situations, we are better off using collections.

5.2 Collections in C#

Collections are slowly replacing arrays in many ways, not only for their ease of use with hardly any limitations but also for their performance compared to arrays in certain situations. Being clear that arrays are not deprecated and will be a standard for object collection for the foreseeable future, the interest in collections is growing in an upward trend.

Where arrays are helpful if the collection size is and must be predetermined, we prefer using collections in any other situation. Collections provide a much more flexible way to work with groups of objects. These groups can grow and shrink dynamically as the needs of the application change. Even for one collection, we can assign a key to any object that we put into the collection so that we can quickly retrieve the object by using the key.

So let us get through collections and why they are so valuable.

5.2.1 The Definition of a Collection

The collection class is a set of specialized classes purposed for storage and accessibility of data within the System.Collections namespace. Due to its class nature, the collection class has to be instantiated before we can add any elements to that collection.

Collections are designed to store, manage, and manipulate similar data more efficiently, including adding, removing, finding, and inserting data in the collection.

Within a standard use case of a collection containing like-typed elements, for example, being a set of int elements, the use of generic collections can be made. Under the System.Collections.Generic namespace, we can find collections that offer us the advantage of enforced type safety to avoid type mismatching and remove the need for type conversion.

Let us go over some of the functionalities that collections offer us.

Adding, Inserting, and Removing Items in a Collection

Unlike arrays, collections allow adding, inserting, and removing elements within, making it a much more dynamic system with the freedom to adapt to any program.

Finding, Sorting, and Searching Items

With the available collection methods, we can easily find, sort, and search an element within the collection, for example, using its name or part of its name.

Replacing Items

Allowing for directly replacing elements with others by pointing out the element to be replaced and the element to replace with.

Copy and Clone Collections and Items

Allows copying and cloning functions to create shallow copies³ and deep copies⁴ of the referenced collections.

Capacity and Count Properties

Simplifying the process of finding the length and sizes of the referenced collections.

Current .NET supports three types of collections. Prior to .NET 2.0, we still only had collections by themselves. However, with the introduction of generics and with that the introduction of the concept of type parameters to .NET that made it possible to design classes and methods that defer the specification of one or more types until the class or method is declared and instantiated by the client code, generic collections were also added.

Each Type of collection serves a specific purpose, with each functioning with a fixed ruleset and limitations that benefit depending on the use case.

Currently, we can find the following three collections in the System.Collections namespace.

- System.Collections.Generic classes
- System.Collections.Concurrent classes
- System.Collections classes

³A Shallow Copy is about copying an object's value type fields into the target object and the object's reference types are copied as references into the target object but not the referenced object itself.

⁴A deep copy occurs when an object is copied along with the objects to which it refers.

5.2.2 System.Collections Classes

In a non-generic collection, each element represents a value that can be of a different type. These collection sizes are not fixed and can be resized to fit more or fewer elements at any moment. The lists within the non-generic collections are considered legacy types, so it is advised to, whenever possible, use generic and concurrent types.

The lists contained within the non-generic collections namespace are still being used and offer unique advantages. However, these are slowly being replaced by similar advantages in modern generic types. The most notable advantage of generic collections over non-generic collections is code reusability. Generics assist in reusing an already written code, thereby not forcing redeclaration of collections when a similar one is needed.

Nevertheless, because non-generic classes are not declared deprecated, we will list the following collection types within the non-generic System.Collection namespace.

- **Hashtable.** This class represents a collection of key/value pairs organized based on the key's hash code. In the following example, we are putting together a sentence informing us about the number of elements within the Hashtable.

```
Hashtable hashtable = new Hashtable();
hashtable.Add(1, " Element 1");
hashtable.Add(2, " Element 2");
hashtable.Add(3, " Element 3");
Console.WriteLine("There is a total of " + hashtable.Count + " elements
within this hashtable.");
//Output: There is a total of 3 elements within this hashtable.
```

- **Queue.** This class represents a first-in, first-out (FIFO) collection of objects. The following example shows a series of possible inputs for the queue collection.

```
Queue queue = new Queue();
queue.Enqueue("String");
queue.Enqueue(1);
queue.Enqueue(null);
queue.Enqueue(2.4);
foreach (var element in queue)
{
    Console.WriteLine(element);
}
//Output:
/*
String
1
2.4
*/
```

- **Stack.** This class represents a last-in, first-out (LIFO) collection of objects. In the following example, we are populating a stack with one string, null, int, and decimal, and displaying them in a series of console writes.

```
Stack stack = new Stack();
stack.Push("String");
stack.Push(1);
stack.Push(null);
stack.Push(2.4);
foreach (var element in stack)
{
    Console.WriteLine(element);
}
//Output:
/*
2.4

1
String
*/
```

As already mentioned, non-generic collections, although useful in some situations, are not recommended to be used within a production environment, since for that, we should be making use of the more industry-standard generic classes. However, before we continue with the generic classes, let us review concurrent classes, a fairly new series of classes valuable for their efficiency.

5.2.3 System.Collections.Concurrent Classes

Since the birth of the .NET 4 framework, we have the addition of concurrent collection classes within the System.Collections namespace. These pretty new classes provide the technology stack efficient thread-safe operations, the technique which manipulates shared data structure to guarantee the safe execution of a piece of code by multiple threads (execution path of a program) simultaneously, for accessing collection items from multiple threads.

Concurrent collection classes are meant to be used instead of corresponding generic and non-generic classes whenever multiple threads access the collection concurrently.

Although we will not need to use them currently, these collections are an essential tool for, for example, server-side Web development, where working in a multithreaded context is the norm since every request will run in a separate thread. Or in environments like WPF⁵

⁵Windows Presentation Foundation is a UI framework that creates desktop client applications.

and Xamarin,⁶ where a UI can modify collections alongside background task modifications. So we had to at least mention them.

For this book's scope, concurrent collections are not a focus, so as we always recommend, we can find further information in the official Microsoft documentation (Microsoft Corporation, n.d.) (direct link <https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent?view=net-6.0>).

For now, let us continue with the next section. We will cover the most recommended collection type for the vast majority of applications, the `System.Collections.Generic` namespace.

5.2.4 System.Collections.Generic Classes

Generic collections in C# are introduced together with generics in C# 2.0. These strongly typed collections can be seen as an extension to the already mentioned non-generic collections. Strongly typed meaning that they will only accept one Type in a collection, reduce mismatch errors, and improve performance by removing the need for boxing⁷ and unboxing at runtime.

So before we go over some of the most widely used Collections in this namespace, we must first understand what generics are. In short, generics in C# are a way to write code that can work with multiple types in a type-safe manner. A generic collection, therefore, is designed to store and manipulate a collection of items, where the Type of the items is specified as a type parameter when the collection is instantiated. This allows the collection to be used with any datatype as long as the datatype meets specific requirements (e.g., implementing a particular interface).

If our program needs a collection of like-typed elements, we can use these collections by implementing the `System.Collections.Generic` namespace and any of its containing classes. Let us go over some of those and their example use cases.

- **Dictionary< TKey, TValue >** This class represents a collection of key/value pairs that are organized based on the key. In the following example, we create a dictionary of a pair of elements for the sentence “Hello World!” and display its content with a console method.

⁶Xamarin is an open-source platform for building modern and performant applications for iOS, Android, and Windows with .NET.

⁷Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. Unboxing extracts the value type from the object.

```
IDictionary<int, string> genericDictionary =
new Dictionary<int, string>();
genericDictionary.Add(1, "Hello");
genericDictionary.Add(2, "World!");

foreach (KeyValuePair<int, string> kvp in genericDictionary)
    Console.WriteLine("Key: " + kvp.Key + " Value: " + kvp.Value);

/*
Output:
Key: 1 Value: Hello
Key: 2 Value: World!
*/
```

- **List<T>** This class represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists. In the following example, we create a simple list of four numbers and write them to the console.

```
List<int> genericList = new List<int>();
genericList.Add(1);
genericList.Add(2);
genericList.Add(3);
genericList.Add(4);

foreach (var item in genericList)
    Console.Write(item + ","); //Output: 1,2,3,4,
```

- **Queue<T>** This class represents a first-in, first-out (FIFO) collection of objects. In the following example, we create a simple queue of type int with four numbers and display them to the console.

```
Queue<int> genericQueue = new Queue<int>();
genericQueue.Enqueue(1);
genericQueue.Enqueue(2);
genericQueue.Enqueue(3);
genericQueue.Enqueue(4);

foreach (var item in genericQueue)
    Console.Write(item + ","); //Output: 1,2,3,4,
```

- **Stack<T>** This class represents a last-in, first-out (LIFO) collection of objects. In the following example, we create a simple stack of type int with four numbers and display them to the console.

```
Stack<int> genericStack = new Stack<int>();
genericStack.Push(1);
genericStack.Push(2);
genericStack.Push(3);
genericStack.Push(4);

foreach (var item in genericStack)
    Console.Write(item + ","); //Output: 4,3,2,1,
```

Both Queue<T> and Stack<T> differentiate themselves from their non-generic counterpart by them needing to specify their Type.

We can still find more types of collection classes for each Type of collection. Like always, we invite you to visit the official documentation to get a complete list of all the collection types. For now, though, we might have noticed some methods used in the previous examples, like “Push” and “Add.” Collections in C# count with an array of methods and properties. Let us go over these in the following subsection.

5.2.5 Collection Properties and Methods

In C#, we have many properties and methods to use to work with collections. These can be non-generic, concurrent, or generic.

While some properties and methods are shared, we will generally have to use different ones depending on the collection that is being used.

We will go through some example use cases and their methods varying according to the applied collection,

starting with a typical use case, adding.

Adding New Element to a Collection

Shared by most collections, we regularly implement the addition of a new element like this:

```
List<int> genericList = new List<int>();
genericList.Add(1);
```

While this stands true to most, there can be variations, for example, in generic queues.

```
Queue<int> genericQueue = new Queue<int>();
genericQueue.Enqueue(1);
```

Additionally, stacks also get elements added differently.

```
Stack<int> genericStack = new Stack<int>();
genericStack.Push(1);
```

Removing Elements from a Collection

Typically we see its usage in this form.

```
List<int> genericList = new List<int>();  
genericList.Add(1);  
genericList.Remove(1);
```

Similarly shared among all collections, this action can also find some collections whose implementation varies, like with queues. However, this will remove the first position in the collection. There is no method of removing a specific element in a queue.

```
Queue<int> genericQueue = new Queue<int>();  
genericQueue.Enqueue(1);  
genericQueue.Dequeue();
```

Then we can also see differences in stacks. When using the pop method, we remove what is at the top of the stack, then return it. This one also has no method of removing a specific element.

```
Stack<int> genericStack = new Stack<int>();  
genericStack.Push(1);  
genericStack.Pop();
```

Moreover, in C# collections, we generally use the Count property or the Count() extension method to obtain the number of elements in a collection. It's worth noting that the Count() method checks if the object implements the ICollection interface and then returns the underlying Count property. Using the Count property directly can lead to slightly better performance, as it bypasses the type check performed by the Count() method. In the following example, both lines output the same result—the number of elements in the collection:

```
List<int> collection = new List<int>();  
Console.WriteLine(collection.Count());  
Console.WriteLine(collection.Count);
```

Then, if the collection object we are using is a simple array, although possible, no longer the count method is used. Now the Length property is the way to go.

```
int[] array = { 0, 1};  
Console.WriteLine(array.Count()); //technically possible but not common  
Console.WriteLine(array.Length);
```

As we can see, there are many methods and properties to achieve a single goal depending on the collection used. To not overextend the scope of this subsection, we think that the ideal way to cover this topic further will be learning more about them whenever we use them.

In programming, learning every bit of information and every last piece of code is an impossible task to achieve. It is common practice for beginners and long-time veterans to learn how to do the precisely given task at the moment of development, be it through the use of online or offline media. Books with scopes more directed to serve as documentation on a topic, the actual official documentation, or even online forums are mostly where we will be able to learn how to achieve our specific goals. So we need to keep our attention focused on learning as much as we can during the following chapters. For only in this way can we become the independent software engineers we strive to be.

Nevertheless, now that we have mentioned arrays again, we never delved deep into the differences between arrays and collections. Let us continue by clearing up all possible confusion about why we make this particular distinction.

5.2.6 Differences Between Array and Collection

Although arrays and the previously mentioned collections can both be considered types of collections, as they share the fundamental functionality of holding elements, there are some key differences that justify treating them separately. While it is true that some collections can hold elements of multiple types simultaneously (e.g., non-generic collections like `ArrayList`), both arrays and generic collections are designed to hold elements of a specific type. These distinctions make it essential to understand the capabilities and limitations of each type of collection when working with them in C#.

As a simple overview, we can immediately point out their sizing variances. As we already know, collections are of dynamic size, meaning that their length can be modified at runtime for addition or subtraction. At the same time, if we remember from the array subsection, their limitation fixed the size to the initialized length, forbidding any wanted change to that preset size, clearly differentiating them from common collections.

Unless desired by the introduction of generics, collections can also be heterogeneous, meaning that no type enforcing is set in place, gaining the option to add whatever Type we may need to the collection. Arrays are strongly typed, meaning that only one Type can be used per array.

Also, unless changed by the usage of generics, when collections remain heterogeneous, a boxing and unboxing process is introduced, whereas in arrays, there is none.

When making a selection between a List and an array, you'll find that the former provides more capabilities; however, this comes with an increased utilization of both CPU and memory resources. Consequently, one must weigh the advantages and disadvantages when making a decision. If the purpose of the collection is basic and its size is definite, an

array is the optimal choice. Conversely, if the collection is subject to change and requires processing, a List is the more appropriate option.

Although both ultimately serve the same purpose, our individual project needs will require one of the two, and while within collections, the requirement needs for the use of any is minimal, the decision to use either arrays or collections will entail a more significant need.

Now, with that out the way, we can continue with a shared feature they both present, a SQL feature used to work and modify arrays and collections efficiently: the System.Linq namespace.

5.2.7 The Shared System.Linq Namespace

Shared between both data element collector types is the System.Linq namespace. This system component provides us a set of methods for query capabilities directly into the C# language, meaning a set of instructions that describes what data to retrieve from a given data source (or sources) and what shape and organization the returned data should have.

The Language Integrated Query, known as LINQ, is the name given to the set of technologies with which a query becomes a first-class language construct, just like classes, methods, and events.

Traditionally, queries against data are expressed as simple strings without type checking at compile time or Intellisense support. Apart from that, a different query language must be learned for each data source type, like Structured Query Language (SQL⁸) databases or eXtensible Markup Language (XML⁹) documents.

The query expression is the most visible “language-integrated” part of LINQ for a developer who writes queries. Query expressions are a way to write LINQ code in a declarative way. This option means that instead of writing code to filter, order, and group data, we can write a query that describes what we want to do.

With the usage of LINQ, queries can be written directly to strongly typed collections of objects using language keywords and familiar operators. Furthermore, this provides a consistent query experience for objects (LINQ to Objects), relational databases (LINQ to SQL), and XML (LINQ to XML) and any collection of objects that supports IEnumerable or the generic IEnumerable<T> interface. Third parties also provide LINQ support for many Web services and database implementations.

⁸ Structured Query Language (SQL) is the language used in relational database management systems (RDBMS) to query, update, and delete data.

⁹ XML is short for eXtensible Markup Language. It is a very widely used format for exchanging data, mainly because it's easily readable for both humans and machines.

By implementing `IEnumerable<T>`, we are provided a set of static methods for integrating the standard query operators for querying data sources. These query operators are general-purpose methods that follow the LINQ pattern.

The majority of the methods in this class are defined as extension methods that extend `IEnumerable<T>`. This definition means they can be called like an instance method on any object that implements `IEnumerable<T>`. We can go through some of the commonly used query methods that LINQ provides us with the following examples.

- **The OrderBy() method.** Sorts the elements of a sequence in ascending order according to a key. The following example shows an unordered list of numbers being ordered by the LINQ method. Note that the `listOfNumbers` collection does not get modified by LINQ operations; rather a new collection is created instead, seen by the `MS` variable in this case.

```
List<int> listOfNumbers = new List<int>() { 6, 5, 4, 3, 7, 8, 9, 1, 2 };
Console.WriteLine("Before: ");
foreach (var item in listOfNumbers)
{
    Console.Write(item + " ");
}
Console.WriteLine();
var MS = listOfNumbers.OrderBy(num => num);
Console.WriteLine("After: ");
foreach (var item in MS)
{
    Console.Write(item + " ");
}
/*
Output:
Before:
6 5 4 3 7 8 9 1 2
After:
1 2 3 4 5 6 7 8 9
*/
```

- **The Any() method.** Determines whether a sequence contains any elements, that is, if it is empty or not. In the following example, we will generate a `Console.WriteLine` depending on if the list is empty or not.

```
List<int> listOfNumbers = new List<int>() { 6, 5, 4, 3, 7, 8, 9, 1, 2 };
bool hasElements = listOfNumbers.Any();
Console.WriteLine("The list " + (hasElements ? "is not" : "is")
    + " empty.");
/*Output:
The list is not empty.*/
```

As we can see from these two examples, these methods provide an easy way to handle and transform our collections, but they are not limited to those featured.

In the following table, we can find another eight examples of LINQ-provided methods we can use in our programs. Note that there are by far too many to include here, so make sure to visit the official documentation for further information (Table 5.3) (Microsoft Corporation, n.d.) (direct link <https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable?view=net-6.0>).

Table 5.3 This table describes a selected list of LINQ-provided methods

LINQ method	Description
First()	Returns the first element of a sequence
FirstOrDefault()	Returns the first element of a sequence, or a default value if the sequence contains no elements
Single()	Returns the only element of a sequence, and throws an exception if there is not exactly one element in the sequence
SingleOrDefault()	Returns the only element of a sequence, or a default value if the sequence is empty; this method throws an exception if there is more than one element in the sequence
Select()	Projects each element of a sequence into a new form by incorporating the element's index
Where()	Filters a sequence of values based on a predicate
OrderByDescending()	Sorts the elements of a sequence in descending order according to a key.
ToList()	Creates a List<T> from an IEnumerable<T>

NOTE Methods that are used in a query that return a sequence of values do not consume the target data until the query object is enumerated. This sequence of action is known as deferred execution. However, a method used in a query that returns a singleton value executes and consumes the target data immediately.

Now that we have learned everything essential about arrays and collections, let us learn something we have not seen yet. Previously in this chapter, we used some code to visualize our collections' contents. These were things like the foreach statement. Nevertheless, we still lack a proper understanding of these concepts. So for the following subsection, let us go over one of the vital elements in practically every programming language, well, except Haskell, one of the Functional programming languages. He does not have a looping construct. We do not talk to Haskell. He is weird.

(DISCLAIMER: Here at TutorialsEU, we do respect Functional and logic programming languages, and we do know that recursion can be used to substitute looping constructs within them; the previous declaration was purely made for comedic purposes.)

5.3 Iteration Statements in C#

Whenever we want to read and use either arrays or collections, iteration statements will make an appearance. And even though the System.Linq namespace allows us proper manipulation of them, and other options for their usage exist, including but not limited to

following the collections' name with a pair of square brackets and the index we want to retrieve, iteration statements will consistently continue to be a necessity when it comes to this task.

So since that is what we will learn about in this subsection, let us start to understand iterators and loops in C# better.

In short, an iteration statement, also called a looping statement, controls the repeated execution of a single or compound statement (a block of statements).

We can use iteration statements in C# to achieve the repeated retrieval of each element within a collection in a controlled manner, or in other words, we iterate over a collection of objects.

We can also use that element iteration for other purposes, like console writing.

A basic example of an iteration through a pre-established collection would be the following `for` loop retrieving elements from an `array`.

```
int[] numbers = { 6, 5, 4, 3, 7, 8, 9, 1, 2 };
for (int i = 0; i < numbers.Length; i++)
{
    Console.Write(numbers[i]);
}
/*Output:654378912*/
```

As visible in the previous example, the basic anatomy of an iterator statement is its declaration, a value to keeping our iteration amount controlled, the set condition for the iteration to take place, and the method we use to reach that limit, followed by the body of the iteration where we can either return or make use in any way we need of the current iterated-on element.

In other words, we declare the iterator, when to stop, and what to do in each iteration.

As you have probably noticed, we have again made a unique distinction between loops and iterators, just as we did with arrays and collections. It is important to note that, although their iterating nature is shared, there are some distinct differences between what is commonly known as a loop and what is known as an iterator.

5.3.1 Loops and Iterators

Although misleadingly similar and both highly valued concepts in programming often used interchangeably, there is an essential difference between the two.

An iterator is an object that allows you to iterate over a data structure, such as a list or an array. On the other hand, a loop is a control flow construct that lets us execute a block of code multiple times.

There are several types of loops in C#, including `for` loops, `while` loops, and `do-while` loops. Each Type of loop has its syntax and semantics. For example, a `for` loop ideally requires us to specify the number of times it should run before entering the body of the

loop, whereas a while loop will keep executing its body until some condition evaluates to false.

In contrast, an iterator does not have any explicit termination condition—it will continue running until all elements in the data structure have been visited. Iterators also typically allow us to modify their behavior.

Let us make use of some examples to clarify the difference between these two.

So, we know that an iterator is a method that runs once for each element in a collection. Using the foreach iterator, we can exemplify this:

```
foreach (var item in collection)
{
    // do something with 'item' here
}
```

This behavior is advantageous if this kind of complete run of a collection is needed. On the other hand, loops do not just run through an entire collection unless indirectly specified. Loops are mainly used if we want to control the iteration amount like in the following example, where we specified the `while` to loop for all elements of a collection until the end of or ten iterations is reached:

```
int i = 0;
while (i < collection.Count && i < 10)
{
    Console.WriteLine(collection[i]);
    i++; // Don't forget to increment 'i'!
}
```

Note that for the while loop, we had to explicitly create an end condition, incrementing a value until it reaches one of the determined termination points, so either until `collection.Count` is reached or until ten is reached.

This step is crucially important with loops because, in the case of not doing so or incorrectly doing so, we will create what is known as an “infinite loop.” These neverending loops will just run until the end of time, or until our system crashes, which will happen.

Both iterators and loops have their use cases, so it is essential to know when to use each one. As a rule of thumb, if we are working with a collection and do not need any special control over how many times the code runs, then an iterator is probably what we want.

5.3.2 Types of Iterators and Loops in C#

Now that we know the difference between iterators and loops and when to use what, we lastly need to learn what tools we have available for both. Luckily, this comprehensive list stays fairly short, easing the task of making a decision.

As per the official documentation, C# includes the following iteration statements: (Microsoft Corporation, 2022) (direct link <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements>).

- The for statement executes its body while a specified Boolean expression evaluates to true.
- The foreach statement enumerates the elements of a collection and executes its body for each element of the collection.
- The do statement conditionally executes its body one or more times.
- The while statement conditionally executes its body zero or more times.

Let us go over them in a bit more detail.

The for Statement

The for statement, probably the most popular because of its clarity and existence within most programming languages, is a type of loop that will repeat until the specified condition returns false.

We use the previous example again:

```
int[] numbers = { 6, 5, 4, 3, 7, 8, 9, 1, 2 };
for (int i = 0; i < numbers.Length; i++)
{
    Console.Write(numbers[i]);
}
/*Output:654378912*/
```

We see that it is composed, as mentioned before, of a value to keep our iteration amount controlled, the set condition for the iteration to take place, and the method we use to reach that limit, followed by the body of the iteration. These elements are the initializer, the condition, the iterator, and the body.

The initializer declares and initializes an integer counter variable used to determine the termination condition.

```
int i = 0;
```

The condition determines if the next iteration will be executed since that will only happen if it results in true. In our example, the condition states that as long as our initializer variable is less than the length of our numbers collection, it will result in true. This conditional example is a simple way to simulate the “iterate each collection element” behavior of iterators.

```
i < numbers.Length;
```

The last element within defines the iterator, responsible for executing the following action after a true condition. This element is typically used to increase the number value of the initializer variable to reach the determined end of the iteration loop sequence, but is not limited to it.

```
i++
```

Finally, the loop's body is only executed if the previous condition returns true. It is typically used to perform an action on the given current iteration through a loop, in our case, simply displaying the current element.

```
Console.WriteLine(numbers[i]);
```

The contents of a for statement typically will present these elements but do not have to. Technically this allows for a for statement declared like this:

```
int[] numbers = { 6, 5, 4, 3, 7, 8, 9, 1, 2 };
for (int i = 0; i < numbers.Length;)
{
    Console.WriteLine(numbers[i]);
    i++;
}
/*Output:654378912*/
```

Although not typically done, this can be used in certain situations, even if it were just to ease the read of the iteration.

The Foreach Statement

The foreach statement is an iterator that executes a statement or block for each element in a collection. As we previously learned, this is typically used for instances where a complete iteration through a collection is desired, and there is no explicit need for iteration amount control.

Going back to our example for the for statement, we can try to use the foreach to achieve the same result.

```
int[] numbers = { 6, 5, 4, 3, 7, 8, 9, 1, 2 };
foreach(int number in numbers)
{
    Console.WriteLine(number);
}
/*Output:654378912*/
```

As we can see, we achieved the same output by simply using the foreach with an initializer, being our number variable.

```
int number
```

And the collection that we wish to iterate through, preceded by an “in” keyword,

```
in numbers
```

Finally, in the iterator body, we perform our actions. However, this time, note that we display our number variable. This change is due to the iterator not using our variable to determine the termination condition but to store the currently iterated over element from the given collection. This change means that our desired element will be stored within the number variable on each iteration.

```
Console.WriteLine(number);
```

This change does mean that this variable must be of the Type of elements contained within the collection. Nonetheless, we can always use the var keyword if that is not known.

NOTE As per the official documentation, beginning with C# 8.0, we can use the await¹⁰ foreach statement to consume an asynchronous stream of data, that is, the collection type that implements the `IAsyncEnumerable<T>` interface. Each loop iteration may be suspended while the next element is retrieved asynchronously. We will go through asynchronous programming in Chap. 7. The following example shows how to use the await foreach statement:

```
await foreach (var item in GenerateSequenceAsync())
{
    Console.WriteLine(item);
}
```

The While Statement

The while statement executes a block of statements while the specified condition continues returning true. As we noted before, loops like these can cause “infinite loops” if not properly implemented. In the following example, we can see a functioning implementation of a while loop.

```
int n = 0;
while (n < 5)
{
    Console.WriteLine(n);
    n++;
}
// Output:
// 01234
```

¹⁰The await operator suspends evaluation of the enclosing async method until the asynchronous operation represented by its operand completes.

As we can see, the variable `n` was initialized before the `while` declaration. A condition was set to loop until `n` was bigger than 5. And, within the `while` body, we `Console.WriteLine` the current value of `n` and increment its value by 1.

We probably have noticed that, fundamentally, a `while` loop is simply a deconstructed `for` loop. It includes an initializer, a condition, an iterator, and a body, just separated in their lines outside the conditional block.

Since a `while` loop states a specific condition, a `while` loop can execute zero times if that condition is never returned as true. This differs from our last Type of loop: the `Do` statement.

The Do Statement

The `do` statement, also known as the `do-while` loop, executes a statement or a block of statements while a specified Boolean expression evaluates to true. Since that condition is checked after the first execution of the body, the `do` statement will always execute at least once, even if that condition returns false on the first execution.

The following example shows the usage of the `do` statement:

```
int n = 0;
do
{
    Console.WriteLine(n);
    n++;
} while (n < 5);
// Output:
// 01234
```

As noted, the execution flow will enter the `do` body, execute its content, then continue to check the `while` condition. If that `while` condition returns true, it will repeat the `do` body execution.

When deciding between `foreach`, `for`, `while` loops, and `do while` loops, we should be able to find the correct candidate if we follow this simple set of guidelines:

- Use `foreach` for iterating over Collections.
- Use `for` when you know the precise number of iterations.
- Choose `while` when the continuation condition is clear and potentially complex.
- Use `do-while` when you want the loop to execute at least once, regardless of the condition, and then repeat based on the condition being true.

Additionally, `while` loops are also used as a means of controlling the flow of our programming, since we could control how often a piece of code is repeated depending on a certain given condition. We will see this in action in future chapters.

Finally, with this, we conclude the introductory part of this book. Let us go over everything we learned in the past chapters, including everything we saw during this last chapter, by going through our fifth project: Tic Tac Toe.

5.4 Tic Tac Toe

As in the previous chapters, we must start with a base by creating a new .NET Console App project. Simply remove everything prewritten and this should be what we get (Fig. 5.1).



Fig. 5.1 The beginning of our fifth project

Let us analyze what we want to achieve here to determine the next step.

5.4.1 The Project

We probably all know what Tic Tac Toe is about: a game with a nine-grid board where the first of two players to complete a row, column, or diagonal with either three Os or three Xs drawn in the spaces wins. This game seems simple at first, and as a concept, it continues to be pretty straightforward. However, when discussing developing this project programmatically, we face a few challenges (Fig. 5.2).



Fig. 5.2 A complete Tic Tac Toe game in console format

Everything seemingly simple, from the board to the win condition detection, will require most of what we covered in the previous chapters. We will utilize C# constructs like loops, arrays, conditional statements, and random numbers. Also, we will include System.Linq and the development of a simple AI.

Believe us when we say that this project will be a challenge.

5.4.2 Our Code

A typical Tic Tac Toe game will present a nine-grid table where the players can input their moves. In a console context, we will achieve this by using a sequence of characters representing the lines and a set of numbers representing the positions of the board. These numbers represent board positions that players will replace with their signatures (X or O).

The “gameplay” consists of each player marking the position they want their move to be at, intending to draw a three-character line. This procedure will be accomplished by a player inputting the corresponding number where they would like to place their signature. This input will override the number in the array with the player’s signature (X or O). On each input, the board is checked for a win condition. If it is not met, the next player gets a turn.

The concept so far is clear, so that is it for the introductory overview. Now, on to the programming. We are going to start by creating our board. This phase is the basis of the whole game. As with the previous project, we want to start by declaring a new class.

```
public class Program  
{  
}
```

Once set up, let us start with the first step to a Tic Tac Toe game, the board.

Let us define a character array to hold the nine positions of the board in our newly defined class. This array is the same array into which either X or O will be replaced during the gameplay.

```
char[] boardPositions = { '1', '2', '3', '4', '5', '6', '7', '8', '9' };
```

The array we just defined holds the player input but does not draw the board. Let us now draw our board based on the `boardPositions` array. Let us define a method called `DrawBoard` that draws a board filled with the previous positions that we can reuse each time a player inputs a new move.

```
public static void DrawBoard() {}
```

To get a board drawn on the console, we will draw a shape with underscores, pipes, and array positions. Something like this should do:

```
Console.WriteLine(" -----");  
Console.WriteLine(" |     |     |     |");  
Console.WriteLine(" | {0} | {1} | {2} |", boardPositions[0],  
    boardPositions[1], boardPositions[2]);  
Console.WriteLine(" |     |     |     |");  
Console.WriteLine(" -----");
```

Using string concatenation, we set up the position of each array element within a square build with several layers of `Console.WriteLine` methods. We probably already want to see a result for that. Remember that a static main method is required when using custom classes, so let us set that up real quick.

```
static void Main(string[] args)
{
}
```

Then within, we want to call our `DrawBoard` method. For that, we now have two options. If we can remember from the previous chapter, our `Main` method is a `Static` method, meaning that it will not be able to call non-static methods like the `DrawBoard` method. So we can specify the class instance we are referring to using the `new` keyword.

```
Program program = new Program();
```

And then continue by calling it using the referred instance.

```
program.DrawBoard();
```

This will already work correctly. Alternatively, we can opt to set everything to static, including the variables. Then this instance specification would not be needed anymore.

```
public static char[] boardPositions = { '1', '2', '3', '4', '5', '6',
'7', '8', '9' };

public static void DrawBoard()
{
    Console.WriteLine(" -----");
    Console.WriteLine(" |     |     |     |");
    Console.WriteLine(" | {0} | {1} | {2} |",
boardPositions[0], boardPositions[1], boardPositions[2]);
    Console.WriteLine(" |     |     |     |");
    Console.WriteLine(" -----");
}

static void Main(string[] args)
{
    DrawBoard();
}
```

Although both options are correct, generally, if not necessary, the use of static methods is not advised. Static methods have their advantages and drawbacks, and working with static methods will lead to issues we do not need for such a simple project, so we will opt for the former. Nevertheless, both will result in the same console application (Fig. 5.3).

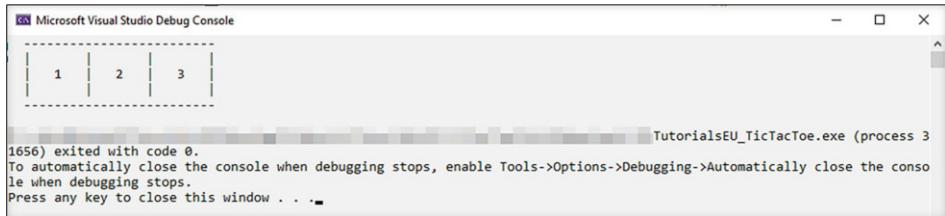


Fig. 5.3 Result of drawing our first table row

Which is a start. As we can see, the three positions we took from the array were successfully set on the board. Let us go back to the DrawBoard method and complete the rest of the board because the board looks a bit small at the moment.

So in the DrawBoard method, add the following lines.

```
public void DrawBoard()
{
    // Draw row 1 of the board
    Console.WriteLine("  -----");
    Console.WriteLine("  |      |      |      |");
    Console.WriteLine("  | {0} | {1} | {2} |",
        boardPositions[0], boardPositions[1], boardPositions[2]);
    Console.WriteLine("  |      |      |      |");
    Console.WriteLine("  -----");
    // Draw row 2 of the board
    Console.WriteLine("  |      |      |      |");
    Console.WriteLine("  | {0} | {1} | {2} |",
        boardPositions[3], boardPositions[4], boardPositions[5]);
    Console.WriteLine("  |      |      |      |");
    Console.WriteLine("  -----");
    // Draw row 3 of the board
    Console.WriteLine("  |      |      |      |");
    Console.WriteLine("  | {0} | {1} | {2} |",
        boardPositions[6], boardPositions[7], boardPositions[8]);
    Console.WriteLine("  |      |      |      |");
    Console.WriteLine("  -----");
}
```

And this is what we get if we rerun the project (Fig. 5.4).

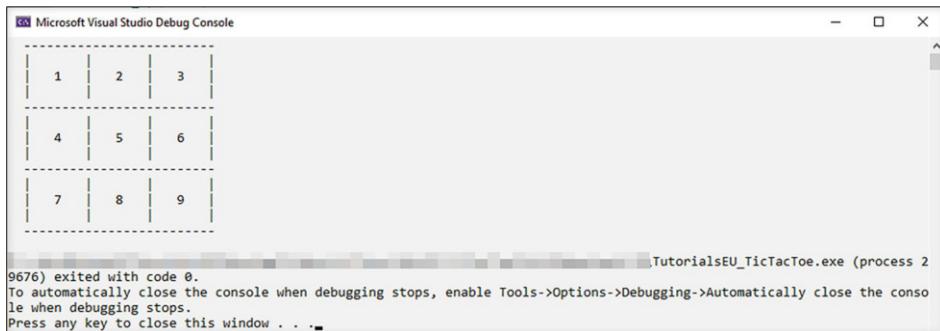


Fig. 5.4 A complete Tic Tac Toe game table drawn using the `Console.WriteLine` method

Now that we have the board drawn, we need to take care of the player input to start the gameplay. For that, we will need a `Play` method that will allow a player to select a board position to set it to their player character. So let us do that.

First, we must define a character variable to hold the player character.

```
public char playerCharacter = ' ';
```

Now, we will need to define the method itself. The method will take two parameters, an `int` for the player playing and an `int` for his input. We will show how to use that in a moment.

```
public void Play(int player, int input)
{
}
```

This method will be called specifying the calling player and his input, so one thing we can already do is determine the “sign” each player will use, so the X and O keys. Since we know the current player, we can set that up first. If it is the first player, it is an X. If it is the second player, an O.

```
public void Play(int player, int input)
{
    if (player == 1) playerCharacter = 'X';
    else if (player == 2) playerCharacter = 'O';
}
```

We have the character ready with that. We need to replace the array element position with the player-given input. For that, we will make use of a switch case. For each case, so the numbers 1 through 9, we replace the corresponding array position with the current player’s symbol.

```
switch (input)
{
    case 1: boardPositions[0] = playerCharacter; break;
    case 2: boardPositions[1] = playerCharacter; break;
    case 3: boardPositions[2] = playerCharacter; break;
    case 4: boardPositions[3] = playerCharacter; break;
    case 5: boardPositions[4] = playerCharacter; break;
    case 6: boardPositions[5] = playerCharacter; break;
    case 7: boardPositions[6] = playerCharacter; break;
    case 8: boardPositions[7] = playerCharacter; break;
    case 9: boardPositions[8] = playerCharacter; break;
}
```

Looks good. Let us ensure we have the Play method set up correctly by inputting a fake move in the Main method. Something like this should suffice:

```
program.Play(1, 5);
program.DrawBoard();
```

Just make sure to redraw the board as we did. This action should result in a fully drawn board with position five replaced with an X. Let us run the program and look at the result (Fig. 5.5).



Fig. 5.5 The fifth position of our table was replaced by the character of the first player

As expected, the correct field was replaced with the correct marker.

Our next task is to develop the methods that will check for the winning conditions. A player can win by arranging three of their characters in line horizontally, vertically, or diagonally.

First, let us explore how to win horizontally.

In concept, we need to check the following. Three of the player's characters must be in line in any of the three rows: positions {1,2,3}, {4,5,6}, and {7,8,9}. So in our method, we

need to check each row of three positions in our table to see if any of these trios contains exclusively one of the two players' symbols. If yes, we cue the winning scenario for that particular winner.

Let us start by defining the `HorizontalWin` method.

```
public void HorizontalWin()
{
}
```

In this method, we will want to check the table's three positions for each row to see if it contains either player character. To make that happen, we create an if condition to check just that for both characters. A straightforward way would be with an array of both possible characters and then iterating through for both. Let us go through how it works with its implementation.

First, create an array that contains both characters.

```
char[] playerCharacters = { 'X', 'O' };
```

Now, we want to iterate through both positions to make the conditional check twice. Since we want to always run through the entire array, we can use an iterator like `foreach`.

```
foreach (char playerCharacter in playerCharacters)
{
}
```

This iterator will run twice with the `char` variable `playerCharacter` being one of the two characters. Now, to set up the conditional statement, we can use that variable. We need an if statement that checks if a row has all three characters filled with a player's character, then move to the next row and do the same check. If any of the three have it, we can cue the win scenario.

As always, we recommend trying to achieve that before seeing our solution. Following this step-by-step guide will always make us learn new things, but going through trial and error ourselves is easily the best way to become a professional C# developer. So, we want to achieve the following action programmatically (Fig. 5.6):

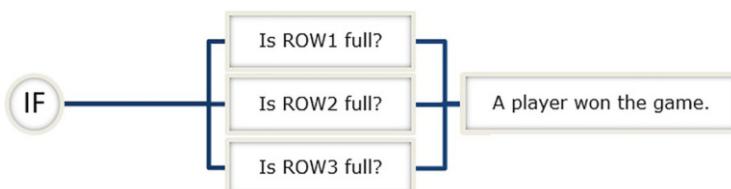


Fig. 5.6 A visualization of our conditional flow structure

So let us then try to do that. Achieving this task while using the given current player would look like the following.

```
foreach (char playerCharacter in playerCharacters)
{
    if (((boardPositions[0] == playerCharacter) && (boardPositions[1]
== playerCharacter) && (boardPositions[2] == playerCharacter))
        || ((boardPositions[3] == playerCharacter) && (boardPositions[4]
== playerCharacter) && (boardPositions[5] == playerCharacter))
        || ((boardPositions[6] == playerCharacter) && (boardPositions[7]
== playerCharacter) && (boardPositions[8] == playerCharacter)))
    {
    }
}
```

As we can see, we check on each row position if it contains the playerCharacter variable and one check for each position on one horizontal line. Then, once checked, with an or “||” symbol, we define an if condition that executes its containing block if any of the three conditional sets return true.

So as we explained in the previous chapter, with “&&” we define that both sides of that symbol have to return true, and with “||” we define that either one of the two can be for the if statement to execute its containing block.

In short, if

position 1 **AND** position 2 **AND** position 3 equals the playerCharacter **OR**
position 4 **AND** position 5 **AND** position 6 equals the playerCharacter **OR**
position 7 **AND** position 8 **AND** position 9 equals the playerCharacter

the if statement returns true.

This addition will already notice if any of the two players reached a win condition, so let us set up the final scenario.

Within the if statement, we need to clear the console, since a board is needed anymore and write a message depending on who achieved the horizontal win.

Something like this should look good, but we can naturally change it to whatever we want.

```
Console.Clear();
if (playerCharacter == 'X')
{
    Console.WriteLine("Congratulations Player 1.
    \nYou have achieved a horizontal win!");
}
```

```
else if (playerCharacter == 'O')
{
    Console.WriteLine("Congratulations Player 2.
        \nYou have achieved a horizontal win!");
}
break;
```

So if playerCharacter is X, we know that player 1 won, and we let him know he was the one who won. Same with player 2.

Also, remember to use `break` at the end to stop the continued iteration of our foreach iterator.

To test that this winning condition is working correctly, let us call our `Play` method thrice for player 1, setting positions 1, 2, and 3, after which we can call the “HorizontalWin” Method.

So inside our `Main` method, replace the last two lines we added to test with these following lines:

```
program.Play(1, 1);
program.Play(1, 2);
program.Play(1, 3);
program.HorizontalWin();
```

Run the program, and we get the following result (Fig. 5.7).



Fig. 5.7 It's a horizontal win!

The program correctly draws the board, “plays” the game, and detects a suitable horizontal win condition that triggers a win scenario for player 1, perfect as expected.

Alright, now that we have the horizontal win condition covered, we should continue with the two missing winning conditions, vertical and diagonal. So let us do so.

For the vertical win condition, three of the player’s characters must be in line in any of the three columns: positions {1,4,7}, {2,5,8} and {3,6,9}. Let us define the method “VerticalWin” to check if a user has this condition.

```
public void VerticalWin()
{
}
```

Now, this new method will be practically the same as the HorizontalWin method, meaning that we first get the two player characters, iterate through them, check if the specified condition is met, and if true, cue the win scenario.

For the HorizontalWin, we checked if any rows had characters exclusively from one player to detect a three-in-a-line win. As we may have guessed, we will check the columns this time.

So start this method in the same way we started the previous method.

```
public void VerticalWin()
{
    char[] playerCharacters = { 'X', 'O' };
    foreach (char playerCharacter in playerCharacters)
    {
    }
}
```

Now, check the columns for a match of three characters. The result should accomplish the following check. If

position 1 **AND** position 4 **AND** position 7 equals the playerCharacter **OR**
position 2 **AND** position 5 **AND** position 8 equals the playerCharacter **OR**
position 3 **AND** position 6 **AND** position 9 equals the playerCharacter

the if statement returns true.

We should be able to get to this result with the following if statement configuration.

```
if (((boardPositions[0] == playerCharacter) && (boardPositions[3] == playerCharacter) && (boardPositions[6] == playerCharacter))
    || ((boardPositions[1] == playerCharacter) && (boardPositions[4] == playerCharacter) && (boardPositions[7] == playerCharacter))
    || ((boardPositions[2] == playerCharacter) && (boardPositions[5] == playerCharacter) && (boardPositions[8] == playerCharacter)))
{
}
```

Finally, finish with the same content as the HorizontalWin method. So, clear the board, check who the winner was, congratulate them, and break the iteration.

```
Console.Clear();
if (playerCharacter == 'X')
{
    Console.WriteLine("Congratulations Player 1.
        \nYou have achieved a vertical win!");
}
```

```
else
{
    Console.WriteLine("Congratulations Player 2.
        \nYou have achieved a vertical win!");
}
break;
```

After that, we can test our new winning condition in the same way we tested our previous one, only this time playing for positions 1, 4, and 7, for example.

```
program.Play(1, 1);
program.Play(1, 4);
program.Play(1, 7);
program.VerticalWin();
```

Run the game to see the results (Fig. 5.8).



Fig. 5.8 It's a vertical win!

Perfect result as expected again. This is going pretty smoothly.

Now, we have one more winning condition left, the diagonal win. Three player characters must be in line in any of the two diagonal lanes: positions {1,5,9} and {3,5,7}.

Yes, you have guessed right. It will be the same, with the only difference being the winning check conditional statement. This time, we are checking for diagonal lines, so the result should check for a condition like this. If

position 1 **AND** position 5 **AND** position 9 equals the playerCharacter **OR**
position 3 **AND** position 5 **AND** position 7 equals the playerCharacter

the if statement returns true.

It seems simple enough. Let us set up this final win condition like last time. Define the method we are going to be using.

```
public void DiagonalWin()
{
}
```

Follow that with the player characters and the respective iteration.

```
char[] playerCharacters = { 'X', 'O' };
foreach (char playerCharacter in playerCharacters)
{
}
```

The conditional statement check for our aforementioned winning positions.

```
if (((boardPositions[0] == playerCharacter) && (boardPositions[4] == playerCharacter) && (boardPositions[8] == playerCharacter))
    || ((boardPositions[6] == playerCharacter) && (boardPositions[4] == playerCharacter) && (boardPositions[2] == playerCharacter)))
{
}
```

Finish that with the win scenario.

```
Console.Clear();
if (playerCharacter == 'X')
{
    Console.WriteLine("Congratulations Player 1.
        \nYou have achieved a diagonal win!");
}
else
{
    Console.WriteLine("Congratulations Player 2.
        \nYou have achieved a diagonal win!");
}
break;
```

Let us try that out by playing positions 1, 5, and 9.

```
program.Play(1, 1);
program.Play(1, 5);
program.Play(1, 9);
program.DiagonalWin();
```

Then run the program to see our results (Fig. 5.9).



Fig. 5.9 It's a diagonal win!

Exactly as we expected. With that, we managed to cover all possible winning conditions now. Furthermore, that is for both players as well. Nevertheless, we have yet to cover the outcome if only player characters cover the board positions and no win conditions are met. In other words, there is no draw scenario yet. Let us tackle that next by defining a method for checking if a draw scenario has been met.

```
public void Draw()  
{  
}
```

We want to check here if all positions on the board have player characters on them, or, if no position on the board has numbers, which would result in the same outcome.

Note that this method is built to execute after the win condition checks to avoid calling a false draw. If the last position to be filled results in a win condition, we don't want to trigger a draw, so always call this method at the end.

To implement this feature, in a manner consistent with our existing conditions, we need to verify that none of the positions on the board contain the numbers that are supposed to be in those positions. We can accomplish this verification by using a lengthy, yet straightforward, 'if' statement.

```
if (boardPositions[0] != '1' && boardPositions[1] != '2' &&  
    boardPositions[2] != '3' && boardPositions[3] != '4' &&  
    boardPositions[4] != '5' &&  
    boardPositions[5] != '6' && boardPositions[6] != '7' &&  
    boardPositions[7] != '8' && boardPositions[8] != '9')  
{  
}
```

As we can see, we are going through all the positions and checking if they do not contain their respective number. This, if used at the right moment in the execution flow, should correctly cover the draw condition.

Finally, add the draw scenario if the condition is met. A simple message should do.

```
Console.Clear();  
Console.WriteLine("It's a draw!");
```

There we go. We could also test this by simply playing all board positions in a no-win condition triggering way like the following.

```
program.Play(1, 1);  
program.Play(1, 3);  
program.Play(1, 5);  
program.Play(1, 8);
```

```
program.Play(1, 4);
program.Play(2, 6);
program.Play(2, 7);
program.Play(2, 9);
program.Play(2, 2);
program.Draw();
```

Then run our program to test for results (Fig. 5.10).

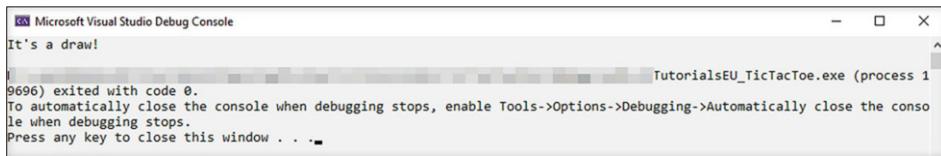


Fig. 5.10 It's a draw!

Perfect result! At this point, we have implemented every way to end our game, either by one player winning or a draw. Currently, once the game ends, we have to close the execution and rerun it, which is a bit annoying. Usually, games have some sort of “play again” or “reset” button. We definitely will need such a feature as well.

After each game, the board needs to be reset in case another game is going to be played. Therefore, we need to write logic to reset the board in case a player wins, or a draw happens.

So let us set this up. Start by defining a new method called ResetBoard.

```
public void ResetBoard()
{
}
```

This method will be responsible for reinitializing the board positions, filling them up again with their respective numeration, and then simply redrawing the board. Doing that would require us to start with the setup of an array containing our desired numbers. This array will be equal to the one made at the beginning of the project.

```
char[] ArrBoardInitialize = { '1', '2', '3', '4', '5', '6', '7',
'8', '9' };
```

What we want to do now is fill our boardPositions variable with these original numbers again. So let us do that.

```
boardPositions = ArrBoardInitialize;
```

Since we have been using the boardPositions array to determine the current situation of the game, by doing this, we essentially have already reset the game and are ready to start over. Moreover, starting over means calling the DrawBoard method again for a new clean board to be drawn.

```
DrawBoard();
```

That is it for the ResetBoard method itself. To be able to use it, we want to set up a way of toggling a board reset. A ReadKey method easily achieves a toggle before calling the ResetBoard method. We can also add a small message to let the player know that, like so:

```
Console.WriteLine("Please press any key to reset the game");
Console.ReadKey();
ResetBoard();
```

Now add these three lines to any end condition, so the Draw method, the HorizontalWin method, the VerticalWin method, and the DiagonalWin method, right at the end after informing the player of the outcome of the last played game. This is exemplified here with the Draw method.

```
public void Draw()
{
    if (boardPositions[0] != '1' &&
        boardPositions[1] != '2' &&
        boardPositions[2] != '3' &&
        boardPositions[3] != '4' &&
        boardPositions[5] != '6' &&
        boardPositions[6] != '7' &&
        boardPositions[7] != '8' &&
        boardPositions[8] != '9')
    {
        Console.Clear();
        Console.WriteLine("It's a draw!");
        Console.WriteLine("Please press any key to reset the game");
        Console.ReadKey();
        ResetBoard();
    }
}
```

If done correctly, we should get this result with any end condition we meet, like in this example where both players ended in a draw (Fig. 5.11).



Fig. 5.11 The reset function working as intended

Now, as soon as we press any key, the board will be reset to be ready for another round.

Congratulations on getting this far. We have now built all the building blocks for our game. Let us now put them together and make the game playable. Start with clearing up our Main method for everything except the new instance creation in case we opted not to set everything as static earlier, as we did.

```
static void Main(string[] args)
{
    Program program = new Program();
}
```

A classical Tic Tac Toe game is played in turns, starting with player 1 and then player 2. In each new game, the previous loser will start to get a fair advantage for the next round. We also need to validate player moves so already covered board positions cannot be selected again. Each time an invalid board position is input or the board position does not exist, we warn the user and ask for another input.

For that, we will use some new variables, starting with a variable that informs about the game's current state, specifically, if the game ended or is still running.

We want to modify this variable from everywhere in our code, since we need to set the game to end upon reaching a win scenario and to run again upon resetting the board.

So, at the very top, underneath the playerCharacter variable, set a new bool called gameEnded to false:

```
public bool gameEnded = false;
```

This variable needs to be set to false for the first round to start.

We need to set a few new variables in the Main method. We will need one for the currentTurn, one for the playerInput, and one bool for if the player has given a valid input, called validInputGiven.

```
Program program = new Program();
int currentTurn = 1;
int playerInput = 0;
bool validInputGiven = true;
```

These three will be responsible for storing the game management data.

Good, with that out of our way, let us dive into the logic.

DISCLAIMER We will be using while loops for the main game logic, meaning that infinite loops and therefore freezing and crashing computers are bound to happen unless we only run to test our software when we know that no infinite loops are present.

Ideally, only run to test your game whenever we do, so as to avoid any complications. With that said. We need a while loop that will run as long as the gameEnded variable stays false. That should look something like this:

```
while (!program.gameEnded)
{
}
```

Within this while loop all the game logic will happen, starting with the first draw of our board. Since we already know how, let us just do so.

```
while (!program.gameEnded)
{
    program.DrawBoard();
}
```

Great. Now, within this while loop, we will need another while loop. This time, it will be responsible for giving the game logic a valid player input to use in our case testing.

The way that will work is by using our validInputGiven bool, and as long as it is set to false, repeating the loop until a valid answer is given and the bool is set to true.

So for that to properly work, we first need to set the bool to false just before we enter the loop.

```
program.DrawBoard();
validInputGiven = false;
```

Right after, set up our second while loop to loop as long as the validInputGiven bool is set to false.

```
while (!validInputGiven)
{
}
```

Currently, we have two while loops that will infinitely loop, so do not run the program.

Let us first cover how to stop the first loop from infinitely looping. Since the idea is to stop it on the game end, we can use the win conditions to stop the game and the reset method to restart it. So, first, in the HorizontalWin method, add the following.

```
else if (playerCharacter == 'O')
{
    Console.WriteLine("Congratulations Player 2.
        \nYou have achieved a horizontal win! ");
}
gameEnded = true;
Console.WriteLine("Please press any key to reset the game");
Console.ReadKey();
ResetBoard();
```

This addition will, upon reaching the win condition, before asking for a game reset, end the game by setting the gameEnded variable to true. This action effectively will stop the first while loop from continuing.

Let us do the same for the VerticalWin, DiagonalWin, and Draw methods. Remember to include that variable assignment in all four win conditions.

After that, we want to start the game again once we enter the ResetBoard method since the game needs to start over, so let us do just that.

```
public void ResetBoard()
{
    char[] ArrBoardInitialize = { '1', '2', '3', '4', '5', '6', '7',
    '8', '9' };
    boardPositions = ArrBoardInitialize;
    DrawBoard();
    gameEnded = false;
}
```

Alright, we got our first while loop covered. However, the second while loop is not yet taken care of, so it is still not safe to run our program. Let us work on that next.

So, we want to continuously ask the player for input within this second loop until they type a valid input. We can start by prompting our players with the next required move so they have some sort of feedback. A Console.WriteLine method like this should suffice:

```
Console.WriteLine("\nReady Player {0}: It's your move!", currentTurn);
```

Next, we want to read the player input line. If we remember correctly from previous chapters, for this, we will make use of the Console.ReadLine method.

A simple way of accomplishing this task would be the following.

```
playerInput = Convert.ToInt32(Console.ReadLine());
```

This line should grab whatever the user entered into the console and save it to our playerInput. However, this simple solution is prone to error, specifically type error.

PlayerInput is a variable we declared as an int, so a type mismatch would already occur there. Furthermore, in that line, we are converting the return value of ReadLine to int since we already know that ReadLine will return a string. Now, if the user inputs something other than an int, like a letter, this convert method will return an error, which will break our program.

Luckily, we can use this error to our advantage and try to catch it, allowing the execution flow to freely continue and giving us the option to warn the user of its error.

For that, we can use the try-catch statement.

To avoid unhandled exceptions, C# uses the try-catch statement. Simply put, the try block contains the guarded code that may cause the exception. The block is executed until an exception is thrown or it is completed successfully, and when an exception is thrown, the common language runtime (CLR) looks for the catch statement that handles this exception.

Let us look at an unhandled exception compared to a handled exception.

```
// Risk of unhandled exception
playerInput = Convert.ToInt32(Console.ReadLine());  
  
// No risk for a unhandled exception
try
{
    playerInput = Convert.ToInt32(Console.ReadLine());
}
catch
{
    Console.WriteLine("Please enter a number.
        \nGive it another Try...");
    continue;
}
```

As we see in the example, if the user inputs the letter “a,” the first ReadLine would provoke an unhandled error, potentially breaking our program, while the second ReadLine is, if incorrectly input, caught by the catch statement. This even gives us the option to warn the user and use the continue keyword not to permit the while loop to execute the current block further. Instead, it instructs the while loop to continue with the next iteration, which means asking for input again.

So make sure to use the try-catch method for our Tic Tac Toe project. That way, we will not have left any error uncontrolled.

Alright, now that we have the players’ input, there is one more thing we need to check: whether the input value is valid in the context of our game board. So, not only if it is within the nine positions of our board, but also if it is an available position since we do not want a player to overwrite another player’s position.

This task we can accomplish by simply using an if statement tree. In other words, a bunch of ifs and elseifs. Let us go over the first one and continue from there.

```
if ((playerInput == 1) && (program.boardPositions[0] == '1'))
    validInputGiven = true;
```

What we are doing here is, in the conditional statement, making two checks. First, if the player input is equal to our first position on the board, so if the player intends to place his character at the first position. Then, if that first position is still available or already occupied, for our case, a simple check to see if a "1" is already enough, since if it were not available, that position would contain either an "X" or an "O."

If both results are true, we finally set the validInputGiven variable to true, stopping the while loop from looping, and the execution flow continues.

Great, now we have to do this eight more times. Have fun!

```
if ((playerInput == 1) && (program.boardPositions[0] == '1'))
    validInputGiven = true;
else if ((playerInput == 2) && (program.boardPositions[1] == '2'))
    validInputGiven = true;
else if ((playerInput == 3) && (program.boardPositions[2] == '3'))
    validInputGiven = true;
else if ((playerInput == 4) && (program.boardPositions[3] == '4'))
    validInputGiven = true;
else if ((playerInput == 5) && (program.boardPositions[4] == '5'))
    validInputGiven = true;
else if ((playerInput == 6) && (program.boardPositions[5] == '6'))
    validInputGiven = true;
else if ((playerInput == 7) && (program.boardPositions[6] == '7'))
    validInputGiven = true;
else if ((playerInput == 8) && (program.boardPositions[7] == '8'))
    validInputGiven = true;
else if ((playerInput == 9) && (program.boardPositions[8] == '9'))
    validInputGiven = true;
```

Alright, this should cover all positions. We are just missing one thing, the case if the player inputs a valid number but not one available or on the board. So we need an `else` statement at the end.

```
else if ((playerInput == 9) && (program.boardPositions[8] == '9'))
    validInputGiven = true;
else
{
    Console.WriteLine("Please choose a valid and available
        board position. \nGive it another Try...");
}
```

There we go. If the `else` statement is entered, the user will know what he did wrong, and the while loop will repeat, asking for new input.

So now we got our valid input. The next step is to play the game. We can start by handling the turns for the players, so if it is player 1's turn, we start the game for him and change the current player to the next one.

```
if (currentTurn == 2)
{
    program.Play(currentTurn, playerInput);
    currentTurn = 1;
}
else if (currentTurn == 1)
{
    program.Play(currentTurn, playerInput);
    currentTurn = 2;
}
```

That will do it. That makes sense, right? Now we need to check for win conditions, and we are pretty much done.

```
program.HorizontalWin();
program.VerticalWin();
program.DiagonalWin();
program.Draw();
```

Now we can run the game, invite over a friend, and play a round of Tic Tac Toe together (Fig. 5.12).



Fig. 5.12 Players 1 and Player 2 working. The game can now be played in its entirety by both players

You can also pretend you are playing against someone else by playing against yourself! It is fun! I have been doing it throughout my entire childhood!

Alright, we might need to create a second player. Let us get into the bonus section and develop a small AI (Artificial Intelligence) to play against.

The way we will develop it is that the computer will select a random position from the available board positions to place their character. Not exactly the sharpest AI, but we should be able to play against it at least.

To accomplish this, we will need first to check who the current player is in the try-catch statement we created earlier.

```
try
{
    if (currentTurn == 2)
    {
        // AI player
    }
    else
    {
        playerInput = Convert.ToInt32(Console.ReadLine());
    }
}
catch
{
    Console.WriteLine("Please enter a number.
    \nGive it another Try...");
    continue;
}
```

Now we can set input for the AI instead of the player typing in one. First, we need to determine our options for available boardPositions. For this, we can make use of our System.Linq namespace, specifically, the Where and Select methods. Let us see how.

```
int[] options = program.boardPositions.Where(x => x != 'X' && x != 'O').Select(x => x - '0').ToArray();
```

Looks funny. However, do not fret. This line is not doing anything other than just selecting the boardPositions that don't have an 'X' or an 'O' with the Where method, then converting those positions to integers with the Select method. Finally, those values get converted to an array using the ToArray method. More straightforward than it looks but perfect for our purposes since we now have an array of currently valid inputs.

Now we need to select a random value of that array and set it to the playerInput variable. After that, the code can just continue as normal.

```
Random random = new Random();
int randomIndex = random.Next(options.Length);
playerInput = options[randomIndex];
```

That is it! Run the project and see the results (Fig. 5.13).



Fig. 5.13 The final result of our project including enemy AI

The computer will automatically play as the second player. Naturally, we could always make our AI smarter by knowing the game rules and trying to win, but we will leave that to you to try to accomplish. Make sure to share with us if you did it, though!

Have fun playing!

With this final project, we have reached the end of Part 1 of this book. We are officially introduced to the C# language and ready to be treated like professionals from now on. With the following increasingly advanced chapters, we will be using everything we learned throughout the first chapters and every topic we went over in this chapter, including but not limited to arrays, collections, loops, iterators, the Linq namespace, and the try-catch statement. Let's get ready for the next chapter.

5.4.3 Source Code

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_TicTacToe

5.5 Summary

This chapter taught us:

- An array is a structure representing a fixed-length ordered collection of values or objects of the same Type.
- A C# array variable is declared similarly to a non-array variable, adding square brackets ([]) after the type specifier to denote it as an array.
- We can find other types of arrays than single-dimensional arrays, like multi-dimensional arrays and jagged arrays.

- The collection class is a set of specialized classes purposed for storage and accessibility of data within the System.Collections namespace.
- We can find three types of collections in the System.Collections namespace: Generic classes, Concurrent classes, and Non-Generic classes.
- Non-generic collections, although useful in some situations, are not recommended to be used within a production environment.
- Collections are of dynamic size, meaning their length can be modified at runtime for addition or subtraction. While with arrays, their limitation is that they are fixed in size to the initialized length.
- Both Arrays and Collections have properties and methods to read and write to them.
- Additionally, shared among both data element collector types is the System.Linq namespace. This system component provides a set of methods for query capabilities directly into the C# language.
- An iteration statement, also called a looping statement, controls the repeated execution of a block of statements.
- An iterator is an object that allows you to iterate over a data structure, such as a list or an array. On the other hand, a loop is a control flow construct that lets us execute a block of code multiple times.
- C# Iteration statements are composed of the for statement, the foreach statement, the do statement, and the while statement.
- To avoid unhandled exceptions, C# uses the try-catch statement. Simply put, the try block contains the guarded code that may cause the exception, and when an exception is thrown, the common language runtime (CLR) looks for the catch statement that handles this exception.

References

- ankita_saini. (2021, February 4). Iterators in C#. Retrieved May 30, 2022 from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/iterators-in-c-sharp/](https://www.geeksforgeeks.org/iterators-in-c-sharp/)
- Chand, M. (2020, July 6). What is SQL. Retrieved June 27, 2022 from [www.c-sharpcorner.com: https://www.c-sharpcorner.com/article/what-is-sql/](https://www.c-sharpcorner.com/article/what-is-sql/)
- Codecademy. (n.d., n.d. n.d.). Learn C#: Arrays and Loops. Retrieved May 22, 2022 from [www.codecademy.com: https://www.codecademy.com/learn/learn-c-sharp/modules/learn-csharp-arrays-and-loops/cheatsheet](https://www.codecademy.com/learn/learn-c-sharp/modules/learn-csharp-arrays-and-loops/cheatsheet)
- csharp.net-tutorials.com. (n.d., n.d. n.d.). Introduction to XML with C#. Retrieved June 27, 2022 from [csharp.net-tutorials.com: https://csharp.net-tutorials.com/xml/introduction/](https://csharp.net-tutorials.com/xml/introduction/)
- Ershad, G. M. (2015, November 14). Thread Safety In C#. Retrieved June 27, 2022 from [www.c-sharpcorner.com: https://www.c-sharpcorner.com/UploadFile/1c8574/thread-safety369/#:~:text=So%2C%20Thread%20safety%20is%20a,without%20the%20breaking%20of%20functionalities](https://www.c-sharpcorner.com/UploadFile/1c8574/thread-safety369/#:~:text=So%2C%20Thread%20safety%20is%20a,without%20the%20breaking%20of%20functionalities)
- GeeksForGeeks. (2021, October 26). C# | Arrays. Retrieved May 25, 2022 from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/c-sharp-arrays/](https://www.geeksforgeeks.org/c-sharp-arrays/)

- Jallepalli, K. (2021, November 25). Collections in C#. Retrieved May 25, 2022 from www.c-sharpcorner.com: https://www.c-sharpcorner.com/UploadFile/736bf5/collection-in-C-Sharp/
- Jones, M. (n.d., n.d. n.d.). Arrays and Collections - C# in Simple Terms. Retrieved May 25, 2022 from <exceptionnotfound.net: https://exceptionnotfound.net/csharp-in-simple-terms-13-arrays-and-collections/>
- Microsoft Corporation. (2021, October 1). Arrays (C# Programming Guide). Retrieved May 25, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/>
- Microsoft Corporation. (2021, September 15). Collections (C#). Retrieved May 25, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/collections>
- Microsoft Corporation. (2021, September 15). Covariance and Contravariance (C#). Retrieved June 9, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/>
- Microsoft Corporation. (2021, September 15). Covariance and Contravariance (C#). Retrieved June 27, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/covariance-contravariance/>
- Microsoft Corporation. (2021, September 15). Jagged Arrays (C# Programming Guide). Retrieved May 25, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/jagged-arrays>
- Microsoft Corporation. (2021, September 15). Multi-dimensional Arrays (C# Programming Guide). Retrieved May 25, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/multidimensional-arrays>
- Microsoft Corporation. (2021, November 3). Single-Dimensional Arrays (C# Programming Guide). Retrieved May 25, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/arrays/single-dimensional-arrays>
- Microsoft Corporation. (2021, December 16). What is Xamarin? Retrieved June 27, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/xamarin/get-started/what-is-xamarin>
- Microsoft Corporation. (2022, March 11). await operator (C# reference). Retrieved June 27, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await>
- Microsoft Corporation. (2022, March 18). Generic classes and methods. Retrieved June 27, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/types/generics>
- Microsoft Corporation. (2022, January 25). Iteration statements (C# reference). Retrieved May 30, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements>
- Microsoft Corporation. (2022, February 18). Language Integrated Query (LINQ) (C#). Retrieved May 25, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/>
- Microsoft Corporation. (2022, February 18). Query expression basics. Retrieved June 27, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/linq/query-expression-basics>
- Microsoft Corporation. (2022, April 30). What is Windows Presentation Foundation (WPF)? Retrieved June 27, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/visualstudio/designers/getting-started-with-wpf?view=vs-2022>
- Microsoft Corporation. (n.d., n.d. n.d.). Array Class. Retrieved May 25, 2022 from <docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.array?view=net-6.0>

- Microsoft Corporation. (n.d., n.d. n.d.). Array Class. Retrieved May 25, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.array?view=net-6.0
- Microsoft Corporation. (n.d., n.d. n.d.). Collection<T> Class. Retrieved May 25, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.collections.objectmodel.collection-1?view=net-6.0#properties
- Microsoft Corporation. (n.d., n.d. n.d.). Console Class. Retrieved April 27, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.console?view=net-6.0
- Microsoft Corporation. (n.d., n.d. n.d.). Enumerable Class. Retrieved May 25, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable?view=net-6.0
- Microsoft Corporation. (n.d., n.d. n.d.). Enumerable Class. Retrieved June 17, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable?view=net-6.0
- Microsoft Corporation. (n.d., n.d. n.d.). IEnumerable Interface. Retrieved June 27, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.collections.ienumerable?view=net-6.0
- Microsoft Corporation. (n.d., n.d. n.d.). IList Interface. Retrieved June 27, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.collections.ilist?view=net-6.0
- Microsoft Corporation. (n.d., n.d. n.d.). System.Collections.Concurrent Namespace. Retrieved June 14, 2022 from docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.collections.concurrent?view=net-6.0
- Singh, A. (2022, February 4). Shallow Copy and Deep Copy Using C#. Retrieved June 27, 2022 from www.c-sharpcorner.com: https://www.c-sharpcorner.com/UploadFile/56fb14/shallow-copy-and-deep-copy-of-instance-using-C-Sharp/
- Tutorials Point. (n.d., n.d. n.d.). C# - Multithreading. Retrieved June 27, 2022 from www.tutorialspoint.com: https://www.tutorialspoint.com/csharp/csharp_multithreading.htm



C# File System

6

This Chapter Covers

- Working with The System.IO namespace
- Developing with System.IO through Operating Systems
- Learning the secrets of the String data type
- Applying advanced string manipulation techniques
- Developing the classic “Hangman” application

We have finally reached the second part of this book with this chapter. This means that we are officially starting the advanced topics.

So far, we have learned everything about object-oriented programming we need. From this point onwards, we will get into projects that either use advanced real-life topics or are complete usable projects fit for a portfolio that, with a bit more polish, can be a finished product.

And this chapter is just the beginning, with the following chapters introducing asynchronous programming, hashing, and APIS.

Without further delay, what is this chapter’s project about? The project consists of a well-known game popularly known as “Hangman.” A random word gets displayed, and character by character, we try to guess it. This project will pick random words from a file existing within the user’s hard drive and display the unguessed word on the console. The player then tries to guess it letter by letter. This chapter will teach us about the integrated file class from the System.IO namespace and demonstrate its most common use cases. Additionally, it will serve as a proper introduction to string manipulation.

In this iteration, we will be working entirely within the main class, but it should be known that this technique is not necessary, as we have already learned in the previous chapter. However, it will show us alternative ways of reaching the same goal.

So let us not hesitate and start with our main topic for this chapter, the C# System.IO namespace.

6.1 The System.IO Namespace

Until now, we only had to work with data stored within our code, like those included in variables, arrays, and lists. However, that is not always the case; in some scenarios, data must be fetched from a given outside source. These sources could encompass anything from simple text files to complex databases.

To start covering these advanced cases, we can first introduce ourselves to file handling. Luckily, C# provides us with a dedicated namespace, System.IO.

6.1.1 System.IO Definition

The System.IO namespace is a dedicated namespace that contains types that allow us to read and write to files and data streams, implementing classes that permit advanced file and directory support.

These classes are Directory, DirectoryInfo, File, FileInfo, FileStream, Stream, StreamReader, and StreamWriter, among others. Each of them also provides a set of valuable methods.

NOTE The System.IO namespace should not be confused with the System namespace since, although similar in naming, the latter is the one containing general but fundamental base classes like those used in the previous chapters. These classes define commonly used value and reference datatypes, events and event handlers, interfaces, attributes, and processing exceptions, including, but not limited to, classes such as Array, Buffer, Console, Convert, Math, Object, Random, and String. Most of those are already pretty familiar to us.

To start this off, let us check out a simple file read-write application. In the following example, we are first creating a new path variable, where the path to our desired file is stored. Then we check if that file exists. If not, we create it and write two lines into it. After that, we read its contents and display them to the console.

Listing 6.1 This Code Exemplifies a Typical Program Using the SystemIO Namespace

```
public static void Main()
{
    var path = Path.Combine(@"..\..\..\text1.txt");
    if (!File.Exists(path))
```

```
{  
    Console.WriteLine("File not existent, creating now...");  
    using (StreamWriter sw = File.CreateText(path))  
    {  
        sw.WriteLine("Hello");  
        sw.WriteLine(" World!");  
    }  
}  
using (StreamReader sr = File.OpenText(path))  
{  
    string s;  
    while ((s = sr.ReadLine()) != null)  
    {  
        Console.Write(s);  
    }  
}
```

After the first execution, since the searched-for file is not present, the final output will include the `Console.WriteLine("File not existent, creating now...");` line, since a file had to be created and written to. For the second execution, however, the file is just read, and no additional line is written.

These are just a few of the classes available within the System.IO namespace, from pathing to stream handling. This namespace allows us to perform nearly any action we need to accomplish our goals. So let us go through those classes and discover their use cases and valuable methods.

6.1.2 System.IO Classes

Generally, the System.IO namespace should cover nearly every advanced case, meaning an exception to a rule or situation that requires additional or special handling when it comes to file handling. It includes classes ranging from the rather simple Path class, responsible for performing operations on String instances that contain file or directory path information, to the classes responsible for stream and buffer handling.

Although this list is rather long, a few classes stand out as some of the most widely used within the C# development space. These include but are not limited to the File class, the Path class, and the Directory class.

The System.IO File Class

Starting with the most widely used, the File class provides static methods for creating, copying, deleting, moving, and opening a single file and aids in creating FileStream objects.

The `File` class has many static methods that allow us to manipulate files in different ways. These methods are more efficient than the corresponding methods in the `FileInfo` class if we only need to perform one action. If we plan on reusing an object several times, in that case, we should consider using the `FileInfo` class's corresponding instance methods since these do not perform security checks and may not be necessary. This statement is true for most `System.IO` classes.

All `File` methods require the path to the file that we are manipulating. The path can refer to a file or a directory for members who accept a path. We will go more in-depth when explaining the `Path` class.

Furthermore, the `file` class allows for getting and setting file attributes or `DateTime` information related to a file's creation, access, and writing.

The number of available file class methods is pretty vast, but a few notable examples can further demonstrate this class's reach.

- `AppendText()` appends text at the end of an existing file.
- `Copy()` copies a file.
- `Create()` creates or overwrites a file.
- `Delete()` deletes a file.
- `Exists()` tests whether the file exists.
- `ReadAllText()` reads the contents of a file.
- `Replace()` replaces the contents of a file with the contents of another file.
- `WriteAllText()` creates a new file and writes the contents to it. If the file already exists, it will be overwritten.

As usual, we can find a complete list in the official Microsoft documentation (Microsoft Corporation, n.d.) (direct link <https://learn.microsoft.com/en-us/dotnet/api/system.io.file?view=net-6.0>).

In the following example, we write and read a simple “Hello World!” message from a file.

```
// A string variable that holds our text
string writeText = "Hello World!";
// A file is created if none is existent and our text is written
File.WriteAllText("filename.txt", writeText);
// The newly created file is read and stored in a new variable
string readText = File.ReadAllText("filename.txt");
// The read text is output
Console.WriteLine(readText);
```

In this example, we can see that we used a string as a direct path to the file for writing and reading. However, that is not the only way. There is a class solely responsible for this task, the `Path` class.

The System.IO Path Class

The Path class in C# provides cross-platform compatibility when performing operations on String instances that contain file or directory path information. A path is a string that locates a file or directory, but the format of a path is different depending on the platform. For example, some systems include a drive or volume letter at the beginning of the path while others do not. Additionally, the characters used to separate path elements and those that cannot be used in paths vary by platform. Therefore, the fields of the Path class and the behavior of some of its members are platform-dependent.

A path can contain either absolute or relative location information. An absolute path fully specifies a location: the file or directory in question can be uniquely identified regardless of the current working directory. On the other hand, a relative path only specifies a partial location: it is interpreted as relative to the current working directory when locating a file specified with that path.

```
string readTextAbsolute = File.ReadAllText(@"C:\Windows\filename.txt");
Console.WriteLine(readTextAbsolute);

string readTextRelative = File.ReadAllText("filename.txt");
Console.WriteLine(readTextRelative);
```

In the .NET Core framework, starting with version 1.1, support has been added for accessing files using so-called device names, which are file paths that begin with the prefix “`\\?`”. These device names allow access to files and directories that may not be accessible using regular file paths.

The device name prefix “`\\?`” is used to indicate that the path should bypass normal path parsing and should be passed directly to the file system API. This can be useful for accessing files or directories with long names, or for accessing files or directories that are located at deeper levels of the file system hierarchy than the normal 260-character limit for file paths.

For example, the device name “`\\?C:`” refers to the root directory of the C: drive and can be used to access files and directories located on that drive, regardless of the length of their names or the depth of their location in the file system hierarchy.

```
string filePath = @"\\?C:\\VeryLongDirectoryName\\File.txt";
```

NOTE Device names are not recommended for general use and should only be used in specific cases where normal file paths are not sufficient. They can be difficult to work with and can have unexpected results in some situations.

All the following relative paths are acceptable:

- “c:\\\\MyDir\\\\MyFile.txt”
- “c:\\\\MyDir”
- “MyDir\\\\MySubdir”
- “\\\\\\\\MyServer\\\\MyShare”

Although most methods of the Path class do not interact with the file system and do not verify the existence of the file specified by a path string, some members do validate the contents of a specified path string and throw an ArgumentException if the string contains characters that are not valid in path strings, as defined in the characters returned from the GetInvalidPathChars method.

For example, on Windows-based desktop platforms, invalid path characters might include a quote ("), less than (<), greater than (>), pipe (|), backspace (\b), null (\0), and Unicode characters 16 through 18 and 20 through 25.

```
string firstPath = Path.Combine(@"C:\Windows\filename.txt");
if (firstPath.IndexOfAny(Path.GetInvalidPathChars()) < 0)
{
    Console.WriteLine("The path " + firstPath + " is valid");
}
else
{
    Console.WriteLine("The path " + firstPath + " is invalid");
}
```

The previous example will output “The path C:\\Windows\\filename.txt is invalid” since it contains one invalid path character, the “|” character.

The members of the Path class give us the ability to do things such as determining if a file name extension is part of a path or combining two strings into one path name. Since all members of the Path class are static, we do not need an instance of a path to use them.

Many different operations can be carried out on strings containing directory path information. Some examples are gone over in more depth below.

- **HasExtension()**: We can check whether a string object that contains file path or directory information has an extension or not using the HasExtension() method provided by the Path class. Here, a path is a string object that contains a file path or directory information.
- **IsPathRooted()**. We can check whether a string object that contains file path or directory information is rooted using the IsPathRooted() method of the Path class. A rooted path is a file path fixed to a specific drive or UNC path; it contrasts with a path relative to the current drive or working directory. Here, a path is a string object that contains a file path or directory information.

- **GetFullPath():** To get the full path of a file, we can use the GetFullPath() method. Here, a path is a string object that contains a file path or directory information.
- **GetTempPath():** To get the location of temporary files, GetTempPath() is used.
- **GetTempFileName():** Get the temporary file name that is available for use.

Let us take a look at an example use of the Path class.

Listing 6.2 This Code Is Checking Whether the File Paths “pathWithExtension” and “pathWithoutExtension” Have File Extensions. If “pathWithExtension” Has an Extension, It Will Print a message Saying So. If “pathWithoutExtension” Does Not Have an Extension, It Will Print a Message Saying So. The Code Also Prints the Location of Temporary Files and Creates a Temporary File and Prints Its Name

```
string pathWithExtension = @"c:\temp\path.txt";
string pathWithoutExtension = @"c:\temp\path";
if (Path.HasExtension(pathWithExtension))
{
    Console.WriteLine("{0} has an extension.", pathWithExtension);
}
if (!Path.HasExtension(pathWithoutExtension))
{
    Console.WriteLine("{0} has no extension.", pathWithoutExtension);
}
Console.WriteLine("{0} is the location for temporary files.",
Path.GetTempPath());
Console.WriteLine("{0} is a file available for use.",
Path.GetTempFileName());
```

To work with directories, however, we need to look at the Directory class.

The System.IO Directory Class

The Directory class provides methods for the manipulation of directories and subdirectories. We can use the Directory class for typical operations such as copying, moving, renaming, creating, and deleting directories. Some example methods include:

- CreateDirectory() creates all directories and subdirectories in the specified path unless they already exist.
- Delete() deletes an empty directory from a specified path.
- GetCurrentDirectory() gets the current working directory of the application.
- SetCreationTime() sets the date and time when the specified file or directory was created.

The following example demonstrates how to move a directory and all its files to a new directory. The original directory no longer exists after it has been moved.

```
string originalDirectory = @"C:\origin";
string finalDirectory = @"C:\final";
try
{
    Directory.Move(originalDirectory, finalDirectory);
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

This example will finally output “Could not find a part of the path ‘C:\origin.’”

Naturally, these three classes are not all of those available in the System.IO namespace, but it should be apparent by now that a complete breakdown of every single class would be way out of this book’s scope. Nonetheless, we can include an additional set of five classes notable for their usefulness in the following table (Table 6.1).

Table 6.1 This table describes a selected list of SystemIO classes

SystemIO class	Description
FileStream	Provides a Stream for a file, supporting both synchronous and asynchronous read and write operations
MemoryStream	Creates a stream whose backing store is memory
StreamReader	Implements a TextReader that reads characters from a byte stream in a particular encoding
StreamWriter	Implements a TextWriter for writing characters to a stream in a particular encoding
TextReader	Represents a reader that can read a sequential series of characters

We will get a better understanding of these classes throughout their usage in this chapter’s project, as well as in future projects.

Other than classes, System.IO also counts with structs, enums, and delegates.

6.1.3 System.IO Structs, Enums, and Delegates

Although this chapter’s main focus is on the system.io classes, leaving out structs, enums, and delegates would not be correct. We will not go in-depth with these but let us follow a simple overview of what these are.

System.IO Structs

System.IO structs are a value type datatype that represents data structures. They can contain a parameterized constructor, static constructor, constants, fields, methods, properties, indexers, operators, events, and nested types. For example, the `WaitForChangedResult` struct contains information about the type of change that occurred (e.g., a file was created, modified, or deleted), the name of the file or directory that was changed, and the old name of the file or directory (if it was renamed).

System.io Enums

System.io enums are a set of named constants whose underlying type is an Integer type. If no underlying type is explicitly declared, `Int32` is used. For example, we can take the `FileAccess` enum. The `FileAccess` enum is a type that represents the possible ways in which a file can be accessed. It has the following members:

- Read: Specifies read access to the file. Data can be read from the file, but the file cannot be written to.
- Write: Specifies write access to the file. Data can be written to the file, but the file cannot be read from.
- ReadWrite: Specifies read and write access to the file. Data can be both read from and written to the file.

System.io Delegates

System.io delegates are a data structure that refers to a static method or a class instance and an instance method of that class. For example, we can see the `FileSystemEventHandler`. It is used to specify a method that will be called whenever a file or directory in the file system is changed, created, or deleted.

In case any of these catch your interest, do not hesitate to check them out on the documentation page. We will mostly not use them, so diving deeper would not make much sense, but their use cases are significant, and depending on our app, they can be highly valuable.

So, at this point, it should be pretty clear that C# is a Microsoft corporation product, but we mentioned at the beginning of this book that using C# in other operating systems other than Windows is possible. Let us go over doing so for the file system.

6.1.4 System.IO for Linux and macOS

When using the file system in a cross-platform context, we may find incompatibility issues when trying to path the files directory, especially if hard coding the path is necessary for our app. Usually, using the forward-slash (/) character, we can avoid these issues since it is the only recognized directory separator on Unix Systems and is the `AltDirectorySeparatorChar` on Windows. Alternatively, we can use string concatenation to

retrieve the path separator at runtime using the DirectorySeparatorChar method in the Path class, like so:

```
separator = Path.DirectorySeparatorChar;
path = $"{separator}users{separator}user1{separator}";
```

Another way to avoid errors and directory incompatibilities is by avoiding specifying the directory separator altogether by using path.Combine. In this example, we retrieved a file located at “..\test\file.txt.”

```
Path.Combine("../", "test", "file.txt");
```

However, one of the downsides of Path.Combine is its inability to resolve relative paths to absolute paths. Luckily, .NET and the Path class provide another method to accomplish this task, the GetFullPath method. In the following example, we retrieve the full path to the Combined file path we got earlier.

```
Path.GetFullPath(Path.Combine("../", "test", "file.txt"));
```

Aside from the aforementioned, C# does not hinder our development process in terms of cross-platform file handling. Therefore, we have covered the basics of C# File handling and should now have a firmer grip on the System.IO namespace.

Let us continue with the following topic needed for this chapter’s project, string manipulation.

6.2 String Manipulation

In the previous examples, we used some string manipulation, for it is simply modifying the value of a string variable in its most basic form. However, in C#, there is much more to string manipulation than we have seen.

From declaring and initializing strings to composite strings, verbatim and substrings, we will cover everything we will be making use of during the development of the projects in this book.

Let us start with a curiosity that we never really notice, mainly because C# works around it in the background, which is strings’ immutability.

6.2.1 The Immutability of Strings

Strings being immutable means that they actually cannot be changed after they have been created. This seems weird since we commonly modify and alter our strings to adapt to our needs and notice no issues.

The behind-the-scenes process is that all string methods and C# operators that appear to modify a string are simply returning the results in a new string object.

In the following example, the string variable s1 gets concatenated with the value of s2 to form a new string stored in s1. However, since strings are immutable, the `+=` operator assigned a new string reference to s1, and the original string referenced by s1 was released for garbage collection after losing its reference.

```
string s1 = " First String ";
string s2 = " Second String ";
s1 += s2;
System.Console.WriteLine(s1);
// Output: " First String  Second String "
```

This unique behavior brings unique advantages and disadvantages. Like everything, it depends on our use case if those are relevant to us or not.

There are several advantages to using immutable strings, the most notable of which is that they are thread-safe. If we work with a multi-threaded system, there is no risk of concurrency issues because a new object is created in memory when modifying the string. Since immutable strings cannot be changed accidentally, we also do not need to worry about additional safety measures as we might for a mutable object.

This creates a unique edge case where references to strings can continue pointing to the original string after it has been modified, creating unexpected results.

```
string str1 = "Hello ";
string str2 = str1;
str1 += "World";
Console.WriteLine(str2);
//Output: Hello
```

However, performance costs can be a potential issue, too, as shown in the following example. Since the CLR is re-allocating the new string each time it is modified, we can quickly notice performance hits and garbage accumulation within our apps if enough modifications have to be made.

```
string str = "Welcome user 0! ";
Console.WriteLine(str);
for (int i = 1; i < 10; i++)
{
    str += " And user " + i + "!";
    Console.WriteLine(str);
}
```

At the end of the day, we will barely notice this, but it is always an advantage to keep it in mind while developing.

In previous examples, we used strings within quotation marks, which is the standard way of using these language features. Nevertheless, we have not yet seen all of its features and characteristics and have not even mentioned verbatim string literals yet.

It seems like we have yet to tie up several loose strings before we can continue. Let us do so then.

6.2.2 Verbatim and Quoted Strings

Quoted string literals start and end with a quote character. This string is the type we are most used to in our code. String literals, also called constants, can be of any basic datatype like an integer constant, a floating constant, a character constant, or a string literal. If they need to be used together with escape sequences, they have to be embedded within the literal, like so:

```
string columns = "Column 1\tColumn 2\tColumn 3";
Console.WriteLine(columns);
//Output: Column 1           Column 2           Column 3
```

On the other hand, verbatim string literals are a more convenient string multi-liner that can contain either backslash characters or embedded double quotes. These are written with the addition of, commonly named, an at “@” sign or address sign. Like the writing of a file path, as shown in the following example.

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\
```

Makes sense. However, we most probably do not yet know what escape sequences are.

6.2.3 String Escape Sequences

In C#, string escape sequences usually represent a new-line character, single quotation, as well as other characters commonly not usable within strings. By combining a backslash (\) and a letter or combination of digits we can determine various behavioral changes to our strings.

These can typically be scenarios like tab movements on terminals or the literal representation of non-printing characters.

With examples like the new-line sequence “\n” or the horizontal-tab “\t”, we can find five commonly found escape sequences for the C# language in the following table (Table 6.2).

Table 6.2 This table explains what each string escape sequence represents

String escape sequence	Represents
\a	Bell (alert)
\b	Backspace
\\\	Backslash
\r	Carriage return
\"	Double quotation marks

So, to print a string that contains a new-line, the escape character “\n” is used. Let us see how it would work in practice.

```
string strNL = "Hello\nWorld";
Console.WriteLine(strNL);
/*OUTPUT
Hello
World*/
```

Then, if we want to string file paths, we use the Backslash “\\” escape characters.

```
string strBS = "C:\\Test\\file.txt";
Console.WriteLine(strBS);
/*OUTPUT
C:\Test\file.txt*/
```

Other than these, C# naturally presents many more escape sequences. We should be able to get a better look at them over time while we continue implementing them in our applications.

Lastly, a common way to dynamically form strings in runtime, a task we had to accomplish several times during the development of the previous projects, is using format strings. Similar to adding a variable to a string using the addition sign, format strings provide efficient ways to concatenate our strings.

6.2.4 Format Strings

In simple words, a format string is a string that is set at runtime. By using braces ({}), we can essentially concatenate variables within strings in a more efficient way than our previous method of using the addition sign (+) between strings of strings.

We have probably seen an example like the following before.

```
string word = "World";
Console.WriteLine("Hello {0}!", word);
//Output: Hello World!
```

Mainly an alternative to standard concatenation, these methods can present essential support for some use and edge cases we might encounter, apart from readability improvements.

We can find two methods for format strings, string interpolation, which includes verbatims, and composite formatting.

String Interpolation

Available in C# 6.0 and later, interpolated strings are identified by the \$ special character and include interpolated expressions in braces.

```
string word = "World";
string interpolation = "This is string interpolation!";
Console.WriteLine($"Hello {word}! {interpolation}");
//Output: Hello World! This is string interpolation!
```

Composite Formatting

The String.Format utilizes placeholders in braces to create a format string. As we can see, the same method as our first test.

```
string word = "World";
string composite = "This is composite formatting!";
Console.WriteLine("Hello {0}! {1}", word, composite);
//Output: Hello World! This is composite formatting!
```

We have previously talked about how, technically, a string is just a sequence of characters. However, we have not yet talked about what that fact entails and how we can make use of that. For that, let us take a look at substrings.

6.2.5 Substrings

Any sequence of characters within a string is considered a substring. For example, in the string “Project,” “Pro” could be considered a substring. A substring can be a single character or more characters.

Substrings allow us to modify and control each character of a string in detail. To achieve this, we have got several methods at our disposal. We can use methods like the substring method to retrieve a specific substring from a string. By specifying at what index our desired substring starts and how long it is, we will get returned that substring as a new string.

```
string mainString = "Small C# Example Projects";
System.Console.WriteLine(mainString.Substring(6, 2));
// Output: "C#"
```

We can also make use of the Replace method to replace certain parts of a string with something else.

```
string mainString = "Small C# Example Projects";
System.Console.WriteLine(mainString.Replace("Small", "Huge"));
// Output: "Huge C# Example Projects"
```

Returning a specific index of a character contained within a string can be achieved using IndexOf.

```
string mainString = "Small C# Example Projects";
System.Console.WriteLine(mainString.IndexOf("#"));
// Output: "7"
```

Array notation, through the use of a loop, can be achieved with an index value in the following example.

```
string s5 = "Printing backwards";
for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

Generally, these will be all the topics we will cover in this chapter. As we can see, we are getting introduced to the more advanced topics in C# development mixed with some additional tips and tricks on general programming knowledge.

With our newfound knowledge about the System.IO namespace and advanced string manipulation abilities, it is time to tackle our sixth project so far, finally, the Guess the Word game, also known as “Hangman.”

Just without the hang. Or the man. Just word.

6.3 Guess the Word

Although not necessarily complex, this project will teach us a direct implementation of a file reading system and solidify our previous knowledge gained over the last five chapters. As in those previous chapters, we should always start with a clean slate. An empty .NET 6 Console project with the program.cs file open and cleaned out, ready to start writing the code.

However, first, let us analyze what we want to achieve here to determine the next step.

6.3.1 The Project

The Guess the Word game, also known by its more popular name, Hangman, is a game where an unknown word gets shown in a format where only the number of individual letters is shown. The player then has to one by one guess letters to hopefully uncover some of the letters contained within that word. The goal is to guess the word before the available number of guesses runs out.

Its conceptual simplicity will emphasize its rather complex mechanics. Mechanics that will show us in a practical manner everything we have learned so far, including this chapter's topics, like file handling and string manipulation.

Let us dive into this chapter's project and explore the development of our Guess the Word game.

6.3.2 Our Code

As usual, we must start by creating a class Program with a static Main method. We previously mentioned that we intend to maintain everything within this main method that the project in question permits and present an alternative to our previous approach.

So this will be our starting point and where our entire code base will be contained, just the following lines.

```
class Program
{
    public static void Main(string[] args)
    {

    }
}
```

From here, we must now determine what we want to start with. Our intention for the beginning of this game is to ask the player if the word to be guessed should be randomly selected or if the custom word is to be specified by the player, like in a two-player scenario.

So let us start with a prompt to receive an initial answer to work with.

```
Console.WriteLine("Welcome to 'Guess the word!'"
    "\nDo you want to type your own word to guess?");
Console.WriteLine("Type 'Y' if you want to type your own word
    or 'N' if you want a randomly selected word");
string randomOrNot = Console.ReadLine();
```

Here we welcome the player and ask how to proceed with the word to be guessed. A simple “Y” or “N” will work for us here.

Then we read the following input given by the user and save it within “randomOrNot.” In the following figure, we will show what it should look like (Fig. 6.1).

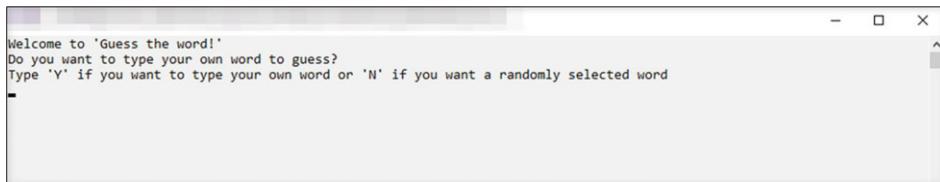


Fig. 6.1 We are prompting the player with a Y/N question to decide the play mode for this game

Here we will encounter our first potential error if we read for a “Y” and the user types “y,” which will most likely happen. Once we run a check, this response will not trigger our check since virtually anything related to strings is case-sensitive.¹ This issue can be fixed in two ways: we check for both “Y” and “y,” or turn whatever input is given to uppercase. The latter would allow us to work with uppercase letters without the need to double our logic.

```
string randomOrNot = Console.ReadLine();  
randomOrNot = randomOrNot.ToUpper();
```

Before we can use this response, we will need to ensure that the response is just “Y” or “N.” Right now, nothing stops the user from responding with an “A” or “B.” A while loop checking if the given response is “valid” for our purposes should do the trick.

First, populate a bool with a true if the response is either one of the wanted responses or a false if not. This should be doable with an if statement. So create a new bool variable.

```
bool isValidResponse;
```

Then the if statement.

```
if (randomOrNot == "Y" || randomOrNot == "N")  
{  
    isValidResponse = true;  
}  
else  
{  
    isValidResponse = false;  
}
```

Then, depending on the result, we keep asking until we get the correct answer. So, while isValidResponse is false, we ask again while alerting the user, as follows.

¹Case sensitivity defines whether uppercase and lowercase letters are treated as distinct.

```
while (isValidResponse == false)
{
    Console.WriteLine("Please, only type 'Y' or 'N'");
    randomOrNot = Console.ReadLine();
    randomOrNot = randomOrNot.ToUpper();
    if (randomOrNot == "Y" || randomOrNot == "N")
    {
        isValidResponse = true;
    }
    else
    {
        isValidResponse = false;
    }
}
```

That should do the trick. Once past this, we should have the answer we need. Now, let us start the first step for our game to start, getting the word depending if it is using a word included in a file or one given by the user.

Let us start by clearing our console to start the game without the initial prompt.

```
Console.Clear();
```

Now, we can implement the logic, beginning with a “Y” response, so the player wants to give their own word to be guessed. This will be the more familiar process. We ask the user for a word, read it, check if it is only letters since numbers are not part of the game, and convert the letters to uppercase like before.

We create a new string variable.

```
string wordToGuess = null;
```

In an if statement, start by prompting the player.

```
if (randomOrNot == "Y")
{
    Console.WriteLine("Host, please enter a word
                      for the guest to guess: ");
    wordToGuess = Console.ReadLine();
}
```

Looks good. Now we need to find a way to check if every introduced character is a letter, and not anything else. Luckily C# has a character method, the IsLetter method, which will return true if the character is a letter.

And not only that, string also has a method that will easily go through all elements of a source sequence, and if it satisfies a preset condition, like our IsLetter method, it will return true, the All method.

Combining both, we can quickly check if every letter within a string is a letter, with a line that looks like this:

```
bool isOnlyLetters = wordToGuess.All(Char.IsLetter);
```

Once the result is stored in the isOnlyLetters variable, we can use this to check if the given word to guess is valid for our game. And like before, we loop the question until the response satisfies our validation checks.

```
while (isOnlyLetters == false || wordToGuess.Length == 0)
{
    Console.WriteLine("Please enter only letters.
        \nGive it another Try...");
    wordToGuess = Console.ReadLine();
    isOnlyLetters = wordToGuess.All(Char.IsLetter);
}
```

Then, we uppercase it, and it is ready to go.

```
wordToGuess = wordToGuess.ToUpper();
```

Perfect. Next, we need to tackle the “N” response. Here is where we will be working on a bit of file handling.

In an else if statement for if randomOrNot is “N,” we will have to start with defining the path we will have to use for the file that has to be read.

```
else if(randomOrNot == "N")
{
}
```

Defining the path can be done using the method we learned earlier, specifically, the Combine method. The intention is to access a file that would be stored at the root of the current project, in our case, on the “FileSystem” file. We can find the file we are looking for by right-clicking on the currently open tab and accessing “Open Containing Folder” (Fig. 6.2).



Fig. 6.2 We are opening the containing folder by right-clicking on the open tab

This folder contains our project on our machine. Since we will be using it to store our text file where the random words will be stored, we can easily access it using Path.Combine.

```
var path = Path.Combine("WordsToGuess.txt");
```

However, by default, this method will not use the root directory we have seen earlier. It will search for the referenced text file within the bin, Debug, and net6.0 folder.

```
// Defaults to ProjectName\bin\Debug\net6.0\WordsToGuess.txt  
var path = Path.Combine("WordsToGuess.txt");
```

We could place the file in that directory, but in our case, we want this to direct to the root. For that, we can use the following path.

```
// Directs to ProjectName\WordsToGuess.txt  
var path = Path.Combine(@"..\..\..\WordsToGuess.txt");
```

NOTE Although we could have simplified this process by adding our file to the project solution, we intended to use this project to get a deep understanding of the File system in C#, so these last few additional steps have hopefully helped with that.

This change should now make it possible to successfully find our file. At least it would if there was one. Let us now set up this text file next.

As previously mentioned, right-click on the tab > Open containing folder. In the file explorer of our machine, create a new .txt file, and call it **WordsToGuess**. This name can be changed to whatever we want. Remember to modify the path so as to direct to the new file name (Fig. 6.3).

📁 .vs	6/30/2022 2:03 PM	File folder
📁 bin	6/30/2022 2:03 PM	File folder
📁 obj	6/30/2022 2:03 PM	File folder
📄 FileSystem.csproj	6/30/2022 2:03 PM	C# Project file 1 KB
🔧 FileSystem.sln	6/30/2022 2:03 PM	Visual Studio Solu... 2 KB
📄 Program.cs	7/11/2022 5:08 PM	C# Source File 6 KB
📄 WordsToGuess.txt	6/30/2022 3:07 PM	Text Document 1 KB

Fig. 6.3 We created a new text file called WordsToGuess to store our words in

Now, open our new file, and make sure to fill it with whatever words we may want. Consider dividing each word into separate lines and for them only to contain letters. Like in our example, these are the words we have chosen.

Listing 6.3 The Content for the File We Will Be Using for Our Example. The Words Are Composed of Only Letters and Separated Into Different Lines

```
APPLE
PEAR
BANANA
GRAPE
BIRD
DOG
CAT
```

With our file ready and our code referencing it, we can continue to the “read and pick a random word” logic. Doing so will first require reading and storing each one of our words in an array.

Using the `ReadAllLines` method from the `File` class, we should be able to read every line separately and store them individually as elements in an array, like so:

```
string[] lines = File.ReadAllLines(path);
```

After this line, we have an array variable called `lines` that will contain each of our words from our file on separate elements. We can use that to select a random one from that list. Getting a random index between 0 and the length of our array should already be familiar to us, like this:

```
Random random = new Random();
int randomIndex = random.Next(lines.Length);
```

Now, we will need a variable to store our word in. Let us create a new variable at the beginning of our code, above our initial prompt.

```
public static void Main(string[] args)
{
    string wordToGuess = null;
    Console.WriteLine("Welcome to 'Guess the word!'"
        "\nDo you want to type your own word to guess?");
```

Back down where we were, we can set this variable to the randomly selected index within the `lines` array.

```
int randomIndex = random.Next(lines.Length);
wordToGuess = lines[randomIndex];
```

And to finish off our word fetch, convert the word to uppercase, just to make sure.

```
wordToGuess = wordToGuess.ToUpper();
```

Great, we can now close our else if, and we should have our word ready no matter what the player chooses to use.

Right after our current code, clear our console again, and prompt the user that the game is about to start.

```
Console.Clear();
Console.WriteLine("Game Start!");
```

So this is what our current game will look like after starting and successfully choosing one of the two options (Fig. 6.4).



```
int randomIndex = random.Next(lines.Length);
wordToGuess = lines[randomIndex];
wordToGuess = wordToGuess.ToUpper();

}
Console.Clear();
Console.WriteLine("Game Start!");

Microsoft Visual Studio Debug Console
Game Start!
FileSystem.exe (process 15848) exited with code
0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Fig. 6.4 After successfully implementing the logic for choosing the word to be guessed, this is our current result

Alright, now on to the game logic itself. The basic concept of how we will be setting up the logic is the following.

- We want to visualize the number of letters in our chosen word using underscores.
- We want to check every position of our word with each guess to see if that guess is contained.
- We must keep an eye on the number of lives a player has left and subtract one on every failed guess.
- With every guess match, we need to reveal that letter and show that to the player.
- We need to cue the win scenario if each letter has been guessed.
- Finally, we need to cue the lose scenario if the word has not been guessed yet and no more lives are left.

Then let us start by visualizing our word to be guessed.

To start, we will need to know the length of our word. As we already know, we can determine the length of a string by using the Length method, like so:

```
int wordLength = wordToGuess.Length;
```

Using this value, we can create a new array responsible for storing our word to be guessed in its progressive states with the predetermined length of our string.

```
char[] positionsToGuess = new char[wordLength];
```

Now we need to populate each position of our array with an underscore “_”. We can achieve that with a for loop.

```
for (int i = 0; i < positionsToGuess.Length; i++)
{
    positionsToGuess[i] = '-';
}
```

There we go. It is not yet printed, but we already got what we needed. To print our result, we should store the current state in a string responsible for receiving the changes made during the game. Let us first create that variable and store our array as a single string, using the Concat method.

```
string printProgress = String.Concat(positionsToGuess);
```

Now, use WriteLine to print it.

```
Console.WriteLine("Word to guess: " + printProgress);
```

Also, remove the previous “Game Start!” we printed earlier as it is no longer needed. These additions should result in the following (Fig. 6.5).



Fig. 6.5 Our chosen word to be guessed was correctly displayed hidden

That is the first step done, now on to the second step. We want to check every position of our word with each guess to see if that guess is contained.

Let us start by prompting the player to guess, getting the player's input, and, as usual, check if it is all letters, and if not, repeat. Then set the successful player input to all uppercase.

```
Console.WriteLine("\n\n\nGuess a letter: ");
string playerGuess = Console.ReadLine();
bool guessTest = playerGuess.All(Char.IsLetter);

while (guessTest == false || playerGuess.Length != 1)
{
    Console.WriteLine("Please enter only a single letter!");
    Console.Write("Guess a letter: ");
    playerGuess = Console.ReadLine();
    guessTest = playerGuess.All(Char.IsLetter);
}
playerGuess = playerGuess.ToUpper();
```

Notice that we used the escape sequence “\n” in our prompt. As we can probably remember, the used escape sequence will provide vertical space between the last line and the line to be printed. This results in the prompt printing a few lines lower to create a cleaner user interface. This is what it should look like (Fig. 6.6).

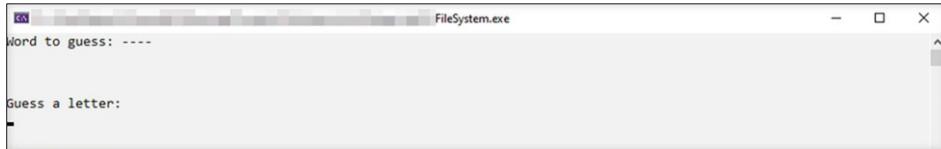


Fig. 6.6 The result of successfully prompting the user to guess a letter

Next, we convert the player guess to a char variable since we will need the guess to be of char type to compare it with the word's letters correctly.

```
char playerChar = Convert.ToChar(playerGuess);
```

Using our player input char, we can now proceed to loop through the word to be guessed to check if there is a match.

```
for (int i = 0; i < wordToGuessChars.Length; i++)
{
    if (wordToGuessChars[i] == playerChar)
    {
    }
}
```

If that if statement returns true, we need to set the positionToGuess array to include that already guessed letter and then let the code know that a letter has been found. Let us begin taking care of the first question. Setting the correct position of the positionToGuess array to include the now guessed letter is as simple as setting that position we are currently looping through to be replaced with the player input letter.

```
for (int i = 0; i < wordToGuessChars.Length; i++)
{
    if (wordToGuessChars[i] == playerChar)
    {
        positionsToGuess[i] = playerChar;
    }
}
```

Now, we need a bool to let the code know that we successfully guessed a letter. Let us create one just after the printProgress string declaration.

```
string printProgress = String.Concat(positionsToGuess);
bool letterFound = false;
```

Now, set it to true in our if statement.

```
for (int i = 0; i < wordToGuessChars.Length; i++)
{
    if (wordToGuessChars[i] == playerChar)
    {
        positionsToGuess[i] = playerChar;
        letterFound = true;
    }
}
```

With the letterFound bool set to true after successfully guessing a letter, we can display a text informing the user.

```
if (letterFound)
{
    Console.Clear();
    Console.WriteLine("Found letter {0}!", playerChar);
}
else
{
    Console.Clear();
    Console.WriteLine("No letter {0}!", playerChar);
}
```

As we can see, if a letter was found, we inform the user whether the input letter was found in our word or not, if no letter was found. This, on successful guess, will result in the following (Fig. 6.7).

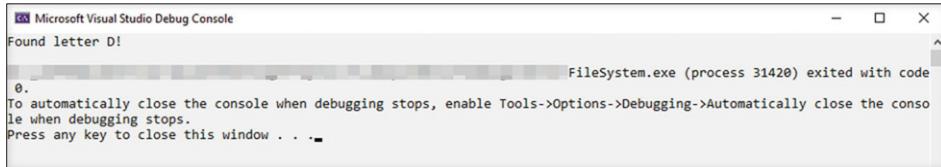


Fig. 6.7 Our logic is already working and displaying the correctly guessed letter

We already have our base game logic working. However, we also need to know if that letter was contained more than once since that is frequently the case. Currently, we are not checking for multiples, but doing so only requires three steps.

First, create a new variable under the letterFound bool variable called multiples.

```
string printProgress = String.Concat(positionsToGuess);
bool letterFound = false;
int multiples = 0;
```

Then, add to multiples each time a letter is found.

```
for (int i = 0; i < wordToGuessChars.Length; i++)
{
    if (wordToGuessChars[i] == playerChar)
    {
        positionsToGuess[i] = playerChar;
        letterFound = true;
        multiples++;
    }
}
```

Finally, we display it on successful guess.

```
if (letterFound)
{
    Console.Clear();
    Console.WriteLine("Found {0} letter {1}!", multiples, playerChar);
}
```

This covers step two, checking if the player input is contained in the word to be guessed.

Perfect, now we can proceed with step three. We must keep an eye on the number of lives a player has left and subtract one on every failed guess.

Doing so requires implementing our final game loop. The idea is that our code will ask for player input, checking if it is contained and displaying the result as long as the player has more than one life left. Doing so only requires a while loop to encompass the entire latter part of the code, starting from our printProgress declaration. First, create a new variable, the lives variable, right after the wordToGuessChars array declaration.

```
char[] wordToGuessChars = wordToGuess.ToCharArray();
int lives = 5;
```

Then, use that value to loop as long as it stays above 0.

```
while (lives > 0)
{
    string printProgress = String.Concat(positionsToGuess);
    bool letterFound = false;
```

Before executing, subtract one each time a letter is not guessed. So, if letterFound is false.

```
if (letterFound)
{
    Console.Clear();
    Console.WriteLine("Found {0} letter {1}!", multiples, playerChar);
}
else
{
    Console.Clear();
    Console.WriteLine("No letter {0}!", playerChar);
    lives--;
}
```

Next, we can display the number of lives remaining before prompting the user to guess a letter, thereby informing the user of how many lives he has left.

```
Console.WriteLine("You have {0} lives!", lives);
Console.WriteLine("Word to guess: " + printProgress);
```

That should make that system work. However, there is one small aspect we have not addressed yet. There is the possibility that the user forgets the letters he previously used to guess, and there is one rule that this “Guess the Word” game has, and that is that we should not lose lives if we are guessing the same wrong letter again.

Luckily, implementing that is just about adding our guesses to a list and checking in each round if we already guessed that letter.

So right after the declaration of the lives variable, declare a new list called lettersGuessed.

```
int lives = 5;
List<char> lettersGuessed = new List<char>();
```

Then, after storing our player input in a char, we want to encompass everything after that in an if statement, checking if that guess is already contained in our list. If not, then add that guess to the list.

```
char playerChar = Convert.ToChar(playerGuess);
if (lettersGuessed.Contains(playerChar) == false)
{
    lettersGuessed.Add(playerChar);
    for (int i = 0; i < wordToGuessChars.Length; i++)
    {
```

If it is, we want to clear our console and inform the user of the already used guess.

```
Console.WriteLine("No letter {0}!", playerChar);
    lives--;
}
else
{
    Console.Clear();
    Console.WriteLine("You already guessed {0}!", playerChar);
}
```

And this is everything we will need for our main game loop. If we were to run the game now, we would get the following result (Fig. 6.8).

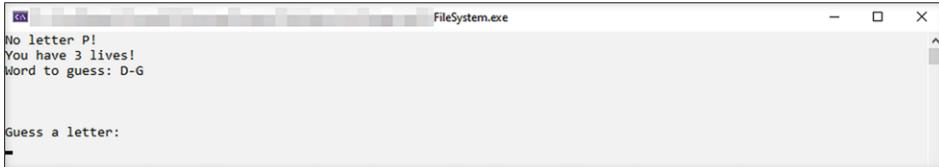


Fig. 6.8 After implementing our main game loop, we got the life system working

That is step three done. Now, in step four, with every guess match, we need to reveal that letter and show that to the player. But, wait just for a second there, I believe we already achieved that.

Indeed, because we are including the print of the word in our game loop and already setting the correctly guessed letter in our positionsToGuess array, each time the game loops the updated word gets printed. Great, fewer steps left to reach our goal.

We successfully reached our final steps in this chapter's project development. Steps five and six will be easily covered by simply adding an in-loop if statement to check if the word to be guessed already resembles our initial word again, to stop the loop and tell the code that a win condition should be cued. So, we need to create our final variable, a bool called gameWon.

```
List<char> lettersGuessed = new List<char>();  
bool gameWon = false;
```

And set it to true if printProgress and wordToGuess match, right after our in-loop variables.

```
int multiples = 0;  
if (printProgress == wordToGuess)  
{  
    gameWon = true;  
    break;  
}  
Console.WriteLine("You have {0} lives!", lives);
```

Finally, clear the console, and prompt the player with either a win scenario or a lose scenario, depending on whether gameWon is set to true or false. This should go after our main while loop at the very end.

```
Console.Clear();  
if (gameWon)  
{  
    Console.WriteLine("The word was: {0}", wordToGuess);  
    Console.WriteLine("You Won!");  
}  
else  
{  
    Console.WriteLine("The word was: {0}", wordToGuess);  
    Console.WriteLine("You Lose...");  
}
```

Congratulations! To celebrate our success as professional C# developers, let us now play a game of Guess the Word! (Fig. 6.9).



Fig. 6.9 This is our final result after successfully completing the project for this chapter

As per the last few chapters, we can find the source code at the following link. Check it out if you get stuck or generally want to see the completed code done by us.

Other than that, we congratulate you for finishing the first chapter of Part II of this book, the first advanced chapter.

Now, in the next chapter, where we will start to tackle asynchronous programming in C#, it is going to get harder. A topic strongly requested in every industry, so a must for our programming portfolio. Let's get ready for the next chapter.

6.3.3 Source Code

Link to the project on [Github.com](https://github.com/tutorialseu/TutorialsEU_GuessTheWord):

https://github.com/tutorialseu/TutorialsEU_GuessTheWord

6.4 Summary

This chapter taught us:

- The System.IO namespace is a dedicated namespace that contains types that allow us to read and write to files and data streams.
- System.IO includes classes ranging from the rather simple Path class, responsible for performing operations on String instances that contain file or directory path information, to the classes responsible for stream and buffer handling.
- The File class provides static methods for creating, copying, deleting, moving, and opening a single file and aids in creating FileStream objects.
- The Path class in C# provides cross-platform compatibility when performing operations on String instances that contain file or directory path information.
- The Directory class provides methods for the manipulation of directories and subdirectories.
- System.IO structs are a value type datatype that represents data structures. It can contain a parameterized constructor, static constructor, constants, fields, methods, properties, indexers, operators, events, and nested types.
- System.io enums are a set of named constants whose underlying type is an integral type.

- System.io delegates are a data structure that refers to a static method or a class instance and an instance method of that class.
- Strings are immutable, meaning they cannot be changed after they have been created.
- Quoted string literals start and end with a quote character.
- Verbatim string literals are a more convenient string multi-liner that can contain either backslash characters or embedded double quotes.
- A format string is a string that is set at runtime. By using braces ({}), we can essentially concatenate variables within strings in a more efficient way than our previous method of using the addition sign (+) between strings of strings.
- We can find two methods for format strings, string interpolation, which includes verbatims, and composite formatting.
- Any sequence of characters within a string is considered a substring. For example, in the string “Project”, “Pro” could be considered a substring.

References

- bhuwanesh. (2022, January 26). C# Path Class – Basics Operations. Retrieved July 1, 2022 from [www.geeksforgeeks.org: https://www.geeksforgeeks.org/c-sharp-path-class-basics-operations/](https://www.geeksforgeeks.org/c-sharp-path-class-basics-operations/)
- CodeBuns. (n.d., n.d. n.d.). C# Escape Sequences. Retrieved July 1, 2022 from [codebuns.com: https://codebuns.com/csharp-basics/escape-sequences/](https://codebuns.com/csharp-basics/escape-sequences/)
- JavaTpoint. (n.d., n.d. n.d.). C# System.IO Namespace. Retrieved July 1, 2022 from [www.javatpoint.com: https://www.javatpoint.com/c-sharp-system-io](https://www.javatpoint.com/c-sharp-system-io)
- Microsoft Corporation. (2021, August 3). Escape Sequences. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/cpp/c-language/escape-sequences?view=msvc-170](https://docs.microsoft.com/en-us/cpp/c-language/escape-sequences?view=msvc-170)
- Microsoft Corporation. (2021, September 15). How to: Perform Basic String Manipulations in .NET. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/standard/base-types/basic-manipulations](https://docs.microsoft.com/en-us/dotnet/standard/base-types/basic-manipulations)
- Microsoft Corporation. (2022, June 21). Strings and string literals. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/strings/)
- Microsoft Corporation. (n.d., n.d. n.d.). Directory Class. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.io.directory?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.io.directory?view=net-6.0)
- Microsoft Corporation. (n.d., n.d. n.d.). File Class. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.io.file?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.io.file?view=net-6.0)
- Microsoft Corporation. (n.d., n.d. n.d.). Path Class. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.io.path?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.io.path?view=net-6.0)
- Microsoft Corporation. (n.d., n.d. n.d.). Path.Combine Method. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.io.path.combine?redirectedfrom=MSDN&view=net-6.0#System_IO_Path_Combine_System_String_System_String](https://docs.microsoft.com/en-us/dotnet/api/system.io.path.combine?redirectedfrom=MSDN&view=net-6.0#System_IO_Path_Combine_System_String_System_String)
- Microsoft Corporation. (n.d., n.d. n.d.). Path.DirectorySeparatorChar Field. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.io.path.directoryseparatorchar?redirectedfrom=MSDN&view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.io.path.directoryseparatorchar?redirectedfrom=MSDN&view=net-6.0)
- Microsoft Corporation. (n.d., n.d. n.d.). System.IO Namespace. Retrieved July 1, 2022 from [docs.microsoft: https://docs.microsoft.com/en-us/dotnet/api/system.io?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.io?view=net-6.0)

- Microsoft Corporation. (n.d., n.d. n.d.). System.IO.Enumeration Namespace. Retrieved July 1, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.io.enumeration?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.io.enumeration?view=net-6.0)
- W3Schools. (2019, December 24). C Sharp System.IO Namespace. Retrieved July 1, 2022 from [www.w3schools.blog: https://www.w3schools.blog/c-sharp-system-io-namespace](https://www.w3schools.blog/c-sharp-system-io-namespace)
- W3Schools. (n.d., n.d. n.d.). C# Files. Retrieved July 1, 2022 from [www.w3schools.com: https://www.w3schools.com/cs/cs_files.php](https://www.w3schools.com/cs/cs_files.php)
- Witspry Technologies. (n.d., n.d. n.d.). String manipulations. Retrieved July 1, 2022 from [witcad.com: https://witcad.com/course/csharp-basics/chapter/string-manipulations](https://witcad.com/course/csharp-basics/chapter/string-manipulations)



C# Async Operations

7

This Chapter Covers

- Learning the technology behind asynchronous programs
- Developing code to work asynchronously
- Comparing values through the use of pattern matching
- Using Records to define classes
- Developing a timed “Diffuse the Bomb” math project

Welcome to this chapter, and welcome to the first big challenge. While the previous file system chapter included the first official advanced project, it was just the introduction to the second part of this book. With this chapter, we enter the world of asynchronous programming. If we have ever heard about await and async, we have encountered an asynchronous program in its natural habitat. However, if we are not yet familiar with these, we will not know what we can do with them or what advantages they offer. So, it is for that reason that this chapter exists.

In this chapter’s project, we will use async operations and pattern matching to develop a project that will prompt the user with a timed, randomly generated math question. Using asynchronous operations, we will be able to run a timer while simultaneously running our main logic throughout the entire process, from prompting the question to receiving the answer and comparing results.

This project will teach us the correct implementation of asynchronous operations, reliable pattern matching, and using records properly to declare classes in as compact and brief a format as possible. So let us get straight into this seventh chapter of this book, starting with a small introduction. What really is a CPU, and how does asynchrony work on the CPU?

7.1 The Central Processing Unit

Any computing unit, meaning all technological devices capable of processing information, possesses some sort of processing unit. In short, the CPU is the hardware component responsible for processing and executing data and instructions. If a variable is created or a mathematical calculation is resolved, the CPU will be the circuitry that will make that happen.

We can visualize it using the warehouse worker analogy. If we, as the input providers, provide the warehouse worker with a package of data, the warehouse worker will know from us where to place that package within the warehouse. In this case, mirroring a computer's functionality, the warehouse itself is not able to process the package and place it where it belongs. It needs a worker to take the data and correctly process it. Similar to how a hard drive cannot process data independently but needs a processing unit to process for it.

7.1.1 How Does Asynchrony Work on the CPU

Imagine what would happen if several packages arrived to be processed simultaneously. The worker would have to process each package individually, making the next deliveries wait longer with each package arrival. Although we could make the worker work extra fast, comparable to how a faster CPU in terms of Hertz (Hz) (which is the number of clock cycles that the CPU can complete every second, commonly referred to as the clock speed) can accomplish given tasks faster, there comes a point where being able to move multiple packages simultaneously, even if slower, becomes the better option.

That is where asynchronous programs come in compared to synchronous programs. In a synchronous program, the worker has first to finish his given task of processing a package before he can process another, whereas in an asynchronous context, while one worker processes one package, a second worker can come in and take up the next package, and so on until the thread limit is reached. But wait, we still do not know what a thread is.

Alright, even though programming, in general, involves a large number of complex processes occurring in the background, which we could not thoroughly explain even within an infinitely thick book, we can continue using our warehouse worker analogy to really understand how asynchronous programming works and its involvement in threads, which are the sequential processes that a processor can perform, and differences from multi-threading or the use of more than one thread.

7.1.2 Asynchrony Versus Multi-threading

In its simplest form, take our warehouse situation. The worker has a long line of arriving packages that he needs to process. Our solution was to bring in a new, second worker. So

from an asynchronous point of view, the first worker can take one package and tell the second worker that instead of waiting, he can go ahead and also take a package at the same time, so both can do the same job simultaneously, doubling the processing speed of the newly arrived packages. Essentially, both workers work in different threads, so in a multi-threaded point of view, each one has to acknowledge the newly arrived package, register it, and place it on the shelf, in that order. So both run simultaneously, independent from each other, but on their own in an ordered, step-by-step way. We could see it as two synchronous processes within two separate asynchronous processes, each process on a separate thread within the CPU.

Thus, we can conclude from that analogy that multi-threading is about workers, and asynchrony is about tasks.

Good, enough analogies. We need some real-life examples of asynchronous programs. Well, look no further than the very tool we use daily for Web browsing. Ever opened a tab and entered a Web site, and because we knew it would take a while to load, we just decided to do something else on another tab while waiting? In a synchronous context, that would not be possible since we would need to wait for the first tab to finish loading before we could open another, so asynchrony is used. By leaving the first tab on another thread, we can answer the user's request to open another tab without having to wait for the previous process to finish executing, just like how our warehouse worker was able to pick up a second package while our first worker was still processing the first one. *Again back to the analogy!*

So with that, we hopefully cleared up a bit of the mystery behind asynchrony. We can now get back to our C# context and dive deeper into asynchronous programming within C#.

7.2 Asynchronous Programming in C#

Optimization should be one of the first things that comes to mind when it comes to best practices in programming. So then, giving our software the ability to run on several threads simultaneously, therefore indirectly improving performance, seems like an easy path to take.

We have probably heard about `async` and `await` and maybe have even seen one implemented or done so ourselves, but unless we have deeply studied the topic before, we will likely still be in the copy-paste phase as developers when it comes to asynchronous implementation. Looking up an easy implementation of an `async` code for our software and having it work on the first try can be easy enough, but we would stand clueless if that were not the case.

The .NET Framework provides us with various tools for asynchronous programming, including native functions, classes, and reserved keywords. With these at our disposal, we can develop asynchronous tasks and workflows into our projects as needed.

So even though simply hoping it will work, no questions asked, and no study needed, is an attractive thought we sometimes can get away with, it will not serve us well always.

Therefore, we are here to learn the ins and outs of this programming language we call C#, so then comes the question: ***Ok, what then is asynchronous programming all about?***

7.2.1 What Is Asynchronous Programming in C#

So far, we have learned that asynchronous programming essentially enables our software to run tasks that may take a substantial amount of time to complete while still staying responsive to continue executing other commands, so even if we have asked our software to load a heavy file, we can continue searching for the next file while we wait.

In C#, this would be presented with, as an example, a specific piece of code running to complete a given task, like asking for data from an external server while still allowing the user to continue browsing. As soon as that async task finishes, it will notify the main thread about the result.

Generally, it is these I/O-bound needs, so those where the time it takes to complete an execution depends mainly on the period spent waiting for a response or CPU-bound needs, so those that depend on waiting for the CPU to complete a heavy processing command given, that greatly benefit from asynchronous programming.

7.2.2 Common Use Cases and When to Use

Alright, great, this is better observed via an actual real-life example, so let us try the following, a somewhat important one for that matter (referenced from the book *The Hitchhiker's Guide to the Galaxy* by Douglas Adams).

Listing 7.1 A Simple, Multi-Million-Years-Taking Example of an Asynchronous Program That Calculates the Meaning of Life

```
using System;
using System.Threading.Tasks;

namespace CPUAsync
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("I'll ask something big, watch this, ");
            Console.WriteLine("what is the meaning of life?");
            var asyncTask = AsyncTask();
        }
    }
}
```

```
        Console.WriteLine("Alright, now we wait...");  
    }  
  
    static async Task<float> AsyncTask()  
{  
        Console.WriteLine("Starting Async task, calculating the  
            meaning of life, the universe, and everything...");  
        int result = await Task.Run(() => (1000 * 1000 * 80 / 500 *  
            70 + 68954 - 98 * 2) / 268303);  
        Console.WriteLine("Finished Async task, The answer to the  
            ultimate question of life, the universe and everything is  
            " + result);  
        return result;  
    }  
}  
}
```

See What We Are Doing Here? In the previous example, as we can see by the reasonably complex calculation in the `AsyncTask` method, we are in front of a CPU-bound example, meaning that, hypothetically speaking, if that calculation were to take around 7.5 million years to resolve, it would be of great help if that were to be calculated in an `async` function.

In our case, luckily, it takes just a few milliseconds, almost too negligible to really be considered a use case, but for our demonstrative purposes, it works very well to show the execution order of each line in our code. If we were to run the previous script, our result would look like this (Fig. 7.1).

Fig. 7.1 The answer to the ultimate question of life, the universe and everything

As we can see, it starts by writing our first `Console.WriteLine`, which we can find in our main function running on our main thread. Then, it calls the `async` function, which will immediately display the first `Console.WriteLine` contained within. After that, it will be set to run a calculation on a different thread. Using the `Task` class and the contained `Run` method, we can move the operation to a different thread to process it asynchronously. Then, notice how instead of continuing with the following `Console` method, it simply continues executing the main thread while the `async` method, now on a separate thread,

continues executing its code. It is after the operation finishes when finally the result is printed, long after the main thread finishes.

The following execution flow diagram should properly visualize the process (Fig. 7.2).

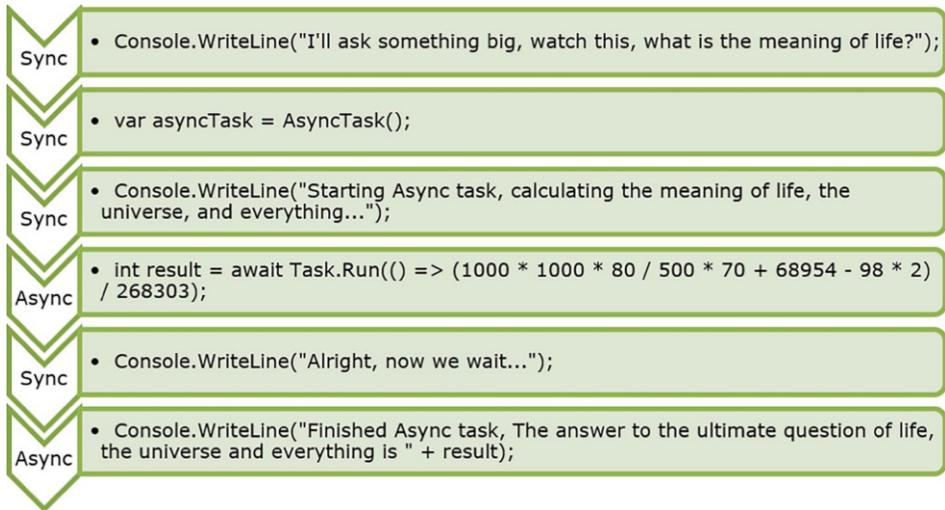


Fig. 7.2 Graphically visualizing the execution flow of our asynchronous program, to see how the async operation moved to the second thread since the main thread continued executing

We accomplished an asynchronous execution flow using one of the many methods in the Task class. We will go a bit more in-depth with the different types of methods that we have available for us later in this chapter. For now, though, let us look at another example. However, let us make it I/O-bound this time.

Listing 7.2 A Modification to the Previous Example I/O Bounding the Application This Time to Exemplify the Differences Between the Two

```

namespace IOAsync
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Read file using an async function");
            var asyncTask = ReadAsync();
            Console.WriteLine("Alright, now we wait...");
        }

        static async Task<string> ReadAsync()
        {

```

```
        Console.WriteLine("Starting async task, reading file...");  
        string filePath = "simple.txt";  
        string text = await File.ReadAllTextAsync(filePath);  
        Console.WriteLine("File was read, the contents were the  
        following: " + text);  
        return text;  
    }  
}  
}
```

In this I/O-bound scenario, only `async` and `await` are used. There is no need to use `Task.Run`.

Other than that, the `File` method changed to use its `async` counterpart, `ReadAllTextAsync`.

Exactly in parallel to our previous example, it starts printing our first `Console.WriteLine` within the `Main` method, then calls the `async` method, which prints its containing print method. After that, an `async` file read is performed; meanwhile, the main thread execution continues. Once the file is read, the next `Console.WriteLine` is printed, resulting in this final execution (Fig. 7.3).



Fig. 7.3 The final result of our I/O-bound asynchronous program example resulted in a similar execution flow to our previous example

Interestingly enough, depending on processing speed, our file reading can occur fast enough to be finished and the final result to be printed before our main thread print. To understand that, we can compare that to our warehouse worker example again.

It is as if the first worker, while processing an incoming package, asks a second worker to process the second package, and before the first worker completes his task, the second worker is already finished. In both results, asynchronous execution is still happening, but depending on execution speed, the final result can vary.

And this is one of the key complications when trying to implement `async` operations properly. Being isolated and, well, asynchronous by nature, our code cannot directly depend on the result, nor can we perfectly know what our final flow may look like since both are running in parallel.

Going back to the two guys at the warehouse, the first worker does not know, nor can he know, if the second worker will be there once he is back. So if the completion of his task

depends on the completion of the second worker's task, problems are bound to happen, unless **he waits**, of course.

Although seemingly counterproductive at first, since the entire point behind asynchrony is not having to wait for other tasks to complete, sometimes, in certain situations, this may be advantageous.

Take our first example, where the main thread's execution flow finishes before the async's thread execution, but **what if we want the data coming from that async thread in the first thread?** Well..., we `Wait()`.

Listing 7.3 Example Use of the `Wait()` method to Stop the Process Until the Asynchronous Task Is Done Executing

```
static void Main(string[] args)
{
    Console.Write("I'll ask something big, watch this, ");
    Console.WriteLine("what is the meaning of life?");
    var asyncTask = AsyncTask();
    Console.WriteLine("Alright, now we wait...");
    asyncTask.Wait();
    Console.WriteLine(asyncTask.Result + "? What does that mean...?");
}
```

That way, we could print the “Alright, now we wait...” message, then wait, and as soon as the async function completes, we continue with the given result on the main thread.

So, when do we use asynchronous programming then? **What would be some good use cases for this technology?** Take the following examples.

- **I/O-bound scenarios** like writing/reading to a file or database, making API calls, and calling devices like printers and scanners. These scenarios could be the following:
 - A social media app where a user's feed has to be updated while he scrolls, ideally without freezing the application while that fetch is done
 - A weather forecast app where the user must be able to check other data while the forecast for the following days is being loaded, commonly from a server
 - A server that must be developed with scaling in mind, where a single fetch may be fast, but a potentially high number of simultaneous fetches may happen
- **CPU-bound scenarios** like looping through many objects, complex calculations, processing large amounts of data, and similarly CPU-intensive operations. These scenarios could be the following:
 - A video streaming application that needs to run compression algorithms in real time
 - A videogame or graphic application where the processing of background calculations needs to be executed without interrupting the flow

- An application that includes a search engine whose search algorithm has to process a user's request while requesting to a server and letting the user potentially modify his search

These examples leave us with one question: If there are so many good and appropriate use cases for asynchronous programming, **why would it not be a good choice?**

Understanding that is simple through the following points.

- **Non-scalable I/O-bound services.** A request base that will not necessarily scale or databases that could not make use of the added asynchrony would simply bottleneck the process and not benefit from the added performance
- **A not SLA-bound service.** Meaning CPU-bound or I/O-bound services that, although initially presenting a need, do not have any strict time service-level agreement (SLA defines the level of service you expect from a vendor)
- **Dependant operations.** Services where computation results directly impact the continuing computations, making them dependent and non-viable for asynchrony. Similar to our previous example, where we had to wait for a response before being able to continue the main thread

Like we already mentioned before, there is more, vastly more, and we will try to cover as much of this as possible during this and the following chapters, but for now, let us deep dive into the asynchronous model and learn about everything involving this technology.

7.2.3 Overview of the Asynchronous Model

Regarding the asynchronous model, the Task and Task<T> objects supported by async and await are pretty much the core. Besides, generally, by just knowing the following, we know almost everything we need to know to be able to jumpstart our async apps:

- **In I/O-bound scenarios,** await starts an operation that returns a Task inside of the async method.
- **In CPU-bound scenarios,** await starts an operation that runs on a separate thread with the task.Run method.

However, learning about the various return types and methods available to us and a few more examples of more niche use cases should be able to solidify our understanding further. Now, async operations, and any programming for that matter, are wholly based on one person's ability to Google search the exact issue we have efficiently and proficiently copy-paste the first search result, so do not worry if any of this currently seems too

convoluted. Ideally, we want to start working on some applications that use this technology, then come back to this or quickly search on some forums for a more specific solution. That way, we should get a firm grasp on parallel programming as seamlessly as possible. This book is, before anything else, a compressed and practical guide for us to start understanding the various complex topics covered during the chapters, but we are the ones who will have to put in the effort of learning everything we need to be able to call ourselves professionals in a given field. **We WILL still try to provide you everything you need to know, though, so you better pay attention.**

For now, let us get into the details and wrap up our async theory. Starting with learning about the anatomy of a CPU-bound example.

**Listing 7.4 A Simple Example of a CPU-Bound Program That Uses Asynchrony.
The Commented Numbers Represent Points in the Code That Will Be Talked About
in the Following Paragraphs**

```
static /*1*/async /*2*/Task<float> AsyncTask()
{
    Console.WriteLine("Starting Async task, calculating the meaning
        of life, the universe, and everything...");
    int result = /*3*/await /*4*/Task.Run(() => (1000 * 1000 * 80 / 500 *
    70 + 68954 - 98 * 2) / 268303);
    Console.WriteLine("Finished Async task, The answer to the ultimate
        question of life, the universe and everything is " + result);
    return result;
}
```

- **One.** Using this example, we can directly see the first point, the declaration of an async function with the `async` keyword. Naturally, a requirement to declare async functions. Used to define that a method, lambda expression, or anonymous method is asynchronous.
- **Two.** Followed by the return type being the `Task<T>` return type. Async functions can have more return types than those commonly used `Task` and `Task<T>` types, like the following.
 - **Task.** Used to perform async operations but not return any value to the caller.
 - **Task<TResult>.** Like `Task`, but this time actively returning a value to the caller. Like `float` as in our case.
 - **Void.** If the declared async method is meant as an event handler. Although not recommended to be used, since the original caller will not get a notification of the completion of the async operation, it can be helpful if the application permits its usage.

- **ValueTask**. Similar to Task, but defined as a struct instead of a Task. Being a struct, it is especially performance-wise for short-term memory allocation-based applications. However, usually not relevant to most applications.
- **ValueTask<T>**. Like ValueTask but returning a value to the original caller.
- **Three**. The await. Usually, the point where a function starts becoming asynchronous since an async method runs synchronously until it reaches its first await expression. At this point, the process is suspended until the awaited task is complete. In the meantime, control returns to the caller of the method.
- **Four**. The Task instantiation. Although await can be followed and completed by other classes and C# components, we will be focusing specifically on the Task class. So let us continue with that now.

We know about the **Task class** as the representation of an asynchronous operation, whose work is typically performed on a separate thread pool, a pre-instantiated set of threads waiting to be assigned a task, asynchronously.

However, here we want to specifically talk about **Task instantiation** since it is what we have to learn for this chapter's project.

Like class instantiation, Task instantiation is the declaration of a specific instance of a Task, commonly to perform a particular operation asynchronously. Most commonly, though, it is done via the Run method. The Run method is a simple way to declare a piece of code to be run exclusively async, separated from the main thread. As seen in our example:

```
int result = await Task.Run(() => (1000 * 1000 * 80 / 500 * 70 + 68954 - 98 * 2) / 268303);
```

This operation is performed asynchronously due to it potentially taking “*7.5 million years*.” Simply by encapsulating that operation with a Task.Run method, we achieve what we are after.

We are more often than not going to use the Run method for our instantiation.

Speaking of Task methods, there are some other commonly used methods in the Task class. One of them is directly relevant to one of the tasks we have to achieve in this chapter's projects, the Delay method.

The **Delay method** is responsible for creating a task that will complete after a time delay. This time delay can be specified in milliseconds, for **Delay(Int32)**, in a specified time interval, for **Delay(TimeSpan)**, or after a specified number of milliseconds, but that can be cancellable, for **Delay(Int32, CancellationToken)** as well as **Delay(TimeSpan, CancellationToken)** if instead of milliseconds, a specified time interval is given.

Other than this bunch of methods, many more methods exist in the Task class. Although not necessary to this chapter's project, these methods are still necessary for other tasks we may encounter in future chapters.

Let us look at the following table, which briefly explains ten common methods we will make use of most of the time when working in an asynchronous task-based context. As always, we can find more information in the official documentation (direct link <https://learn.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-7.0>) (Table 7.1).

Table 7.1 This table provides a selected list of Task methods and their descriptions

Task method	Description
Start()	Starts the Task, scheduling it for execution to the current TaskScheduler
ContinueWith(Action<Task>)	Creates a continuation that executes asynchronously when the target Task completes
Wait()	Waits for the Task to complete execution
WhenAny(Task[])	Creates a task that will complete when any of the supplied tasks have completed
WhenAll(Task[])	Creates a task that will complete when all of the Task objects in an array have completed
RunSynchronously()	Runs the Task synchronously on the current TaskScheduler
GetAwaiter()	Gets an awaiter used to await this Task
Dispose()	Releases all resources used by the current instance of the Task class
FromException(Exception)	Creates a Task that has completed with a specified exception
WaitAsync(TimeSpan)	Gets a Task that will complete when this Task completes or when the specified timeout expires

We'll go into specifics once we reach that point during the project's development. This amount of theory about asynchrony is probably already enough. This is supposed to be a practical book, after all.

Asynchronous programming is a really vast field that can easily fill a book all by itself. Even if we tried to condense it to its barest of bones, we would not be able to fit this into this book, even as a subsection in a chapter. However, for our purposes, we will be able to cover everything needed. As always, if the interest in more information about asynchronous programming arises, either the official documentation or any other Tutorials.eu material should be able to enrich your learning path. Additionally, we suggest reading 'C# in Depth' by Jon Skeet, particularly the chapter on asynchrony, for a deeper understanding of this subject.

For now, it is in our best interest to continue with a different topic that we will need for our project: **Pattern matching**.

7.3 Pattern Matching

Pattern matching is a relatively new feature of C#. With its official introduction in C# 7.0 and since being improved upon greatly, it serves as an efficient and easy-to-use method for pattern matching on any data type and value extraction on expressions.

If we have ever developed a code including if or switch statements, we probably encountered a scenario where pattern matching could come in handy. Consider the following example.

Listing 7.5 Example Code That Contains Both an if Statement and Pattern Matching

```
string book = "C#ByExample";  
  
/* if-else */  
bool choice;  
if (book == "C#ByExample")  
{  
    choice = true;  
}  
else  
{  
    choice = false;  
}  
  
/* pattern-matching */  
bool choice = book switch  
{  
    "C#ByExample" => true,  
    _ => false,  
};  
  
if (choice)  
{  
    Console.Write(book + " book! I see! Good choice!");  
}  
else  
{  
    Console.Write(book + " book? Not so sure about it,  
        but probably a good choice!");  
}
```

Although generally ok and functionally correct, the final result feels somewhat muddy. Extending this scenario to a broader range of options will result in even less readability. At

the same time, using the pattern-matching example, we were able to simplify our if-else statement. What we just saw was the use of a switch expression for pattern matching. However, that is not the only way to use pattern matching and optimize our code.

So then, **what really is pattern matching?**

7.3.1 What Is Pattern Matching

Pattern matching is a method of checking whether a given sequence conforms to a specific pattern. Unlike pattern recognition, the process of looking for a similar or most likely pattern in a given set of data, which can often be imprecise, pattern matching requires an exact match. Either the sequence conforms to the pattern or does not.

Patterns generally take the form of either sequences or tree structures. Because C# is primarily an imperative language, meaning that its focus is on how to solve problems rather than what problems need to be solved, pattern matching is not as common as it is in functional languages. However, when used judiciously, it can make code more declarative and therefore easier to read and understand.

One way to perform a pattern match in C# is by using the *is* operator followed by the desired pattern. For example, to check whether a given object is of type int, you would write:

```
if (obj is int)
{
    // do something with the integer
}
```

If it is, the code within the if statement will be executed; otherwise, it will be skipped. Simple enough. This is getting interesting; let us dig a bit deeper now.

7.3.2 Pattern Matching in C#

All patterns are evaluated as either true or false, meaning that the expression either meets the requirements of the pattern or does not. That is a standard that continues in the context of C# and that we can easily observe in our previous example. So, relative to other statements, it is fairly similar. However, other than that, we can specify some other ways of using pattern matching to control code flow. Pattern matching uses three main ways to integrate its functionality, that is, *is expressions*, *switch statements*, and *switch expressions*.

- **Is expression.** Allows you to test an expression and declare a new variable to the result if it matches the specified pattern.

```
var newNum = 5;
bool newVarIsInt = newNum is int;
```

- **Switch statement.** Enables pattern matching directly within generic switch statements. The value of `person.Name` is assigned to `name`.

```
switch (person.Name)
{
    case string name:
        //Do something
        break;
    case null:
        //Do another thing
        break;
}
```

- **Switch expression.** Enables you to perform actions based on the first matching pattern for an expression.

```
object choice = book switch
{
    "C#ByExample" => true,
    _ => false,
};
```

In our case, we will focus on the “switch expression.” So let us learn about some of its advantages over other methods using a simple switch example.

One sought-after advantage is that pattern matching is both a control structure and a binding mechanism. Meaning that not only does it offer “if/else”-like functionality, but it also associates a value directly to a variable. Shortening the code to a much more compact result, as we saw in the previous example.

```
string book = "C#ByExample";

bool choice = book switch
{
    "C#ByExample" => true,
    _ => false,
};

if (choice)
{
    Console.WriteLine(book + " book! I see! Good choice!");
}
```

```

else
{
    Console.Write(book + " book? Not so sure about it,
        but probably a good choice!");
}

```

It also offers compatibility for compiler errors if some matches were left not covered, warning the developer of possible runtime issues before execution. We get this result by using the previous example and removing the “default” state (Fig. 7.4).



Fig. 7.4 The IDE warned the user of a possible non-exhaustive scenario. This means that a scenario where there are no matches can occur

And if our type accepts several types, for example, if it’s an *object*, a match can be done with more than one type. As seen in this example.

```

string book = "C#ByExample";

object choice = book switch
{
    "C#ByExample" => true,
    "C++ByExample" => "C++? Are you sure?",
    _ => false,
};

if (choice.Equals(true))
{
    Console.Write(book + " book! I see! Good choice!");
}
else
{
    Console.Write(book + " book? Not so sure about it,
        but probably a good choice!");
}

```

As we can see, the choice variable has been set to *object*. Now that it can accept several types, we can perform the matching process and check for our *true* value using an *Equals()* method.

Nevertheless, there is more to pattern matching than these few examples. Although we will not be covering every single use case, let us at least go over the various types of patterns that C# offers us.

7.3.3 Types of Patterns

There are many ways to define patterns that reflect the various operations supported by pattern matching. Depending on our particular use case, some could be more beneficial than others.

To be able to judge what pattern corresponds to our needs correctly, it is essential to have at least a rudimentary understanding of those. So let us go over the different patterns that are included.

- **Declaration pattern.** Used to check the runtime type of an expression and, if a match succeeds, assign an expression result to a declared variable.

```
if (greeting is string message)
{
    Console.WriteLine(message);
}
```

- **Type pattern.** Used to check the runtime type of an expression.

```
if (person is Teacher)
{
    Console.WriteLine(person + " is a Teacher");
}
```

- **Constant pattern.** Used to test whether or not an expression result equals a given constant.

```
if (greeting is "Hello, World!")
{
    Console.WriteLine(greeting);
}
```

- **Relational patterns.** Used to compare an expression result with a specified constant value.

```
if (heightInCM is <= 160)
{
    Console.WriteLine("Tiny haha");
}
```

- **Logical patterns.** Used to test if an expression matches a logical combination of other patterns.

```
if (input is not null)
{
    Console.WriteLine("We have an input!");
}
```

- **Property pattern.** Tests if the properties or fields of an expression match nested patterns.

```
calculate switch
{
    { Add: "10" } => 10 + current, // "Add" is a property of "calculate"
};
```

- **Positional pattern.** Deconstructs an expression result and tests if the resulting values match nested patterns. This allows us to match and extract values from structured data types based on their position or structure, such as tuples.

```
(string name, int age) person = ("Alice", 30);

if (person is ("Alice", int age))
{
    Console.WriteLine($"{person.name} is {age} years old");
}
```

- **Var pattern.** Matches any expression and assigns its result to a declared variable.

```
string a = "a";
var alphabet = new string[] { "a", "b", "c" };
switch (a)
{
    case var aa when (alphabet).Contains(a):
        Console.WriteLine("It is a letter in the alphabet!");
        break;
}
```

- **Discard pattern.** Matches any expression without assigning it to a variable.

```
Console.WriteLine(IsItTimeYet(DayOfWeek.Friday));
static string IsItTimeYet(DayOfWeek? dayOfWeek) => dayOfWeek switch
{
    DayOfWeek.Friday => "It is time for the Diffuse the bomb project!",
};
```

NOTE Logical, property, and positional patterns can all be recursive, meaning they can contain nested patterns.

And there we go, the two most important topics are covered for this chapter's project. Now that we know about asynchronous operations and pattern matching, we think that you are ready to embark on this mission to diffuse the bomb.

So let us not waste any more pages and start right away.

7.4 Diffuse the Bomb

With that, we reached this chapter's project. For this project, we need to develop an asynchronous app that includes some sort of pattern matching during the process. That should be easily doable with our idea for an app. A Diffuse the Bomb project.

This app will teach us about async operations running in parallel to our main mechanics, and how to compare given results to a generated correct answer. Great, **but what is this Diffuse the bomb game?**

7.4.1 The Project

The Diffuse the Bomb game is about a math challenge. This challenge revolves around us as contestants resolving a randomly generated math question within a set given time frame. In other words, **What is $2 + 2$? Think fast!** And try not to cave into the pressure set by a 10s asynchronously running timer.

Sounds doable, so let us start right away.

7.4.2 Our Algorithm

As usual, we must start by creating a class Program with a static Main method. We previously mentioned that we intend to maintain everything within this main method that the project in question permits and present an alternative to our previous approach.

So this will be our starting point and where our entire code base will be contained, just the following lines.

```
class Program
{
    public static void Main(string[] args)
    {
    }
}
```

From here, we must now determine what we want to start with. As we mentioned, this project will be composed of a generated question, a pattern-matching comparison of answers, an asynchronous timer, and an end scenario. Following this chapter's order, starting with the asynchronous timer sounds like the most suitable approach, so let us do so.

To make the idea clear, the way we are going to develop this timer is by using the *Delay* method from the *Task* class. As we probably already know, the *Delay* method delays the execution of that given asynchronous task by the time set in milliseconds. So if we simply iterate a loop and execute a 1-ms delay for our desired duration, for example, 1000 iterations on a 1-ms delay for a 1000-ms, or 10-s, timer, we should achieve what we are looking for.

That makes sense. Let us do so with our iterator. As we said, we want an iterator that iterates for the amount of milliseconds we want our app to run. So 1000 iterations should suffice.

```
for (int a = 1000; a >= 0; a--)
{}
```

Great, then let us add a slight delay within the iterator so that it waits for 1ms on each iteration. If we remember right, we did that using await and then the method we needed, like so:

```
for (int a = 1000; a >= 0; a--)
{
    await Task.Delay(1, CancellationToken.None);
}
```

This also already includes a *CancellationToken*, since we have to use it if the user successfully diffuses the bomb to stop the timer and cue the win condition.

However, this, as is, will not work. It will warn us that the *await* operator can only be used within an *async* method (Fig. 7.5).



Fig. 7.5 We got a CS4033 error for trying to use the *await* operator outside of an *async* method

Therefore, we need to encapsulate this iterator in an *async* method. Let us do so with an *ActivateBombAsync* *Task* method.

```
static async Task<string>
ActivateBombAsync(CancellationToken cancellationToken)
{
    for (int a = 1000; a >= 0; a--)
    {
        await Task.Delay(1, CancellationToken.None);
    }
}
```

As we can see, we used a `Task<string>` method, for which we will need a return for, a string return to be exact. Since that return would mean an end-game scenario where the timer would run out, let us use this as a message for the contestant.

```
static     async     Task<string>     ActivateBombAsync(CancellationToken
cancellationToken)
{
    for (int a = 1000; a >= 0; a--)
    {
        await Task.Delay(1, CancellationToken.None);
    }
    return "Timer run out! Game Over!";
}
```

Perfect, there are currently no errors to speak of. However, we are not calling it, nor are we displaying the timer yet. So for the latter, a simple `Console.WriteLine` with a character we have learned about in previous chapters, the `\r`, which commands the carriage to go back leftwards until it hits the leftmost stop, meaning it'll return to the beginning. This will prove helpful to be able to replace each iteration timer with the updated version, like so:

```
for (int a = 1000; a >= 0; a--)
{
    await Task.Delay(1, CancellationToken.None);
    Console.WriteLine("\r {0}:{1} <- Answer quick! = ", a / 100, a % 100);
}
```

This should give us a precise representation of the current leftover time. Plus, we already have a little text prompting the user for an answer. We will need that later!

Now, to be able to see our timer in action, at least as a test, we will need three simple lines of code. First, we need to create a cancellation token to use for our cancellable `Delay` method. So right after our `async` method, add the following.

```
CancellationTokenSource cancellation = new CancellationTokenSource();
```

Then, we need to call the `async` method like so:

```
var result = ActivateBombAsync(cancellation.Token);
```

And finally, so that the main thread doesn't just stop running right after calling it, a `Wait()` method.

```
result.Wait();
```

And then, well, just execute to get this nice result (Fig. 7.6).



Fig. 7.6 Our first result: the asynchronous timer is running and continuously displaying the time left on the console

There we go. Step one is completed. Now on to the next step, the random operator generator.

So for the random operator generator, we have the goal of randomly generating a mathematical operation based on some preset parameters so that ideally, every game can be different from the previous.

Our operation will be based on a left side operand, an operator, and a right side operand. So a simple $2 + 2$ style operation, with a range of 1, 10 (10 not inclusive) and the option of randomly using addition, subtraction, multiplication, or division.

So, let us do so then. Starting with the random generation since that is what we are most used to. Simply add the following below our previous code.

```
result.Wait();

var rnd = new Random();
var left = rnd.Next(1, 10);
var right = rnd.Next(1, 10);
```

Alright, this will generate our numbers now. Next, let us do a quick print to be able to visualize later on.

```
Console.WriteLine($"Left: {left}, Right:{right}");
```

Now, we need our operators. For that, we can create a simple enumerator of type int and list our operators within it. Just keep in mind that this would be placed outside our `Main` method, so just after the closing bracket for our `Main` method, like so:

```
enum Operators : int
{
    Sum = 0, Minus = 1, Multiply = 2, Divide = 3
}
```

With this, we have the option to randomly pick one of the four operators. Let us then create a variable that will do so. Just before our `Console.WriteLine` we wrote:

```
var operation = rnd.Next(3);
```

And also, let us cast the operation index to the `Operations` enumerator.

```
var operation = (Operators)rnd.Next(3);
```

Then, let us include the operand in our `Console.WriteLine()` so we can visualize it.

```
Console.WriteLine($"Left: {left}, Operator: {operation}, Right:{right}");
```

Alright, if we now take the `result.Wait()` line from before and place it below our `Console.WriteLine()` like this

```
Console.WriteLine($"Left: {left}, Operator: {operation}, Right:{right}");
result.Wait();
```

we can do a swift test to see if our random generation works correctly (Fig. 7.7).

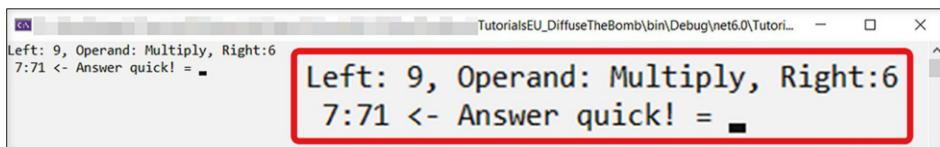


Fig. 7.7 After testing our operation generation, we see that it correctly generates the left-hand operand, the operator, and the right-hand operand

It seems to be working as expected. Then to be able to use it, wrap this whole code into a single function called `generateAnEquation()`. And we should end up with this result:

```
void generateAnEquation()
{
    var rnd = new Random();
    var left = rnd.Next(1, 10);
    var right = rnd.Next(1, 10);
    var operation = (Operators)rnd.Next(3);
    Console.WriteLine($"Left: {left}, Right:{right}");
}
```

And instead of console logging, we can return the values. Just remember that, if we want to return the randomly generated operation variables, we must change the function from void to a fitting return type, as follows.

```
static (int Left, Operators Operator, int Right) generateAnEquation()
{
    var rnd = new Random();
    var left = rnd.Next(1, 10);
    var right = rnd.Next(1, 10);
    var operation = (Operators)rnd.Next(3);
    return (left, operation, right);
}
```

Next, we can call the function, to get and store the values we will be using later to generate the final operation.

```
var (Left, Operator, Right) = generateAnEquation();
```

This will now store the numbers to be operated on and the operand within these variables.

NOTE The previous variable declaration consists of three separate variables (Left, Operator, and Right), and them being given the return of generateAnEquation(), which was return (left, operation, right).

Now, let us build the operation. Essentially, depending on the operator, we want to build a specific operation and get the final answer calculated beforehand. Using a simple if/else if statement tree, we can get something like this.

```
string puzzle = "";
if (Operator == Operators.Sum)
{
    puzzle = $"{Left} + ? = {Left + Right} ==> {Right}";
}
else if (Operator == Operators.Minus)
{
    puzzle = $"{Left} - ? = {Left - Right} ==> {Right}";
}
else if (Operator == Operators.Multiply)
{
    puzzle = $"{Left} * ? = {Left * Right} ==> {Right}";
}
else if (Operator == Operators.Divide)
{
    puzzle = $"{Left} / ? = {Left / Right} ==> {Right}";
}
```

As we can see, we first create a new string variable which we call *puzzle*. This will hold our final operation. Then, in the if statements, we, depending on the operator, create an operation with the final result. This will only serve us for testing purposes but will greatly help us understand how the inner workings are going.

Remember that `result.Wait();` has to come after these last lines of code we implemented so that the waiting occurs after they have been written.

So let us test this out by printing the final operation.

```
else if (Operator == Operators.Divide)
{
    puzzle = $"{Left} / ? = {Left / Right} ==> {Right}";
}
Console.WriteLine(puzzle);
result.Wait();
```

And now, execute the code, and we can see the final result (Fig. 7.8).

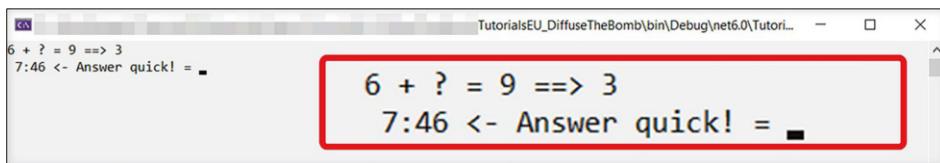


Fig. 7.8 We have correctly generated an operation, calculated a result, and formatted it in an understanding manner. And all of that while the counter kept on going asynchronously

Perfect. We have generated the operation and displayed it successfully. Let us simplify this by using a switch statement and a bit of that good old pattern matching we learned about in this chapter. So, **how would we turn an if statement into a pattern-matching switch expression?**

If we remember from the theory, a pattern-matching switch expression was just a switch statement with extra steps. But also, extra features; do not forget that. So to set this up, we simply need to think about it as if it was a switch statement. First, we had the unique advantage of being able to use the expression to set a variable directly, so let us build the base like this:

```
var puzzle = Operator switch {};
```

Now, we need to include the conditions on each expression case, like so:

```
var puzzle = Operator switch
{
    Operators.Sum =>,
    Operators.Minus =>,
    Operators.Multiply =>,
    Operators.Divide =>
};
```

Then, we need to create the final string we will use depending on each case, just like with the if statements.

```
var puzzle = Operator switch
{
    Operators.Sum => $"{Left} + ? = {Left + Right} ==> {Right}",
    Operators.Minus => $"{Left} - ? = {Left - Right} ==> {Right}",
    Operators.Multiply => $"{Left} * ? = {Left * Right} ==> {Right}",
    Operators.Divide => $"{Left} / ? = {Left / Right} ==> {Right}"
};
```

And at last, we need a default state, which we will be using for an **ArgumentOutOfRangeException** error, to let our user recognize that something went wrong during the execution. Although this condition will never be met since there are only four options to pick from, a fail-safe is still recommended and needed to avoid the warning we previously discussed.

```
var puzzle = Operator switch
{
    Operators.Sum => $"{Left} + ? = {Left + Right} ==> {Right}",
    Operators.Minus => $"{Left} - ? = {Left - Right} ==> {Right}",
    Operators.Multiply => $"{Left} * ? = {Left * Right} ==> {Right}",
    Operators.Divide => $"{Left} / ? = {Left / Right} ==> {Right}",
    _ => throw new ArgumentOutOfRangeException($"nameof(Operator): {Operator}")
};
```

And there we go. This will now replace our if statements. We have successfully implemented pattern matching in our application. Using the `Console.WriteLine` from before, we can now go ahead and visualize the result (Fig. 7.9).

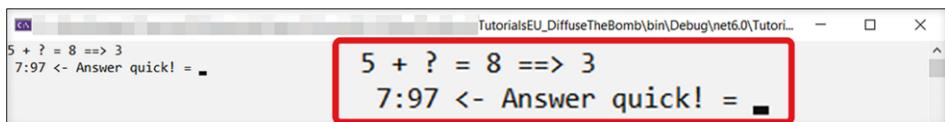


Fig. 7.9 After implementing the pattern-matching switch expression to replace our previous if tree, we achieved the same result in a cleaner and more optimized way

As we can see, we achieved the same result using pattern matching, with a cleaner code and a more optimized workflow. We achieved the result we were after using better ways to do so. Now though, we need to dig deeper into C#. Then **it is time to start using Records, but what are Records in C#?**

In short, record types are immutable reference types that provide value semantics for equality. Great, what does this even mean?

The most important bit one needs to understand is that the properties of an instance of a reference type, like records, cannot change after their creation, which means that outside processes cannot modify them during the execution of the application.

So if our application, which is true for our case, is one of those cases where we do not want the properties to change during execution, using records can be beneficial.

Although there are many more specific advantages over classes and structs than the one mentioned, that is usually the main reason to decide if records are the right fit for our applications.

Since we do not want our properties to change during execution since that would affect our final operation, we can use records to our advantage. And doing so is as straightforward as adding the following right after our enumerator.

```
record Puzzle(string Question, string Answer);
```

And now, add new Puzzle to each one of our switch cases.

```
var puzzle = Operator switch
{
    Operators.Sum => new Puzzle($"{Left} + ? = {Left + Right} ==> {Right}"),
    Operators.Minus => new Puzzle($"{Left} - ? = {Left - Right} ==> {Right}"),
    Operators.Multiply => new Puzzle($"{Left} * ? = {Left * Right} ==> {Right}"),
    Operators.Divide => new Puzzle($"{Left} / ? = {Left / Right} ==> {Right}"),
    _ => throw new ArgumentOutOfRangeException($"nameof(Operator) : {Operator}")
};
```

This will not work yet, since we will need to give the Puzzle record a second value for the answer property, and if we look closely, we are just providing a string as a single value. Therefore, simply separate the {Right} value into a distinct string as follows:

```
Operators.Sum => new Puzzle($"{Left} + ? = {Left + Right}", $"{Right}"),
Operators.Minus => new Puzzle($"{Left} - ? = {Left - Right}", $"{Right}"),
Operators.Multiply => new Puzzle($"{Left} * ? = {Left*Right}", $"{Right}"),
Operators.Divide => new Puzzle($"{Left} / ? = {Left / Right}", $"{Right}"),
_ => throw new ArgumentOutOfRangeException($"nameof(Operator) : {Operator}")
```

And with that, we can again run the app and see our result (Fig. 7.10).

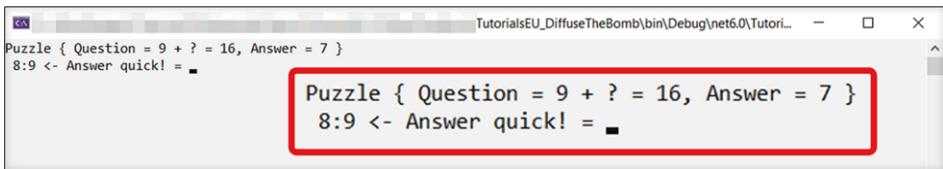


Fig. 7.10 After using record, we get the following result, assuring us that the correct values are being sent and that our code works correctly

Great, we got our question ready. Now we need to go with the next step, getting our contestant's answer to compare it with the correct answer.

So, to get our contestant's answer, before the wait and after the `Console.WriteLine`, write a `ReadLine` method.

```
Console.WriteLine(puzzle);
var usersAnswer = Console.ReadLine();
result.Wait();
```

Naturally, we are not done here. Although with that, we would already get the user's answer, we still need to do something with it.

We can start by checking if, well, it is the correct answer; let us do so with an if statement.

```
var usersAnswer = Console.ReadLine();
if (usersAnswer == puzzle.Answer)
{
    Console.WriteLine($"You Won! Your answer of {usersAnswer}
    $was correct!");
}
else
{
    Console.WriteLine($"The correct answer is {puzzle.Answer},
    $but you entered: {usersAnswer}! Disqualified!");
}
```

Also, let us not forget the possible case where the user says nothing, like this:

```
var usersAnswer = Console.ReadLine();
if (usersAnswer == "")
{
    Console.WriteLine("You didn't answer! Disqualified!");

}

else if (usersAnswer == puzzle.Answer)
{
    Console.WriteLine($"You Won! Your answer of {usersAnswer}
        $was correct!");

}

else
{
    Console.WriteLine($"The correct answer is {puzzle.Answer},
        $but you entered: {usersAnswer}! Disqualified!");
}
```

And this would already work. However, we have already learned how to make this better and cleaner. **By using switch expressions and pattern matching!**

```
var usersAnswer = Console.ReadLine();
Console.WriteLine(usersAnswer switch
{
    "" => "You didn't answer! Disqualified!",
    _ when usersAnswer == puzzle.Answer =>
        $"You Won! Your answer of {usersAnswer} was correct!",
    _ => $"The correct answer is {puzzle.Answer},
        $but you entered: {usersAnswer}! Disqualified!"
});
```

Alright, that is better. Nonetheless, it can be cleaner, not in and of itself, but by being part of our previously created record. To make sure that our final result uses the correct values, the best-case scenario would be to use our record's properties, since those are immutable.

Then to do so would simply be creating a new method within our record, as if it was another class, like so:

```
record Puzzle(string Question, string Answer)
{
    public string Evaluate(string usersAnswer)
    => usersAnswer switch
```

```

    {
        "" => "You didn't answer! Disqualified!",
        _ when usersAnswer == Answer => $"You Won!
        Your answer of {usersAnswer} was correct!",
        _ => $"The correct answer is {Answer},
        but you entered: {usersAnswer}! Disqualified!"
    };
}

```

It is getting better and better, isn't it? It is, right? ***Please respond.***

Next, we can turn our Puzzle expression into a method as well, to be able to easily call it in our code. Doing so would be replacing our “var puzzle” with the following.

```

Var puzzle = static Puzzle createPuzzle(int Left, Operators Operator,
int Right) => Operator switch
{
    Operators.Sum => new Puzzle($"{Left} + ? = {Left + Right}",
 $"{Right}"),
    Operators.Minus => new Puzzle($"{Left} - ? = {Left - Right}",
 $"{Right}"),
    Operators.Multiply => new Puzzle($"{Left} * ? = {Left * Right}",
 $"{Right}"),
    Operators.Divide => new Puzzle($"{Left} / ? = {Left / Right}",
 $"{Right}"),
    _ => throw new ArgumentOutOfRangeException($"'{nameof(Operator)}': {Operator}'")
};

```

And underneath, set the puzzle variable again.

```

};

var puzzle = createPuzzle(Left, Operator, Right);
Console.WriteLine(puzzle);

```

Alright, with this, we have covered most of the functionality of our app. Now we are only missing one thing: to make everything work together. Right now, we only have loose functions hanging around, and if we execute the project, it is not yet working as intended. For that to change, we need to do something about it. We need to organize, put together, and understand the flow to create a final app that runs and displays everything correctly.

So let us do that now in the next few steps because we are finally reaching the end of this chapter.

First, we will no longer need the `Console.WriteLine(puzzle);` line in our code.

Then, we need to wrap `generateAnEquation()` and `createPuzzle` into a function called `GenerateAPuzzle()`.

```
static Puzzle GenerateAPuzzle()
{
...
}
result.Wait();
```

Next, since the new function returns `Puzzle`, we need to return the values we previously stored in a variable, like so:

```
var puzzle = createPuzzle(Left, Operator, Right),
return createPuzzle(Left, Operator, Right);
var usersAnswer = Console.ReadLine();
```

Then, just outside the `GenerateAPuzzle()` function, we need first to call our method and store the returned variable in a variable.

```
var usersAnswer = Console.ReadLine();
}
Puzzle puzzle = GenerateAPuzzle();
result.Wait();
```

Then, we need to console write the puzzle question.

```
Puzzle puzzle = GenerateAPuzzle();
Console.WriteLine($"\\r{puzzle.Question}");
result.Wait();
```

And now we need to see that, currently, we are asking the contestant for an answer before we show the operation, which seems a tad unfair if you ask me.

Let us change that then.

```
var usersAnswer = Console.ReadLine();
}
Puzzle puzzle = GenerateAPuzzle();
Console.WriteLine($"\\r{puzzle.Question}");
var usersAnswer = Console.ReadLine();
result.Wait();
```

And if we run the code right now, we should be getting a result that already looks like what we expect to see for our final product (Fig. 7.11).

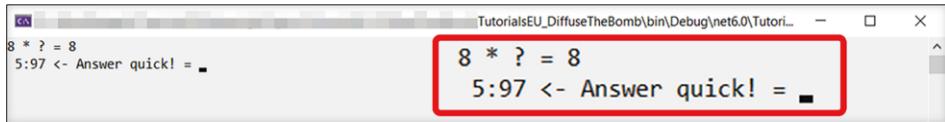


Fig. 7.11 We can see a reasonably finished-looking result after reorganizing our code a bit. We already have our question on top and the timer underneath

However, it is not yet working by any means. So first, let us get the user's answer to get it evaluated. Using our Evaluate method, we set up in the Puzzle record.

```
Console.WriteLine($"\\r{puzzle.Question}");
var usersAnswer = Console.ReadLine();
string message = puzzle.Evaluate(usersAnswer);
result.Wait();
```

This should already determine if the user's answer is correct or not and return the correct message. Now, we need to use this message to end the game. For that, let us first wrap the GenerateAPuzzle() method and the block of lines we just wrote within a new function called PuzzleApp().

```
static string PuzzleApp()
{
    static Puzzle GenerateAPuzzle()
    {
        ...
    }
    ...
    result.Wait();
}
```

And immediately after doing so, we can finally remove the result.Wait() line from here. We will add that back in later on, and add a return to our message variable, since PuzzleApp() returns a string.

```
string message = puzzle.Evaluate(usersAnswer);
result.Wait();
return message;
}
```

Great, now if we were to run the game, we would successfully view... nothing. That is not a mistake or an oversight, that is simply the fact that now everything is within its own method and nothing is being executed anymore, except for our async timer, but since we are no longer waiting for it, it just stops the execution of the entire app and no longer lets the async timer run.

Obviously, that is not the last step we need to take to finish our program. We need a method that is being called from the beginning that is responsible for the correct execution and flow of our app. And that will be the static async Task RunTheGame(). Still, within the Main method, add the following.

```
static async Task RunTheGame()
{
}
```

Here, we need to do the following. From further up in our code, we should have two lines that look somewhat like this:

```
CancellationTokenSource cancellation = new CancellationTokenSource();
var result = ActivateBombAsync(cancellation.Token);
```

Although the second one is no longer needed in its current form, we can move both lines to the newly created RunTheGame() method.

```
static async Task RunTheGame()
{
    CancellationTokenSource cancellation = new CancellationTokenSource();
    var result = ActivateBombAsync(cancellation.Token);
}
```

Now, we need to make use of a Task class method, the WhenAny method. If we remember from earlier in this chapter, the WhenAny method creates a task that will complete when any of the supplied tasks have been completed. If any of the two tasks finishes, the WhenAny task is completed. For us, we can make use of that to finally **combine the timer with our puzzle**.

As we can remember, both the timer and the puzzle will return a string, to be specific, a string that informs the contestant of the outcome of the game. So if any of those finishes, their return string can be saved in a finalMessage variable which we can later use to prompt it to the user. The implementation of this would look something like this:

```
static async Task RunTheGame()
{
    CancellationTokenSource cancellation = new CancellationTokenSource();
    var result = ActivateBombAsync(cancellation.Token);
    string finalMessage = await await Task.WhenAny(ActivateBombAsync
(cancellation.Token), Task.Run(PuzzleApp));
}
```

On a finalMessage string variable, the return value of either ActivateBombAsync or PuzzleApp is stored.

With that, if the user answers on time, the evaluated response gets returned as the finalMessage; if not, and the timer runs out, the “Timer run out! Game Over!” return message gets returned.

Perfect, not much missing. Now that the timer has run out, or, stopping the timer is no longer needed, we can cancel our cancellation token.

```
static async Task RunTheGame()
{
    CancellationTokenSource cancellation = new CancellationTokenSource();
    string finalMessage = await await
        Task.WhenAny(ActivateBombAsync(cancellation.Token),
        Task.Run(PuzzleApp));
    cancellation.Cancel();
}
```

This means essentially that, while the cancellation token is not canceled, it can still be used to stop the async process from executing, which could leave room for unwanted behavior when another process tries to cancel the timer when that should not happen anymore. To avoid further manipulation of our async processes, the token is usually canceled, making it impossible to use further.

Although that is not necessarily needed in our case, it is best practice always to do so if we do not want our async processes to be further worked with.

Other than that, we only need two more lines of code within this method, one to clear the screen, and one to display our finalMessage, like this:

```
static async Task RunTheGame()
{
    CancellationTokenSource cancellation = new CancellationTokenSource();
    string finalMessage = await await
        Task.WhenAny(ActivateBombAsync(cancellation.Token),
        Task.Run(PuzzleApp));
    cancellation.Cancel();
    Console.Clear();
    Console.WriteLine($"\\r{finalMessage}");
}
```

And that is it for our flow handler method. Now simply call this method at the Main method level to execute our entire script correctly, and we are ready for the final execution.

```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to Diffuse the Bomb!
        Here is your question!");
    var asyncTask = RunTheGame();
    asyncTask.Wait();
    static async Task<string> ActivateBombAsync(CancellationToken)
```

And for our final run (Fig. 7.12).

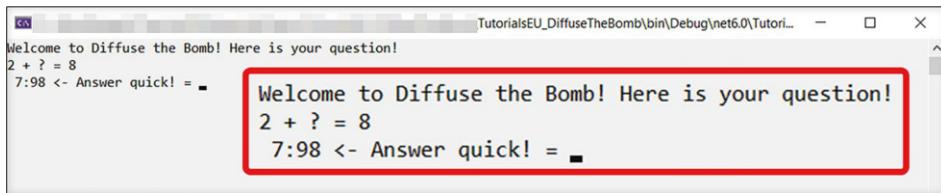


Fig. 7.12 We can see our last project execution, resulting in a successful launch and correct answers from our returns depending on our input

Congratulations! We have concluded this chapter. With that, this project's chapter correctly works and returns the values we expected from our methods. We have our timer running asynchronously and some pattern matching doing our... well... pattern matchings. We also included records and made everything run smoothly in a fun math-solving minigame.

Now, we will continue to the next topic, where we will tackle a checksum checker in C#, another must-have for sure. So, let's get ready for the next chapter.

7.4.3 Source Code

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_DiffuseTheBomb

7.5 Summary

This chapter taught us:

- The CPU is the hardware component responsible for processing and using data and instructions. If a variable is created or a mathematical calculation is solved, the CPU will be the circuitry that will make that happen.
- Asynchronous programming works on the CPU by assigning a asynchronous task to a separate Thread, leaving the main thread to continue executing instead of having to wait for its completion.
- There are two types of asynchronous programming scenarios that are well suited.
 - I/O-bound scenarios like writing/reading to a file or database
 - CPU-bound scenarios like looping through many objects or complex calculations
- Some scenarios where asynchronous programming is not well suited are the following.
 - Microservices
 - Non-scalable I/O-bound services

- A not SLA-bound service
- Dependent operations
- The basic anatomy of an async method is the `async` keyword, the return type (which can be `Task`, `Task<T>`, `Void`, `ValueTask`, or `ValueTask<T>`), the `await` keyword, and the `Task` instantiation.
- Pattern matching is a method of checking whether a given sequence conforms to a specific pattern.
- Pattern matching uses three main ways to integrate its functionality, that is, *is expressions*, *switch statements*, and *switch expressions*.
- There are nine types of pattern matching available in C#.
 - Declaration pattern
 - Type pattern
 - Constant pattern
 - Relational patterns
 - Logical patterns
 - Property pattern
 - Positional pattern
 - Var pattern
 - Discard pattern
- A record in C# is an immutable class or struct.
- Properties of an instance of a reference type, like records, cannot change after their creation, which means that outside processes cannot modify them during the execution.

References

- Bevans, D. (2022, March 28). Asynchronous vs. Synchronous Programming: Key Similarities and Differences. Retrieved August 8, 2022 from [www.mendix.com: https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/](https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/)
- Kayal, S. (2022, January 3). C# Asynchronous Programming - Return Type of Asynchronous Method. Retrieved August 8, 2022 from [www.c-sharpcorner.com: https://www.c-sharpcorner.com/UploadFile/dacca2/asynchronous-programming-in-C-Sharp-5-0-part-2-return-type-of-as/](https://www.c-sharpcorner.com/UploadFile/dacca2/asynchronous-programming-in-C-Sharp-5-0-part-2-return-type-of-as/)
- Kumar, V. (2022, July 21). Async And Await In C#. Retrieved August 8, 2022 from [www.c-sharpcorner.com: https://www.c-sharpcorner.com/article/async-and-await-in-c-sharp/](https://www.c-sharpcorner.com/article/async-and-await-in-c-sharp/)
- Microsoft Corporation. (2021, December 9). `async` (C# Reference). Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/async](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/async)
- Microsoft Corporation. (2021, December 17). Create record types. Retrieved August 8, 2022 from [docs.microsoft.com/: https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/records](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/records)
- Microsoft Corporation. (2022, January 25). Async return types (C#). Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/async-return-types](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/async-return-types)

- Microsoft Corporation. (2022, March 11). Asynchronous programming. Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/async](https://docs.microsoft.com/en-us/dotnet/csharp/async)
- Microsoft Corporation. (2022, April 8). Asynchronous programming with async and await. Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/)
- Microsoft Corporation. (2022, August 4). Asynchronous programming with async and await. Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/)
- Microsoft Corporation. (2022, March 11). await operator (C# reference). Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/await)
- Microsoft Corporation. (2022, July 19). Pattern matching overview. Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/pattern-matching](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/functional/pattern-matching)
- Microsoft Corporation. (2022, July 2). Patterns (C# reference). Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns)
- Microsoft Corporation. (2022, September 20). Records (C# reference). Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record)
- Microsoft Corporation. (n.d., n.d. n.d.). CancellationToken Struct. Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.threading.cancellationtoken?view=net-6.0)
- Microsoft Corporation. (n.d., n.d. n.d.). Task Class. Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task?view=net-6.0)
- Microsoft Corporation. (n.d., n.d. n.d.). Task.Delay Method. Retrieved August 8, 2022 from [docs.microsoft.com: https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.delay?view=net-6.0](https://docs.microsoft.com/en-us/dotnet/api/system.threading.tasks.task.delay?view=net-6.0)
- TutorialsTeacher. (2022, January 28). Asynchronous programming with async, await, Task in C#. Retrieved August 8, 2022 from [www.tutorialsteacher.com: https://www.tutorialsteacher.com/articles/asynchronous-programming-with-async-await-task-csharp](https://www.tutorialsteacher.com/articles/asynchronous-programming-with-async-await-task-csharp)



C# Hashing and Checksum

8

This Chapter Covers

- Why we should care about data integrity and its role in data security
- Using File Checksum and its use in detecting compromised data
- Working with hashing algorithms for checksum generation
- Implementing the various hash methods
- Developing a “File Checksum Checker” application

After successfully building an asynchronous application and with several complete projects under our belt, we are ready to venture into data integrity, its importance, and the tools we have available for that with C#.

With online data nowadays a must in almost every aspect of our lives, the advantages of learning about data security become apparent, especially when we consider the ever-growing need for experts in this field.

Our programming language offers many options to choose from, so we can experiment with various data security methods. As we will see shortly, these methods cover everything from ensuring the correct manipulation of data to anything hashing-related.

We will see what role checksum plays in this context and how we can use cryptographic hashing algorithms to secure our file’s integrity.

Then, we will continue to build a small application that will take a given file and run it to check its checksum with a given or generated checksum value. This application can be seen as a small file integrity tester, technically able to serve as a tool for small-volume businesses and hopefully useful as a base to develop further into a more feature-rich application.

However, before we can jump into the latter, we must begin by learning about data integrity as our context. So let us get straight into this chapter. What is data integrity, and why should we care?

8.1 Data Integrity

Simply put, if we want to maintain our data's trustworthiness, we must ensure its integrity.

Over the past few decades, tech companies have gradually increased their data sets. With that, previously unheard of problems became more apparent. In situations where the data's correctness is vital, a transfer error or an unauthorized modification can be fatal.

Take a hospital as an example. Although we may not think about it that way, the data for any given patient is entirely stored within a database. As we may imagine, this data includes vital information in the literal sense. Just like the data on a hard drive is susceptible to corruption, the data on a hospital's database can be lost due to a multitude of factors. A corrupt allergy report or a missing patient medical warning could not only mean monetary hardships for that hospital, but also potentially the loss of a patient's life.

For that, monitoring tools were built to help system administrators control a file's integrity through methods like the "CheckSum" method, where a number of bits are generated unique to that file for later comparison with a future version. That way, if the file is unaltered, the results should match. Let us get a bit more into detail.

8.1.1 What Is Data Integrity

Data integrity discusses the accuracy and completeness of our data. It is essential for maintaining trust in our databases and ensuring that they can be relied upon for decision-making. Several things can threaten data integrity, including hardware failures, software bugs, human error, and malicious attacks.

Data integrity initiatives aim to ensure that information is easy to find, can be traced back to its source, is reliable, and can be used. Throughout an organization, data integrity leads to better decision-making, lower costs, and improved efficiency.

At its core, data integrity is the reliability and trustworthiness of data from its creation or acquisition to its storage, backup, and archiving or destruction. For data to be considered as maintaining its integrity, it must be validated as uncorrupted and uncompromised.

Data integrity is used to describe the state of data, either valid or invalid, and the process of ensuring the data's validity through error checking and anomaly detection, important in so much of the world's activities because of the ripple effect inaccurate data can have.

Data integrity is the method of maintaining and securing data accuracy, reliability, and consistency throughout the data's lifecycle, using practices for the control of cybersecurity, physical data safety, and database management. The aspects that make information reliable in terms of its physical and logical accuracy are also part of data integrity.

8.1.2 Why Do We Care About Data Integrity

Our database is only as reliable as the data that it contains. In other words, if the databases contain inaccurate or corrupted data, the organization cannot make sound decisions based on that information. This fact is why data integrity is so important.

Data integrity is essential for maintaining the credibility of an organization. Suppose people cannot rely on the accuracy of a company's data. In that case, they may lose faith in their ability to make decisions, damaging their reputation. Finally, data integrity is also crucial from a legal standpoint. Inaccurate or corrupted data could lead to individuals or organizations taking action against them if they suffer damages as a result.

Let us go over the main points with the following list to remind ourselves why this topic is so important.

- Safeguards users' privacy by reducing the risk of data breaches when preventing unauthorized access to an administrative password
- Promotes positive customer experiences customized with quality data avoiding wrong or repeated data
- Offers the infrastructure that safeguards data throughout its lifecycle so as not to lose or corrupt it
- Promotes reliable analysis and valuable reports to back up informed decisions
- Guarantees the quality of products and services
- Reduces storage and improves performance by minimizing or eliminating incomplete and duplicate records
- Moreover, it generally just strengthens users' faith and certainty in the data

As noted above, we see two main types of data integrity: physical and logical. Physical data integrity is about how data is stored so that it is safe from things like security breaches or disasters. Logical data integrity is about how data is protected from being corrupted by things like human error.

8.1.3 Types of Data Integrity

There are two types of data integrity, physical integrity and logical integrity. Physical data integrity protects stored data from corruption or loss due to physical damage (e.g., fires, floods, hardware failure). Logical data integrity, on the other hand, encompasses all aspects related to ensuring that stored data remains accurate and consistent even when subjected to logical operations (e.g., updates, deletions) or system changes (e.g., configuration changes).

Thus, achieving comprehensive data integrity requires taking into account both physical and logical Data integrity. Let us look closer at each one.

Physical Data Integrity

Physical Data Integrity (PDI) is a crucial aspect of data management that focuses on protecting data from physical damage or loss. Ensuring data integrity is vital for preserving availability, safeguarding sensitive information, ensuring business continuity, maintaining regulatory compliance, and enhancing an organization's trust and reputation.

To maintain PDI, organizations should implement regular backups and offsite storage, utilize proper access controls, deploy physical security measures, ensure proper environmental controls, conduct regular audits and assessments, train staff on data management best practices, and develop a comprehensive disaster recovery plan. By adopting a proactive approach to PDI, organizations can effectively safeguard their data and ensure long-term operational success.

Logical Data Integrity

Unlike Physical Data Integrity, which focuses primarily on the protection of stored data, Logical Data Integrity encompasses all aspects related to ensuring the accuracy and consistency of stored data, even when subjected to logical operations or system changes. In other words, if our objective is to maintain correct and consistent information throughout the entire lifecycle of that information, then we need to ensure Logical Data Integrity. Data integrity is enforced by a series of integrity constraints or rules to achieve this.

- **Entity Integrity Constraints.** This type deals with the relationships between entities in a database. It ensures that each entity has a unique identifier and that all attributes for an entity are valid values within their respective domains. In addition, it prevents duplicate entities from being created.
- **Domain Integrity Constraints.** This type governs the range of allowable values for specific attributes/columns. For example, if we have an integer field called “Age,” then domain integrity would ensure only whole numbers within a specific range and type could be entered into that field. Invalid entries would be rejected.
- **Referential Integrity Constraints.** As implied by its name, referential integrity requires every foreign key (the unique link between two tables) value to reference a valid primary key (the unique identifier of a table) value in another table; otherwise, the entry will be rejected. An example might help illustrate. Let’s consider an example with two tables: one containing information on job positions (“Jobs”) and the other holding details about employees (“Employees”). In this case, the “Jobs” table lists various positions (e.g., Construction Worker, Accountant), each with a unique identifier (primary key). The “Employees” table stores information about individual employees, including their job positions, which is referenced through a foreign key that links to the primary key in the “Jobs” table. Suppose we want to add a new employee, “Bob,” who is a Construction Worker. We must first ensure that the “Construction Worker” position exists in the “Jobs” table. If it does, we can proceed to add “Bob” to the “Employees” table and reference the “Construction Worker” position using the foreign key. This way, referential integrity is maintained, as the foreign key in the “Employees” table points to a valid primary key in the “Jobs” table.

- **User-Defined Constraints.** These types of constraints are set by users when they want stricter rules regarding what constitutes invalid data than what those previous standard kinds provide. For example, a company might decide that birthdates must always fall on weekends for all new employee records added going forward, so they will not take a work day off to celebrate anymore. Cruel.

After this comes the question: How would one perform data integrity checks? One of the main methods of data integrity checking used by many is the checksum method.

8.2 Checksum

In essence, a checksum is a unique value generated from a file's data, which helps verify the file's integrity. For example, consider a file containing the text "Hello World!" A checksum algorithm processes the file's data and generates a unique string of characters representing the file in its current state. This string, known as the checksum, can then be used to verify the file's integrity.

When you want to confirm that a file has not been altered or corrupted, you can use the same checksum algorithm to recompute the file's checksum. If the newly generated checksum matches the original one, you can be confident that the file's contents have not been changed. If the two checksums do not match, it indicates that the file has been altered or corrupted since the original checksum was generated.

We will focus on that functionality in this chapter, starting with checksum and its role in the data integrity world.

8.2.1 What Is Checksum

With logical data integrity being one of the major causes of data loss or compromised files, many integrity-checking methods have been developed, starting in 1963 when the US Food and Drug Administration (FDA) published its first guideline for data integrity (direct link <https://doi.org/10.5731/pdajpst.2017.007765>).

The importance of data integrity has grown exponentially with today's modern cryptography hashing algorithms, a topic we will dive deeper into later in this chapter, like PBKDF2 and Argon2, which generate a string of characters from the file's checksum representing the contents of our file. Let us simplify this to make this clearer.

Let us say we "developed" our own "checksum" method. In our case, each letter in a text file gets represented by its position in the alphabet. So if the text states "Hi," its "Checksum" would result in "89." Now, what if we were to change the text to something like "Ha." If we were to generate its "Checksum" now, the result would be "81." Now, when comparing this new result with our previous result, we can identify a difference, meaning that the text file is no longer in the same state we had before, leading

us to conclude that the file has been altered. The following diagram should properly visualize the process (Fig. 8.1).

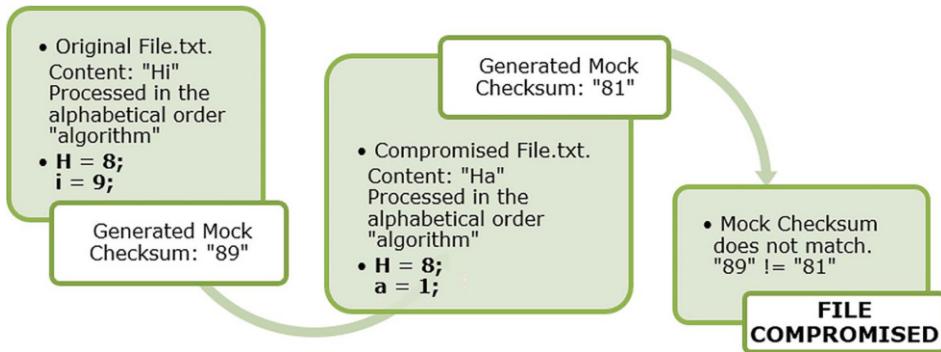


Fig. 8.1 Graphically visualizing the process of checking a file for changes using the simplified mock checksum method by purely using the alphabetical position of each letter contained in the given file

NOTE This previous example is a vast oversimplification and does not represent the real functionality of a checksum or hashing algorithm but only simulates its behavior to ease our understanding. There would be no correlation between the checksum value and the file's contents in real checksum functions.

In a real-world scenario, every piece of data or file can be assigned a checksum value generated after running a cryptographic hash function. This generated result, which generally is a long string of letters and numbers acting as a sort of fingerprint for a file, is then later used to check if a file has been compromised, be it due to transmission errors, storage failures, or human manipulation.

If the checksum calculated by the end-user differs even slightly from the original, it can inform that the file on hand was corrupted, and further actions must be taken.

The following example shows the result of a checksum generation for a file containing the text “Hey.” We will have to use the MD5 hashing algorithm for this example, but do not worry; we will explain that topic further in this chapter.

Listing 8.1 This Example Shows the Result of a Checksum Generation for a File Containing the Text “Hey”

```

// For this we need to import the
// System.Security.Cryptography namespace
using System.Security.Cryptography;
// Create an instance of the MD5 hash algorithm
using var md5 = MD5.Create();
// Read the file whose hash is going to be generated.
Content of the text file: "Hey"
  
```

```
using var stream = File.OpenRead("TestFile.txt");
// Compute the Checksum of the file
byte[] checksum = md5.ComputeHash(stream);
// Convert the byte array to base64
string base64Hash = Convert.ToBase64String(checksum);
// Write to the console
Console.WriteLine(base64Hash);
// Output: 007beZWE2FD92AL9PCeuNA==
```

If we try to run this, we will get the result of 007beZWE2FD92AL9PCeuNA== written to our console. That would be the real checksum value for our file. Also, if we were to run it a second time, we would observe that the checksum would not change. However, this might not always be the case. Maybe your case shows a different result for your second run right now. In the checksum generation process, many things could affect the final result, some more complex and worrying, some a bit simpler. Let us learn about some common causes of checksum inconsistencies.

8.2.2 Checksum Inconsistencies and Why We Shall Not Forget the Simple Things

While we already briefly mentioned some reasons why a checksum mismatch can occur, especially during the “types of data integrity” subsection, we should go through some more mundane reasons for these inconsistencies. Oftentimes during the development of context-specific software like this chapter’s project, where not only the data being worked on has to stay consistent but also the algorithm used, we encounter errors that we cannot relate to any expected outcome.

This is true for not only this chapter’s topic; it is and will always be essential to keep possibilities of a more superficial nature in mind. So our code does not need to return a checksum inconsistency because some big-time hacker is trying to steal our famous cupcake recipe, but simply because we changed the number of eggs used, therefore modifying the file and forgetting to account for that. So then, let us go through some examples.

- The most basic reason, and often the most common, is that the **data being checked has simply been changed without notifying that change**, and the new data produces a different checksum than expected. This can happen due to human error (e.g., someone accidentally changes a file), or it can be intentional (e.g., someone deliberately modifies a file). It is important to keep track of intentional file modifications and suspicious administrators to avoid any possible headaches due to trivial issues.
- A less common but still probable reason is that there was **an error in generating the original checksum**. This could be due to software bugs, hardware errors, or other problems during transmission or storage of the checksums themselves, for example, if two systems generate slightly different results for identical input.

- Another could be that **the wrong file has been used**. Two files can have the same extension and name but contain different data. Although this seems too asinine a reason, this inconsistency can occur if we use an automated file-checking code that looks for a specific name and extension. Or it could also just be that we selected the wrong file. Simply make sure you use the correct file. Trust me, I'm speaking from experience.
- Another reason could be that the **recent file modifications are being tracked and consistently updated, but not their checksum**. As we already know, each change, no matter how small, will affect the outcome of the checksum generation. Suppose the file modifications are being tracked consistently but the latest file has no generated checksum yet. In that case, the latest checksum will be from a previous version, producing a mismatch in our checksum check. Always keep your versioning up to date!
- **A corrupted download** could also be the culprit. If we are getting our files from an external source, the download of the file can produce an error that corrupts a file, be it due to network issues or file transfer errors. Sometimes a simple re-download could make our problems disappear.
- And lastly, it may very well be of a malicious nature. **Malware and generally malicious software can compromise the integrity of our files**, be it intentionally or collaterally. If several apparently unexplainable inconsistencies have appeared, it might be wise to invest in a system cleanup or reset to avoid further damage. Keep your data backed up!

Generally, we can only recommend always checking for the more common user errors before we go and declare the file in question corrupt and lost. Nevertheless, before discovering possible inconsistencies in generated checksum values, we must first learn how to generate these. With that, we finally learn about hashing and how to do so using C#.

8.3 Hashing

When it comes to cybersecurity, we often hear about hashing. Per definition, hashing algorithms produce a fixed-size string of bits that represent the given data. We can see this as an algorithm that takes a piece of data, such as a string or the contents of a file, and generates a shorter value or key that can be used to ensure its security.

But what exactly is hashing, and how can we use it with C#?

8.3.1 What Is Hashing

We are presented with a long string of seemingly random characters and led to believe that this will ensure our data's safety and integrity. As we have already seen, these characters are not random at all but rather the result of an algorithm generating these strings out of the data it is given. As discussed, we can generate a hashed checksum from a file to detect

integrity compromises, and any minor change to the given data will change the final result of our hashing algorithm; furthermore, there is also the concept of the “avalanche effect.” This concept refers to the unique characteristic of secure hashing algorithms that result in an entirely different hash, even if the data has only been altered slightly. For it to be considered to have the avalanche effect, an average of one-half of the resulting hash should change if we change 1 bit of the given data. The proper definition of avalanche effect is given in the paper at direct link https://link.springer.com/chapter/10.1007/978-3-030-39799-X_41#page-1. However, we should be able to exemplify this behavior through the following example (Table 8.1).

Table 8.1 This table shows two code examples of hashing using two different inputs. Both show their output at the end

First run. File Content "Hey"	Second run. File Content "Hei"
<pre>using System.Security.Cryptography; using var md5 = MD5.Create(); StreamWriter sw = File.CreateText("example.txt"); sw.WriteLine("Hey"); sw.Close(); using var stream = File.OpenRead("example.txt"); byte[] checksum = md5.ComputeHash(stream); string base64Hash = Convert.ToBase64String(checksum); Console.WriteLine(base64Hash); // Output: DzCIrvuj0oPWDNkXucq6DVw==</pre>	<pre>using System.Security.Cryptography; using var md5 = MD5.Create(); StreamWriter sw = File.CreateText("example.txt"); sw.WriteLine("Hei"); sw.Close(); using var stream = File.OpenRead("example.txt"); byte[] checksum = md5.ComputeHash(stream); string base64Hash = Convert.ToBase64String(checksum); Console.WriteLine(base64Hash); // Output: aAGBsc3juRN9PzB8YZ+Keg==</pre>

As we can see, in the second result, **although the file changed only slightly, the resulting checksum is completely different.**

If we take our mock checksum as an example, the only difference between our two results was one number, “89” compared to “81.” This would be an example of an algorithm not containing the avalanche function; furthermore, it would be an example of an algorithm that is reversible. After a few generations, one could easily correlate the number in the checksum with the letter’s position in the alphabet, making it, thus, highly insecure. ***That is why, at least in hashing algorithms where security is a desired feature, a non-reversible result is a must.***

Now, technically, every hash algorithm is somewhat reversible. If we run a piece of code that generates the checksum of a random three-letter word until both checksums match, we will eventually discover the original message. However, not only are the hashed inputs usually not three-letter words, but they frequently use several **iterations** of hashing, so how many times we hash a password before we store it. Or the so-called **salting**, which is the addition of a unique, random string of characters known only to the site to each password before it is hashed, further complicating the brute forcing of the original data.

An attempt to brute force a secure password, for example, could take an infinite amount of time, even with modern machines.

But, other than brute force, why can we not simply figure out the input using the output?

You may have noticed that in our mock checksum example, the final output would grow proportionally together with the given input. Naturally, that is not the behavior of actual hashing algorithms since they stay of a fixed size. To accomplish this, **some data must be discarded in the hashing process**.

Let us take as an example the following operation:

$$256 + 256 = 512$$

The final result of this operation is 512; we obviously understand that by adding together $256 + 256$ we get 512. However, since our result would only be 512, comparable to our checksum result, we would not have any information about how we got to that result.

$$\begin{aligned}1 + 511 &= 512 \\500 + 12 &= 512 \\-10 + 522 &= 512\end{aligned}$$

There is practically an infinite quantity of possible combinations that all lead to the same result of 512. This makes reversing a hash practically impossible, for it is only an option if the brute force method is used. Also, we must say that if it were possible to reverse a hash, it would give us the ability to compress data of any size into a mere few bytes of data, and as we know, this is currently not a possibility as of 2022.

If you are reading this from the future, let us know if that changed! Contact us directly under the following hashed phone number! **WzXMF/sgExVztJJcDXi/sA==**

Until now, we have been talking about hashing and its implications in cybersecurity. Typically, this behavior is unique to the so-called cryptographic hashing algorithms. You may have heard about hashing functions and cryptographic hashing functions alike, but either did not notice an apparent differentiation between the two or did not know that there even was a differentiation between the two, to begin with. Yes, there are **cryptographic** hashing algorithms and **non-cryptographic** hashing algorithms, **but what is the difference?**

8.3.2 What Is the Difference Between Cryptographic Hashing and Non-cryptographic Hashing

In simple words, cryptographic hash functions are just hash functions with extra steps.

Every cryptographic hash function is essentially the same as a non-cryptographic hash function, apart from the fact that cryptographic hash functions aim to guarantee a number of security properties.

Generally, non-cryptographic hash functions are intended for accidental non-malicious data integrity failure detection. They present collision detection, meaning that they intend to prevent the generation of the same hash for two or more different inputs, aiming to detect accidental changes in data (CRCs or Cyclical Redundancy Checking). However, the algorithms used are generally susceptible to attacks and deemed too insecure for use in a sensible context. Nonetheless, they have the unique advantage of faster hash generation speed.

On the other hand, there are some properties that cryptographically secure hash functions strongly require in order for it to be effective. These include:

- **Preimage resistance:** For a given hash value, it should be computationally infeasible to find an input that, when hashed, results in that given hash value. In other words, it should be extremely difficult to reverse-engineer the original input from its hash.
- **Second preimage resistance:** Given a specific input, it should be challenging to find a different input that produces the same hash value when processed through the hash function. This property ensures that an attacker cannot easily create an alternative input with the same hash, potentially bypassing security measures.
- **Strong collision resistance:** It should be computationally difficult to find any two distinct inputs that generate the same hash value when processed through the hash function. This property prevents attackers from finding hash collisions, which could compromise the integrity and security of the hashed data.

In addition, there is a property in which older cryptographic hashes fail but which newer ones like SHA-3 and Blake 2 are designed specifically to achieve:

- **The property of Random oracle indifferentiability.** Finding any correlation between the resulting output and any given input must be difficult.

For our purposes, we primarily intend to work with cryptographic hashing algorithms, **so what algorithms could we use for our applications?**

8.3.3 Types of Cryptographic Hashing Algorithms

There are multiple cryptographic hash functions that programmers can use to generate checksum values. A few common ones include:

SHA-0

The first algorithm of its kind, created in 1993, was withdrawn shortly after its release due to an undisclosed “significant flaw” in its revised SHA-1 version.

Due to its quick withdrawal, there is no direct implementation with C#. However there is a Github-hosted implementation by Andrey Rusyaev called acryptohashnet, which

includes a .Net implementation of SHA-0 (direct link <https://github.com/AndreyRusyaev/acryptohashnet>).

SHA-1

The successor to SHA-0, although cryptographically broken and no longer deemed secure as of 2010, it is still being used as a reasonably cheap data integrity check for non-malicious errors.

```
using System.Security.Cryptography;
using var sha1 = SHA1.Create();
StreamWriter sw = File.CreateText("example.txt");
sw.WriteLine("Hey");
sw.Close();
using var stream = File.OpenRead("example.txt");
byte[] checksum = sha1.ComputeHash(stream);
string base64Hash = Convert.ToBase64String(checksum);
Console.WriteLine("Hashed Checksum using SHA-1: " + base64Hash);
```

SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512)

This family of hash functions is considered very secure and was published as the official crypto standard, widely used in the Bitcoin blockchain for transaction identification and proof-of-work. However, it lacks the property of Random oracle indifferentiability, exposing it to data breaches if its hash pattern is learned.

```
using System.Security.Cryptography;
using var sha256 = SHA256.Create();
StreamWriter sw = File.CreateText("example.txt");
sw.WriteLine("Hey");
sw.Close();
using var stream = File.OpenRead("example.txt");
byte[] checksum = sha256.ComputeHash(stream);
string base64Hash = Convert.ToBase64String(checksum);
Console.WriteLine("Hashed Checksum using SHA-256: " + base64Hash);
```

SHA-3 (SHA3-224, SHA3-256, SHA3-384, SHA3-512)

Although similar to SHA-2, this family is considered more secure than SHA-2, mainly due to the property of indifferentiability. It is currently the recommended crypto standard being used as its variant, the Keccak-256 from the SHA3-256, in the Ethereum blockchain.

NOTE Currently there is no direct implementation yet. However, there are available implementations through external libraries like BouncyCastle (direct link <https://www.nuget.org/packages/SHA3.Net/>) and Keccak (direct link <https://github.com/MrMatthewLayton/CORE/tree/master/Core/Security/Cryptography>).

MD5

The MD5 hash function creates a secure checksum value. However, each file might not have a unique number. This flaw allows hackers to exploit vulnerabilities by swapping out a file with the same checksum value. Despite its potential shortcomings, the MD5 hash function is one of the most widely used algorithms because it is pretty fast and easy to implement.

```
using System.Security.Cryptography;
using var md5 = MD5.Create();
StreamWriter sw = File.CreateText("example.txt");
sw.WriteLine("Hey");
sw.Close();
using var stream = File.OpenRead("example.txt");
byte[] checksum = md5.ComputeHash(stream);
string base64Hash = Convert.ToBase64String(checksum);
Console.WriteLine("Hashed Checksum using MD5: " + base64Hash);
```

PBKDF2

The Password-Based Key Derivation Function 2 is the recommended hashing algorithm for passwords. Implementing a native iteration and salt feature is considered one of the best choices for low-risk data security. However, it is not recommended for high-risk situations due to its still existent vulnerability to GPU-based attacks. Specialized hardware, such as used in bitcoin mining rigs, can perform upwards of 50 billion hashes per second, and with that number increasing continuously, a more GPU-resistant algorithm is needed.

```
using System.Security.Cryptography;
int saltSize = 24;
int hashSize = 24;
int iterations = 100000;
string input = "Admin123";
RNGCryptoServiceProvider provider = new RNGCryptoServiceProvider();
byte[] salt = new byte[saltSize];
provider.GetBytes(salt);
Rfc2898DeriveBytes pbkdf2 = new Rfc2898DeriveBytes(input, salt,
iterations);
byte[] bytes = pbkdf2.GetBytes(hashSize);
string base64Hash = Convert.ToBase64String(bytes);
Console.WriteLine("Hashed Checksum using pbkdf2: " + base64Hash);
```

Bcrypt

Similar to PBKDF2, it is a fairly secure hashing algorithm with integrated iteration and salting capabilities. However, like PBKDF2, although better, it is still susceptible to GPU attacks.

NOTE No direct implementations have been added yet. It has to be installed via (direct link) <https://www.nuget.org/packages/BCrypt.Net-Next>.

Scrypt

Now explicitly designed to prevent custom hardware GPU-based attacks, it is deemed one of the most secure hashing algorithms. Nevertheless, it has an increased cost of usage due to its increased use of memory-heavy functions. It is typically not used for password hashing.

NOTE No direct implementations have been added yet. It has to be installed via (direct link) <https://www.nuget.org/packages/Scrypt.NET/>.

Argon2

A relatively new hashing algorithm that, if configured correctly, can be more secure than PBKDF2, Bcrypt, and Scrypt. Although beginning to increase the complexity, the use of these later hashing algorithms becomes more niche. Currently, older and simpler as well as faster-hashing algorithms are more commonly used.

NOTE No direct implementations have been added yet. It has to be installed via (direct link) <https://www.nuget.org/packages/Konscious.Security.Cryptography.Argon2/>.

There are even more hashing algorithms in existence. But for our purposes, knowing this list of the most commonly known algorithms should suffice. Moreover, we simply want to develop a program capable of checking a checksum value in a non-malicious context, meaning that a simpler hashing algorithm could be used for our use case.

We will be using MD5 for our checksum generation. Although not the most secure, it is fast and easy to implement, making it ideal for a small project like ours.

With that, let us not waste any more time and get right into developing our checksum checker project.

8.4 Checksum Checker

We have reached the part where we develop this chapter's project, and this time, we need some checksum checking to happen. In this chapter we learned about checksum, about hashing, and about data integrity. Even a bit about data security was taught to introduce us to the world of cybersecurity, which we will go over a bit further in the next chapter.

To make use of all we have learned, we will need to develop an application capable of generating a Checksum through a reliable hashing algorithm. So what will this project be about?

8.4.1 The Project

The Checksum Checker project will be an application that will allow a user to either feed it a file or create a new file to then check its checksum for possible inconsistencies.

An option will also be given to provide an existing Checksum so that the given file can be checked with a hypothetical previous version of itself.

We will use what we learned in the previous chapter about the file and its methods to handle the files. Then, through the MD5 cryptographic hashing algorithm, we will generate a Checksum to keep for future reference.

With that out of the way, we should be able to get going with our code.

8.4.2 Our Code

As usual, we must start by creating a class Program with a static Main method. To maintain consistency throughout the continuing chapters, we will maintain this code as our starting point.

So then, create a new project, clean it, and add the following lines.

```
class Program
{
    public static void Main(string[] args)
    {

    }
}
```

Starting with the basics, we first want to get the file that will be checked from the user. So we will need to ask the user for the name and specify the naming convention. In our case, that will be no special characters, and it must end in “.txt” for its extension.

Easy enough, let us do so. First, prompt the user with the initial message. Start with a welcome and end by asking for the file name and stating the naming rules, like so:

```
static void Main(string[] args)
{
    //We get the file name from the user
    Console.WriteLine("Welcome to the file integrity checker,
        please introduce the name and extension of the file
        you want to check.");
    Console.WriteLine("File must not contain any illegal characters
        and must end in '.txt' (ex. TestFile.txt)");
}
```

Great, now we will need to take the user's answer and handle the format checking. Ideally, we should do that in a separate method to keep the Main code clean. For that, let us create a new static method, so it can be called from the static Main method, just after the Main method, called GetUserFileName, that returns a string.

```
public static string GetUserFileName()  
{  
}
```

Since we set it to return a string, we now need to get that string from somewhere. As you may have guessed, that string will be the correctly formatted file name provided by the user. So start by implementing the ReadLine Console method to let the user input a text.

```
string? fileName = Console.ReadLine();
```

We set the string variable to nullable since there is the possibility that the user inputs nothing, and so to prevent any possible errors, we simply set it as nullable, so a variable that can take any of its underlying value types and an additional null value.

Now, we need some sort of name validation. For that, let us also create a dedicated method. This time, it will be called IsFileNameValid. It will return a bool and take the nullable string filename as a parameter, like so:

```
public static bool IsFileNameValid(string? fileName)  
{  
}
```

We create so many methods because, generally, separating different functionalities into separate functions is the better approach for larger-scale projects to be able to reuse code more efficiently. If we were to use the file validation method more than once during our code, we would greatly benefit from calling it instead of rewriting the contents where needed.

This directly follows the principle of Single Responsibility, one of the core principles in SOLID programming. In Chap. 10, we will come back with further detail on this topic when we apply this principle to classes. Nonetheless, in short, the Single Responsibility principle dictates that we must always separate methods, and as we will see in Chap. 10, also classes, to only do one thing and to have a single responsibility. So we could see that, in a way, we have been applying this principle during pretty much every chapter of this book.

By splitting functionalities into separate methods, we ensure clean code, keep it readable for future coders, and make it easily maintainable.

So then, the way we will be checking for the correctness of the user's input is by first checking the entire input for invalid file name characters, or if the entire thing is a null or whitespace, then we will check if the input ends with the ".txt" extension. Both these tests

will initialize a bool type variable which we will be using to return to the previous method to inform of its correctness. This would be done by the following.

```
public static bool IsFileValid(string? fileName)
{
    bool fileNameValid = !string.IsNullOrWhiteSpace(fileName) &&
    fileName.IndexOfAny(Path.GetInvalidFileNameChars()) < 0;

    bool fileNameExtension = fileName!.EndsWith(".txt");

    return fileNameValid && fileNameExtension;
}
```

We can see that using the method `IsNullOrWhiteSpace`, we could check for empty input. Then, through `IndexOfAny` and `GetValidFileNameChars`, we counted the number of invalid characters included in that input and checked for it to be true if the amount counts to less than 0 to detect any possible invalid characters that our operating system (OS) cannot use.

Then, we simply checked if the input ended with a “.txt” to ensure that the desired extension was included. Finally, we returned both variables with an “`&&`” operator to conclude either a true or false depending on whether both variables resulted in true or not.

Using that, we will already get returned correctly if the file is valid or not. The only thing missing before we can send the input back as valid is to handle the situation where the value is incorrect. We need to find a way to inform the user that his input was wrong and that he should try again.

This can be quickly achieved by using a while loop.

```
public static string GetUserFileName()
{
    string? fileName = Console.ReadLine();

    while (!IsFileValid(fileName))
    {
        Console.Clear();
        Console.WriteLine("File name is not valid.
Name must not contain any illegal characters and must end in
'.txt' (ex. TestFile.txt)");
        fileName = Console.ReadLine();
    }

    return fileName!;
}
```

Each time the IsFileValid method returns a false, the while loop will loop again, and within it, we simply clear the console and ask again for new input.

Once the while concludes, we return the file name.

If you noticed the exclamation mark beside the fileName variable at the return and are unsure what that is about, let us explain.

Let us remove that exclamation mark first and leave it like so:

```
return fileName;
```

As you can probably see, a green underline appeared under fileName stating “‘fileName’ may be null here” (Fig. 8.2).

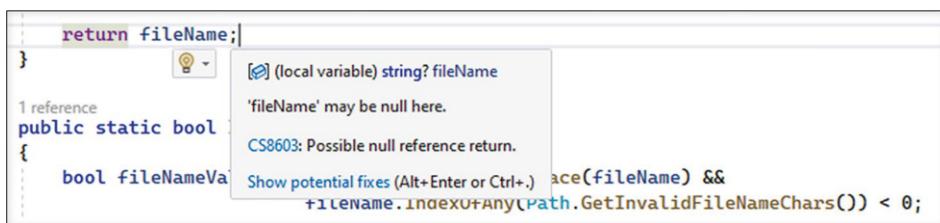


Fig. 8.2 By removing the exclamation mark from the returned variable, we got a warning for a possible unhandled null value

Although not a code-breaking error (*most of the time*), it is always better to keep track of these warnings. Since we know in our case that there will never be a null value assigned to that returned variable, we can let the compiler know that it does not need to warn us. And for that we use that exclamation mark from before, and leave it like so:

```
return fileName!;
```

In C#, that exclamation mark is known as the null forgiving operator. Essentially, it tells the compiler, “Trust me, there is never going to be a null here,” so it does not continue warning us of a possible null reference.

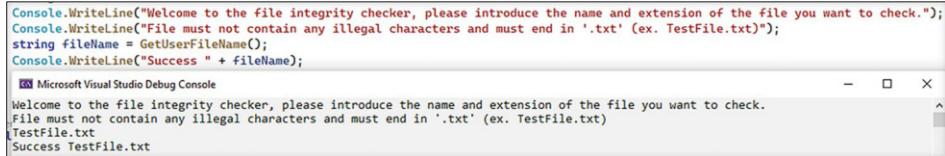
Great, we have the file naming ready. Let us try it out by calling it and storing it into a variable for later use. So back in the Main method, under our welcome messages, add this:

```
Console.WriteLine("File must not contain any illegal characters  
and must end in '.txt' (ex. TestFile.txt)");  
string fileName = GetUserFileName();
```

And temporally, to make sure that it works, add the following.

```
String fileName = GetUserFileName();  
Console.WriteLine("Success " + fileName);
```

Now, let us run it to try it out (Fig. 8.3).



```
Console.WriteLine("Welcome to the file integrity checker, please introduce the name and extension of the file you want to check.");
Console.WriteLine("File must not contain any illegal characters and must end in '.txt' (ex. TestFile.txt)");
string fileName = GetUserFileName();
Console.WriteLine("Success " + fileName);
Microsoft Visual Studio Debug Console
Welcome to the file integrity checker, please introduce the name and extension of the file you want to check.
File must not contain any illegal characters and must end in '.txt' (ex. TestFile.txt)
TestFile.txt
Success TestFile.txt
```

Fig. 8.3 Successfully validating a file name given by the user and displaying it in the console

As we can see, the given file name of “TestFile.txt” has been input, validated, returned, and displayed correctly. Great, so we already have the name. Next, we need to confirm if it exists, and if not, create a new one. So let us do so.

First, delete our last Console message since we will not need that anymore.

Now, to continue and use the file for our checksum operations, we must have it exist, so there is no reason to let it continue executing if that is not the case. This means that we can make use of a while loop again, this time with the Exists method from the File class, like so:

```
while (!File.Exists(fileName))
{
}
```

This will continue looping until that input file exists. So here is where we must either create it or prompt the user to add it to the correct directory. So first, clear the console again, prompt the user that there is no such file with the given name, and ask if he wants to create one, like this:

```
while (!File.Exists(fileName))
{
    Console.Clear();
    Console.WriteLine($"'{fileName}' does not exist, $would you like to
create it? Y|N");
}
```

Remember: Do NOT run this yet. You do not want to have an infinite loop crash your machine.

Great, as you have seen, we prompted a Yes or No type of question; however, we have yet no way to handle a yes or no response correctly. For that, we can again create a dedicated method. The YNQuestion method returns a bool for true, if it is a “Yes” answer, or false, if it is a “No” answer.

```
public static bool YNQuestion()
{
}
```

In this new method, we will also be handling the input reading, so start by creating a variable where we can store that response.

```
public static bool YNQuestion()
{
    string userInput;
}
```

Now, using a Do-While loop, we can read the input, check if it is “Y” or “N,” and read again if it is not one of these two. If it is, return one of the two as a bool, like so:

```
public static bool YNQuestion()
{
    string userInput;
    do
    {
        userInput = Console.ReadLine() ?? string.Empty;
        userInput = userInput.ToUpper();
    }
    while (userInput != "Y" && userInput != "N");
    return userInput == "Y";
}
```

As we can see, we are reading the user’s input with the ReadLine method and setting the userInput variable to either that or empty if nothing is read to also here. So we would receive either input or, if the user inputs nothing, instead of a null, it will be a "", so empty string. This later addition will handle the null warning we would otherwise receive, similar to our previous situation.

Then, we uppercase the response so it is not necessarily case sensitive and check if it is either of the two responses we are after. If not, repeat; if it is, continue and return “userInput == ‘Y’.” If this last return sounds confusing, remember that our method returns a bool, so if we return that, essentially what we are doing is returning a true if userInput is “Y,” or false if it is not.

That is our Yes or No handling method now; let us use it to get our next user response in our Main method.

Back where we were in the Main method, store in a new bool variable called createFile the return of our just created method, like this:

```
while (!File.Exists(fileName))
{
    Console.Clear();
    Console.WriteLine($"{fileName}' does not exist,
                      would you like to create it? Y|N");
    bool createFile = YNQuestion();
}
```

Now we need to set up the loop end where either the user asks us to create the file or we prompt him to add the file to the correct directory. For that, start with a familiar if statement.

```
bool createFile = YNQuestion();
if (createFile)
{
    // Create
}
else
{
    //Don't create
```

Starting with us creating the file, we simply need to use the Create method from the File class, and then remember to close the FileStream again so it can be used later again, so as not to get an IO exception, like this:

```
if (createFile)
{
    FileStream newFile = File.Create(fileName);
    newFile.Close();
}
```

So, we create a temporary variable of type FileStream where we store the newly created file. Then, using the Close method, we close the file again.

Great, now, if the user does not want us to create a new file but rather add his own, we can let him know where to do so, like so:

```
else
{
    Console.WriteLine($"Please add a file with the name {fileName}
                    under the following directory: " + Directory.GetCurrentDirectory());
    Console.WriteLine($"Press any key to check for the file {fileName}
                    again.");
    Console.ReadKey();
}
```

With the GetCurrentDirectory method from the Directory class, we display the desired directory. Then, with the ReadKey method, we wait for a key press to repeat the loop and check if the file finally exists.

Perfect. For us to check if our code works, let us write a quick message to the console. So right after our while loop, add this:

```
Console.Clear();
Console.WriteLine("File exists " + fileName);
```

Now, let us run the code and see how it goes (Fig. 8.4).



Fig. 8.4 We should get this result by either letting the code create a new file or providing the file in the correct directory

Once we get the desired result, we should be able to see the final file in the “CurrentProject/bin/Debug/net6.0 directory.” Now that our file is ready to go, we can continue with the fun part, the checksum checking. However, we do need first to ask the user if he wants to add a checksum first. If not, well, then we do have to generate one ourselves.

Alright, to start this off, let us prompt the user to provide an original checksum, if he wants, of course. Remove the previous WriteLine we used to test and add the following.

```
Console.Clear();
Console.WriteLine("Would you want to provide an original checksum
for the provided file (Y) or should the current
checksum be used? (N) Y|N");
```

And since this is also a Yes or No question, we can reuse our Yes or No method, like so:

```
Console.Clear();
Console.WriteLine("Would you want to provide an original checksum
for the provided file (Y) or should the current
checksum be used? (N) Y|N");
bool provideChecksum = YNQuestion();
```

Now, create a string variable to use as a place to store our original checksum, so the one where hypothetically the file has not yet been tampered with and still preserves its integrity.

```
string originalChecksum;
```

And next, we can use an if statement to read the user’s input if he does indeed want to provide his own checksum, like so:

```
string originalChecksum;
if (provideChecksum)
{
    Console.WriteLine("Please provide a valid checksum.");
    originalChecksum = Console.ReadLine() ?? string.Empty;
}
else
{}
```

There is no apparent need for validation here, so we can assume that the user will correctly input his checksum. Now, for the else statement, if the user does not want to provide his checksum, we need to generate an initial checksum. So it is time to get some hashing going, finally. As we have previously said, we will be using the MD5 algorithm. Although not extremely secure, it is still enough for a non-risk environment like internal file integrity checking procedures. On top of that, it is pretty fast and easy to implement.

So let us create a new static method called `GenerateCheckSumMD5`, which takes a string variable called `filepath`, as a parameter.

```
public static string GenerateCheckSumMD5(string filepath)
{
}
```

Great. For the generation of the checksum, no more than five lines will be needed. Let us go through each one of them, starting with the creation of an instance of the MD5 hash algorithm.

```
public static string GenerateCheckSumMD5(string filepath)
{
    MD5 md5 = MD5.Create();
}
```

This will give us an error “The type or namespace name ‘MD5’ could not be found” (Fig. 8.5).

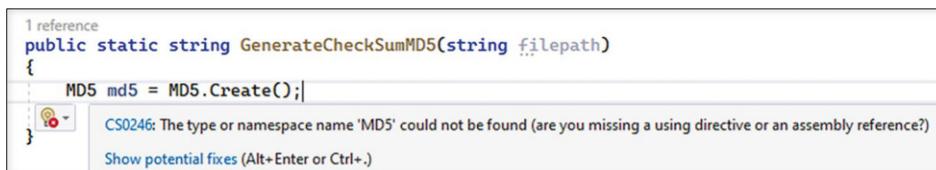


Fig. 8.5 Trying to implement a new instance of the MD5 hashing algorithm, we get the error “The type or namespace name ‘MD5’ could not be found”

This is simply due to us not adding the necessary using statements into our code. So, in other words, as the error states, “are you missing a using directive,” yes, we are, specifically the Cryptography directive. So let us add that.

At the very top of our code, we add the following.

```
using System.Security.Cryptography;
class Program
```

Now the error should disappear, and we should be able to continue.

For line two, we will be opening our file on a new FileStream, essentially reading its contents. For that, create a new variable called stream of type nullable FileStream and initiate it with the OpenRead method of the File class opening our given filepath, like this:

```
public static string GenerateCheckSumMD5(string filepath)
{
    MD5 md5 = MD5.Create();
    FileStream? stream = File.OpenRead(filepath);
}
```

It sounded more complicated than it is for some reason. Great, we have our file ready to go. Now, onto line three.

We now want to take our new FileStream and use the integrated ComputeHash method from the HashAlgorithm class to store its returned value in a variable called checksum of type byte, like this:

```
public static string GenerateCheckSumMD5(string filepath)
{
    MD5 md5 = MD5.Create();
    FileStream? stream = File.OpenRead(filepath);
    byte[] checksum = md5.ComputeHash(stream);
}
```

This will take the given stream and use the given file's contents to generate a Hash of the Checksum.

Believe it or not, technically, that is it. We have the file's checksum. In three lines, we successfully generated a file's checksum using the MD5 hashing algorithm. Told you it was easy to implement.

The only thing we are missing for lines four and five is simply converting our hashed checksum to base64 so we can work with it, then returning that converted value for further use. The reason for converting the checksum to base64 is to represent the binary data in a human-readable, text-based format that can be easily transmitted, stored, or displayed.

The base64 encoding is commonly used when there is a need to encode binary data, like checksums, that needs to be stored and transferred over media that are designed to handle textual data. This ensures that the data remains intact without modification during transport and storage.

```
public static string GenerateCheckSumMD5(string filepath)
{
    MD5 md5 = MD5.Create();
    FileStream? stream = File.OpenRead(filepath);
    byte[] checksum = md5.ComputeHash(stream);
    string base64Hash = Convert.ToBase64String(checksum);
    return base64Hash;
}
```

And there we go, we are generating our checksum. However, right now there is one issue. Simply taking in the file to hash without checking for its size is prone to error, because the size right now has no limitation. If the user decides to hash a several-terabyte file there is nothing stopping him. We may actually want to give that option if we handle it in a different way, but in our case, where we simply take it and hash it, a size check is necessary. To do so, we can simply set a hard limit of 10 MB, for example, and check the file just before we continue the execution, like so:

```
public static string GenerateCheckSumMD5(string filepath)
{
    // Check the file size
    long fileSize = new FileInfo(filepath).Length;
    if (fileSize > 10485760) // 10 MB threshold
    {
        throw new Exception("File size is too large to calculate
check sum");
    }

    // Calculate the check sum
    MD5 md5 = MD5.Create();
    FileStream? stream = File.OpenRead(filepath);
    byte[] checksum = md5.ComputeHash(stream);
    string base64Hash = Convert.ToBase64String(checksum);
    return base64Hash;
}
```

That could be a way to handle this scenario. We can always decide later if we want to roll back and make the user change the given file or handle it in a different way. For this chapter, we can leave it at that and continue with the rest of the project.

Next, we can use the returned value to store as the originalChecksum so that we have a reference point for possible future file manipulations.

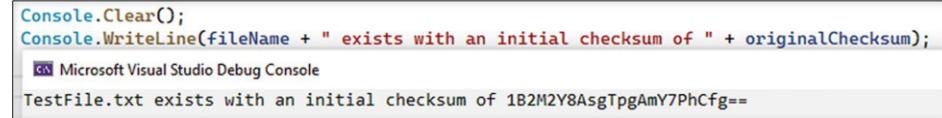
For that, in our previous if statement where we asked the user if he wanted to provide his own checksum, call the checksum generator in the else statement and store it in the originalChecksum variable, like this:

```
if (provideChecksum)
{
    Console.WriteLine("Please provide a valid checksum.");
    originalChecksum = Console.ReadLine() ?? string.Empty;
}
else
{
    originalChecksum = GenerateCheckSumMD5(fileName);
}
```

Let us now check if everything is working as intended. Clear the console and write to console a message like the following.

```
else
{
    originalChecksum = GenerateCheckSumMD5(fileName);
}
Console.Clear();
Console.WriteLine(fileName + " exists with an initial checksum of " +
originalChecksum);
```

Let us see what result we are getting after running it (Fig. 8.6).



```
Console.Clear();
Console.WriteLine(fileName + " exists with an initial checksum of " + originalChecksum);
Microsoft Visual Studio Debug Console
TestFile.txt exists with an initial checksum of 1B2M2Y8AsgTpgAmY7PhCfg==
```

Fig. 8.6 We are getting the checksum from our file correctly displayed on the console

And we did it. So we have a file, and we have its original checksum. The only thing missing is to create a “Press any key to check for the file’s data integrity” state.

For that, let us create a new method called PromptFileIntegrityCheck that returns void, so nothing, and takes the file name and original checksum as parameters.

```
public static void PromptFileIntegrityCheck(string fileName, string
originalChecksum)
{
}
```

Here, we will create the “end state” of the application, meaning the state where we have received all the data we need to achieve the execution of our main task, which is checking the given file’s integrity through its checksum.

The program will stay in a state of “Press any key to check the file’s integrity.” Each time a key is pressed, the current checksum is compared with the original checksum, giving us the result if the file has changed since its initial state.

Naturally, we could add a way to change the given file or the given checksum, but that is beyond the scope of this chapter’s project, so we will stick with this final loop.

Then let us not wait any longer and finally implement it.

Start by prompting the user with the next action that is going to be taken.

```
public static void PromptFileIntegrityCheck(string fileName, string
originalChecksum)
{
    Console.WriteLine($"Press any key to check
    ${fileName}'s file integrity.");
    Console.WriteLine("The given file will be checked
        for a match with the checksum: " + originalChecksum);
}
```

After this, we need to listen to the user's key press to continue, so just like last time, a `ReadKey` will do just that.

```
public static void PromptFileIntegrityCheck(string fileName, string
originalChecksum)
{
    Console.WriteLine($"Press any key to check
    ${fileName}'s file integrity.");
    Console.WriteLine("The given file will be checked
        for a match with the checksum: " + originalChecksum);
    Console.ReadKey();
}
```

Once a key is pressed, we will need the last method we will be implementing, the `CheckFileIntegrity`. It also returns void and needs the file name and the original checksum. After the `PromptFileIntegrityCheck` method, add the following.

```
public static void CheckFileIntegrity(string fileName, string
originalChecksum)
{
}
```

In this method, we will need to generate a checksum again; this checksum will be for the file's current checksum, so something like this:

```
public static void CheckFileIntegrity(string fileName, string
originalChecksum)
{
    string currentChecksum = GenerateCheckSumMD5(fileName);
}
```

Then, depending on the result, this shows the user the appropriate message. To use the result, we will simply make use of the `Equals` method from the `String` class to compare both checksums. So, first clear the console, and then, using a clean ternary conditional operator, we write to the console the correct message.

```
public static void CheckFileIntegrity(string fileName, string originalChecksum)
{
    string currentChecksum = GenerateCheckSumMD5(fileName);
    Console.Clear();
    Console.WriteLine(currentChecksum.Equals(originalChecksum)
        ? "CHECKSUM MATCHES."
        : "CHECKSUM MISMATCH.");
    Console.WriteLine("File Integrity Is Observed." +
        : "File Integrity Is Compromised.");
}
```

This already compares and correctly writes to the console a message letting the user know if their file has been compromised or if it still maintains its integrity.

Now we just have to connect everything up and run for the last time to check if everything is working. So, at the end of our Main method, call the PromptFileIntegrityCheck method.

```
Console.Clear();
Console.WriteLine(fileName + " exists with an initial checksum of " +
originalChecksum);
PromptFileIntegrityCheck(fileName, originalChecksum);
```

Then, in the PromptFileIntegrityCheck, at the end, call the CheckFileIntegrity method.

```
public static void PromptFileIntegrityCheck(string fileName, string originalChecksum)
{
    Console.WriteLine($"Press any key to check
{fileName}'s file integrity.");
    Console.WriteLine("The given file will be checked
for a match with the checksum: " + originalChecksum);
    Console.ReadKey();
    CheckFileIntegrity(fileName, originalChecksum);
}
```

And lastly, to loop right back, at the end of the CheckFileIntegrity method, call PromptFileIntegrityCheck again.

```
public static void CheckFileIntegrity(string fileName, string originalChecksum)
{
    string currentChecksum = GenerateCheckSumMD5(fileName);
    Console.Clear();
```

```
Console.WriteLine(currentChecksum.Equals(originalChecksum)
    ? "CHECKSUM MATCHES.
        File Integrity Is Observed."
    : "CHECKSUM MISMATCH.
        File Integrity Is Compromised.");
PromptFileIntegrityCheck(fileName, originalChecksum);
}
```

And the cycle is complete. Although we could still add a way to exit this last loop, I kind of don't like how I am "stuck" in here. In the PromptFileIntegrityCheck method, just before we call the CheckFileIntegrity method, add the following.

```
public static void PromptFileIntegrityCheck(string fileName, string
originalChecksum)
{
    Console.WriteLine($"Press any key to check
{fileName}'s file integrity.");
    Console.WriteLine("The given file will be checked
        for a match with the checksum: " + originalChecksum);
    ConsoleKeyInfo key = Console.ReadKey();
    if (key.Key == ConsoleKey.Escape)
    {
        Environment.Exit(0);
    }
    CheckFileIntegrity(fileName, originalChecksum);
}
```

Now, if the user presses the escape key, the application terminates. Better.

Let us try it out then! Let us see if our TestFile.txt still conserves its data integrity (Fig. 8.7).



Fig. 8.7 Our application's final result shows our TestFile.txt running through the Checksum checker test, resulting in "CHECKSUM MATCHES. File Integrity Is Observed"

As a final touch, we can include a little message to also let the user know that pressing the ESC key is an option.

```
public static void PromptFileIntegrityCheck(string fileName, string originalChecksum)
{
    Console.WriteLine($"Press any key to check
{fileName}'s file integrity. Or press ESC to exit.");
    Console.WriteLine("The given file will be checked
for a match with the checksum: " + originalChecksum);
    ConsoleKeyInfo key = Console.ReadKey();
    if (key.Key == ConsoleKey.Escape)
    {
    }
    Environment.Exit(0);
    CheckFileIntegrity(fileName, originalChecksum);
}
```

There we go, everything is working as it should: we have a perfectly running checksum checker ready to go to check some checksums. As we said, our goal for this project was to simply create an application that could be useful for internal use. While MD5 is somewhat secure, it is not applicable for use in risk-heavy environments since plenty of malicious attacks are capable of breaking the security it provides.

Nonetheless, there are more secure options out there, like, for example, SHA256, or PBKDF2 if passwords are being handled.

Migrating MD5 code to any of these is fairly easy most of the time, and if you want to maximize security, it might be worth investing some time into researching these other algorithms.

One way of accomplishing this could be using an interface that implements each algorithm variation.

Other than that, with the knowledge gained in this chapter, you should better understand file integrity, file handling, and even some cybersecurity.

And while we are mentioning it, why not delve a bit deeper into cybersecurity? This calls for another security-focused chapter! The next chapter will tackle a Secret Message Encryptor in C#. We will build a console application able to encrypt any message we pass in safely using the AES encryption algorithm. Another must-have for our portfolios.

After that, we will finally set foot into the outside world with APIs and Mail protocols. Let's get ready for the next chapter.

8.4.3 Source Code

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_ChecksumChecker

8.5 Summary

This chapter taught us:

- Data integrity discusses the accuracy and completeness of our data.
- There are two types of data integrity, physical and logical.
- Physical data integrity protects stored data from corruption or loss due to physical damage.
- Logical data integrity encompasses all aspects related to ensuring that stored data remains accurate and consistent even when subjected to logical operations.
- Data integrity is enforced by a series of integrity constraints or rules: Entity, Domain, Referential, and User-defined constraints.
- A Checksum is a specific string of characters generated by a hashing algorithm that represents the data on a file.
- Checksums are used to compare files for data corruption or loss.
- Hashing algorithms produce a fixed-size string of bits that represent the given data.
- Any minor change to the given data will change the final result of our hashing algorithm.
- The concept of the “Avalanche effect” refers to the unique characteristic of secure hashing algorithms that result in an entirely different hash, even if the data has only been altered slightly
- Hashing algorithms often use iterations (how many times we hash a password before we store it) and salting (the addition of a unique, random string of characters known only to the site to each password before it is hashed).
- There is no method to reverse a Hash since some data must be discarded in the process.
- Every cryptographic hash function is essentially the same as a non-cryptographic hash function, apart from the fact that cryptographic hash functions aim to guarantee a number of security properties.
- These are some properties that cryptographically secure hash functions require: strong collision resistance, preimage resistance, and second preimage resistance.
- There are multiple cryptographic hash functions, including SHA-0, SHA-1, SHA-2, SHA-3, MD5, PBKDF2, BCRYPT, SCRYPT, and ARGON2.

References

- Arias, D. (2021, February 25). Hashing in Action: Understanding bcrypt. Retrieved August 29, 2022 from [auth0.com: https://auth0.com/blog/hashing-in-action-understanding-bcrypt/](https://auth0.com/blog/hashing-in-action-understanding-bcrypt/)
- ChrisMcKee. (2022, March 3). BCrypt.Net-Next. Retrieved August 29, 2022 from [www.nuget.org: https://www.nuget.org/packages/BCrypt.Net-Next](https://www.nuget.org/packages/BCrypt.Net-Next)
- dariogriffo. (2021, December 27). SHA3.Net. Retrieved August 29, 2022 from [www.nuget.org: https://www.nuget.org/packages/SHA3.Net/](https://www.nuget.org/packages/SHA3.Net/)
- DevTut. (n.d., n.d. n.d.). Hash Functions. Retrieved August 29, 2022 from devtut.github.io: <https://devtut.github.io/csharp/hash-functions.html#complete-password-hashing-solution-using-pbkdf2>

- Egnyte. (2021, July 29). What is Data Integrity and Why is it Important? Retrieved August 29, 2022 from [www.egnyte.com](https://www.egnyte.com/guides/governance/data-integrity): <https://www.egnyte.com/guides/governance/data-integrity>
- Fitzgibbons, L. (2019, April n.d.). checksum. Retrieved August 29, 2022 from [www.techtarget.com](https://www.techtarget.com/searchsecurity/definition/checksum): <https://www.techtarget.com/searchsecurity/definition/checksum>
- freeCodeCamp. (2020, January 26). What is Hashing? How Hash Codes Work - with Examples. Retrieved August 29, 2022 from [www.freecodecamp.org](https://www.freecodecamp.org/news/what-is-hashing/): <https://www.freecodecamp.org/news/what-is-hashing/>
- GitBook. (n.d., n.d. n.d.). Argon2. Retrieved August 29, 2022 from [cryptobook.nakov.com](https://cryptobook.nakov.com/mac-and-key-derivation/argon2): <https://cryptobook.nakov.com/mac-and-key-derivation/argon2>
- GitBook. (n.d., n.d. n.d.). Bcrypt. Retrieved August 29, 2022 from [cryptobook.nakov.com](https://cryptobook.nakov.com/mac-and-key-derivation/bcrypt): <https://cryptobook.nakov.com/mac-and-key-derivation/bcrypt>
- kmaragon. (2022, May 20). Konscious.Security.Cryptography.Argon2. Retrieved August 29, 2022 from [www.nuget.org](https://www.nuget.org/packages/Konscious.Security.Cryptography.Argon2/): <https://www.nuget.org/packages/Konscious.Security.Cryptography.Argon2/>
- Layton, M. (2018, June 18). CORE. Retrieved August 29, 2022 from [github.com](https://github.com/MrMatthewLayton/CORE/tree/master/Core/Security/Cryptography): <https://github.com/MrMatthewLayton/CORE/tree/master/Core/Security/Cryptography>
- Nidecki, T. A. (2019, July 2). Insecure Default Password Hashing in CMSs. Retrieved August 29, 2022 from [www.acunetix.com](https://www.acunetix.com/blog/web-security-zone/insecure-default-password-hashing-cms/): <https://www.acunetix.com/blog/web-security-zone/insecure-default-password-hashing-cms/>
- Rattan, A. K. (2018, March 23). Data Integrity: History, Issues, and Remediation of Issues. Retrieved August 29, 2022 from [journal.pda.org](https://journal.pda.org/content/72/2/105): <https://journal.pda.org/content/72/2/105>
- Rusyaev, A. (2021, September 1). acryptohashnet. Retrieved August 29, 2022 from [github.com](https://github.com/AndreyRusyaev/acryptohashnet): <https://github.com/AndreyRusyaev/acryptohashnet>
- Synopsys. (2015, December 10). What are cryptographic hash functions? Retrieved August 29, 2022 from [www.synopsys.com](https://www.synopsys.com/blogs/software-security/cryptographic-hash-functions/): <https://www.synopsys.com/blogs/software-security/cryptographic-hash-functions/>
- Tutorials Point. (2022, March 10). What are the types of data integrity? Retrieved August 29, 2022 from [www.tutorialspoint.com](https://www.tutorialspoint.com/what_are_the_types_of_data_integrity.html): https://www.tutorialspoint.com/what_are_the_types_of_data_integrity.html
- viniciuschiele. (2017, February 2). Scrypt.NET. Retrieved August 29, 2022 from [www.nuget.org](https://www.nuget.org/packages/Scrypt.NET/): <https://www.nuget.org/packages/Scrypt.NET/>
- Webster, A. T. (2000, December 1). On the Design of S-Boxes. Retrieved August 29, 2022 from [link.springer.com](https://link.springer.com/chapter/10.1007/3-540-39799-X_41#citeas): https://link.springer.com/chapter/10.1007/3-540-39799-X_41#citeas



C# Cryptography

9

This Chapter Covers

- Why we should care about data encryption
- Differentiating between hashing and encrypting
- Finding out the role of encryption in cybersecurity
- Discovering the various encryption methods in C#
- Learning how to encrypt and decrypt data
- Developing a “Message Encryptor” application

In our interconnected world, the significance of cybersecurity is increasingly undeniable. Just think about the vast number of devices and systems we depend on daily: from personal smartphones to financial systems, healthcare databases, and national security infrastructures, all potential targets for cyberattacks. The repercussions of such attacks can range from inconvenient to catastrophic. For instance, imagine a situation where a major hospital’s patient records were breached. Confidential data could be exposed, medical procedures might be delayed or canceled, and patients’ lives could be put at risk. In a business context, a breach could result in millions, even billions, in financial losses, along with irreparable damage to a company’s reputation.

To stay ahead of evolving threats, cybersecurity requires continuous innovation and learning. This is not just a job for seasoned programmers—it is a call to action for aspiring developers who are passionate about safeguarding our digital world. As educators, it is our mission to inspire you, the next generation of innovators, to rise to this challenge. We believe you have the potential to devise creative, cutting-edge solutions to the cybersecurity problems of tomorrow.

In the previous chapter, we explored data hashing, a process that secures our data and shields it from loss. Now, we will broaden our scope to cybersecurity as a whole, beyond just data, to include all elements of our digital lives. To illustrate this, let us consider a

real-life scenario: an individual sends a confidential message to a friend, unaware that their communication is being intercepted by a cybercriminal. This situation highlights the pressing need for secure communication methods—precisely what we will be tackling in this chapter’s project.

The project for this chapter is the creation of a message encryptor. This console application will be designed to encrypt any message you input, ensuring secure transmission. While primarily a learning tool, this encryptor could serve as the foundation for more complex, real-world applications. As we build it, we will delve into the various encryption technologies available, their applications, benefits, and limitations.

By the end of this chapter, we intend for you to have a solid understanding of the often-intimidating realm of cybersecurity. But more than that, we hope to inspire you to continue exploring this field, to develop your expertise, and, ultimately, to contribute to the ongoing effort to secure our digital world.

So let us get straight into this ninth chapter of this book, starting with a small introduction. What is cybersecurity?

9.1 Cybersecurity

Why are we addressing cybersecurity in this chapter of a book primarily dedicated to small, varied knowledge applications within C# projects? Given the book’s scope, delving into cybersecurity and data security, as we did in the previous chapter, might appear overly ambitious or even extraneous. However, even if this will not be the path we will ultimately choose as our future careers, specializing in the tech industry and not learning about cybersecurity would be like specializing in the car industry without any knowledge about safety standards.

Additionally, as this generation’s teaching figures, we share a joint responsibility to teach the importance of everyone’s safety and well-being, be it as a chemical engineer with health precautions or as a software engineer with cybersecurity.

9.1.1 What Is Cybersecurity and Why Should We Care

As per definition, cybersecurity can be seen as the protection of Internet-connected systems from unauthorized access or theft. Also known as information technology (IT) security, this includes everything from viruses and malware to phishing attacks and computer hacking.

Since its limited early years in the late 1960s up until the current 2020s, cybersecurity and threats have been consistently present. Although the first known computer virus written by Bob Thomas called “Creeper” was conceptualized in 1971, it was not until the 1990s that cybersecurity saw global adoption due to the Internet’s ever-growing usage.

With that, in 1995, the Secure Socket Layer (SSL) was implemented to protect online activities such as transactions and was deemed as one of the first big steps forward against cyber threats.

We can learn more about the history of cybersecurity in the book *The Evolution of Viruses and Worms* by Thomas M. Chen and Jean-Marc Robert (direct link <https://ivan-lefou.fr/repo/madchat/vxdev/papers/avers/statmethods2004.pdf>).

From there, the number of malware samples that existed grew from just a few thousand to a massive 677.66 million programs as of March 2020, according to Statista (direct link <https://www.statista.com/statistics/680953/global-malware-volume/#:~:text=As%20of%20March%202020%2C%20the,surpass%20700%20million%20within%202020>).

But why should we care? The importance of cybersecurity cannot be understated. Cybersecurity is essential to protect sensitive data and prevent attacks in today's world, where more and more individuals and businesses are connected to the Internet.

As the number of Internet users and the amount of data available online continue to grow, so do the opportunities for cybercriminals to exploit vulnerabilities. That is why everyone needs to be conscious of the importance of cybersecurity and take steps to protect themselves and their data.

There are many reasons why cybersecurity is so critical. First, the Internet has become a vital part of our lives and the way we do business. Each day more of our personal and professional lives is visible online, which means that our private data and files are at risk of being accessed and stolen without us even noticing. Second, the sheer amount of online data makes it a prime target for cybercriminals. They can easily find and exploit vulnerabilities in our systems and use our data for their purposes. Finally, the consequences of a cyberattack can be devastating. We can lose important data, be exposed to identity theft, or suffer other financial or reputational damage. That is why it is so essential to protect ourselves and our data.

Much of this sounds similar to what we have learned in the previous chapter. Isn't this simply the same as datasecurity? Well yes, but actually no. Let us understand this topic better.

9.1.2 What Is the Difference Between Datasecurity and Cybersecurity

Datasecurity and cybersecurity are terms often used interchangeably, but they actually refer to slightly different concepts.

Although they both present similarities, several differences can be found. In short, datasecurity is focused on protecting digital and analog forms of information from unauthorized access or modification. While cybersecurity, on the other hand, protects anything in cyberspace, from data breaches to technological threats.

While both datasecurity and cybersecurity professionals work to protect sensitive information, they each prioritize different things. For example, datasecurity professionals may focus on ensuring that all data resources are properly encrypted so that only authorized

users can access them. Meanwhile, cybersecurity specialists might proactively scan for potential threats and vulnerabilities in order to prevent attacks before they happen.

There is an overlap between these two; nevertheless, they are distinct disciplines with different goals.

However, while it is true that there are distinctions between the two, we must also ponder the fact that the distinction between cybersecurity and datasecurity is becoming increasingly blurred over time.

This change is mainly because most companies only have cybersecurity experts on staff, who must take on both responsibilities at the end of the day. And these specialists may not have the necessary expertise to evaluate and protect data properly.

In response to this problem, many organizations are reconsidering their approach to security. Datasecurity is now seen as just as necessary as cybersecurity, resulting in rapid comprehensive data protection systems advancements, ensuring that their information is safe from potential threats.

Even though these two fields originated separately, they are quickly merging into a single larger discipline due to recent developments.

So then, what is it? Do we have to consider their differences, or is there no difference between them anymore?

Ultimately, although there is a difference in the core focus of each term, data on one side and cyber threats on the other, both strive to protect the user from potential attacks, be it through malware or someone walking into your facility and smashing a hard drive.

So get yourself an antivirus and back up that data!

Back to pure cybersecurity, though. Now that we know where the distinction and similarities between data and cybersecurity are, is there any distinction between pure cybersecurity categories?

9.1.3 Cybersecurity Categories

Cybersecurity, just like datasecurity, can be divided into several pillars that define each specialized sector within its spectrum. These seven pillars are network, cloud, endpoint, mobile, IoT, application, and the so-called Zero Trust security.

- Network security refers to security measures responsible for identifying and blocking network attacks. An example would be NAC, or Network Access Control, which supports access management through policy enforcement.
- Cloud security includes cybersecurity solutions, controls, policies, and services that help to protect an organization's entire cloud deployment. Generally, systems like IAM, or identity and access management, similar to NAC, a framework of policies that ensures that the right people get authorized access to the stored data, is used to ensure cloud security.
- Endpoint security is securing endpoints of end-user devices, like desktops, laptops, and mobile devices, from being attacked. Methods like EPPs or Endpoint protection

platforms, which work by inspecting files as they enter the network, are used for such purposes.

- Mobile security, through attacks like malicious apps, zero-day, phishing, and IM (Instant Messaging), is often one of the main points of unauthorized data access. MDM, or Mobile Device Management solutions, are often used to prevent such situations.
- With IoT security or Internet of Things security, we ensure the secure handling of those smart products that are often victims of sometimes fairly straightforward attacks. Network Segmentation is often used to prevent the entire network from being affected by a single point of compromise.
- Application security is a cybersecurity branch focused on protecting computer applications from vulnerabilities, tampering, and data loss. This includes input validation, output encoding, access controls, encryption, and application firewalls. If an attacker exploits vulnerabilities in an application, it could lead to sensitive data being accessed or taken over completely. Application firewalls help prevent these attacks by blocking any malicious traffic before it reaches the app. It also prevents bot attacks and stops malicious interactions with applications and APIs.
- Lastly, the Zero Trust security model ensures that data and authorized access are only granted to those that should have them, securing data from the outside and the inside, through micro-segmentation, monitoring, and enforcement of role-based access controls. This is especially important in an ever-growing remote work environment.

Lastly, we will turn our focus to application security. This key area involves addressing vulnerabilities that arise from insecure practices during the design, coding, and publishing of software or Web sites. Specifically, we will explore the process of converting data or information into an encrypted string to ward off unauthorized access.

As we delve into application security, it is worth noting the significance of the OWASP Top 10 (<https://owasp.org/www-project-top-ten/>). This is an industry-standard list compiled by the Open Web Application Security Project, which outlines the most critical security risks developers should be aware of. While we will not directly reference this list in our discussion, it is a highly recommended resource for developers aiming to enhance their understanding of secure coding practices.

9.2 Encryption in C#

So then, what about application security interests us this much? We just learned that application security covers a wide range of security methods that ensure our application's safety. One of those methods is what is called encryption, a method of cybersecurity often used for data transmission, the transfer of credit card information or identity details, for a person with the intent of accessing data without authorization could, for example, intercept the transfer process and either read what is being transferred or redirect the data to be received by a third party.

Through encryption, the intercepted message can become unintelligible and, therefore, unusable, but we can still decrypt our message to get our original data for further use. Let us get more into detail.

9.2.1 What Is Encryption

Encryption, as the name suggests, is a process of transforming regular text, known as plaintext, into an unreadable form, ciphertext. This transformation is achieved through mathematical cryptography algorithms.

In other words, encryption is similar to hashing in that it intends to change the input text into an unrecognizable version of itself.

However, unlike hashing, encrypted data can be decrypted—meaning that given the right key, it is possible to recover the original text from its encrypted form.

Thus, encryption provides a way to protect data from being accessed by unauthorized individuals, as only those who possess the decryption key will be able to read it.

This creates the question: Why do we need encryption in the first place? Moreover, how does it provide a valuable form of security if the encrypted can be decrypted? Sure we may find uses to secure authorized access. However, outside of closed environments, its security benefits seem nonexistent since anyone could get access to the key, unlocking countless pieces of data.

Although that is somewhat true, there is more to encryption than meets the eye. So, why do we encrypt?

9.2.2 Why Do We Encrypt?

While encryption does not directly convey a sense of security, there are a myriad of reasons how and why data encryption can prove to be useful. To really understand why we are learning about these concepts in this chapter, let us first learn about ten good reasons why data encryption is such an invaluable asset to anyone dealing with sensitive information.

- Encryption protects our data from being accessed by unauthorized individuals or systems, making it more difficult for cyber-criminals to steal sensitive information.
- Encrypting data can prevent breaches and protect our organization's confidential information from falling into the wrong hands.
- Encryption can help ensure compliance with privacy regulations, such as the General Data Protection Regulation (GDPR) or Health Insurance Portability and Accountability Act (HIPAA), by ensuring that only authorized individuals can access customer or patient data.
- Encrypted data is more secure in transit, ensuring that it cannot be intercepted and read by third parties en route to its destination. This reason is especially important for businesses that transmit large amounts of sensitive information electronically daily.

- Encrypting cloud-based services and applications provides an additional layer of security, helping to deter cyber-attacks and providing peace of mind knowing that critical information is protected even if there is a breach of the overall system.
- Suppose encrypted data needs to be backed up. In that case, this can be done securely, meaning only authorized individuals would have access should something happen to the original files (i.e., corruption or overwrite).
- The use of encryption can help deter cyber-attacks, as attackers may move on to easier targets if they know their efforts will not be successful against a well-protected organization. This could save our business time and money in the long run by avoiding costly clean-up and reputation damage associated with data breaches.
- In the event of a data breach, encrypted data may be less valuable to attackers and, therefore, less likely to lead to identity theft or other types of fraud—meaning businesses have a better opportunity of weathering the storm should an attack occur.
- Encryption can help prolong the useful life of digital information by protecting it from being accessed or corrupted by unauthorized individuals, meaning important files can be stored electronically for extended periods of time without deterioration.
- Implementing encryption shows that our organization takes data security seriously and is committed to protecting its customers' data or clients' information privacy, building trust between them and us.

But wait, although we did go over how encryption differs from hashing, all of this reminds me a lot of hashing, even down to why we even use it. Is there really a considerable difference between the two?

9.2.3 Difference Between Hashing and Encryption

Naturally, the process and reasons of existence for hashing and encryption share common points; however, not all are the same. Let us clear up the existing distinction between the two.

In short, encryption is a process that transforms readable data into an unreadable format. This unreadable data can only be converted into a readable format using a decryption key. On the other hand, hacking is a process that takes readable data and transforms it into a unique code that cannot be reverse-engineered back into the original data.

See, for example, the hashing of a password where the company that takes it does not need to be able to read the original password, and the encryption of banking information, where the original is required to be able to fulfill the transfer of funds.

Hashing is used to check whether the content has been changed or not. If the content is changed, the hash output will be different. Encryption is used to make sure that the data is confidential and secure. Only those who have the private key can change the encrypted text to plaintext.

Hashing and encryption serve different but similar purposes in computing systems. They both change data into a different format, making it tougher (but not impossible) for unauthorized users to reverse the process and read the original message.

Hopefully now that the differences are clear, or at the very least clearer, we can get into the types of encryption and which one we will use for this chapter's project.

9.2.4 Types of Encryption

Generally, we can differentiate two types of encryption: asymmetric and symmetric. In an overview, the differences between both are that asymmetric encryption is used for high-security low-performance scenarios. In contrast, symmetric encryption is used for high-performance lower-security scenarios. We will go over what that means in the following.

Asymmetric Encryption As we already know, encryption is the process of encrypting a piece of data with a key that we can later use to decrypt. However, in asymmetric encryption, the key used for encryption and the key used for decryption differ. That is to improve security further, sacrificing speed in the process since more complex mechanisms are needed to accomplish the decryption successfully, and computationally heavy mechanisms at that. Apart from offering us high-security standards, asymmetric encryption is also used when identity verification is required.

We often see asymmetric encryption in digital signatures for online documents and even transaction authorization for cryptocurrency in the blockchain.

RSA is a popular asymmetric encryption algorithm used for encryption. It makes use of a public key for encryption paired with a private key for decryption. This ensures secure data transmission. Public key infrastructure (PKI) is responsible for issuing and managing digital certificates that encrypt data.

Symmetric Encryption Symmetric encryption, in contrast, uses one secret symmetric key for both encryption and decryption. This encryption is usually used in credit card transactions, sensitive client data transfers, and any other transmission and storage of data that requires security measures. One common symmetric encryption algorithm is DES or Data Encryption Standards. This low-level encryption block cipher algorithm converts simple text into blocks of 64 bits and converts them to ciphertext by means of keys of 48 bits. Another is AES (Advanced Encryption Standard), the general standard in many often-used systems worldwide, including the US government.

And it is this latter we will be further focusing on for this chapter's project, precisely the advanced encryption standard. We will use that to learn more about its inner workings and the process of encrypting and decrypting data using AES and C#.

NOTE There is another! Hybrid Encryption. When both are used together in an attempt to get the best of both worlds. This one isn't used too often though, since its implementation can be quite hard.

9.3 The Process of Encrypting and Decrypting Data

The number of potential permutations for an encryption key determines its strength. Learning about the origins of encryption, we know that in the late twentieth century, Web developers used either 40-bit encryption, a key with 240 possible permutations, or 56-bit encryption. This level of security was enough for practically the entire century. However, hackers had successfully broken those keys by the end of that century.

This development in cyberattacks led to the development of a 128-bit system with the Advanced Encryption Standard (AES). The Advanced Encryption Standard, created in 2001 by the United States National Institute of Standards and Technology, made key lengths of 128, 192, and 256 bits available.

Naturally, the exponential security strength achieved by increasing the bit length made it a valid option for highly sensitive data handling, leading to today, where most banks, militaries, and governments use 256-bit encryption.

So how would we be able to make use of this encryption standard? How can this be applied to this chapter's project?

9.3.1 Advanced Encryption Standard

AES is a type of encryption that uses a block cipher to encrypt and decrypt data. It was developed by two Belgian cryptographers, Vincent Rijmen and Joan Daemen. They submitted a proposal to the National Institute of Standards and Technology (NIST) in November 1997, and NIST approved AES as a Federal Information Processing Standard (FIPS) in December 2001.

AES is more secure than DES (developed in the 1970s, one of the first commercial block ciphers) because it uses a larger key size and has a more complex structure. AES is also faster than DES, making it suitable for real-time applications.

This algorithm uses a block size of 128 bits, meaning it can encrypt data in blocks of 128 bits. AES can also use a 192-bit or 256-bit block size, but all AES implementations do not yet support these sizes.

As we previously mentioned, AES is also a symmetric-key algorithm, which means that the same key is applied to encrypt and decrypt data. AES uses a key size of 128, 192, or 256 bits, similar to its block size. The key size is the number of bits in the key, and, simply put, the larger the key size, the more secure the algorithm.

Now, for the question we have been waiting for: How do we encrypt data?

9.3.2 How Do We Encrypt Data

We will work with the AES encryption algorithm as we have already mentioned. Although its implementation can generally become difficult, we can consider ourselves lucky that .NET provides us with the System.Security.Cryptography namespace.

This namespace will allow us to use AES class, which represents the base class from which all implementations must inherit.

This will remove several layers of complexity from our code and simplify its readability.

So then, what do we need to implement still? By implementing the AES class, let us learn a bit about the general anatomy of an encryption method.

Take the following implementation as an example.

```
static byte[] Encrypt(string simpletext, byte[] key, byte[] iv)
{
    byte[] cipheredtext;
    using (Aes aes = Aes.Create())
    {
        ICryptoTransform encryptor = aes.CreateEncryptor(key, iv);
        using (MemoryStream memoryStream = new MemoryStream())
        {
            using (CryptoStream cryptoStream = new
CryptoStream(memoryStream, encryptor, CryptoStreamMode.Write))
            {
                using (StreamWriter streamWriter = new
StreamWriter(cryptoStream))
                {
                    streamWriter.WriteLine(simpletext);
                }

                cipheredtext = memoryStream.ToArray();
            }
        }
    }
    return cipheredtext;
}
```

We can see that we are starting with creating a method responsible for processing the encryption.

```
static byte[] Encrypt(string simpletext, byte[] key, byte[] iv)
```

This method must receive as parameters the following variables.

- **String plaintext.** The message that has to be encrypted, provided in its plain form, that is, not in encrypted form.

- **Byte[] Key.** The key used to encrypt the plaintext. That is the Key used both for encryption and decryption since AES, as we learned earlier, is symmetrically encrypted. This is passed in in Byte[] format, so an array of single bytes.
- **Byte[] IV.** The Initialization vector, also called the starting vector, is an array of bytes used to provide randomness or pseudo-randomness applied before its encryption. This can be used to ensure the safety of data even if the key is leaked. Generally, we see the IV in use to avoid repeating encryptions in case a fixed Key is used. If the same key encrypts two identical texts, they will result in the same encryption, giving possible attackers valuable information if no IV is used.

Let us now look at them in use within the method's body.

After creating a new variable to store our encrypted data in, we follow up with the using statement. We have not covered the using statement yet, so let us do so.

The using statement ensures that the object is read-only within the using block and cannot be modified or reassigned. A variable declared with a using declaration is read-only. If we use the following example, we will see how Visual Studio will warn us, “Cannot assign to ‘ms’ because it is a ‘using variable.’”

```
using MemoryStream memoryStream = new MemoryStream();  
memoryStream = new MemoryStream();
```

NOTE The using statement shall not be confused with the using directive we previously saw when implementing our namespaces. So these two are used first, to turn its presented variable to read-only, and second, to use types defined in that namespace without having to specify the namespace itself each time.

Additionally, the using statement ensures the correct disposal of a created variable since once that statement is left, the created variable is immediately set to be destroyed by garbage collection.

Back to the anatomy of our encryption method, we can see that this particular using statement creates an AES class variable using the A.E.S.Create() method. This method creates a cryptographic object used to run the symmetric algorithm.

```
using (Aes aes = Aes.Create()) {}
```

Within, we generate an encryptor using our provided key and IV. These are stored in an ICryptoTransform variable using the A.E.S.CryptoEncryptor() method. This creates the basis for our encryption process.

```
ICryptoTransform encryptor = aes.CreateEncryptor(key, iv);
```

Right after, we create a new memory stream through the MemoryStream class. MemoryStream in C# programs allows us to use in-memory byte arrays or other data as though they are streams. So we can, instead of storing data in files, store data in memory

and access them directly from there. Memory streams can reduce the need for temporary buffers and files in an application and improve performance. Generally speaking, we should reserve the use of memory streams for situations where they are truly needed. This approach is only more efficient when dealing with files that are too large to manage by other means, which, as we will discuss shortly, applies to our case.

```
using (MemoryStream memoryStream = new MemoryStream()) {}
```

Once a memory stream is created, we must create a CryptoStream instance. It defines a stream that links data streams to cryptographic transformations. At least, this is how the official documentation states it. However, this could be misleading. If we assume that its third argument is CryptoStreamMode.Read:

```
using (CryptoStream cryptoStream = new CryptoStream(memoryStream,
    encryptor, CryptoStreamMode.Write))
```

However, if the third argument is CryptoStreamMode.Write, as in our case, the description should be changed to “The stream on which the result of the cryptographic transformation is written to.”

There is no need to dive much deeper into this, it is just a quick side note. So simply put, it allows read/write for both Encrypt and Decrypt.

Here the memory stream and encryptor were used, and the crypto stream was set to write since we attempted to encrypt.

Now, we simply have to write our plaintext onto the generated crypto stream, like so:

```
using (StreamWriter streamWriter = new StreamWriter(cryptoStream))
{
    streamWriter.WriteLine(simpletext);
}
```

And get our encrypted plaintext back by retrieving it from our memory stream object.

```
cipheredtext = memoryStream.ToArray();
```

Then return our encrypted text to the caller. With that, we have successfully implemented an AES encryption method ready to be used. Now, we need to know how to get our text back, so let us see how decryption is done.

9.3.3 How Do We Decrypt Data

Once we understand encryption, decryption becomes completely straightforward. Nonetheless, as a reminder, let us go quickly through each step and see where the differences lie.

We use this decryption method as an example:

```
static string Decrypt(byte[] cipheredtext, byte[] key, byte[] iv)
{
    string simpletext = String.Empty;
    using (Aes aes = Aes.Create())
    {
        ICryptoTransform decryptor = aes.CreateDecryptor(key, iv);
        using (MemoryStream memoryStream = new MemoryStream(cipheredtext))
        {
            using (CryptoStream cryptoStream = new
CryptoStream(memoryStream, decryptor, CryptoStreamMode.Read))
            {
                using (StreamReader streamReader = new
StreamReader(cryptoStream))
                {
                    simpletext = streamReader.ReadToEnd();
                }
            }
        }
    }
    return simpletext;
}
```

We can immediately see how both are practically the same regarding implementation. Let us go over each step to find the differences.

First, we naturally need a method to call for the decrypter. This takes in the encrypted text, the key used to encrypt it, and the IV used to initialize the encryption.

```
static string Decrypt(byte[] cipheredtext, byte[] key, byte[] iv) {}
```

Now, we create a string variable to store our decrypted text. This is already understandably different since now we want to get plaintext as a final result.

```
string simpletext = String.Empty;
```

Then we, just like the encryptor, create an AES class object.

```
using (Aes aes = Aes.Create())
```

And this time, use the method A.E.S.CreateDecryptor() method to create a symmetric decryptor object instead of an encryptor object like in our previous method.

```
ICryptoTransform decryptor = aes.CreateDecryptor(key, iv);
```

Then, we create a memory stream again, passing in the cipher text as a variable for the memory stream's buffer.

```
using (MemoryStream memoryStream = new MemoryStream(cipheredtext))
```

Then, back to the crypto stream, we create a new instance passing in the memory stream, the decryptor object, and setting it this time to CryptoStreamMode.Read.

```
using (CryptoStream cryptoStream =
new CryptoStream(memoryStream, decryptor, CryptoStreamMode.Read))
```

Then simply read the crypto stream to retrieve our plaintext again.

```
using (StreamReader streamReader = new StreamReader(cryptoStream))
{
    simpletext = streamReader.ReadToEnd();
}
```

In the end, simply return our plaintext to the caller.

```
return simpletext;
```

Essentially, that is it. Now we can pass in a text to encrypt, and a cipher to decrypt. We should be ready to turn these two methods into a fully fledged project able to do both tasks. So, let us do so.

9.4 The Message Encryptor

Now that we are at the project development section, we can see our newly acquired encryption skills finally in action. In the last section, we learned how to implement simple encryption and decryption methods. However, we have yet to see them in action.

With this final chapter for our advanced section, we conclude with a valuable lesson in cybersecurity and some new skills we can show off and even vastly improve upon to develop real-world applications for real-world companies.

So then, let us check out what this chapter's project is all about.

9.4.1 The Project

For this chapter's project, we want to develop a console app capable of taking in a user input text to return an encrypted cipher text. In this case, if no key is given, a random one is generated and given, together with the randomly generated initialization vector.

The intention is to use the same key and IV to be able to decrypt the encryption and get our text back successfully.

With this functionality prepared, we can see how we could now use this app to, as an example, automatize the encryption of a messaging service or implement it in any app we develop where some security is needed. Although it seems to be limited just by itself, nothing truly limits it.

By the way, in the end, we will throw in a little message for you all to decipher. Let us see if those algorithms work! But first, let us get to work.

9.4.2 Our Code

As usual, we must start by creating a class Program with a static Main method. To maintain consistency throughout the continuing chapters, we will maintain this code as our starting point.

So then, create a new project, clean it, and add the following lines.

```
class Program
{
    public static void Main(string[] args)
    {

    }
}
```

Now we must start by introducing the user to our application. Let us kick this off simply with a greeting, like so:

```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to the Message Encryptor");
}
```

That was for the warm-up. What we need now for this app is first to ask the user what the desired action he would like to take is. Since we know that our app will be able to either encrypt or decrypt, we must ask him which of the two he wants to do. For that, we will make a small function that will take care of that, called PromptMessageEncryptor(). So right after our Main() method, add the following.

```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to the Message Encryptor");
}
```

```
public static void PromptMessageEncryptor()
{}
```

This first method will be responsible for prompting the user and handling the responses. However, it will use three other methods to process the response. Let us see which ones.

First, we will go back to our previous chapters and borrow our YNQuestion() method since it works well for what we need here. This means that our decision prompt must let the user know that he has to type either Y or N.

```
public static void PromptMessageEncryptor()
{
    Console.WriteLine("Would you want to Encrypt (Y)
        or Decrypt (N) a message? ( Y | N )");
}
```

Then throw in our method to take care of the user's response.

```
public static void PromptMessageEncryptor()
{
    Console.WriteLine("Would you want to Encrypt (Y)
        or Decrypt (N) a message? ( Y | N )");
    bool encryptOrDecrypt = YNQuestion();
}
```

Naturally, we do not yet have it implemented, but we will do so right now.

NOTE We make use of the YNQuestion() method for the sake of reusing existing code and improving our workflow. However, we could surely make a variation of it by duplicating that method and using it here as a EDQuestion() ("E" for Encrypt and "D" for Decrypt) method or something similar. We will use our existing code just to simplify our process.

Before adding our YNQuestion() method in, let us finish what is left for the PromptMessageEncryptor() method. Once a response is received, the decision is used to execute either a method that will start the encryption process or a method that will start the decryption process.

```
public static void PromptMessageEncryptor()
{
    Console.WriteLine("Would you want to Encrypt (Y)
        or Decrypt (N) a message? ( Y | N )");
    bool encryptOrDecrypt = YNQuestion();
    if (encryptOrDecrypt) TakeMessageToEncrypt();
    else TakeEncryptionToDecrypt();
}
```

Once we reach this point, we can incorporate the missing methods into our program. Starting with our YNQuestion method mentioned above (), we must take it from our previous projects and add it after the PromptMessageEncryptor() method. Here is the method in the case that going back to our previous projects does not sound like something we want to do.

```
public static bool YNQuestion()
{
    string userInput;
    do
    {
        userInput = Console.ReadLine() ?? string.Empty;
        userInput = userInput.ToUpper();
    }
    while (userInput != "Y" && userInput != "N");
    return userInput == "Y";
}
```

If we want, we can already start testing by commenting out the last if statement within our PromptMessageEncryptor() method so that no errors remain. Let us do so to make sure we are on the right track. And while we are here, let us write to console our result as well.

```
public static void PromptMessageEncryptor()
{
    Console.WriteLine("Would you want to Encrypt (Y)
        or Decrypt (N) a message? ( Y | N )");
    bool encryptOrDecrypt = YNQuestion();
    /*if (encryptOrDecrypt) TakeMessageToEncrypt();
    else TakeEcryptionToDecrypt();*/
    Console.WriteLine(encryptOrDecrypt);
}
```

Also do not forget to call our PromptMessageEncryptor() method in the Main() method.

```
Console.WriteLine("Welcome to the Message Encryptor");
PromptMessageEncryptor();
```

Great, now we need the TakeMessageToEncrypt() and TakeEcryptionToDecrypt() methods. Let us start by going the route of encryption.

Before that, simply leave the PromptMessageEncryptor() method as it was before.

```
public static void PromptMessageEncryptor()
{
    Console.WriteLine("Would you want to Encrypt (Y)
        or Decrypt (N) a message? ( Y | N )");
}
```

```
    bool encryptOrDecrypt = YNQuestion();
    if (encryptOrDecrypt) TakeMessageToEncrypt();
    else TakeEncryptionToDecrypt();
}
```

Now, below, let us add our `TakeMessageToEncrypt()` method in.

```
public static void TakeMessageToEncrypt()
{ }
```

This method and the other method for decrypting will manage the entire process within them. We will have to add more methods to cover specifics, but we can see these two as the encryption and decryption managers. This means we need to cover the complete execution flow for each section, starting with a prompt asking for the text that the user wants to be encrypted.

```
public static void TakeMessageToEncrypt()
{
    Console.WriteLine("Please introduce a message to encrypt.");
    string userInput = Console.ReadLine() ?? string.Empty;
}
```

Here we fetched the input the user wants to encrypt.

NOTE Notice that currently, this app has the limitation that only one-line inputs are allowed. I might challenge you to implement multi-line inputs as well as entire files. Because, after all, this is the last chapter of the advanced part, and we should already be advanced C# developers. But let us keep that for the end of this chapter. First, let us continue maintaining this limitation and call it a feature.

Next, we need to know if the user wants to give his encryption key and initialization vector. We can do that by prompting the user with the question and processing his answer with our `YNQuestion()` method.

```
public static void TakeMessageToEncrypt()
{
    Console.WriteLine("Please introduce a message to encrypt.");
    string userInput = Console.ReadLine() ?? string.Empty;
    Console.WriteLine("Would you want to provide your own
(Y) encryption kwy and initialization vector?
If not (N), a new one will be created. ( Y | N )");
    bool provideKeyAndIV = YNQuestion();
}
```

This will now, like in our PromptMessageEncryptor() method, leave us with a split in our execution where we need to either get the user's key and IV or generate a new random key and IV. For that, once more, we need to create two new methods: the FetchUserEncryptionKeyAndIV() method and the GenerateNewEncryptionKeyAndIV() method.

Starting with the assumption that the user does not have a key and IV, we can implement the GenerateNewEncryptionKeyAndIV() first.

```
public static void GenerateNewEncryptionKeyAndIV()  
{ }
```

Alright, first, we should consider clearing the console to keep everything professional-looking. So let us add that quickly.

```
public static void GenerateNewEncryptionKeyAndIV()  
{  
    Console.Clear();  
}
```

Now, we can get into the fun parts. So how would we create a key and an IV? As we have seen before in our encryption breakdown, we were getting the key and IV passed in as a parameter, meaning we were not generating them at that point. Well, that is only partly true.

At the moment of creating the cryptographic object, we are also creating a key and IV automatically. We simply were not directly using them. This means that generating a key and IV is just a matter of creating a cryptographic object and getting its generated key and IV. In other words, this would be enough.

```
public static void GenerateNewEncryptionKeyAndIV()  
{  
    Console.Clear();  
    using (Aes aes = Aes.Create())  
    {  
        permKey = aes.Key;  
        permIV = aes.IV;  
    }  
}
```

That way, we now have a new and unique key and IV. To eliminate any errors, we simply have to implement the variables used and remember to add the using directive for the System.Security.Cryptography namespace.

Starting with the latter.

```
using System.Security.Cryptography;
class Program
{
```

And now, the variables.

```
using System.Security.Cryptography;
public class Program
{
    static byte[]? permKey;
    static byte[]? permIV;
```

Perfect, we have our key and IV generator ready to go. Let us make it so the user can add his key and IV by implementing our FetchUserEncryptionKeyAndIV() method. So right after, or before, the GenerateNewEncryptionKeyAndIV() method, add the following.

```
public static void FetchUserEncryptionKeyAndIV()
{ }
```

Here, we must also start by clearing the console.

```
public static void FetchUserEncryptionKeyAndIV()
{
    Console.Clear();
}
```

And then, we need to clear both the permanent key variables we created and create temporary user key and IV variables. We will go over the logic of why shortly.

```
public static void FetchUserEncryptionKeyAndIV()
{
    Console.Clear();
    permKey = null;
    string? userEK = string.Empty;
    permIV = null;
    string? userIV = string.Empty;
}
```

That should do it. Now, we will use a do/while loop.

```
public static void FetchUserEncryptionKeyAndIV()
{
    Console.Clear();
    permKey = null;
    string? userEK = string.Empty;
```

```
permIV = null;
string? userIV = string.Empty;
do
{
}
while (permKey == null || userEK == string.Empty || permIV == null ||
userIV == string.Empty);
}
```

This loop will simply repeat the questions we will include inside until all variables are set, meaning that if no input is given, it will start over. Here we can see why we had to clear the permanent variables and initialize the temporal ones the way we did, to make sure that we successfully get these new values. It will get more evident once we complete the containing code.

So, what do we need to ask here? First, we need to ask for the encryption key to be used, so prompt the question to the user, and take his input to set the userEK variable.

```
do
{
    Console.WriteLine("Please introduce a valid Encryption key.");
    userEK = Console.ReadLine() ?? string.Empty;
}
while (permKey == null || userEK == string.Empty || permIV == null ||
userIV == string.Empty);
```

Since the “?? String.Empty” will set the userEK variable as string.Empty in case no input is given, this will cover our “validation,” ensuring that at least something is given.

Naturally, this can be expanded to proper key validation. However, for our purposes, that will suffice.

Then, as we have seen earlier, our permKey variable is a nullable byte array, so we need to take the user’s input currently in string format and convert it to a compatible format. Luckily C# provides us with the Convert class method Convert.FromBase64String(). To use it, we just have to set the permKey variable with the return of this method passing in our user input as a parameter.

```
do
{
    Console.WriteLine("Please introduce a valid Encryption key.");
    userEK = Console.ReadLine() ?? string.Empty;
    permKey = Convert.FromBase64String(userEK);
}
while (permKey == null || userEK == string.Empty || permIV == null ||
userIV == string.Empty);
Console.Clear();
```

There we go. Now, we need to repeat the same for the initialization vector, like this:

```
do
{
    Console.WriteLine("Please introduce a valid Encryption key.");
    userEK = Console.ReadLine() ?? string.Empty;
    permKey = Convert.FromBase64String(userEK);

    Console.WriteLine("Please introduce a valid
        Encryption Initialization Vector.");
    userIV = Console.ReadLine() ?? string.Empty;
    permIV = Convert.FromBase64String(userIV);
}
while (permKey == null || userEK == string.Empty || permIV == null ||
userIV == string.Empty);
```

While we do this, we can clear our printed texts from the console since the user no longer needs to see them. And we can also add a little text at the end in case any of the inputs are wrong, and the loop needs resetting.

```
Console.WriteLine("Please introduce a valid Encryption key.");
userEK = Console.ReadLine() ?? string.Empty;
permKey = Convert.FromBase64String(userEK);

Console.WriteLine("Please introduce a valid
    Encryption Initialization Vector.");
userIV = Console.ReadLine() ?? string.Empty;
permIV = Convert.FromBase64String(userIV);
Console.Clear();
Console.WriteLine("Key or IV is not valid.");
```

If the loop repeats, this will result in a new message appearing at the top stating that “**Key or IV is not valid**” before letting the user retype them.

Let us test this immediately and see if it is working well. For that, temporarily comment out the call for the still missing TakeEcryptionToDecrypt() method in the PromptMessageEncryptor() method.

```
public static void PromptMessageEncryptor()
{
    Console.WriteLine("Would you want to Encrypt (Y)
        or Decrypt (N) a message? ( Y | N )");
    bool encryptOrDecrypt = YNQuestion();
    if (encryptOrDecrypt) TakeMessageToEncrypt();
    /* else TakeEcryptionToDecrypt(); */
}
```

And run the app to see what we get. We will try to get to the point of fetching the user key and IV and input nothing as the IV but input something as the key (anything will be seen as valid, so proceed to button smash).

NOTE Ensure it is within the following: A–Z, a–z, 0–9, +, /, and =. It must also be a multiple of 4. Be careful with spaces!

Here are example “keys” that we could use: “godilovethisbook” or “aosiuufdnaosufbao.”

After doing so, we see the following (Fig. 9.1).

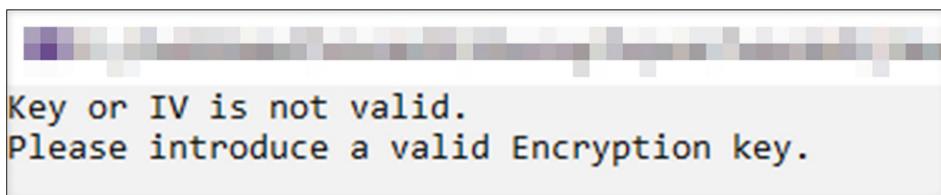


Fig. 9.1 After inputting a random text as the key and nothing as the IV, the loop repeated, detecting that both userIV and permIV are still empty and “not valid.” This is the result of our actions

We got around to repeating the do/while loop once and got the “**Key or IV is not valid**” text printed. However, as we may have noticed, even though we said that we would type in the key correctly but leave the IV blank, it asks us to input the key again. Ideally, it should only ask us for the wrong one. Luckily, fixing that is simple.

```
do
{
    if (permKey == null || userEK == string.Empty)
    {
        Console.WriteLine("Please introduce a valid Encryption
key.");
        userEK = Console.ReadLine() ?? string.Empty;
        permKey = Convert.FromBase64String(userEK);
    }
    if (permIV == null || userIV == string.Empty)
    {
        Console.WriteLine("Please introduce a valid
Encryption Initialization Vector.");
        userIV = Console.ReadLine() ?? string.Empty;
        permIV = Convert.FromBase64String(userIV);
    }
    Console.Clear();
    Console.WriteLine("Key or IV is not valid.");
}
```

```
        while (permKey == null || userEK == string.Empty || permIV ==  
null || userIV == string.Empty);
```

Using a set of two simple if statements, we cover that as well. If one of the two is correct, it will not ask for it again. Simple.

With that, we got the keys and IVs ready for usage. Now we can get into the encrypting, so let us do so. Since we already know about the encryption method we are going to be using by learning about it before starting our project, we are only going over it quickly to not bore you with repeated lessons. We can always go back up if anything does not make sense.

So we need to set up an encryption method. We did that by first creating a method called Encrypt() that took a total of three parameters: the text to encrypt, the key to use, and the initialization vector to use.

```
static byte[] Encrypt(string simpletext, byte[] key, byte[] iv)  
{ }
```

Within, we want to create a variable responsible for storing our ciphered text and create a new cryptographic object, like so:

```
static byte[] Encrypt(string simpletext, byte[] key, byte[] iv)  
{  
    byte[] cipheredtext;  
    using (Aes aes = Aes.Create())  
    {  
    }  
}
```

Inside the using statement, we need to create an encryptor using our key and IV passed in as parameters. This will create a symmetric encryptor agent that will use the given key and IV for encryptions which we will later pass into the crypto stream.

```
using (Aes aes = Aes.Create())  
{  
    ICryptoTransform encryptor = aes.CreateEncryptor(key, iv);  
}
```

But before creating the crypto stream, we will need a memory stream.

```
using (Aes aes = Aes.Create())  
{  
    ICryptoTransform encryptor = aes.CreateEncryptor(key, iv);  
    using (MemoryStream memoryStream = new MemoryStream())
```

```
{  
}  
}  
}
```

And now, using the memory stream instance and the encryptor object, we can create a crypto stream instance and set it to write since we are encrypting.

```
ICryptoTransform encryptor = aes.CreateEncryptor(key, iv);  
using (MemoryStream memoryStream = new MemoryStream())  
{  
    using (CryptoStream cryptoStream =  
        new CryptoStream(memoryStream, encryptor, CryptoStreamMode.Write))  
    {  
    }  
}
```

Now using our crypto stream instance, we create a stream writer instance.

```
using (CryptoStream cryptoStream = new CryptoStream(memoryStream,  
encryptor, CryptoStreamMode.Write))  
{  
    using (StreamWriter streamWriter = new StreamWriter(cryptoStream))  
    {  
    }  
}
```

Write the plaintext to the stream writer instance.

```
using (StreamWriter streamWriter = new StreamWriter(cryptoStream))  
{  
    streamWriter.WriteLine(simpletext);  
}
```

And finally, read the memory stream now that it has been written to with the stream writer to retrieve our encrypted word.

```
using (StreamWriter streamWriter = new StreamWriter(cryptoStream))  
{  
    streamWriter.WriteLine(simpletext);  
}  
cipheredtext = memoryStream.ToArray();
```

Also do not forget to return our final ciphered text.

```
    streamWriter.WriteLine(simpletext);  
}
```

```
        cipheredtext = memoryStream.ToArray();
    }
}
}
return cipheredtext;
}
```

So, in the end, we went through the following steps. Let's see the steps in a linear format:

- The Encrypt method is created, which takes three parameters: text to encrypt, key to use, and initialization vector.
- A variable cipheredtext is created for storing the encrypted text.
- A new AES object is created for cryptographic operations.
- Using the AES object, an ICryptoTransform encryptor is created using the key and IV passed as parameters.
- A MemoryStream object is created.
- Using the MemoryStream object and encryptor, a CryptoStream is created, set to write mode.
- A StreamWriter is created using the CryptoStream.
- The plaintext is written to the StreamWriter.
- The StreamWriter is then used to write to the MemoryStream.
- The MemoryStream is converted to an array, storing the result in cipheredtext.
- The cipheredtext is returned by the Encrypt method.

Perfect. That is an encryption method done. Let us now use it to complete the encryption side of this project.

Back in the TakeMessageToEncrypt() method, we now need to create a variable to hold the return of our Encrypt() method.

```
public static void TakeMessageToEncrypt()
{
    Console.WriteLine("Please introduce a message to encrypt.");
    string userInput = Console.ReadLine() ?? string.Empty;
    Console.WriteLine("Would you want to provide your own
(Y) encryption key and initialization vector?
If not (N), a new one will be created. ( Y | N )");
    bool provideKeyAndIV = YNQuestion();
    if (provideKeyAndIV) FetchUserEncryptionKeyAndIV();
    else GenerateNewEncryptionKeyAndIV();

    byte[] encrypted = Encrypt();
}
```

And as our passed-in parameters, we use the `userInput` from earlier and our `permKey` and `permIV` variables.

```
byte[] encrypted = Encrypt(userInput, permKey, permIV);
```

To resolve the nullable warning we have right now, seen by the `permKey` and the `permIV` being underlined in green, we can simply add an exclamation mark at the end of each. This tells it that we are sure these will never be null.

```
byte[] encrypted = Encrypt(userInput, permKey!, permIV!);
```

Next, we need to use the result to show them to the user. Our app will show the encrypted message and the key/IV pair. Probably not the best for a published and running app, but it should be perfect for our purely demonstrative purposes. For that to be possible, we must first convert them to a readable format. This can be accomplished with the `ToBase64String()` method from the `Convert` class. So doing that for all three values, we should have something like this:

```
if (provideKeyAndIV) FetchUserEncryptionKeyAndIV();
else GenerateNewEncryptionKeyAndIV();
byte[] encrypted = Encrypt(userInput, permKey!, permIV!);

string base64 = Convert.ToBase64String(encrypted);
string keyBase64 = Convert.ToBase64String(permKey!);
string ivBase64 = Convert.ToBase64String(permIV!);
```

With all three ready, we can start thinking about the presentation. Start by clearing the console again, so we have a clean slate to work in.

```
string ivBase64 = Convert.ToBase64String(permIV!);
Console.Clear();
```

At the top, we want to show off the encrypted text. Since we already have it converted to string, we only have to present it in a `Console.WriteLine()` method, paired with a small message informing of the success.

```
Console.Clear();
Console.WriteLine("Your text has been encrypted.");
Console.WriteLine($"Encrypted data: {base64}");
```

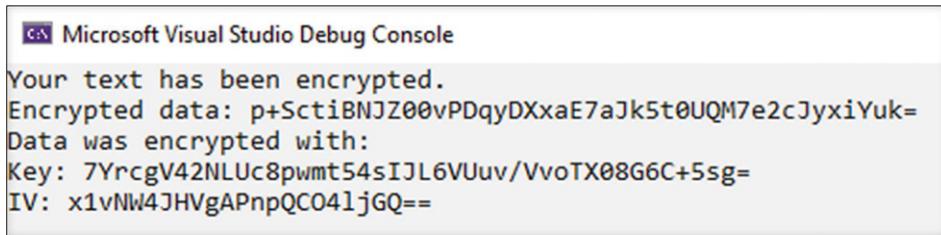
Now, we want to show the user the key and the IV used in the encryption, like so:

```
Console.WriteLine($"Encrypted data: {base64}");
Console.WriteLine($"Data was encrypted with:");
```

```
Console.WriteLine($"Key: {keyBase64}");
Console.WriteLine($"IV: {ivBase64}");
```

That should show us everything we need. To wrap up the encryption portion of our project, we only need to prepare the end of our application with a standard “Press ANY KEY to start over, or press the ESC key to exit.” But before we do that, let us try our app to see if everything is going as planned.

After running the application and inputting a text, we got the following result (Fig. 9.2).



The screenshot shows the Microsoft Visual Studio Debug Console window. The output text is:

```
Microsoft Visual Studio Debug Console
Your text has been encrypted.
Encrypted data: p+SctiBNJZ00vPDqyDXxaE7aJk5t0UQM7e2cJyxiYuk=
Data was encrypted with:
Key: 7YrcgV42NLUc8pwmt54sIJL6VUuv/VvoTX08G6C+5sg=
IV: x1vNW4JHVgAPnpQCO41jGQ==
```

Fig. 9.2 Our result after encrypting a given text and correctly displaying the ciphered text, the key, and the initialization vector

Seems to run just fine! Here you got the transcription in case you want to try and decrypt it later...

```
Your text has been encrypted.
Encrypted data: p+SctiBNJZ00vPDqyDXxaE7aJk5t0UQM7e2cJyxiYuk=
Data was encrypted with:
Key: 7YrcgV42NLUc8pwmt54sIJL6VUuv/VvoTX08G6C+5sg=
IV: x1vNW4JHVgAPnpQCO41jGQ==
```

But now, let us continue where we left off.

We need a conclusion and a way to reset or exit the app. For that, prompt the user first so that he knows what to do.

```
Console.WriteLine($"Key: {keyBase64}");
Console.WriteLine($"IV: {ivBase64}");
Console.WriteLine("Press ANY KEY to start over,
or press the E.S.C. key to exit.");
```

And now, we need a way to retrieve the user’s key press. If we remember from the beginning chapters, the method `ReadKey()` from the `Console` class did just that. So let us use this to get and store the specific key press in a `ConsoleKeyInfo` variable.

```
Console.WriteLine("Press ANY KEY to start over,
or press the E.S.C. key to exit.");
ConsoleKeyInfo key = Console.ReadKey();
```

Why did we store it? That is to determine if the key press was the ESC key or any other key. That way, we can exit the app or return to the beginning. And to do those, starting with exiting the app, we need to check if it is the correct key, and if yes, use the `Exit()` method from the `Environment` class (a class that provides information about, and means to manipulate, the current environment).

```
ConsoleKeyInfo key = Console.ReadKey();
if (key.Key == ConsoleKey.Escape)
{
    Environment.Exit(0);
}
```

Great, and now, if the key is not the ESC key, clear the console and call the `PromptMessageEncryptor()` method again to restart the app.

```
if (key.Key == ConsoleKey.Escape)
{
    Environment.Exit(0);
}
Console.Clear();
PromptMessageEncryptor();
```

And that is it for the encryption section. We only need to implement the decryption components to finalize the app, and we are good to go. Before doing so, let us try our app and see if this side of the loop works right (Fig. 9.3).



Fig. 9.3 After finishing the encryption portion of the project, we reach an end point where we can either repeat the process or exit the application. And after testing, it works correctly, as expected

It looks as expected. We successfully reached our application endpoint and can either reset or escape.

Perfect, now for the decryption. Since we already set up the encryption process, the decryption process is mostly the same. Let us go over its implementation.

Starting with the method that will manage the decryption process, the `TakeEcryptionToDecrypt()` method.

```
public static void TakeEcryptionToDecrypt()
{ }
```

Here, just like our previous method, we want to get from the user the encryption to decrypt, only this time, we will need to convert the given string to a byte array so we can use it for our method later. As we already know, this can be done with the `FromBase64String()` method.

```
public static void TakeEcryptionToDecrypt()
{
    Console.WriteLine("Please introduce an encryption to decrypt.");
    string userInput = Console.ReadLine() ?? string.Empty;
    byte[] bytes = Convert.FromBase64String(userInput);
}
```

Another difference is our next step. Since using the right key and initialization vector is needed for the correct decryption, we cannot simply let the app generate a new key and IV for that purpose. We now must ask the user for a key and IV next.

```
public static void TakeEcryptionToDecrypt()
{
    Console.WriteLine("Please introduce an encryption to decrypt.");
    string userInput = Console.ReadLine() ?? string.Empty;
    byte[] bytes = Convert.FromBase64String(userInput);
    Console.WriteLine("Please introduce a valid" +
        "Encryption Initialization Vector.");
    FetchUserEncryptionKeyAndIV();
}
```

And now, like before, it is time for the `Decrypt()` method. So let us go and set that up.

Start with the method, with the parameters needed being the ciphered text, the key used, and the initialization vector used.

```
static string Decrypt(byte[] cipheredtext, byte[] key, byte[] iv)
{ }
```

Now add our string variable to store the final decrypted text and create a cryptographic object.

```
static string Decrypt(byte[] cipheredtext, byte[] key, byte[] iv)
{
    string simpletext = String.Empty;
    using (Aes aes = Aes.Create())
    {
    }
}
```

Alright, as we learned in Sect. 9.3.3 (How Do We Decrypt Data?), we now need a decryptor. Create one using the passed-in key and IV and store it in an ICryptoTransform variable.

```
static string Decrypt(byte[] cipheredtext, byte[] key, byte[] iv)
{
    string simpletext = String.Empty;
    using (Aes aes = Aes.Create())
    {
        ICryptoTransform decryptor = aes.CreateDecryptor(key, iv);
    }
}
```

Next, we need the memory stream and pass in the ciphered text when instancing, like so:

```
static string Decrypt(byte[] cipheredtext, byte[] key, byte[] iv)
{
    string simpletext = String.Empty;
    using (Aes aes = Aes.Create())
    {
        ICryptoTransform decryptor = aes.CreateDecryptor(key, iv);
        using (MemoryStream memoryStream = new MemoryStream(cipheredtext))
        {
        }
    }
}
```

And within the memory stream instantiation, we instantiate a crypto stream set to CryptoStreamMode.Read.

```
using (MemoryStream memoryStream = new MemoryStream(cipheredtext))
{
    using (CryptoStream cryptoStream = new CryptoStream(memoryStream,
decryptor, CryptoStreamMode.Read))
    {
    }
}
```

And instantiate a stream reader to retrieve the decrypted text from the crypto stream.

```
using (CryptoStream cryptoStream =
new CryptoStream(memoryStream, decryptor, CryptoStreamMode.Read))
{
```

```

        using (StreamReader streamReader = new StreamReader(cryptoStream))
    {
        simpletext = streamReader.ReadToEnd();
    }
}

```

And finally, return our decrypted text.

```

        simpletext = streamReader.ReadToEnd();
    }
}
}
return simpletext;
}

```

That is the decryption method! Good, now, on to the decryption process in our TakeEcryptionToDecrypt() method. Like in our TakeMessageToEncrypt() method, we now need to complete the process of using the user-given data to decrypt and present the results.

```

public static void TakeEcryptionToDecrypt()
{
    Console.WriteLine("Please introduce an encryption to decrypt.");
    string userInput = Console.ReadLine() ?? string.Empty;
    byte[] bytes = Convert.FromBase64String(userInput);
    Console.WriteLine("Please introduce a valid" +
    "Encryption Initialization Vector.");
    FetchUserEncryptionKeyAndIV();
    string decrypted = Decrypt(bytes, permKey!, permIV!);
}

```

We store the return of Decrypt() in a new string variable. Remember to add the correct parameters to the Decrypt() method. This time it is the “bytes” variable since we needed to convert the given string first!

Now it is just the presentation that is left, and since we already know the key and IV, there is no need to print that as well, so let’s just print out the final decrypted text.

```

FetchUserEncryptionKeyAndIV();
string decrypted = Decrypt(bytes, permKey!, permIV!);
Console.Clear();
Console.WriteLine("Your data has been decrypted.");
Console.WriteLine($"Decrypted text: {decrypted}");

```

Notice how we did not have to convert the returned value this time. Since we did not get a byte array returned, there is no need to convert to string.

And to wrap this up, the ending sequence has to be added. You know which one, this one:

```
Console.WriteLine($"Decrypted text: {decrypted}");
Console.WriteLine("Press ANY KEY to start over,
    or press the E.S.C. key to exit.");
ConsoleKeyInfo key = Console.ReadKey();
if (key.Key == ConsoleKey.Escape)
{
    Environment.Exit(0);
}
Console.Clear();
PromptMessageEncryptor();
}
```

Now, to run it, remember to uncomment the method call in the `PromptMessageEncryptor()` method.

```
public static void PromptMessageEncryptor()
{
    Console.WriteLine("Would you want to Encrypt (Y
        or Decrypt (N) a message? ( Y | N )");
    bool encryptOrDecrypt = YNQuestion();
        if (encryptOrDecrypt)    TakeMessageToEncrypt();    else
    TakeEcryptionToDecrypt();
}
```

Nice! And that is officially it! We got a functioning app ready to go! Go try it out. Try to decrypt the following encryption.

```
Encryption:
kifQ+8zxQah6LxUi4+ZPNGqXkaRsIt18BEFvIWQTzZmpDc3/QB0b15Ne/+cB5z93PBeYGk
dlobgm6hAyfo51xzOD5y7PFAJKxBfrGGUeq2N4Xy46K5lUV5wCWYwzeM145MservPbUVF7
WjCrXLMyrUAQHAOSGG8R1yBURqZp3K72hZNm0DExhDgAFQHrBFTe87bIQPY3X0E73NPifb
E1oO0MdEpYYNa0uKxliIsvwOOUS5t4fm7wt42vD/yQKD/5d2wgk/f1mzxm2aT2cVH/
TZWeAtC0iP82Wq/watplzkPZo3d6OcCI8u6DbKkR/oM3sysn4u0iQY+NDNvitr59g==
```

```
Key: TMm/9yukJPJs1NAtB4/aB61r01mRHxtKmH5xasms+7k=
IV: jaZRtsmhHDJg0gcsgexU8g==
```

We will try to decrypt something else here, though. Let us see how it goes (Fig. 9.4).

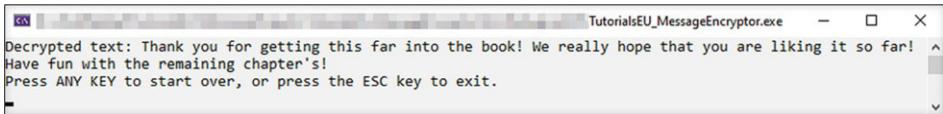


Fig. 9.4 Our app works perfectly as intended. Thank you for completing this chapter!

If we try the decrypter, we will see it work as intended. If we want to encrypt, we can do so by providing our own key and IV, or randomly generated ones. And if we want to decrypt, we simply add the encryption together with the key and IV, and there we go.

The potential should be apparent as to what is possible with such a base. We recommend not only using this as a portfolio piece but expanding on it, potentially even developing something that a company could use. And if we want to get it online, maybe one of our following chapters will help us get a general direction on how we could set up a full app. In the next chapter, we will start covering what goes beyond offline development with an introduction to the development process of a Simple Mail Transfer Protocol (SMTP) application using C# and our trusty Console Application format.

Let's get ready for the next chapter.

9.4.3 Source Code

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_MessageEncryptor

9.5 Summary

This chapter taught us:

- As per definition, cybersecurity can be seen as the protection of Internet-connected systems from unauthorized access or theft. There are two types of data integrity: physical and logical.
- There are many reasons why cybersecurity is so critical. First, the Internet has become a vital part of our lives. Second, the sheer amount of online data makes it a prime target for cyber criminals. Finally, the consequences of a cyber attack can be devastating in the case of data loss or reputation damage.
- Datasecurity is focused on protecting digital and analog forms of information from unauthorized access or modification. While cybersecurity, on the other hand, protects anything in cyberspace, from data breaches to technological threats.
- Cybersecurity, just like datasecurity, can be divided into several pillars that define each specialized sector within its spectrum. These seven pillars are network, cloud, endpoint, mobile, IoT, application, and the so-called Zero Trust security.

- Encryption, as the name suggests, is a process of transforming regular text, known as plaintext, into an unreadable form, ciphertext. This transformation is achieved through mathematical cryptography algorithms.
- Encryption is a process that transforms readable data into an unreadable format. This unreadable data can only be converted into a readable format using a decryption key. On the other hand, hacking is a process that takes readable data and transforms it into a unique code that cannot be reverse-engineered back into the original data.
- Generally, we can differentiate two types of encryption: asymmetric and symmetric. In an overview, the differences between both are that asymmetric encryption is used for high-security low-performance scenarios, and uses two keys for encrypting and decrypting. In contrast, symmetric encryption is used for high-performance lower-security scenarios and uses only one key.
- AES is a type of encryption that uses a block cipher to encrypt and decrypt data. This algorithm uses a block size of 128 bits, meaning it can encrypt data in blocks of 128 bits. AES can also use a 192-bit or 256-bit block size, but all AES implementations do not yet support these sizes. AES is also a symmetric-key algorithm, which means that the same key is applied to encrypt and decrypt data. AES uses a key size of 128, 192, or 256 bits, similar to its block size.
- The key size is the number of bits in the key, and, simply put, the larger the key size, the more secure the algorithm.
- The using statement and the using directive are used to turn its presented variable to read-only, and to use types defined in that namespace without having to specify the namespace itself each time, respectively.
- MemoryStream in C# programs allows us to use in-memory byte arrays or other data as though they are streams. So we can, instead of storing data in files, store data in memory and access them directly from there. Memory streams can reduce the need for temporary buffers and files in an application and improve performance.

References

- AO Kaspersky Lab. (n.d., n.d. n.d.). What is Cloud Security? Retrieved October 10, 2022 from [www.kaspersky.com: https://www.kaspersky.com/resource-center/definitions/what-is-cloud-security](https://www.kaspersky.com/resource-center/definitions/what-is-cloud-security)
- AO Kaspersky Lab. (n.d., n.d. n.d.). What is Cyber Security? Retrieved October 10, 2022 from [www.kaspersky.com: https://www.kaspersky.com/resource-center/definitions/what-is-cyber-security](https://www.kaspersky.com/resource-center/definitions/what-is-cyber-security)
- CHEN, J. (2022, July 27). What Is Encryption? How It Works, Types, and Benefits. Retrieved October 10, 2022 from [www.investopedia.com: https://www.investopedia.com/terms/e/encryption.asp](https://www.investopedia.com/terms/e/encryption.asp)
- Cisco Systems, Inc. (n.d., n.d. n.d.). What Is Network Access Control? Retrieved October 10, 2022 from [www.cisco.com: https://www.cisco.com/c/en/us/products/security/what-is-network-access-control-nac.html](https://www.cisco.com/c/en/us/products/security/what-is-network-access-control-nac.html)
- Clark, S. S. (2022, September n.d.). What is cybersecurity? Retrieved October 10, 2022 from [www.techtarget.com: https://www.techtarget.com/searchsecurity/definition/cybersecurity](https://www.techtarget.com/searchsecurity/definition/cybersecurity)

- Cobb, C. B. (2021, September n.d.). Advanced Encryption Standard (AES). Retrieved October 10, 2022 from [www.techtarget.com](https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard): <https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard>
- IT Governance Ltd. (n.d., n.d. n.d.). What is Cyber Security? Definition and Best Practices. Retrieved October 10, 2022 from www.itgovernance.co.uk: <https://www.itgovernance.co.uk/what-is-cybersecurity>
- Kumar, V. (2019, March 11). Encryption And Decryption Using A Symmetric Key In C#. Retrieved October 10, 2022 from www.c-sharpcorner.com: <https://www.c-sharpcorner.com/article/encryption-and-decryption-using-a-symmetric-key-in-c-sharp/>
- Microsoft Corporation. (2022, July 04). Encrypting data. Retrieved October 10, 2022 from learn.microsoft.com: <https://learn.microsoft.com/en-us/dotnet/standard/security/encrypting-data>
- Microsoft Corporation. (n.d., n.d. n.d.). Aes Class. Retrieved October 10, 2022 from learn.microsoft.com: <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.aes?view=net-6.0>
- Microsoft Corporation. (n.d., n.d. n.d.). CryptoStream Class. Retrieved October 10, 2022 from learn.microsoft.com: <https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.cryptostream?view=net-6.0>
- Musarubra US LLC. (n.d., n.d. n.d.). What Is Endpoint Security? Retrieved October 10, 2022 from www.trellix.com: <https://www.trellix.com/en-us/security-awareness/endpoint/what-is-endpoint-security.html>
- Palo Alto Networks. (n.d., n.d. n.d.). How to Secure IoT Devices in the Enterprise. Retrieved October 10, 2022 from www.paloaltonetworks.com: <https://www.paloaltonetworks.com/cyberpedia/how-to-secure-iot-devices-in-the-enterprise>
- Robert, T. M.-M. (2004, n.d. n.d.). The Evolution of Viruses and Worms. Retrieved October 10, 2022 from ivanlef0u.fr: <https://ivanlef0u.fr/repo/madchat/vxdevl/papers/avers/statmethods2004.pdf>
- Shrestha, J. (2015, September 3). Data Encryption And Decryption in C#. Retrieved October 10, 2022 from www.c-sharpcorner.com: <https://www.c-sharpcorner.com/UploadFile/f8fa6c/data-encryption-and-decryption-in-C-Sharp/>
- ssl2buy.com. (n.d., n.d. n.d.). Difference Between Hashing and Encryption. Retrieved October 10, 2022 from www.ssl2buy.com: <https://www.ssl2buy.com/wiki/difference-between-hashing-and-encryption#:~:text=The%20difference%20between%20hashing%20and%20encryption&text=In%20short%2C%20encryption%20is%20a,unique%20digest%20that%20is%20irreversible>
- Statista Research Department. (2022, July 7). Cumulative detections of newly-developed malware applications worldwide from 2015 to March 2020. Retrieved October 10, 2022 from www.statista.com: <https://www.statista.com/statistics/680953/global-malware-volume/#:~:text=As%20of%20March%202020%2C%20the,surpass%20700%20million%20within%202020>
- Trend Micro Incorporated. (2015, July 24). Encryption 101: What It Is, How It Works, and Why We Need It. Retrieved October 10, 2022 from www.trendmicro.com: <https://www.trendmicro.com/vinfo/us/security/news/online-privacy/encryption-101-what-it-is-how-it-works#:~:text=Encrypting%20your%20data,data%2C%20and%20secure%20intellectual%20property>



C# Simple Mail Transfer Protocol

10

This Chapter Covers

- Working with SMTP and Gmail
- Using the JSON file format
- Getting and Setting C# properties
- SOLID Programming and The Single Responsibility Principle
- Developing a “Mass Email Sender” application

After wrapping up the main bulk of this book with the later cybersecurity chapters, we finally venture out into the unknown territory of the Internet. Although we have already started to consider online services, like in our cryptography chapter, where we talked about a messaging service, we have not yet started to develop for these scenarios. Everything until now has been strictly offline, and that is for a good reason. The Internet is complicated and scary.

From communication protocols to application programming interfaces (APIs), we need to take into account a variety of factors when our application transitions from an offline environment to an online one. Yet, this doesn’t have to be an overly complicated process. In this chapter, we aim to navigate this uncharted territory by introducing communication protocols, with a particular focus on the Simple Mail Transfer Protocol (SMTP).

We certainly have extensive knowledge about using such services that implement SMTP. Take Gmail, for example. Do you have a Gmail account? If yes, then you have surely used it to send some emails before. There you go. You used the SMTP protocol. Well, kind of.

Most email clients, like Gmail, Outlook, Apple Mail, and Yahoo, rely on SMTP to send messages from a sender to a recipient. So comes the question: What even is SMTP? How does this protocol work? And, last but certainly not least, how can we use it to make our mailing service?

In this chapter, we attempt to answer this question by investigating the inner workings of the Simple Mail Transfer Protocol attempting an implementation using C# and our trusty Console Application format. We will attempt to develop an app that will take a subject and body for an email, a JSON file containing names and emails for recipients, and send an email to each of those addresses.

Ideally, we should be left with an application that could be even of use as an email marketing service for a small business.

Let us then get right into this chapter's first topic: What exactly is SMTP?

10.1 Simple Mail Transfer Protocol or SMTP

As the Internet grows, email does too. In 2022, as per this statistic (direct link <https://www.statista.com/statistics/456500/daily-number-of-e-mails-worldwide/>), over 300 billion emails are sent each day with a constant growth expected to reach almost 400 billion by the middle of this decade. However, we generally do not think much about what is happening behind the scenes.

Originating in the 1980s, SMTP quickly became one of the most widely used sender protocols worldwide. Today, it is the dedicated protocol for email service giants like Google and Microsoft with Gmail and Outlook, respectively.

And in the modern day, simple implementations such as the following

```
var smtpClient = new SmtpClient("smtp.gmail.com")
{
    Port = 587,
    Credentials = new NetworkCredential("username", "password"),
    EnableSsl = true,
};

smtpClient.Send("email", "recipient", "subject", "body");
```

allow many developers to start creating their message delivery service using SMTP.

As we see in this example, we have a basic setup for the SMTP client done and then execute the Send() method from the SmtpClient class.

So then, let us get a bit more into detail to understand the definition of SMTP properly.

10.1.1 How Does SMTP Work

Think of a one-way postal office. We want to send a message to a specific address. We tell the postal office who the recipient is and provide the exact message. The postal office then prepares the mail, much like the SMTP protocol prepares our email for sending. In the

case of SMTP, preparing the email involves validating it—checking to make sure the email address is correctly formatted, the message does not exceed size limits, the sending server is authenticated, and other requirements are met to ensure successful delivery.

Next, the email is handed over to the mailman. This mailman, trying to find a way to the recipient, will search for it and find a direct connection between his current position and the destination, like typing the address into a GPS. This clear way, direction, or connection represents our SMTP connection built on TCP, or the Transmission Control Protocol, which is responsible for transferring your email to the recipient's email server.

However, once the mailman (SMTP) has reached the destination (the recipient's email server), he hands over the mail to a different protocol—let's call it the housemaid (Post Office Protocol, POP, or Internet Access Protocol, IMAP). The housemaid is responsible for placing the mail into the recipient's inbox (POP) or managing and synchronizing the email as per the recipient's actions (IMAP).

If the address given to the mailman is correct, he will find a way to the recipient's server and hand over the email. If not, the mailman (SMTP) will get back to the office and let the sender know that he could not deliver his email, most likely because the address was wrong, in a process called "Bounce back" (Fig. 10.1).

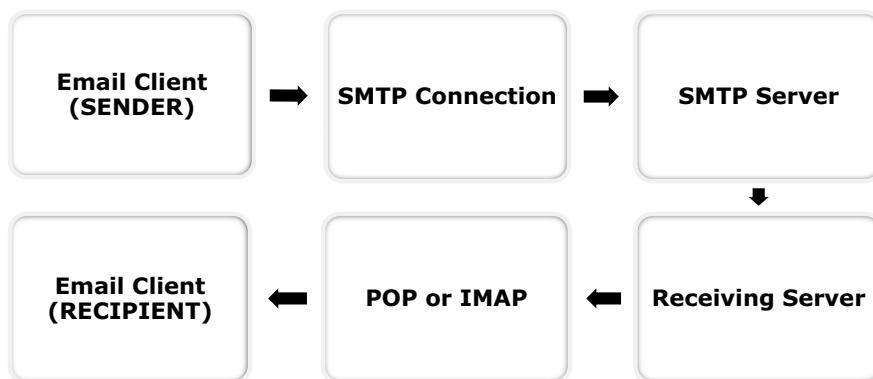


Fig. 10.1 After the email client sends the email through the SMTP connection to an SMTP server, after validation, this server, also through an SMTP connection, will send the email to the Receiving server, which will further process it depending on what it is destined to

NOTE As seen in Figure 10.1, we can also add that once it is received at the target destination, the recipient must use a receiving protocol like POP or IMAP to get and read his email. In essence, that means that since SMTP cannot queue the received messages, other protocols are needed to cover that part of the process. We will not need to go over that part of the process during this chapter, as Gmail covers that section within our application. This chapter will solely cover the preparation of a new email and sending over SMTP to a target address.

So, the email client initiates a connection with the server and sends the message across the connection. The server evaluates the message and, if it is addressed to one or more

recipients on the server, stores the message in a mail queue. The server sends the message to the next hop, either the final destination or another email server. When the message reaches its final destination, the receiving server stores it in the recipient's mailbox.

Great, now there is only one question: How can we use it through C#?

10.1.2 How Do We Use SMTP

Through the `System.Net.Mail` namespace, .NET and .NET Core provide built-in support for, among other things, the implementation of Simple Mail Transfer Protocol functions. Its usage will just require a single prerequisite, preparing a Gmail account to use it as the sender.

NOTE These requisites may change over time since Google's policies are constantly updated. If these steps do not lead you to a working application, you may need to consult Google's newest requirements for using Google accounts with external apps that do not support two-step verification.

Since we will be using Gmail as our server, we need a Google account. For this, we should be able to use any account as long as the password used is considered "strong" by Google. We will be using a TutorialsEU tester account for our process.

Once we get a valid account, since May 2022, we need to set up an App password. If we were to search for a way to prepare our account on our own, we might find the occasional solution that tells us to "Allow less secure apps." However, this changed recently, forcing users to use App passwords instead. Luckily, doing so is reasonably straightforward. Let us go through the steps to find out how.

In our Gmail inbox, click on our profile picture in the top-right corner of the screen and press the button "Manage your Google Account" (Fig. 10.2).

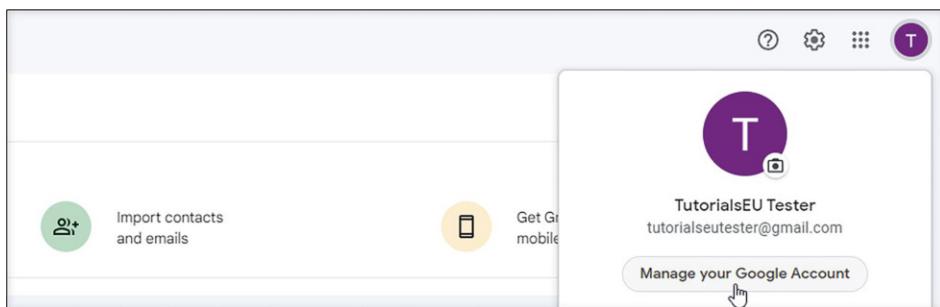


Fig. 10.2 Accessing our Gmail account settings

Next, we need to access the Security tab and set up 2-Step Verification on our account. This step is crucial for the next step even to be available. Doing so should be the same as with any other application, either through a phone number or a verification application (Fig. 10.3).

Now a new option should appear right underneath the 2-Step Verification section called “App passwords” (Fig. 10.4).

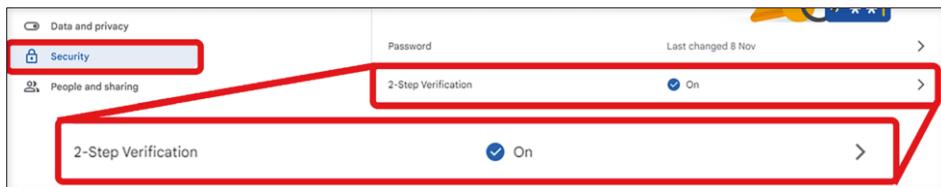


Fig. 10.3 Setting up 2-Step Verification on our Gmail account

If we enter this new section and enter our password again, we encounter a new screen where we can set up a new password for a new app. For that, we must select the App’s

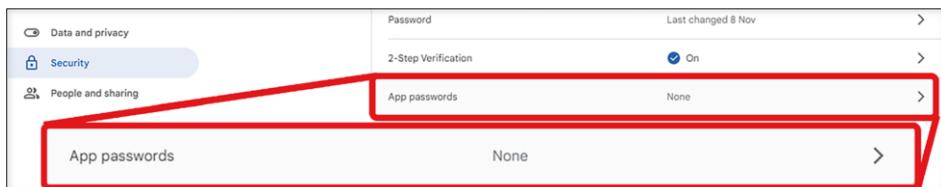


Fig. 10.4 The “App passwords” option should appear right underneath 2-Step Verification as soon as the 2-Step verification setup is done

purpose, target the device, and generate our new password. In our case, it would be “Mail” and “Windows computer” (Fig. 10.5).

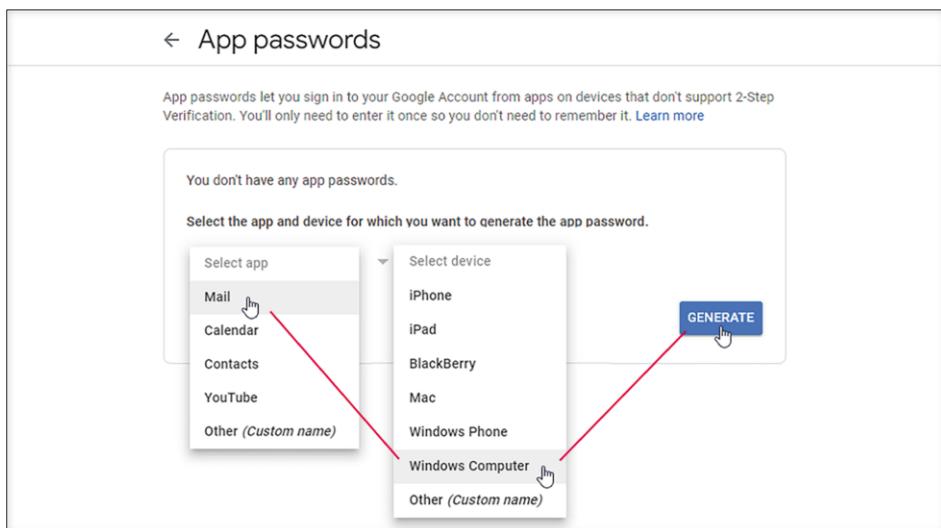


Fig. 10.5 Creating an App password for a “Mail” App with “Windows Computer” as the device

NOTE Google is constantly updating and changing the way its processes are done, meaning that by the time you pick up this book, these steps may not apply anymore. In case that is true, you should always be able to find updated steps directly from Google themselves or from our list of blogs covering a multitude of topics, including Google services.

Once generated, we should see our code on the screen. Now save this code, for we will need it for our app later.

Although we will be working on a more extended version of an SMTP app later in this chapter's project, why not look at a basic SMTP project so we can use our newly generated Gmail password? Let us use the following code and learn a bit more about how an SMTP app works.

```
using System.Net;
using System.Net.Mail;

var smtpClient = new SmtpClient("smtp.gmail.com")
{
    Port = 587,
    Credentials = new NetworkCredential("yourEmail", "yourPassword"),
    EnableSsl = true,
};

smtpClient.Send("fromEmail", "toEmail", "Subject", "Body");
```

We started by implementing the namespace System.Net and System.Net.Mail in our program. As we mentioned earlier, these will permit us to use some of the classes and methods we will need to make our SMTP service work.

```
using System.Net;
using System.Net.Mail;
```

Then, we created a new instance of the SmtpClient class, which we will be using to send emails through the specified server, in this case, the Gmail server.

```
var smtpClient = new SmtpClient("smtp.gmail.com")
{};

};
```

We then specify the use of port 587. Why port 587, we might ask? By default, this port is the one that is used for standard mail submission. It is the one that provides the necessary security and compliance with the The Internet Engineering Task Force (IETF¹)

¹The Internet Engineering Task Force is the body that defines the standards for operating Internet protocols (like TCP/IP).

guidelines. Naturally, we can find other available ports, like port 2525, generally used as an alternative to port 587. For example, we can see port 2525 used by services hosted by Google Compute Engine. However, for our use case, we should typically stick to port 587 since it would still be the most common.

```
var smtpClient = new SmtpClient("smtp.gmail.com")
{
    Port = 587,
};
```

Next, our recently created password comes into play. Using the NetworkCredential Class, we can connect to our prepared Gmail account by passing in our email address together with the password we created.

```
var smtpClient = new SmtpClient("smtp.gmail.com")
{
    Port = 587,
    Credentials = new NetworkCredential("yourEmail", "yourPassword"),
};
```

Then, for the last step in our server setup, we must enable the SSL or Secure Sockets Layer. This certificate will encrypt the connection and issue a STARTTLS command. This command tells the server that the email client wants to upgrade from an insecure connection to a secure one using either TLS or SSL. This command is necessary for the connection to be authorized. If set to false, we will be met with a System.Net.Mail.SmtpException. Besides the exception though, it is always advised to favor security in our apps.

```
var smtpClient = new SmtpClient("smtp.gmail.com")
{
    Port = 587,
    Credentials = new NetworkCredential("yourEmail", "yourPassword"),
    EnableSsl = true,
};
```

Once this is done, we can use this connection to send our first email by using the Send() method provided by the SmtpClient class. As in the example, we need to pass in the “from” email, which will be our TutorialsEU tester email again, the “to” email, which can be whatever, and finally, our subject and body.

```
smtpClient.Send("fromEmail", "toEmail", "Subject", "Body");
```

If we now run this, the application will start, process for a bit, and finish right after sending a new email to our recipient’s inbox (Fig. 10.6).



Fig. 10.6 The email we received on our recipient email address after executing the code. It was sent from the correct address to the correct address. In this concrete example, the email “Body” has been changed to a different text than previously shown in the code listing

Moreover, if we check the email details, we can see that it also includes a valid certificate. To be exact, the Standard encryption (TLS) certificate.

And with that, we have already created a simple SMTP application that can send emails through code.

Although our email reception was handled with Gmail, just as this chapter’s project will be, there are other ways of doing so. Say we were working on a competitor to Thunderbird, a popular open-source email client, then we would need a way also to retrieve emails to an inbox. SMTP cannot help in that regard, as it can only send emails. For that, we will need a receiver protocol.

Even though in this chapter’s project, we use Gmail to cover that base, if we wanted to avoid that or create a completely independent app, we would need a receiver protocol like POP3 or IMAP. Let us talk a bit about them.

10.1.3 SMTP and Receiver Protocols

The Simple Mail Transfer Protocol is used to send, relay, or forward messages, not to receive messages. This limitation means that if our app were to need email reception, we would need a different protocol.

For example, the popular Thunderbird email client allows users to use POP3 or IMAP to receive emails. Both have their use cases, while IMAP is generally still the recommended option. But what even are POP3 and IMAP?

Both POP3, or Post Office Protocol 3, and IMAP, or Internet Message Access Protocol, are MAAs, or Message Accessing Agents. Both serve to retrieve messages from a server to present them in an inbox. The difference between the two, though, is quite significant.

With IMAP, messages that land on a server are retrieved and copied to the email client inbox. Then any changes done to that email within the client will be mirrored in the server. This behavior also avoids accidental data loss since the emails exist on both the server and the local machine. Additionally, emails can be accessed and organized in the server and accessed through multiple devices. Generally, these features are what we would want in a receiver protocol.

On the other hand, POP3 only allows downloading emails from the server to our local computer. It cannot be organized within the server and can only be accessed by a single device at a time. Seemingly this protocol is missing many of the features we are accustomed to in our daily lives.

Nevertheless, nowadays, POP3 is still being used as the preferred option. Why is that?

In essence, independence and privacy are the reason. No need to have to stick to storage limits that popular email service providers often apply, and knowing that our data is safely stored within our own machine and is not mirrored in any external, potentially at-risk server often compel users and businesses alike to switch to POP3 for their personal email needs.

However, at the end of the day, IMAP continues to be the recommended choice for most of our uses. The service we use, Gmail, uses IMAP, for instance. Moreover, services like Microsoft Outlook and Apple Mail do too. So although we will not be using a receiver protocol for this chapter, if we wanted to, and if we had to decide between the two, IMAP would be our recommendation due to its increasing popularity and demand in the industry.

Great, after this journey through SMTP, we must come back down to the basics of programming. There is a concept that although it is not one of the first things that one learns about when beginning our developer journey, it is still crucial to learn about this if we intend to create SOLID applications. We are talking about SOLID Programming.

10.2 SOLID Programming

In our exploration of object-oriented programming (OOP), it's important to highlight the SOLID principles. These principles are a set of design guidelines that, when applied well, can lead to more understandable, flexible, and maintainable code. The acronym SOLID stands for Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle. Let's explore these principles one by one.

Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) is the concept that any single object in OOP should be made for one specific function. This means that each class should only serve one single function within a program.

Consider the following example. The class “SimpleClass” currently provides two functions, preparing a message to show to the console, and performing an addition. To adhere to the SR principle, we must separate these two functions into two separate classes, say in “SimpleClass” and “CalculatorClass.” We then move the CalculateResult() method to the new “CalculatorClass” class.

```
public class SimpleClass
{
    public static void Main(string[] args)
    {
        Console.WriteLine(GetResult(5, 5)); ;
    }

    public static string GetResult(int a, int b)
```

```
{  
    return $"The result of adding {a} and {b} is  
{CalculatorClass.CalculateResult(a, b)}";  
}  
}  
  
public class CalculatorClass  
{  
    public static int CalculateResult(int a, int b)  
    {  
        return a + b;  
    }  
}
```

Although this example seems overly simplified, the advantages of applying this principle become quickly apparent once the codebase grows and multiple developers work on it. Furthermore, adherence to SOLID programming is a requirement in many modern companies and is seen as the correct way of programming.

Let us now continue our understanding of the SOLID principles with the Open/Closed Principle.

Open/Closed Principle (OCP)

The Open/Closed Principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means that a class's behavior can be extended without modifying its source code. Modifications can introduce new bugs in existing functionality; hence being “closed” for modification helps maintain stability.

For instance, if we need to add more mathematical operations in our “CalculatorClass,” instead of modifying it directly, we can create a new class that extends “CalculatorClass” and add our new method there.

Let’s extend our “CalculatorClass” with a “Multiply” operation without modifying it.

```
public abstract class CalculatorClass  
{  
    public abstract int CalculateResult(int a, int b);  
}  
  
public class AdditionCalculator : CalculatorClass  
{  
    public override int CalculateResult(int a, int b)  
    {  
        return a + b;  
    }  
}
```

```
public class MultiplicationCalculator : CalculatorClass
{
    public override int CalculateResult(int a, int b)
    {
        return a * b;
    }
}
```

We've created two subclasses of "CalculatorClass": "AdditionCalculator" and "MultiplicationCalculator." Each of these classes overrides the "CalculateResult" method to perform their specific operations. This keeps the original class closed for modification while being open for extension.

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP) states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of the program. It means that if a class B is a subclass of class A, we should be able to replace A with B without disrupting the behavior of our program.

For instance, if we have a "ComplexCalculatorClass" that is a subclass of "CalculatorClass," any place in our program using "CalculatorClass" should be able to use "ComplexCalculatorClass" without any issues.

Consider the previous classes. A client function using a "CalculatorClass" instance should be able to use any of its subclasses without the function knowing the exact type of class it is using.

```
public void ExecuteCalculation(CalculatorClass calculator, int a, int b)
{
    int result = calculator.CalculateResult(a, b);
    Console.WriteLine($"The result is {result}");
}
```

Here, "ExecuteCalculation" can operate with any instance of "CalculatorClass" or its subclasses, hence adhering to LSP.

Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) states that clients should not be forced to depend on interfaces they do not use. It suggests that large interfaces should be split into smaller, more specific ones so that clients only need to know about the methods that are of interest to them.

For example, if we had a "SuperCalculatorClass" that also had methods for calculating trigonometric values, but a client class only needed basic arithmetic operations, the ISP would suggest creating separate interfaces for each set of operations.

If a class doesn't need a method from an interface, that interface should be broken down into smaller, more specific interfaces.

```
public interface IAdd
{
    int Add(int a, int b);
}

public interface IMultiply
{
    int Multiply(int a, int b);
}

public class BasicCalculator : IAdd
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}

public class AdvancedCalculator : IAdd, IMultiply
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public int Multiply(int a, int b)
    {
        return a * b;
    }
}
```

In this example, “BasicCalculator” only implements the “IAdd” interface since it only needs the “Add” method, while “AdvancedCalculator” implements both “IAdd” and “IMultiply” interfaces.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules. Both should depend on abstractions. Additionally, abstractions should not depend upon details; details should depend upon abstractions.

In essence, this principle inverts the way some people may think about object-oriented design, dictating that both high-level and low-level objects must depend on the same abstraction. This principle enables decoupling.

Instead of directly calling methods from lower-level modules, high-level modules should depend on abstractions.

```
public interface ICalculator
{
    int Calculate(int a, int b);
}

public class Addition : ICalculator
{
    public int Calculate(int a, int b)
    {
        return a + b;
    }
}

public class CalculationExecutor
{
    private readonly ICalculator _calculator;

    public CalculationExecutor(ICalculator calculator)
    {
        _calculator = calculator;
    }

    public int ExecuteCalculation(int a, int b)
    {
        return _calculator.Calculate(a, b);
    }
}
```

Here, the “`CalculationExecutor`” class doesn’t depend directly on the “`Addition`” class. Instead, it depends on the “`ICalculator`” abstraction, fulfilling the Dependency Inversion Principle.

Now, having explored the principles that make up SOLID, we can see how they guide toward creating more robust, maintainable, and scalable software. These principles are not rules, but rather guidelines to be used as part of an overall strategy for designing and organizing code.

Now, let us get back on track and learn about a file format we will use for this chapter's project that will massively extend our app functionality, the JSON file format, and how to use it.

10.3 JavaScript Object Notation or JSON

JSON is a lightweight data-interchange format that has become increasingly popular among Web developers due to its easy-to-read syntax. JSON stands for JavaScript Object Notation and is often used to transfer data between a server and a client as an alternative to XML.

Seen as a standard for data storage, it seems clear why its human-readable format and ease of use through JavaScript and many other languages make it a coder's favorite. But why is it so popular, and why did we choose to use it?

There are three main reasons why JSON is still one of the preferred file formats when it comes to data storage and transfer:

- Readability. Its simplicity and structured syntax make reading and understanding its contents easy, even intuitive.
- Structure. Structured like a JavaScript object, it is simple to parse and easy to implement.
- Standard. Lastly, it just became a standard over time. Although this reason does not come directly from the file format itself, it is one that emerged because of it. Nowadays, when many applications and companies rely on JSON, many systems are already prepared to use JSON as their format. Most languages already have a built-in way to process JSON files.

Because of these reasons, and because we are simply used to it ourselves, we decided to go with this format for this chapter's application. Specifically, we decided to use it for our list of emails and names to which we will be writing our emails.

Now, how do we use it with C# if it is so simple? Let us go over a simple implementation.

10.3.1 How Do We Use a JSON File?

We use the following JSON file as an example.

```
[  
 {  
   "userID": 123,  
   "userName": "John Doe",  
   "userEmail": "johndoe123456@gmail.com",
```

```
"coursesEnrolled": [
  {
    "courseID": 2,
    "courseName": "C# Masterclass",
  },
  {
    "courseID": 5,
    "courseName": "Unity Masterclass",
  }
]
```

The previous code is an example of a simple JSON file that stores user data like name, email, and the courses he is enrolled in. Let us go through how the file is structured.

To start, as we see in the example, JSON files can contain both lists of objects and just objects. Lists are surrounded by square brackets (`[]`), while objects are surrounded by curly brackets (`{}`).

```
[{
  "object": 1
}]
```

Name/Value pair objects are declared by a field name in quotation marks ("") and a field value right after a colon (:).

```
{
  "object": 1
}
```

Additionally, these field values can be of many types, like integers, strings, nulls, another array, or another object.

```
[
  {
    "objectInt": 1
  },
  {
    "objectString": "string"
  },
  {
    "objectNull": null
  },
]
```

```
{  
    "objectArray": []  
,  
{  
    "objectObject": {}  
}  
}]
```

And each consecutive object or array must be comma (,) separated.

```
[  
{  
    "objectInt": 1  
,  
{  
    "objectString": "string"  
}  
]
```

And essentially, that is it. There is more depth to the JSON format; however, with that, we got the most important covered. Also, for our project, we will not need anything more than this anyway.

So then, let us continue by learning how we would go about reading JSON using C#. Luckily, C# already provides us with a namespace for that: the System.Text.Json namespace.

NOTE The Newtonsoft library is still the solution used in most projects. However, it is recommended to use the Microsoft built-in solution as it is steadily taking over as the standard. That is why we are going to work with the System.Text.Json namespace.

Take as an example a JSON file that holds an array of “customer” objects.

```
[  
{  
    "Name": "user1",  
    "EmailAddress": "user1@gmail.com"  
,  
{  
    "Name": "user2",  
    "EmailAddress": "user2@gmail.com"  
}  
]
```

If we wanted an array of customers where the name and email address would be linked, one solution would be to build a “Record” with a name and address parameter.

```
public record Contact(string Name, string EmailAddress);
```

Then create a new array of that record to fill it with the data we need, like so:

```
Contact[] customers = new Contact[] {  
    new("user1", "user1@gmail.com"),  
    new("user2", "user2@gmail.com")  
};
```

This solution works. However, this also means that we “hard-coded” the data about our customers. So each time we want to add or remove a customer, the code itself has to change, which is just simply not a good practice.

If we wanted to improve this, we could use the previously specified JSON file. In essence, the data it held consisted of an array of objects that each had a “Name” and “EmailAddress” field. This leads us to assume that we could use that to take each object in that array as one record object with its fields populating the record’s parameters. But how would we go about doing that?

Implementing the System.Text.Json namespace, we get access to the JsonSerializer class. This class contains a method called Deserialize<>() which we can use to pass in the specific type and JSON file to deserialize. So just something like this:

```
Contact[] customers = JsonSerializer.Deserialize<Contact[]>(JsonFile);
```

Now we are filling the customer’s variable with the return of the Deserialize<>() method. This return, we can imagine, mirrors the “hard-coded” list of customers of the previous example.

So, it would separate each object in the list of objects and populate the record with objects containing a “name” and “email” parameter.

With that said, we essentially implemented our deserialization feature for this chapter’s project. Although we have not started this project yet, a proper JSON integration for a full app is pretty much just what we did. So then, seeing as we are already here, why don’t we go ahead and finally start this chapter’s project development? Let us start with our Mass Email Sender App.

10.4 A Mass Email Sender App

For our second to last project, we are finally reaching out to the outside world, and since we want to do that in style, we will start by building a Mass Email Sender using Gmail. Believe it or not, this could already be used by a company to serve as their mailing list service, so if you get big in the mailing service world, please remember us!

So let us check what we will be needing, shall we?

10.4.1 The Project

For this chapter, we want to build a Mass Email Sender App. This app will consist of a few prompts for the user to know where to place the receiver email list for the app to read from it, to write a subject, a body, and, lastly, confirm if the email is correct and send it out to the mailing list.

To accomplish this, we will need to build our SMTP code using our previously prepared Gmail account, prepare the email using the user's input, process the given JSON file, and send out the email to each email successfully.

Along the way, we will be learning about two new C# properties, Get and Set. Furthermore, we will introduce the concept of SOLID programming, specifically the Single Responsibility Principle, into our application.

There is a lot to cover in this project, so let us get right into it.

10.4.2 Our Code

As usual, we must start by creating a class Program with a static Main method. To maintain consistency throughout the chapters, we will maintain this code as our starting point.

So then, create a new project, clean it, and add the following lines.

```
class Program
{
    public static void Main(string[] args)
    {

    }
}
```

We will start with our basic introduction, as we always want to attempt to create user-friendly applications. So, start by welcoming our future users to our app.

```
Console.WriteLine("Welcome to the TutorialsEU email marketing service.");
```

Sounds good. Now, right away, we need to work on our first implementation of the SMTP code. For we need a message letting the user know who is currently “logged in” to our app.

Disclaimer. It must be said that this project will not include a login system. This only refers to the email with which to send the email. Although you could expand on this project with a dynamic way of updating the current email and server, it is not within the scope of this chapter, so it will stay as just a piece of information for the user to know what is the sender's email in the current instance of the app.

Take the following string, “You are currently logged in as John.” “John” would be the user that owns the “John@Doe.com” email prepared to act as a sender email. How would we be able to get such a string to display?

We could just print out the previous string as is, but this would not update when the email is modified.

```
Console.WriteLine($"You are currently logged in as John");
//Hard-coded, not adaptable to changes.
//It should not be used like that.
```

So we need to fetch a name associated with the user email from somewhere. We want to implement something more like this:

```
Console.WriteLine("Welcome to the TutorialsEU email marketing service.");
Console.WriteLine($"You are currently logged in as
{Sales.Rep.CurrentSalesRep.Contact.Name}");
```

For that to work though, we must create the necessary back-end. In that case, let us change this project's approach by starting with the back-end first. Doing so will permit us to use anything we might need. Let us start right away with our SMTP implementation.

For that, we will make use of a new class. Why? We want to separate the specific code for the SMTP implementation into its own class. And why that? Because of the Single Responsibility Principle. Or the S in SOLID programming.

To work on our SMTP implementation we will want to add a new class called “Sales.” By doing so we attempt to apply the first rule in the SOLID programming principles set, that of Single Responsibility.

So, how do we apply this? We currently have a class called “Program” in a file called “Program.cs.” We will be using this class for anything visual and anything that will be displayed and interacted with. For anything else, especially anything SMTP related, we will create new classes. Starting with the “Sales” class. Creating a new class is fairly straightforward. We must simply right-click on our project in the solution explorer, select “Add,” then “Class...,” and give it a name (Fig. 10.7).

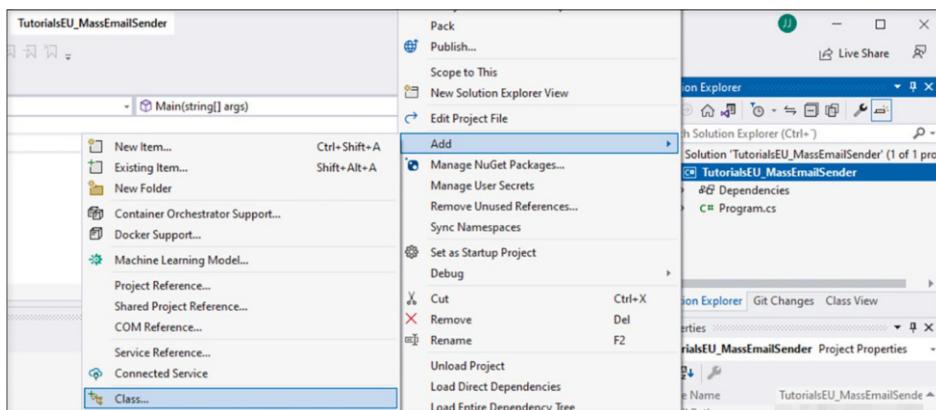


Fig. 10.7 This figure depicts the process of creating a new class within the Visual Studio IDE

Once created, we will find ourselves in front of an empty class that will look something like this:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace TutorialsEU_MassEmailSender
{
    internal class Sales
    {
    }
}
```

It will have a list of using statements at the beginning, followed by the namespace of our project, and the class set to internal.

We want two things for this class; one, to set up a sender email that uses a specific mail server, and to be publicly accessible. Straight away, we won't need any of the using statements included. Furthermore, the class currently set to internal must be set to public so as to be publicly accessible, leaving us with something like the following.

```
namespace TutorialsEU_MassEmailSender
{
    public class Sales
    {
    }
}
```

Great, that looks more like what we are looking for. So, what do we need from this class specifically?

This class's single responsibility is to initialize our data for the future connection to the desired SMTP server. As we mentioned earlier, we will be using Gmail for this project. However, we will still include the connection to the servers for Yahoo!, Office 365, and AOL, in case Gmail is not an option for us.

In our "Sales" class, we will make use of something new, what is known as a "Constructor." Let us briefly go over what a constructor is and why we are using it.

In short, a constructor is a method whose name is the same as the class it is contained in and that does not have any return type. Furthermore, as soon as a class is created, its constructor is called, enabling the definition of "default" values.

```
public class SimpleClass
{
    int defaultNumber;
    SimpleClass()
    {
        defaultNumber = 1;
    }
}
```

This, however, begs the question: Why do we use constructors and not anything else? Why are they the preferred option for this use case?

Constructors allow us to create an object with its properties already set, rather than having to manually set the properties each time we create an instance of the object. This saves time and increases code readability.

Therefore, as a general rule of thumb, if our app needs initialization, it is advised to use constructors to accomplish that goal. As in our project we need to initialize the server list and user before we can start using it, we should make use of a constructor.

So then, to implement a constructor in our Sales class, we simply need the following.

```
public class Sales
{
    Sales()
    {
    }
}
```

Within our constructor, we then need to initialize our default values, these being our servers. Let us create a local variable called “servers” for that.

```
public class Sales
{
    Sales()
    {
        var servers = ();
    }
}
```

NOTE In a real-world setting, the SMTP server configuration should ideally be managed in a separate class, rather than the one referenced here. To simplify this, we have kept it within the same class.

The variable we created is what is called a “Tuple” type variable. Tuple variables are variables that can hold elements of more than one type. We could, to make it clearer, also declare our variable like so:

```
Sales()
{
    (string, int) servers = ();
}
```

This now allows us to implement elements within it that contain both string data types and int data types. Right in time for our server elements, then they implement both, an address string, and a port number. In Sect. 10.1.2 we have already seen those values in action; here we simply want to, instead of directly implementing them in our connection, prepare them for future use. Starting with Gmail, our address would be the “smtp.gmail.com” address, and our port number port “587.” So we add them to the list, like so:

```
var servers = (
    Gmail: (Address: "smtp.gmail.com", Port: 587)
);
```

However, if we do so, we will immediately get an error message stating that Tuple must contain at least two elements. That is simply a requirement that we must obey to be able to use a tuple type. Luckily, though, we wanted to do so anyway since we planned on adding the Yahoo!, Office 365, and AOL servers. So if we do so, we get this result, and no more errors:

```
var servers = (
    Gmail: (Address: "smtp.gmail.com", Port: 587),
    Yahoo: (Address: "smtp.mail.yahoo.com", Port: 587),
    Office365: (Address: "smtp.office365.com", Port: 587),
    AOL: (Address: "smtp.aol.com", Port: 587)
);
```

To go back to tuple types, we can attempt to visualize what the data types are within that tuple. As we can see, we are creating elements that contain string and int elements. To correctly represent this, we get this enormous line:

```
((string Address, int port) Gmail, (string Address, int port) Yahoo,
(string Address, int port) Office365, (string Address, int port) AOL)
servers = (
    Gmail: (Address: "smtp.gmail.com", Port: 587),
    Yahoo: (Address: "smtp.mail.yahoo.com", Port: 587),
```

```
Office365: (Address: "smtp.office365.com", Port: 587),  
AOL: (Address: "smtp.aol.com", Port: 587)  
);
```

As we talked about all the way back in the first chapter, it is correct to use the var keyword for a variable declaration, if legibility is improved. It is safe to say that this instance justifies the use of a var for its declaration. So let us just leave this as it was.

```
var servers = (  
    Gmail: (Address: "smtp.gmail.com", Port: 587),  
    Yahoo: (Address: "smtp.mail.yahoo.com", Port: 587),  
    Office365: (Address: "smtp.office365.com", Port: 587),  
    AOL: (Address: "smtp.aol.com", Port: 587)  
);
```

NOTE It is important to note that in a practical application, this data would typically be stored in an “appSettings.json” file. We would then retrieve this information during the initialization phase, adhering to standard practices. However, in the interest of maintaining the book’s learning-oriented simplicity, we have omitted this step in our discussions.

Now that we have our servers, we need to create a user responsible for connecting us with the server. Let us call it the sales representative.

For that, we will again need a new class, this time the “Sender” class. So in a new file, we should, after a bit of cleanup like before, have a class like this:

```
namespace TutorialsEU_MassEmailSender  
{  
    public class Sender  
    {  
    }  
}
```

Within the Sender class, we need a new Constructor again; this time though, it will be a parametrized constructor, like so:

```
public class Sender  
{  
    public Sender(string displayName, string address, string password,  
(string Address, int Port) server)  
    {  
    }  
}
```

As we can see, the constructor takes a displayName parameter, an address parameter, a password parameter, and a server parameter which will contain the server address and port we set up earlier.

The displayName will function as a way to reference the current sales representative and mostly serves as internal information. The address will be our Gmail account we set up in this chapter. The password will be the password we generated. And we already learned what the server will take. The concept seems clear, so let us set up what is left. What is the constructor going to do with this data it is given?

The given data will be used to populate a few public variables. First, the MailServer variable. The MailServer variable, as it will be holding the data for the address and the port within, will again be Tuple.

So we have to create one like so:

```
public (string Address, int Port) MailServer;
```

Now, within the constructor, we can populate it like so:

```
public Sender(string displayName, string address, string password,
(string Address, int Port) server)
{
    MailServer = server;
}
public (string Address, int Port) MailServer;
```

To make it clearer, we can imagine our MailServer tuple as if it was declared like the following, where the MailServer variable gets populated with an Address string element, and a Port int element.

```
public (string Address, int Port) MailServer =
(
    Address: "smtp.gmail.com",
    Port: 587
);
```

After the MailServer, we need the Credentials. That is where our Gmail account and generated password come in. So set up another tuple like so:

```
public Sender(string displayName, string address, string password,
(string Address, int Port) server)
{
    MailServer = server;
}
public (string Address, int Port) MailServer;
public (string UserName, string Password) Credentials;
```

Then, in our constructor:

```
public Sender(string displayName, string address, string password,
(string Address, int Port) server)
{
    MailServer = server;
    Credentials = (address, password);
}
```

Now, there is one final variable we need. This variable will take in the displayName and the address. A Contact variable. This time though, it won't be a tuple, not because the variables it holds aren't of different data types, but because the Contact variable will be a record.

We have already learned about records from a previous chapter, but essentially, we want to use a record because of its immutability. And because we just want to store a bit of data from our current salesperson. For that, we will again create a new class file, but this time there will be a little twist.

So create a new class and call it "Contact." Now, instead of the usual cleanup, we will simply have this on our file:

```
namespace TutorialsEU_MassEmailSender;
public record Contact(string Name, string EmailAddress);
```

We are replacing the Class with a record so that we can have an entire file dedicated to this record. Why do we do so? And why are we using a record in the first place? Because this is a big opportunity to create reusable code within our project.

Our future customers, or the users that are subscribed to our mailing list, will essentially be a list of users with a first name to refer to, and an email address. So the best scenario here would be to reuse this code for both our salesperson and our customers. Less code, better readability, and less clutter to work with if anything needs updating.

And so, with that record in place, if we go back to our Sender class, we can simply use the following to implement a new Contact variable.

```
public Sender(string displayName, string address, string password,
(string Address, int Port) server)
{
    MailServer = server;
    Credentials = (address, password);
    Contact = new(displayName, address);
}

public (string Address, int Port) MailServer;
public (string UserName, string Password) Credentials;
public Contact Contact;
```

This concludes our Sender class. Now we can finally create the new sales representative and continue with our setup. Back in the Sales class, we first have to create a new variable for our Sender class called “CurrentSalesRep,” for example.

```
Sales()
{
    var servers = (
        Gmail: (Address: "smtp.gmail.com", Port: 587),
        Yahoo: (Address: "smtp.mail.yahoo.com", Port: 587),
        Office365: (Address: "smtp.office365.com", Port: 587),
        AOL: (Address: "smtp.aol.com", Port: 587)
    );
}
public Sender CurrentSalesRep;
```

Then, we can, within our Sales constructor, create a new CurrentSalesRep initialization, like this:

```
Sales()
{
    var servers = (
        Gmail: (Address: "smtp.gmail.com", Port: 587),
        Yahoo: (Address: "smtp.mail.yahoo.com", Port: 587),
        Office365: (Address: "smtp.office365.com", Port: 587),
        AOL: (Address: "smtp.aol.com", Port: 587)
    );
    CurrentSalesRep = new("yourName", "yourEmail", "yourPassword",
servers.Gmail);
}
```

And finally, at the end of the Sales class, now that we got it all ready, let us create a new instance of it using Lambda and call it Rep.

```
public class Sales
{
    public Sender CurrentSalesRep { get; }
    Sales()
    {
        var servers = (
            Gmail: (Address: "smtp.gmail.com", Port: 587),
            Yahoo: (Address: "smtp.mail.yahoo.com", Port: 587),
            Office365: (Address: "smtp.office365.com", Port: 587),
            AOL: (Address: "smtp.aol.com", Port: 587)
        );
    }
}
```

```
        CurrentSalesRep = new("yourName", "yourEmail", "yourPassword",
servers.Gmail);
    }
    public static Sales Rep => new();
}
```

This will create, as noted, a new instance of the Sales class, therefore calling the Sales constructor. And with that we got everything ready to get back to our program class.

So back to the “Program.cs” file, we can continue where we left off. We attempted to implement the following line.

```
Console.WriteLine($"You are currently logged in as
{Sales.Rep.CurrentSalesRep.Contact.Name}");
```

We know that we have a Sales class and we have an instance called Rep we just created, within which there is a CurrentSalesRep, which has a Contact Name. However, if we add this line where it should be, it still will not work. Why is that (Fig. 10.8)?

```
Console.WriteLine("Welcome to the TutorialsEU email marketing service.");
Console.WriteLine($"You are currently logged in as {Sales.Rep.CurrentSalesRep.Contact.Name}");
```

CS0103: The name 'Sales' does not exist in the current context

Fig. 10.8 Although we have our back-end ready, we are still getting an error that “Sales” does not exist in the current context

That is because Sales does not yet exist, at least for the Program class, that is. To give the class access to any other class within the current solution, we must specify a new using statement. One that includes the solution within this project file.

```
using TutorialsEU_MassEmailSender;
public class Program
{
```

Once that is added, our error should go away, and we can finally get to the final stretch for this project. But first, let us try it and see if everything is running fine and we get our desired result.

Remember, we should see something like “Welcome to the TutorialsEU email marketing service,” followed by “You are currently logged in as yourName” (Fig. 10.9).

```
Console.WriteLine("Welcome to the TutorialsEU email marketing service.");
Console.WriteLine($"You are currently logged in as {Sales.Rep.CurrentSalesRep.Contact.Name}");

Microsoft Visual Studio Debug Console
Welcome to the TutorialsEU email marketing service.
You are currently logged in as Jafar

TutorialsEU_MassEmailSender\bin\Debug\net6.0\TutorialsEU_MassEmailSender.exe
e (process 22752) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Fig. 10.9 A successful run with the expected result printed on the console, meaning that our backend has executed correctly

Everything is running flawlessly and we are ready to continue. So what now? Well, let us prompt the start of the process. Since we want to let the user write his own email, we must let him decide when to start the said process. We already know how to do so, like this:

```
Console.WriteLine("Press any key to start the process");
Console.ReadKey();
```

Leaving us with this Main() method:

```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to the TutorialsEU
        email marketing service.");
    Console.WriteLine($"You are currently logged in as
        ${Sales.Rep.CurrentSalesRep.Contact.Name}");
    Console.WriteLine("Press any key to start the process");
    Console.ReadKey();
}
```

And once the user presses “any key,” we can jump right into our PrepEmail() method. So let us create this method to handle the logic for taking the user input and preparing the full email. Start by creating the PrepEmail() method right under the Main() method. This will be a static method returning void, so nothing. This will simply handle the process so no need to return anything.

```
public static void PrepEmail()
{
}
```

Within, we can start by getting and storing the email subject and body from the user and storing it in its respective variable.

```
public static void PrepEmail()
{
    Console.Clear();
    Console.WriteLine("\r Please write an email subject:");
    string emailSubject = Console.ReadLine() ?? string.Empty;
    Console.WriteLine("\r Please write an email body:");
    string emailBody = Console.ReadLine() ?? string.Empty;
}
```

Using the stored subject and body, we can create a quick confirmation window before continuing with it, like so:

```
Console.WriteLine("\rThe following email will be sent:");
Console.WriteLine($"Email Subject: {emailSubject}");
Console.WriteLine($"Email Body: {emailBody}");
```

And using our world-famous YNQuestion method from previous chapters, we can ask for the final confirmation, like so:

```
Console.WriteLine("\rConfirm send? (Y) to send (N) to redo");
bool sendOrRedo = YNQuestion();
```

Naturally, our world-renowned YNQuestion method needs to be implemented for that to work, so simply do so right after the current method.

```
public static bool YNQuestion()
{
    string userInput;
    do
    {
        userInput = Console.ReadLine() ?? string.Empty;
        userInput = userInput.ToUpper();
    }
    while (userInput != "Y" && userInput != "N");
    return userInput == "Y";
}
```

Not the most beautiful but gets the job done. If the user does not confirm their email, we then invoke the PrepEmail() method once more to restart the process.

```
Console.WriteLine("\rConfirm send? (Y) to send (N) to redo");
bool sendOrRedo = YNQuestion();
if (!sendOrRedo) PrepEmail();
```

If the email is confirmed, the code will just continue executing, so we can implement the email sending logic now. For that, we will need a new method responsible for this task, let us call it the `SendEmail()` method. So first, create it as a static, void-returning method.

```
public static void SendEmail()  
{  
}
```

This method will need the data we just fetched from the user as well as some additional info. To get that, we must pass in a few parameters. The sender, the receiver, the subject, and the body, to be exact.

```
public static void SendEmail(Sender from, Contact[] toAll, string subject,  
string body)  
{  
}
```

Within, we must first construct a new `MailMessage` instance. This class comes from the `SmtpClient` class that must be implemented through the using tag `System.Net.Mail`. So let us do so.

```
using System.Net.Mail;  
using TutorialsEU_MassEmailSender;  
public class Program  
{
```

Now, we can create a new `MailMessage` using the data we got through the parameters of the method.

```
public static void SendEmail(Sender from, Contact[] toAll, string subject,  
string body)  
{  
    MailMessage message = new()  
    {  
        From = new MailAddress(from.Contact.EmailAddress,  
                               from.Contact.Name),  
        Subject = subject,  
        Body = body  
    };  
}
```

Now, using the list of email receivers that we have to get as a parameter, we can add each email as a Bcc to our email. This can also be easily achieved by looping through our array of contacts and adding each element to the Bcc of the `MailMessage` we just created. Simply like this:

```
foreach (var contact in toAll)
    message.Bcc.Add(new MailAddress(contact.EmailAddress, contact.Name));
```

This should correctly populate our Bcc list of our new message. Now, we create two new variables initialized with the values from our MailServer Tundle variable in our Sales class.

```
var (Address, Port) = from.MailServer;
```

This will create the variables Address and Port, both initialized with the values of Address and Port from our MailServer variable. Essentially like doing this, but in a single line:

```
// Alternative two-liner
var address = from.MailServer.Address;
var port = from.MailServer.Port;
```

Now, we must create a new SmtpClient class instance and set it up with our user credentials. But to do so, we need to add the System.Net using statement to get access to the NetworkCredential class.

```
using System.Net;
using System.Net.Mail;
using TutorialsEU_MassEmailSender;
public class Program
{
```

And now, in our SendEmail() method, we can create the SmtpClient instance.

```
var (Address, Port) = from.MailServer;
SmtpClient smtpClient = new(Address, Port)
{
    EnableSsl = true,
    UseDefaultCredentials = false,
    Credentials = new NetworkCredential(from.Credentials.UserName,
from.Credentials.Password)
};
```

At last, we use the smtpClient instance we just created to send our message out.

```
smtpClient.Send(message);
```

And that is the sending done. Now back to the PrepEmail() method, we have to call this method and pass in everything it needs.

```
Console.WriteLine("\rConfirm send? (Y) to send (N) to redo");
bool sendOrRedo = YNQuestion();
if (!sendOrRedo) PrepEmail();

SendEmail(
    from: Sales.Rep.CurrentSalesRep
    , toAll: Customers()
    , subject: emailSubject
    , body: emailBody
);

```

The issue is that we do not yet have a list of customers. Remember our JSON example from earlier in this chapter? That will come in handy right now.

First, we need a JSON file with user data. We can create one by right-clicking on the solution > Add > New item... window and adding any file that ends in .json into our project. Creating a new file through the Windows file explorer window that ends in the .json extension is also an option, as well as simply getting our file from the Github repository for this project.

What you will need is simply a JSON file somewhere reachable for your project. We have placed it within the “YourDrive:\YourProject\bin\Debug\net6.0” directory. So we recommend that you do the same.

Once you have a JSON file, open it with any text editing software you have, like, well, Microsoft™ Visual Studio Community 2022 (64-bit) Edition. And edit its contents to have a data structure similar to this:

```
[
{
    "Name": "user",
    "EmailAddress": "user@email.com"
}
]
```

Once you got that, we can proceed with parsing. In the Program class, create a new static method which returns Contact[]. Using our Record from earlier. As you can see, reusing code to have a cleaner project.

```
public static Contact[] Customers()
{
}
```

This method must have a return since we want to get a list of receiver emails out of it. Within, we need to get our file open in our app. If we have placed our “customers.json” file in the directory mentioned earlier, we should simply be able to do the following.

```
public static Contact[] Customers()
{
    string customerList = File.ReadAllText("customers.json");
}
```

Now, we need to deserialize. As we learned earlier, Microsoft already provides us with a namespace for that, the System.Text.Json namespace. So let us just add that at the top of the Program file.

```
using System.Net;
using System.Net.Mail;
using TutorialsEU_MassEmailSender;
using System.Text.Json;

public class Program
{
```

And now we got access to the Deserialize() method from the JsonSerializer class. Back down in our Customers() method, we can now simply deserialize our JSON file and store it in a new Contact[] type variable called customer.

```
public static Contact[] Customers()
{
    string customerList = File.ReadAllText("customers.json");
    Contact[] customer =
JsonSerializer.Deserialize<Contact[]>(customerList);
}
```

Then, finally, just return the new variable.

```
public static Contact[] Customers()
{
    string customerList = File.ReadAllText("customers.json");
    Contact[] customer =
JsonSerializer.Deserialize<Contact[]>(customerList);
    return customer;
}
```

Now, if our implementation is correct, our email-sending functionality should be working, and an email should have already been sent. Let's proceed and try it out.

But before, remember to call PrepEmail() in your Main() method if you don't want to sit there looking at the screen for a solid minute wondering why it isn't executing correctly. Just a little advice here.

```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to the TutorialsEU
                      email marketing service.");
    Console.WriteLine($"You are currently logged in as
                      ${Sales.Rep.CurrentSalesRep.Contact.Name}");
    Console.WriteLine("Press any key to start the process");
    Console.ReadKey();
    PrepEmail();
}
```

And after running our project (Fig. 10.10).



Fig. 10.10 Successfully sent an email through our SMTP program

We did it. It works and runs great. Just as a final touch-up, let us add a message at the end of the process to inform the successful send and to either exit or restart the app.

```
SendEmail(
    from: Sales.Rep.CurrentSalesRep
    , toAll: Customers()
    , subject: emailSubject
    , body: emailBody
);

Console.WriteLine("\rEmail has been sent successfully!
                  Press any key to send another email or
                  Esc to exit the application");
ConsoleKey nextStep = Console.ReadKey().Key;
if (nextStep == ConsoleKey.Escape)
{
    Environment.Exit(0);
}
else
{
    PrepEmail();
}
```

Then, to make the app more intuitive, add a disclaimer at the very beginning informing on where the “customers.json” file has to be located for the program to work properly.

```
Console.WriteLine("Welcome to the TutorialsEU email  
marketing service.");  
Console.WriteLine($"You are currently logged in as  
${Sales.Rep.CurrentSalesRep.Contact.Name}");  
Console.WriteLine($"Make sure to include a  
'customers.json' file in  
the '{Directory.GetCurrentDirectory()}' directory.");  
Console.WriteLine("Press any key to start the process");  
Console.ReadKey();  
PrepEmail();
```

A little bit of polish goes a long way!

We are officially done! The project is complete. Obviously there is still room for improvement, be it in the user experience front, or the app functionality, but that is where you come into play. There are a multitude of ways for expanding this Mass Mail Sender Service, and we hope that you are seeing this end to this chapter as an opportunity to jump-start a much bigger application with complete features that is ready for deployment to any future client you may have.

However, there is also another option. That is to jump ship onto the next project and learn about Application Programming Interfaces, or APIs, with the next chapter, where we will be building a Discord bot from scratch. The decision is up to you.

Let's get ready for the next chapter!

10.4.3 Source Code

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_MassEmailSender

10.5 Summary

This chapter taught us:

- SMTP is a protocol used to send emails. It transfers messages from the email client to the server, and then to the recipient's mailbox. SMTP doesn't handle email reception; for that, you need protocols like POP3 or IMAP.
- SMTP can be used in .NET and .NET Core with the help of the System.Net.Mail namespace. A Gmail account can be prepared to act as the sender.

- Email clients like Thunderbird use POP3 or IMAP to receive emails. IMAP is often the recommended choice.
- The Single Responsibility Principle (SRP) states that each class in a program should have just one reason to change.
- The Open/Closed Principle (OCP) suggests that you should be able to add new features to a system without changing existing code.
- The Liskov Substitution Principle (LSP) states that if a class B is a subclass of class A, B should be able to replace A without affecting the program's functionality.
- The Interface Segregation Principle (ISP) advises against forcing clients to rely on interfaces they do not use and promotes dividing large interfaces into smaller, specific ones.
- The Dependency Inversion Principle (DIP) suggests that high-level and low-level modules should depend on abstractions, not on each other, aiding in decoupling software modules.
- JSON, or JavaScript Object Notation, is a popular, lightweight data-interchange format used for data transfer between a server and a client. It's favored due to its readability, structure, and wide acceptance as a standard.
- JSON's popularity comes from its readability and structured syntax like a JavaScript object. Its adoption as a standard across many applications and systems also contributes to its widespread use.
- Although the Newtonsoft library is widely used, the Microsoft built-in solution, the `System.Text.Json` namespace, is gradually becoming the standard for working with JSON.



This Chapter Covers

- Understanding APIs
- Learning the difference between API, REST, and SDK
- Using HTTP requests to communicate through an API
- Developing our first API application that fetches weather data
- Developing an automatic E-Commerce Discord bot

With this, we have reached the final chapter of this book. We went from learning about variables all the way to this chapter about Application Programming Interfaces (API). Covering asynchronous programming, checksums, cryptography, and many more important topics along the way, we hopefully were able to provide great enrichment to any developer out there, be it intermediate C# developers or even those from a different object-oriented programming language altogether.

And for the grand finale, as we want to work on a big API-focused project, we will attempt to develop a Discord bot dedicated to E-Commerce services. To be specific, this bot will show a list of products, and then users in the chat can write the product they want. After that, the bot will read their request and send them a direct message with the price of the product they requested.

Like many other APIs, the VoIP and instant messaging social platform “Discord” API allows us to access and interact with a specific service or application over the Internet. It is a powerful tool that enables us to expand the functionality of our bot beyond just sending emails.

By the end of this chapter, we will have a solid understanding of how to work with APIs and have developed a functional bot that demonstrates the potential of online applications.

It is an exciting transition from the mechanics of sending emails with SMTP to the dynamic possibilities of the online world through APIs.

So let us start this chapter by learning our first concept. What even is an API?

11.1 APIs

The meaning of API, or application programming interface, often raises questions for many people who may wonder what it stands for. We hear about this acronym repeatedly whenever we work with anything that needs to connect to any outside online service. In modern times, APIs have become ubiquitous. Whether using a ride-sharing app, making a mobile payment, or adjusting our home thermostat from our phone, we are utilizing an API. When you use one of these applications, it connects to the Internet and sends data to a server. The server then processes the data, executes the necessary functions, and returns it to our phone. The application then deciphers the data and presents it in a way that is easy for us to understand. That sounds familiar, but unless we ever decide to dive deeper into the rabbit hole, we never see APIs as more than just a magic bridge between two services. For the vast majority, an API is a technology that we know exists but feels outside of our realm, a tool used by only a chosen few that know something we do not know, an unreachable area of this world.

In reality, that could not be further from the truth. An API is nothing more than an intermediary between two systems, nothing more than, well, a bridge. Let me explain.

An API, or application programming interface, acts as a mediator between two software applications, facilitating communication.

Think of two friends who attempt to solve a problem together, each of them with their own know-how. How would they both communicate together? Well, with speech. Just how an API is a standardized way of communicating between two systems, speech is a standardized way of communicating between two individuals.

Seems too simple a comparison? Well, in its most simple definition, that is really it. That is what an API is, a way for two systems to communicate.

Well, that is it, then. That was the chapter! Thanks for tuning in! Hope to see you in the next video!

Just jokes. But yes, there is really not much else to it. However, although simple in concept, just as with speech, as anyone who has ever tried to learn a new language can confirm, APIs can be pretty complex in execution. So, this is what we are attempting to do in this chapter: learn how to communicate through APIs, the universal bridge of our applications.

11.1.1 What Is an API

APIs are tools that help different software components communicate with each other by following a set of rules. For example, the weather app on our phone uses APIs to get daily weather updates from a weather bureau's software system. See as an example this program.

```
using System.Net.Http;

var httpClient = new HttpClient();
httpClient.DefaultRequestHeaders.Authorization = new
AuthenticationHeaderValue("Bearer", "my-api-key");
var response =
    await httpClient.GetAsync(
        "https://weather-bureau-api.com/data"
    );
var content = await response.Content.ReadAsStringAsync();
Console.WriteLine(content);
```

Here we can see how this sends a GET request to a weather-bureau API endpoint, which requires authorization using an API key. To send the request, an instance of the `HttpClient` class is created, and the API key is set as an authorization header. The `GetAsync()` method sends the GET request. The response content is then read as a string using the `ReadAsStringAsync()` method and the `Console.WriteLine()` method is used to output the content to the console. So this code demonstrates a simple way to retrieve data from an API endpoint. Do not worry; this is just a simple demonstration, and we will go into more detail with a more detailed example shortly.

An API is like an agreement between two pieces of software. It explains how they will talk to each other using requests and responses. Developers follow the API documentation instructions to ensure their software works with the API.

Modern APIs follow specific rules, which make them easy for developers to use and understand. APIs are treated like products that are designed for specific groups of people, and they are carefully documented and managed.

APIs are also monitored to ensure they are working well and securely. For example, the weather app and server share small amounts of data to ensure they only send what they need. APIs also have a process for being created, tested, and retired, and their documentation is updated over time.

Nowadays, APIs are essential, and many businesses rely on them to make money. Thousands of APIs are available, and they help companies create new products and services that would not be possible otherwise. Nevertheless, that is not the only reason they are so essential, so why do we use APIs?

11.1.2 Why Do We Use APIs

APIs can provide businesses access to a broad range of resources, enabling cross-platform communication and improving workflow efficiency. For example, C# code snippets can be used to create an API that powers the connection between a commercial carrier policy administration system and a third-party database with essential vehicle data for policy underwriting.

Real-Time Quoting

Real-time quoting is one of the primary competitive advantages that APIs offer. Businesses can provide potential clients with quick policy quotes by enabling agile API data quality and delivery speed. In many cases, the speed of policy quote delivery becomes the deciding factor for clients when choosing a policy provider.

Efficiency

Efficiency is a huge benefit for API integrations. Third-party APIs can handle many background tasks in Web-centric information gathering and computation tasks, reducing the need for extensive IT department labor. Integrating APIs also enables businesses to outsource business process parts, reducing costs associated with building in-house capabilities.

APIs can help streamline access to a vast asset database in the property and casualty insurance sector. By integrating well-designed APIs with batch-processing analytical methods, businesses can create new channels for business development and financial growth.

Simplicity

APIs also offer simplicity, which reduces the need for manual data manipulation. Using APIs allows data to be automatically distributed externally and internally, allowing businesses to eliminate redundant work.

Customization

The API integration can also be customized to fit specific enterprise needs. Enterprise developers can deploy their central platform alongside APIs, allowing customization to fit each customer's needs. Additionally, API integration eliminates the risk of data transfer loss during low connectivity periods and drives down data handling costs.

Other Benefits

Other API integration benefits include options for mobile development, partnering and onboarding opportunities, decreased time to market for new products, competitive technological advantage, compliance with regulatory requirements, and future-proofing through a focus on innovation. By utilizing APIs, businesses can gain a competitive edge, streamline workflows, and reduce costs, making them an essential tool for companies of all sizes.

This covers the concept and the why, but we are still left with the how. How do APIs work?

11.1.3 How APIs Work

Think of an API as a waiter in a restaurant. The waiter acts as an intermediary between the customer and the kitchen. The customer tells the waiter what they want to order, and the waiter takes that order to the kitchen, where the food is prepared. When the food is ready, the waiter brings it back to the customer.

Similarly, an API acts as an intermediary between a client (such as a Web or mobile application) and a server. The client requests to the API, specifying what data or functionality it needs. The API processes the request and responds with the requested data or functionality. Just like a waiter in a restaurant, the API provides a standardized way for the client to interact with the server without knowing the specific details of how the server works.

APIs are often considered contractual agreements between two parties, with the help of documentation that represents how one party's software will respond to a remote request structured in a particular way from another party. APIs simplify the integration of new application components into existing architecture, enabling business and IT teams to collaborate more effectively. As digital markets continuously shift, new competitors can change the entire industry with a new app, and businesses need to adapt quickly to stay competitive. Cloud-native application development is an identifiable way to increase development speed, which relies on connecting a microservices application architecture through APIs. Let us try to exemplify this process with some pseudocode:

Listing 11.1 Pseudocode Exemplifying the Process of Using an API to Request Data

```
// Party 1 sends a remote request to Party 2's API
Party1.sendRequest(API_URL, requestData);

// Party 2's API receives the request and sends back a response
API.receiveRequest(requestData, function(responseData) {
    Party1.handleResponse(responseData);
});
```

In this example, Party1 is a software application that needs to send a request to an API provided by Party2. API_URL is the URL of the API, and requestData is the data that Party1 wants to send to the API.

API is the software that implements the API provided by Party2. It receives the request from Party1, processes it, and sends back a response in responseData. Party1 then handles

the response by calling handleResponse(responseData), which can be any code that Party1 needs to execute to handle the response.

Connecting our infrastructure through cloud-native app development using APIs simplifies the process, allowing us to share data with customers and external users. Public APIs provide unique business value, enabling us to simplify and expand how we connect with partners and potentially monetize our data. For instance, Google Maps API is a famous example.

In essence, APIs enable us to open access to our resources while maintaining control and security. We can decide how to open access and to whom. Good API management, which includes using an API gateway, is crucial for API security. We can connect to APIs and create applications that consume the data or functionality exposed by APIs using a distributed integration platform that connects everything, including legacy systems and the Internet of Things (IoT).

All of this theory about APIs might sound familiar to some of us if we have ever worked with other developers. However, we might have heard a different term, like “The Rest API,” or “The SDK.” Words that often seem to be used interchangeably to describe the same concept. Well, if you think that, you are not alone. And understandably so, as these terms refer to a pretty similar definition. Nonetheless, they do not mean the same thing. So what do they mean?

11.1.4 API vs REST vs SDK

We hear three terms that seem to mean the same thing but do not simultaneously. We get the following if we attempt to differentiate them from a purely technical point of view.

An API is a set of tools, protocols, and routines utilized to build software applications and facilitate the interaction of different software components. REST is a software architectural style that defines limitations for creating Web services and is commonly used in building APIs. It uses HTTP requests to exchange data. On the other hand, an SDK is a compilation of software development tools that enable developers to create a platform or system-specific applications. It comprises APIs, libraries, documentation, sample code, and other resources that assist in software development. Confusing? Let me clear that up.

Let us go back to our Waiter example.

API

An application programming interface, or API for short, in our example, will be represented by our waiter. Between us, the customer, and the service, the kitchen, there is a medium, the waiter, that allows fluid communication between both parties. This, however, is a way of communicating that, unless trained thoroughly, is difficult to understand and follow along. Well, for that, we have REST.

REST

Representational state transfer, or REST for short, in our example, is the language all parties speak. Just as in English, where grammar and syntax govern how the waiter communicates with each party, REST is a set of rules about using APIs to simplify communication between different parties. REST specifies how an API should receive requests and send responses in a standardized format. But what if we do not know what we need to ask for? How can I ask for my carbonara if I don't know how to ask for it? It can't possibly be expected from me to list out the ingredients I need and how to cook the dish, right? No, we don't need to, so for that we have SDKs.

SDK

In our example, a Software Development Kit, or SDK for short, would be like a menu. Instead of giving the waiter all the individual ingredients and cooking processes, we can just refer to a pre-packaged menu item that already contains all the necessary steps to request our favorite dish. As in SDKs, these present the user with a simplified interface that already delivers preset requests for certain actions, instead of having to construct that request manually.

So, in short, we can summarize the process with the following sentence:

As the customer (One Application service), we use the menu (SDK) to make a request, using the English language (REST) to communicate through the waiter (API) to the kitchen (Other Application service).

Furthermore, if we go back to our technical point of view, we see that APIs and REST are technologies used for building software applications and Web services, while an SDK is a collection of development tools and resources for building said software applications.

Hopefully, this makes the difference between the different terms clear! Great, now we should be able to answer someone if asked what an API is. However, being able to say what it is is not the same as using it. For that, we will have to learn a bit about HTTP requests.

Although we have mentioned them already, we have yet to discuss them. Thankfully, learning what there is to know about the `HttpClient` class in C# is not that big of a step, so let us do that as our last bit of theory before jumping right into the cold water and starting to build some real projects.

11.2 The HTTP Class

The `HttpClient` class, a component of the `System.Net.Http` namespace, is a tool utilized in C# for making HTTP requests and receiving HTTP responses in a modern and asynchronous manner.

One of the key advantages of utilizing the `HttpClient` class is its support for asynchronous programming. This allows for the execution of other tasks while a request is being processed, thereby improving the performance and responsiveness of an application.

To utilize the `HttpClient` class, an instance must be created first. This can be done through the following code snippet:

```
using System.Net.Http;

HttpClient client = new HttpClient();
```

Once an instance of the `HttpClient` class has been created, it can be utilized to make various types of HTTP requests, such as GET, POST, PUT, and DELETE. The class offers a wide range of methods and options for the manipulation of HTTP requests and responses (Table 11.1).

Table 11.1 A list of five common methods in the `HttpClient` class

HttpClient methods	Description
<code>DeleteAsync(String)</code>	Sends a DELETE request to the specified Uri as an asynchronous operation
<code>Dispose()</code>	Releases the unmanaged resources and disposes of the managed resources used by the <code>HttpMessageInvoker</code>
<code>GetAsync(String)</code>	Sends a GET request to the specified Uri as an asynchronous operation
<code>GetStringAsync(String)</code>	Sends a GET request to the specified Uri and returns the response body as a string in an asynchronous operation
<code>GetStreamAsync(Uri)</code>	Sends a GET request to the specified Uri and returns the response body as a stream in an asynchronous operation

For example, the `GetAsync` method allows for the execution of a GET request to a specified URL, returning a `Task<HttpResponseMessage>` object that represents the response.

```
using System.Net.Http;
using System.Threading.Tasks;

HttpClient client = new HttpClient();
Task<HttpResponseMessage> response =
    client.GetAsync(
        "https://www.example.com"
    );
```

The `GetStringAsync` and `GetStreamAsync` methods can also be utilized to retrieve the contents of a response as a string or a stream, respectively.

```
using System.Net.Http;
using System.Threading.Tasks;
```

```
HttpClient client = new HttpClient();
Task<string> response = client.GetStringAsync("https://www.example.com");
```

In addition to the aforementioned methods, the `HttpClient` class also allows for the manipulation of headers, timeouts, and other configurations.

```
using System.Net.Http;
using System.Threading.Tasks;

HttpClient client = new HttpClient();
client.Timeout = new TimeSpan(0, 0, 30); // 30 seconds
```

NOTE The `HttpClient` class is designed to be employed as a “singleton” object, meaning that a single instance should be created and reused throughout an application. This approach improves performance and reduces the risk of socket exhaustion.

“Alright, this seems approachable. Can we now jump into using all of that to make an application?”

You know what? Sure. Let us jump right into it.

11.3 Working with APIs

At the end of the day, the best way to understand a concept is to use it. Be it in maths, our driver’s license, or learning to use APIs. After a round of preparatory theory, we need to jump into some practical examples to get a feel of the concept at hand.

So this final chapter will be special in this regard since it will contain one extra small project before we get into building this book’s flagship small project.

And what will that look like? A simple weather API app. Let us take a closer look at this.

11.3.1 What a Simple API Project Would Look Like

For this example project, we will make use of the OpenWeatherMap API. As its name suggests, it is open source and free to use, meaning that we can use it for testing as much as we desire.

Other than that, there is no more preparation needed. Let us jump right in and learn whatever we might need along the way. So then, what is this project about?

We will build a simple console application using the OpenWeatherMap API to retrieve weather data for a specific city. It should look something like this by the end:

```
class Program
{
    static void Main(string[] args)
    {
        // Create a new HttpClient object
        HttpClient client = new HttpClient();

        // Set the base address for the API
        client.BaseAddress = new Uri("http://api.openweathermap.org/");

        // Set the API key
        String apiKey = "API-KEY";

        // Set the city ID for the city you want to get weather data for
        String cityId = "2172797";

        // Make a GET request to the API
        HttpResponseMessage response =
            client.GetAsync(
                $"data/2.5/weather?id={cityId}&appid={apiKey}"
            ).Result;

        // Read the response as a string
        String? responseString = response.Content.ReadAsStringAsync().Result;

        // Print the response to the console
        Console.WriteLine(responseString);
    }
}
```

We will use the `HttpClient` class for that. In this application, the `HttpClient` class will make a request to the OpenWeatherMap API through, as we can see in the above example, the `GetAsync` method.

Great to know, but let us go a bit more in-depth and discuss each line while implementing this little project step by step.

The OpenWeatherMap API Step by Step

Starting in the same way as all of our projects, we will have our base project be our “Program” Class and `Main()` method. Simple as this:

```
class Program
{
    public static void Main(string[] args)
    {

    }
}
```

Now, since the `HttpClient` class is part of the .NET Framework, there is no need for any using statements. So let us just get into our first step, creating an `HttpClient` object.

An `HttpClient` object is like a browser that you can use to make requests to a Web site. In this case, the Web site is the OpenWeatherMap API. To create an `HttpClient` object, you need to use the following line of code:

```
HttpClient client = new HttpClient();
```

This line of code creates a new `HttpClient` object and assigns it to the variable named `client`. Next, we will need a base address. The base address is the URL of the Web site you want to make the request to. In this case, the Web site is the OpenWeatherMap API, and the base address is “<http://api.openweathermap.org/>”

```
client.BaseAddress = new Uri("http://api.openweathermap.org/");
```

Now, to use the OpenWeatherMap API, you need to have an API key. An API key is like a password that you use to access the API. To get us one of those, we need to register at <https://openweathermap.org/>. Doing so is fairly straightforward.

Once you are on the correct Web site, make sure to either sign in or sign up, depending if we already have an account or not. As soon as you are in, the Web site will prompt you to confirm your email. Just follow their instructions and right after your confirmation email, you should get a second email with all of your data, including the API key.

And that is it, quite simple! However, just in case, if you do not get that key or there is anything else wrong with the process, on the page (direct link) https://home.openweathermap.org/api_keys, we should be able to see our API keys as well, and if none are there, we can create them there directly.

Alright, so once you have an API key, you can set it in a new string variable, like so:

```
String apiKey = "API-KEY";
```

With that we got our setup already done; now, we need an actual thing to request. In our case, we simply want to get the weather data on the city of, for example, Berlin. Naturally, we can go with any city we want; there is no need to go with the same one as us.

To get the weather data for a specific city, you need to know its city ID. You can find the city ID for any city by searching for it on the homepage of openweathermap.org. There you should be able to find a search bar where you can freely type in whatever city you want.

Once you have chosen your city, you should have the city ID in the search bar right in the URL (Fig. 11.1).

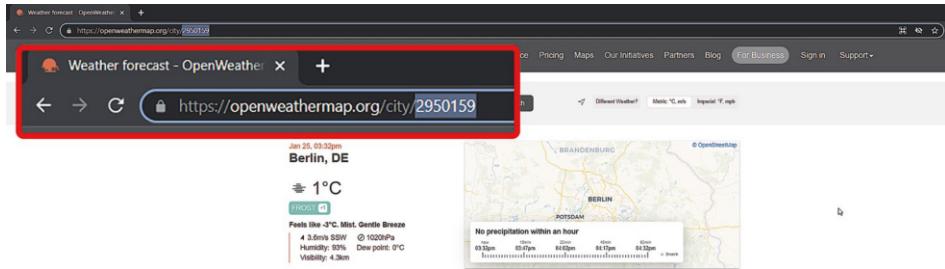


Fig. 11.1 Get a city ID from openweathermap.org by searching your city of choice and getting the ID right from the URL

As soon as we have the city ID, we can set it in our project with a new string variable, like so:

```
String cityId = "2950159";
```

As previously stated, we are using Berlin for our city, so we used the ID “2950159.” If we wanted any other city, we would simply have to replace that ID with a new one.

Now that we have set the base address, API key, and city ID, we can finally make the GET request to the API. To make the GET request, we need to use the `GetAsync` method of the `HttpClient` object, passing in the endpoint of the API as an argument. This endpoint is the specific URL of the API that we want to make the request to. In this case, the endpoint is “`data/2.5/weather?id={cityId}&appid={apiKey}`.”

```
HttpResponseMessage response =
    client.GetAsync(
        $"data/2.5/weather?id={cityId}&appid={apiKey}"
    ).Result;
```

Once the GET request has been made, the API will respond with a message that contains the weather data. This message is stored in the `response` variable. To read the message, you need to use the `ReadAsStringAsync` method of the `Content` property of the `response` object.

```
var responseString = response.Content.ReadAsStringAsync().Result;
```

This line of code reads the message from the `response` object and assigns it to the `responseString` variable.

The final step is to display the weather data on the console. To do this, you will use the `Console.WriteLine` method.

```
Console.WriteLine(responseString);
```

This line of code displays the message stored in the `responseString` variable on the console.

At this point, your application should be able to retrieve the weather data for the specified city and display it on the console. The output should be a string of JSON data, containing all the weather information for the city. However, if we output it now, it will just be a long string of data that is practically unreadable, so let us, before we finish, quickly convert this into a more readable format.

For that, we must first add the `System.Text.Json` using statement to the project.

```
using System.Text.Json;
```

```
class Program
{
```

With that added, we now have access to the `JsonSerializer` class. Using that, we can first deserialize the response into a `dynamic` variable, like so:

```
dynamic? jsonResponse = JsonSerializer.Deserialize<dynamic>(responseString);
```

Then, we indent the resulting text using `JsonSerializeOptions` and by reserializing with those new options added.

```
JsonSerializerOptions? jsonOptions = new JsonSerializerOptions {
    WriteIndented = true };
dynamic? prettyPrintedResponse = JsonSerializer.Serialize(jsonResponse,
    jsonOptions);
```

And at last, we take our previous `Console.WriteLine()` method where we displayed the response and move it to the bottom, also changing the content that is being written from `responseString` to `prettyPrintedResponse`.

```
Console.WriteLine(prettyPrintedResponse);
```

With all that, we should have a finished project ready to launch, so let us check what we get as a response if we use Berlin as the Target location.

```
{
    "coord": {
        "lon": 13.4105,
        "lat": 52.5244
    },
    "weather": [
```

```
{  
    "id": 804,  
    "main": "Clouds",  
    "description": "overcast clouds",  
    "icon": "04d"  
},  
],  
"base": "stations",  
"main": {  
    "temp": 276.29,  
    "feels_like": 272.2,  
    "temp_min": 275.37,  
    "temp_max": 277.03,  
    "pressure": 1014,  
    "humidity": 89  
},  
"visibility": 10000,  
"wind": {  
    "speed": 4.92,  
    "deg": 30,  
    "gust": 8.05  
},  
"clouds": {  
    "all": 100  
},  
"dt": 1674832954,  
"sys": {  
    "type": 2,  
    "id": 2011538,  
    "country": "DE",  
    "sunrise": 1674802580,  
    "sunset": 1674834088  
},  
"timezone": 3600,  
"id": 2950159,  
"name": "Berlin",  
"cod": 200  
}
```

We have now successfully built a console application that retrieves weather data for a specific city using the OpenWeatherMap API. That was fairly simple, right? Well we hope that this example project was able to show us in better form how to work with these APIs and what tools we have at hand to get our data. Also, we have already now started working with an external service, so doing the same now with the Discord API should be pretty simple.

Judging by the number of pages left, this task will surely not be a small one. Nonetheless, we should be prepared and ready for any challenge we might face during our journey. So without delaying this any longer, it is time for our final boss.

11.4 A Discord E-Commerce Bot

As we conclude this final chapter, we find ourselves at the culmination of our learning journey throughout this book. Over the preceding chapters, we have mastered a multitude of concepts in C# programming and explored in-depth the realm of cybersecurity and online applications. It's time to stitch together all the knowledge we have accumulated and showcase the versatility and applicability of these concepts.

Our final project will utilize a range of key topics we've covered throughout this book. We'll be working with APIs, a crucial aspect of modern applications enabling disparate systems to interact and share data. We'll be applying this understanding to engage with the Discord API. Our grasp of JSON files will come into play as we use them to manage our product data. Our knowledge about network protocols and handling data in transit becomes relevant as we navigate sending and receiving messages over Discord. Furthermore, the concept of concurrency will be implemented to efficiently monitor our Discord channel for incoming messages continuously.

So then, with much knowledge and many projects under our belt, we can finally tackle this final application.

11.4.1 The Project

For this grand finale, we will get to build a Discord E-Commerce bot. As we want to focus on the API functionality, this app will take a list of products that we provide as a JSON file, post it to a channel that we provide, and after a few seconds, read all of the messages that were sent to the chat so as to find those interested in the product and send them via direct message the price of the given product. This message reading repeats after 5s until we press the Esc key on our keyboard. So what is it that we need to build?

We need:

- A connection to our Discord server of choice, including the specific Text Channel and the ability to write direct messages
- To create and read a list of available products and prices
- To send a message to the Text Channel to list our products
- Then, in delays of 5s, we need to read the Text Channel to find any responses that contain a product on our list
- When found, to direct message them the price of that list and delete their message from the Channel

Naturally, these steps imply several intermediate steps to achieve such mechanisms. Those are the ones that we will need to pay special attention to. As we will work with the Discord API, there will be some steps that may differ in our current experience; as this book has been written in early 2023, the technology may change and update. However, even if much changes, the code to this project should always be easy and clear enough to allow the understanding of what needs to be modified if these systems become deprecated at some point. Moreover, at the very least, this will always serve well as a guide on how to generally work with any given API, no matter the time period.

11.4.2 Our Code

As usual, we must start by creating a class Program with a static Main method. As we did with every previous project, we made use of this base to attempt to maintain consistency between the projects.

So then, create a new project, clean it, and add the following lines.

```
class Program
{
    public static void Main(string[] args)
    {

    }
}
```

Starting with the basics, let us warm up by introducing our user to our application. As with any application, this is up to you and can change depending on your use case, but for us, we will go with a simple yet effective introduction. So within our main method, we will have the following.

```
Console.WriteLine("Welcome to the Transcendental
ECommerce Discord Online Bot Advanced Console app,
or TECDOBACA in short");
Console.WriteLine("To start the bot press any key");
Console.ReadKey();
```

We can introduce our application in any way we desire, so be creative!

Next, let us wire up our app to Discord. As mentioned before, this will be our first step toward our functionality. “A connection to our Discord server of choice, including the specific Text Channel and the ability to write direct messages.” This can be achieved through Discord’s own API.

In cases like these, the use of external APIs and libraries is not as simple as writing our method or class, then “Ctrl + .” to implement the namespace, as we need to implement this library first into our application. Let us attempt this to exemplify this behavior.

Say we wanted to use the “DiscordBot” class within the Discord namespace.

```
using Discord;

class Program
{
    public static void Main(string[] args)
    {
        Console.WriteLine("Welcome to the Transcendental
                           ECommerce Discord Online Bot Advanced Console app,
                           or TECDOBACA in short");
        Console.WriteLine("To start the bot press any key");
        Console.ReadKey();

        DiscordBot bot = new DiscordBot("DiscordBot code");
    }
}
```

Naturally, this does nothing: there is no such thing as a DiscordBot class or a Discord namespace yet. So this will require a bit more work than just using it. That is going to be a pretty big first step already!

So, what are the specifics that we now need? Well, we need a new namespace, and a new class, as we have just seen. Using the Single responsibility principle, we should separate this from our Program.cs file, so let us create a new class within our project.

As we already know, by right-clicking on our project, then clicking on “Add” and “Class...,” we pop up a new window where we can create a new class. Make sure to call this class `DiscordBot` (Fig. 11.2).

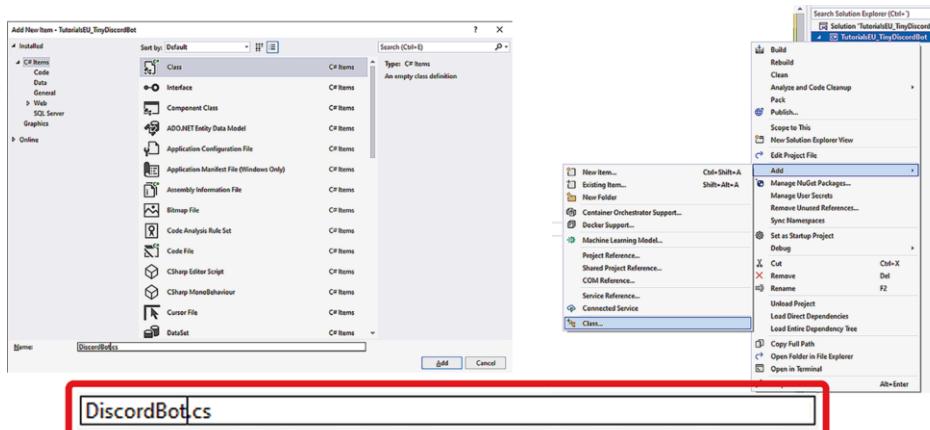


Fig. 11.2 Creating a new `DiscordBot` Class to serve as the connection to the Discord API

Once we have our DiscordBot Class, we have to prepare it. For our purposes, we can set it in the “Discord” namespace and make our internal class a simple public class, like so:

```
namespace Discord;
public class DiscordBot
{
}
```

This will already remove most errors from our Program class, since now there is a namespace to use the using statement to, as well as a public DiscordBot class. However, with our attempt, we still are not able to pass in a string as we would need. As a matter of fact, our DiscordBot class does nothing yet. Well, then, let us build up our DiscordBot class before we go back to our program. Alright then, how do we do this?

Our DiscordBot class represents a Discord bot that can send requests to the Discord API. It will contain methods for creating and sending HTTP requests, as well as fields for storing the bot’s authentication token and the base URL of the Discord API.

To do so, we can start implementing the “CreateRequest” method.

```
public HttpRequestMessage CreateRequest(HttpMethod method, string path)
{
    // method body here
}
```

This method creates a new HttpRequestMessage object that can be used to make an HTTP request to the Discord API. It takes two parameters: the HTTP method to use for the request (e.g., GET, POST) and the path of the API endpoint to send the request to.

Great, now, inside the “CreateRequest” method, a new HttpRequestMessage object has to be created.

```
public HttpRequestMessage CreateRequest(HttpMethod method, string path)
{
    HttpRequestMessage request = new(method, baseUrl + path);
}
```

This line creates a new HttpRequestMessage object with the specified HTTP method and URL. The URL is constructed by concatenating the baseUrl field (which contains the base URL of the Discord API) with the path parameter (which contains the path of the API endpoint being requested).

At this point, the baseUrl has yet to be defined. As this is the URL to the Discord API, we simply need to create a read-only variable that will hold this information; this will be an explicitly private variable that we can create like so:

```
public class DiscordBot
{
    public HttpRequestMessage CreateRequest (HttpMethod method, string path)
    {
        HttpRequestMessage request = new(method, baseUrl + path);
    }
    readonly string baseUrl = $"https://discordapp.com/api";
}
```

This will complete our initial Request message; however, authentication will still be needed to be given access to the API. So for the next step, the Authorization header is added to the request.

```
public HttpRequestMessage CreateRequest (HttpMethod method, string path)
{
    HttpRequestMessage request = new(method, baseUrl + path);
    request.Headers.Add("Authorization", "Bot " + botToken);
}
```

This line adds an Authorization header to the HttpRequestMessage object. The header value is set to “Bot” followed by the bot’s authentication token stored in the botToken field. This header is required by the Discord API to authenticate requests from bots.

This needs a “botToken” variable, which, as with baseUrl, has not yet been defined. Similarly to our previous solution, we must implement a read-only string variable that will hold this information.

```
public class DiscordBot
{
    public HttpRequestMessage CreateRequest (HttpMethod method, string path)
    {
        HttpRequestMessage request = new(method, baseUrl + path);
        request.Headers.Add("Authorization", "Bot " + botToken);
    }
    readonly string botToken;
    readonly string baseUrl = $"https://discordapp.com/api";
}
```

And at last, we can return the HttpRequestMessage object created in the method to complete the “CreateRequest” method.

```
HttpRequestMessage request = new(method, baseUrl + path);
request.Headers.Add("Authorization", "Bot " + botToken);
return request;
```

This should serve us further on to create our request to the Discord API. Now, we need a method that is responsible for sending the created request to the Discord API. As this is an API call, we should implement that in an asynchronous way to avoid unnecessary application freezes. So now, the `SendAsync` method is defined:

```
public async Task<HttpResponseMessage> SendAsync (HttpRequestMessage request)
{
    // method body here
}
```

This method sends an HTTP request to the Discord API and returns the response. It takes an `HttpRequestMessage` object as a parameter, which contains the details of the request. This parameter, as we may have already guessed, comes from the previous method we just created.

Inside the `SendAsync` method, an HTTP request is sent using the `httpClient` object:

```
public async Task<HttpResponseMessage> SendAsync (HttpRequestMessage request)
{
    HttpResponseMessage httpResponse = await httpClient.SendAsync (request);
}
```

This line sends the HTTP request contained in the `HttpRequestMessage` object using the `SendAsync` method of the `HttpClient` object stored in the `httpClient` field. The `await` keyword is used to asynchronously wait for the response.

For this to work, we need an `HttpClient` object. We can simply define one as we did without previous variables.

```
readonly HttpClient httpClient = new ();
readonly string botToken;
readonly string baseUrl = $"https://discordapp.com/api";
```

Next, as with any external request, API calls can lead to errors. One such exception could be an unauthorized access, or something as simple as an unreachable server. Leaving these errors unhandled can lead to errors blocking the execution flow of our program.

We can check for an unsuccessful status code with this:

```
if (!httpResponse.IsSuccessStatusCode)
```

However, we need a way to handle that error if it occurs. In our case, we will simply throw an exception to the user to acknowledge a possible misuse or error in the process. For that, let us create a new class that has the responsibility to do just that. As before,

create a new class called `HttpBotException`. This class named `HttpBotException` extends the built-in `Exception` class. We get this result:

```
public class HttpBotException : Exception
{
}
```

Now, this needs to be able to take an `HttpResponse` and process it as a string. This can be done using a constructor, like the following.

```
public HttpBotException(HttpResponseMessage response)
```

Wait, a constructor? We haven't covered constructors in this book yet. What is a constructor now?

Constructors in C#

A constructor is a special method that is automatically called when an object of a class is created. Constructors are used to initialize the object's state (i.e., its fields and properties) to some initial values.

In this case, we used a constructor in the `HttpBotException` class to ensure that each instance of the class is properly initialized with the response content from a failed HTTP request.

When an HTTP request fails, the Discord API returns an error message as the content of the response. In order to handle these errors and provide meaningful feedback to the user, we need to capture this error message and store it somewhere.

By defining a constructor that takes an `HttpResponseMessage` object as its input, we can capture the response content and store it in an instance of the `HttpBotException` class. This allows us to create a new instance of the `HttpBotException` class for each failed HTTP request and provide access to the error message using the `Response` property.

So then in our case, the constructor for `HttpBotException` assigns the response content to the `Response` property, like so:

```
public HttpBotException(HttpResponseMessage response)
=> Response = response.Content.ReadAsStringAsync().Result;
```

We assign the result of calling the `ReadAsStringAsync()` method on the `Content` property of the `HttpResponseMessage` object to the `Response` property of the `HttpBotException` object. This is done to capture the error message returned by the Discord API in the event of an HTTP error.

In general, constructors are used to initialize the state of an object when it is created. They can take parameters (as in this case) to allow for customization of the initialization

process. Constructors are automatically called when an object is created, so they ensure that the object is properly initialized before it is used.

After this, we simply need to create a public getter-only property named Response of type string that can be used to retrieve the response content from the `HttpBotException` object, like so:

```
public string Response { get; }
```

Leading to this final `HttpBotException` class:

Listing 11.2 The Final `HttpBotException` Class Used to Handle HTTP Errors in a Discord Bot That Uses the `DiscordBot` Class

```
public class HttpBotException : Exception
{
    public HttpBotException(HttpStatusCode response)
        => Response = response.Content.ReadAsStringAsync().Result;

    public string Response { get; }
}
```

Now, we can go back to our `DiscordBot` class to implement the `HttpBotException` class to handle any possible exception thrown by the Discord API.

```
public async Task<HttpResponseMessage> SendAsync(HttpRequestMessage request)
{
    HttpResponseMessage httpResponse = await httpClient.SendAsync(request);
    if (!httpResponse.IsSuccessStatusCode)
        throw new HttpBotException(httpResponse);
}
```

Great, and finally, we have to make sure to return our `HttpResponse`.

```
return httpResponse;
```

That is it for the `SendAsync` method. Next, we need a method that creates an HTTP request using the `CreateRequest` method and sends it using the `SendAsync` method. It takes two parameters: the HTTP method to use for the request (e.g., GET, POST) and the path of the API endpoint to send the request to.

```
public Task<HttpResponseMessage> RequestAsync(HttpMethod method,
string path)
```

```
{  
    // method body here  
}
```

Inside the RequestAsync method, a new HttpRequestMessage object is created using the CreateRequest method. It calls the CreateRequest method with the specified HTTP method and API endpoint path to create a new HttpRequestMessage object.

```
HttpRequestMessage request = CreateRequest(method, path);
```

Then, the SendAsync method is called with the HttpRequestMessage object to send the request and retrieve the response. Calling the SendAsync method with the HttpRequestMessage object created in the previous step, and returning the resulting Task<HttpResponseMessage> object.

```
return SendAsync(request);
```

Now, we need to overload. The RequestAsync method is overloaded with a second version that takes an additional content parameter. This method is similar to the previous one, but it also includes a request body (i.e., content) in the HTTP request. It takes three parameters: the HTTP method to use for the request (e.g., GET, POST), the path of the API endpoint to send the request to, and the content to include in the request body.

```
public Task<HttpResponseMessage> RequestAsync(HttpMethod method, string  
path, object content)  
{  
    // method body here  
}
```

Inside the RequestAsync method with content, a new HttpRequestMessage object is created using the CreateRequest method. Calling the CreateRequest method with the specified HTTP method and API endpoint path to create a new HttpRequestMessage object.

```
HttpRequestMessage request = CreateRequest(method, path);
```

The content is serialized to JSON format and added to the request body as a StringContent object. We use the System.Text.Json namespace to serialize the content object to a JSON string using the JsonSerializer.Serialize method. So, after adding the two missing using statements before our namespace declaration, we add the following line.

```
string jsonPayload = JsonSerializer.Serialize(content);
```

The resulting JSON string is then used to create a new `StringContent` object with UTF-8 encoding and a content type of `application/json`. Finally, the `StringContent` object is added to the `HttpRequestMessage` object's `Content` property.

```
StringContent stringContent = new(jsonPayload, Encoding.UTF8,  
"application/json");  
request.Content = stringContent;
```

The `SendAsync` method is called with the `HttpRequestMessage` object to send the request and retrieve the response. The `SendAsync` method with the `HttpRequestMessage` object created in the previous step returns the resulting `Task<HttpResponseMessage>` object.

```
return SendAsync(request);
```

With this, we have our methods ready and we are just missing one final step to complete this class. We need to define a `DiscordBot` constructor. This constructor initializes a new `DiscordBot` object with the specified `botToken` authentication token.

NOTE The `=>` syntax is a shorthand way of writing a constructor with a single assignment statement.

```
public DiscordBot(string botToken)  
=> this.botToken = botToken;
```

And that is it. Now, the `DiscordBot` class provides methods for sending HTTP requests to the Discord API. The class includes methods for creating and sending HTTP requests, handling HTTP errors, and serializing JSON payloads. And finally also includes a constructor that initializes a new `DiscordBot` object with an authentication token.

There we go! Now that we are done with this class, we can get back to our `Program` class and start using it. So where did we leave off last time? Right, this was our code in our `Program` class:

```
using Discord;  
  
class Program  
{  
    public static void Main(string[] args)  
    {  
        Console.WriteLine("Welcome to the Transcendental  
        ECommerce Discord Online Bot Advanced Console app,  
        or TECDOBACA in short");  
        Console.WriteLine("To start the bot press any key");  
        Console.ReadKey();  
    }  
}
```

```

DiscordBot bot = new DiscordBot("DiscordBot code");
}
}

```

Well, as we can see, right now there are no issues to speak off. Mind you, it is not yet working in any way, but we are getting closer. So what is this class we just created good for? How do we use the DiscordBot class here? By adding our own bot token to the parameter of the newly created DiscordBot object. What token? That is where we will need to get on to Discord to get our own token to use in this example. Luckily, that is not a very difficult task, so let us get one.

DISCLAIMER The following steps are valid as of early 2023. This may change over time. However, the general direction should stay the same.

Getting a Discord Developer API Bot Token

After logging into your Discord account, go to the Discord Developer Portal (<https://discord.com/developers/applications>). Click the “New Application” button in the top-right corner of the screen. Give your application a name and click the “Create” button (Fig. 11.3).

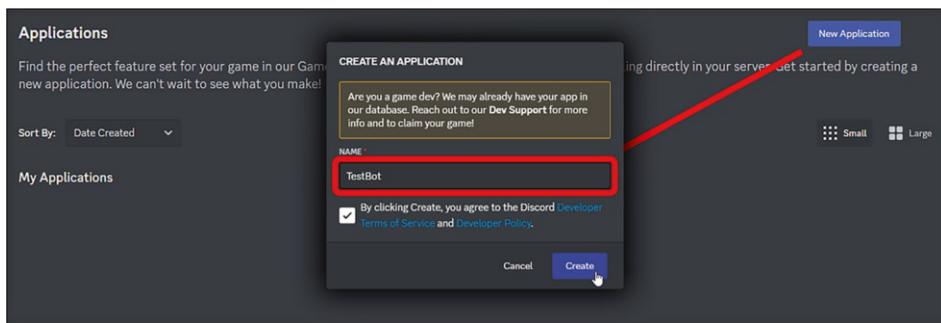


Fig. 11.3 Creating our application in the Discord Developer Portal

On the left-hand side of the screen, click the “Bot” tab. Click the “Add Bot” button. Give your bot a name and click the “Create” button. Under the “Token” section, click the “Copy” button to copy your bot token to the clipboard (Fig. 11.4).

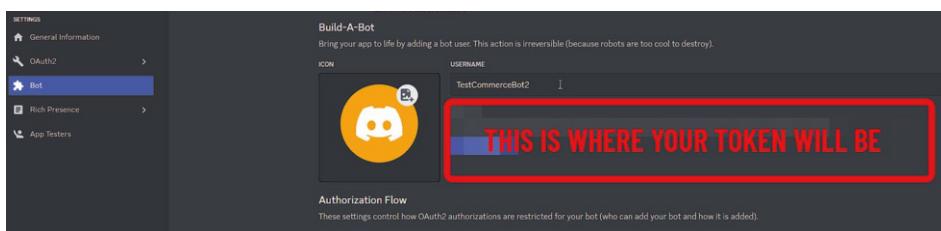


Fig. 11.4 Getting our Bot Token from our new application

Now do not forget to keep your token secure! Do not share it with anyone or commit it to a public repository. This typically is avoided by using a “config” class with this information; however, as this chapter is already gigantic, we did simplify the project not to include this file. Other than that, that’s it! You now have a Discord developer bot token that you can use to authenticate your bot and interact with the Discord API. With the copied token at hand, go back to the Program class and paste it where we wrote “DiscordBotDeveloperToken.”

```
public static void Main(string[] args)
{
    Console.WriteLine("Welcome to the Transcendental
        ECommerce Discord Online Bot Advanced Console app,
        or TECDOBACA in short");
    Console.WriteLine("To start the bot press any key");
    Console.ReadKey();

    DiscordBot bot = new DiscordBot("DiscordBotDeveloperToken");
}
```

Great, if we now run it, we will see that a connection is made with no exceptions thrown whatsoever. This will become more obvious once we start requesting data from the API though. So to get there we must now get our Discord channel in our code. And how will we do that? With another class, the DiscordChannel class. Well, if we want a proper Discord bot, we will be needing this. So let’s keep going and create a new class called DiscordChannel.

In this new class we will be implementing all of the methods responsible for reading the chat, sending the initial message, sending direct messages, and even deleting the read messages. So then, what is the first step? Well, we need to place ourselves within the Discord namespace again.

```
namespace Discord;
public class DiscordChannel
{}
```

Alright, so this will be part of our “Discord” library as well; makes sense. What now? We need a constructor, this class needs initializing before anything else. So let us do that right now.

First, we will be needing a new read-only DiscordBot object, as we need to communicate with the API to be able to make our requests.

```
public class DiscordChannel
{
    readonly DiscordBot bot;
```

Next, we need a read-only resourceAddress variable. This variable represents the resource address of the channel.

```
readonly DiscordBot bot;
readonly string resourceAddress;
```

It is constructed using the channelId parameter that is passed to the constructor. What constructor? The DiscordChannel class constructor.

```
public DiscordChannel(DiscordBot bot, string channelId)
{
}
```

As we can see, this constructor takes a DiscordBot and string parameter. This will be responsible for setting our previously created resourceAddress with a provided channel ID as well as setting the DiscordBot variable declared before with a given DiscordBot. This would look something like this:

```
readonly DiscordBot bot;
readonly string resourceAddress;
public DiscordChannel(DiscordBot bot, string channelId)
{
    resourceAddress = $"{"/channels/{channelId}}";
    this.bot = bot;
}
```

By initializing the bot and resourceAddress fields in the constructor, we ensure that each DiscordChannel object has the necessary information to communicate with the Discord API for a specific channel. This allows us to read and write messages to and from that channel.

Once initialized, we are ready to create our methods. We first need to define a public async method called ReadMessages.

```
public async Task<IEnumerable<UserMessage>> ReadMessages()
{
}
```

This method returns an IEnumerable of UserMessage objects. But wait, what is UserMessage now? Well, a new class we need to define, that is what it is. Essentially, we need a message “template.” So we need a lightweight syntax for defining immutable data types, which makes it easy to define types that represent data without having to write a lot of boilerplate code. If we remember from previous chapters, something that comes to mind that ticks all of the boxes would be a “record.” That’s right, that is what we need. Let

us create a new class, call it “userMessage,” implement the following code, and check out what it does for us.

```
namespace Discord;  
public record UserMessage(string MessageId, string UserId, string Content);
```

Naturally we need to place it within our Discord namespace again, that was to be expected, but what is our record for in this case?

The “UserMessage” record type is defined with three properties:

- “MessageId”—a string property that represents the unique identifier of the user message
- “UserId”—a string property that represents the unique identifier of the user who sent the message
- “Content”—a string property that represents the content of the user message

We will be using this to read the user messages that have been sent to the specific channel and breaking them down into its ID, to be able to reference it, the user’s ID, to be able to direct message that user with this request, and the Content of that message, to be able to determine what the user has requested. We will see its full usage shortly. For now, let us continue with the DiscordChannel class and the ReadMessages method we were building. So back to this point:

```
public async Task<IEnumerable<UserMessage>> ReadMessages()  
{  
}
```

Great, let us keep going then. What is this method going to need to do? The method needs to send a GET request to the Discord API using the RequestAsync method of the DiscordBot object. The resource address for this request is constructed by appending “messages” to the resourceAddress of the channel. Doing this would look something like this:

```
public async Task<IEnumerable<UserMessage>> ReadMessages()  
{  
    HttpResponseMessage response = await bot.RequestAsync(HttpMethod.Get,  
    resourceAddress + "messages");  
}
```

Then, we will be needing to return something. We need to return a UserMessage IEnumerable task. For that, we will make use of a small helper method, the “ReadUserMessagesFrom” method. The ReadUserMessagesFrom method is a private helper method that is called by the ReadMessages method. It takes an HttpContent object as its parameter, which represents the JSON response from the server.

```
private static async Task<IEnumerable<UserMessage>>
ReadUserMessagesFrom(HttpStatusCode response)
{
}
```

The JSON response is then read as a string using the `ReadAsStringAsync` method of the `HttpContent` object.

```
string responseJson = await response.ReadAsStringAsync();
```

The `responseJson` string is then parsed into a `JsonDocument` object using the `JsonDocument.Parse` method.

```
JsonDocument document = JsonDocument.Parse(responseJson);
```

For that parsing to happen though we need to make sure to add the `System.Text.Json` namespace to our .cs file. So add the `using` statement at the top right before our namespace.

```
using System.Text.Json;
namespace Discord;
public class DiscordChannel
{
```

Great, once we get that response parsed, we need to retrieve the root element of the `JsonDocument` object, which is an array of message objects.

```
string responseJson = await response.ReadAsStringAsync();
JsonDocument document = JsonDocument.Parse(responseJson);
JsonElement root = document.RootElement;
```

We need to do this to be able to use the `EnumerateArray` method to iterate over the message objects in the array, and for each message object we create a new `UserMessage` object using the `GetProperty` method to retrieve the values of its `id`, `author`, and `content` properties.

```
IEnumerable<UserMessage> userMessages = root.EnumerateArray() .
Select(message =>
new UserMessage
(
    MessageId: message.GetProperty("id").GetString(),
    UserId: message.GetProperty("author").GetProperty("id").GetString(),
    Content: message.GetProperty("content").GetString()
));
```

Here is the direct use of our UserMessage record that we were promised earlier. As you can see, we are using our record to get the message ID, the user ID, and the message content.

Now, keep in mind that while the direct use of our UserMessage record, as demonstrated above, provides a straightforward way to get the message ID, the user ID, and the message content, it's worth noting that there is an alternative approach. For efficiency and simplicity, one might opt to map the JSON object directly to the record without explicitly selecting every single property. To utilize this method, it's crucial that the fields in the JSON object and record align in both name and type. This approach is widely used due to its streamlined nature, although the method shown here can be particularly helpful for learning purposes.

Next, we will make the userMessages collection available as an IEnumerable<UserMessage> type. This allows the ReadMessages method to use it for its return value.

```
return userMessages;
```

So if we go back to the ReadMessages method, we can now give it a return, by calling our ReadUserMessagesFrom helper method and passing in our response content.

```
public async Task<IEnumerable<UserMessage>> ReadMessages ()  
{  
    HttpResponseMessage response = await bot.RequestAsync (HttpMethod.Get,  
resourceAddress + "messages");  
    return await ReadUserMessagesFrom (response.Content);  
}
```

And that is it for the message reading! Naturally nothing is being sent yet but we should be able to make an early test to see if our bot is working so far. To do that, let us go back to our Program class.

We will be preparing our message processor already, we just won't be implementing any message sending, query checking, direct messaging, or message deleting features yet. However, we will implement a simple read of every message in a given chat. This goes as follows.

We first need to create a DiscordChannel object right after our DiscordBot object.

```
DiscordBot bot = new  
DiscordBot ("MTA2MjMwOTIyNDM1ODYyOTQxNw.GhdEg_."  
FZC6Rsaviw4Ti95jzwquLl7HW5sHxCOrRPFDcg");  
DiscordChannel channel = new DiscordChannel();
```

As the DiscordChannel class has a constructor, it needs to get two parameters passed in along its creation. These being our bot, which we created one line above, and a channel ID.

```
DiscordBot bot = new DiscordBot("MTA2MjMwOTIyNDM1ODYyOTQxNw.GhdEg_.
FZC6Rsaviw4Ti95jzwquLl7HW5sHxCOrRPFDcg");
DiscordChannel channel = new DiscordChannel(bot, channelId: "channelID");
```

Disclaimer. As noted previously, the implementation of our token directly within our project as demonstrated here, although easier to see and understand, is not recommended practice. If you publish this project to a repository, remember to remove your token.

Now where do we get our channel ID from? We can easily get the channel ID of any server channel, but attempting to read a channel through a bot without adding it as one will result in an exception response from the Discord API. Before we do that, we need to add our bot to a Discord server. And for that, you must be an admin on a Discord server.

Alright then, should be easy. Start by joining a random Discord server and gaining their trust by pretending to be their frie... Oh right, you can also simply create a new server. Well, once you have admin rights to a server, we need to go back to the Discord developer platform.

Right above the “Bot” tab we have an OAuth2 tab. There, we need to click on the “URL Generator” sub-section, where we will be able to generate a URL that joins our bot to our server. Make sure to select “bot,” then “Administrator” and copy the given link (Fig. 11.5).

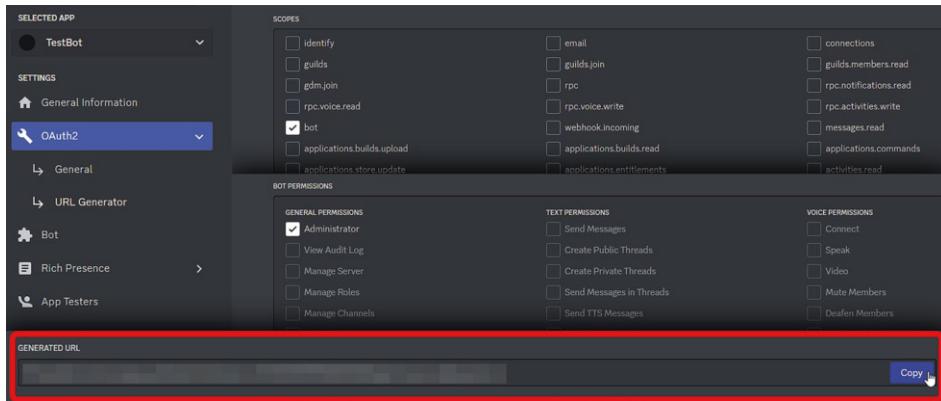


Fig. 11.5 Generating a URL that invites our bot to our server with admin rights

Open the copied link in a new tab. There you should see a few steps to select your server and the rights that you are giving the bot. And after these steps, we should have our bot visible in the Server Settings > Integrations tab.

Now our bot should have administrator rights to read, write, and modify anything on that server.

So for the next step, we would just need to get the channel ID of the channel we want to use for our bot. We get that by simply right-clicking on a channel and clicking “Copy ID” (Fig. 11.6).

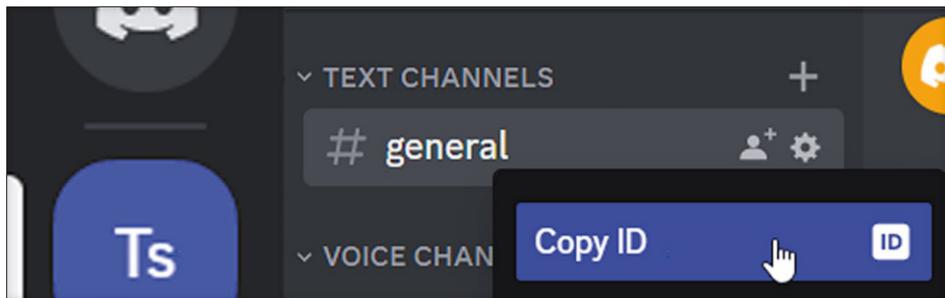


Fig. 11.6 Getting the ID of the channel we are going to be using for our tester bot

Now, add that ID to our new `DiscordChannel` object, right where we wrote “`ChannelID`.”

```
DiscordBot bot = new  
DiscordBot("MTA2MjMwOTIyNDM1ODYyOTQxNw.GhdEg_  
FZC6Rsaviw4Ti95jzwquLl7HW5sHxCOrRPFDcg");  
DiscordChannel channel = new DiscordChannel(bot, channelId: "ChannelID");
```

Now we have our `DiscordBot` and `DiscordChannel` both initialized. So it is time to read some messages.

After our `Main()` method, create a new method called `MessageProcessor`. This should be a static `async` method that returns a `Task` and takes a `DiscordChannel` variable as a parameter.

```
public async static Task MessageProcessor(DiscordChannel channel)  
{}
```

Since we will be needing that anyway later, let us prepare this method already to delay each channel reading by 5 s. But before that, let us inform the user of the application about this procedure. We want to let the user know that the program will read all messages every 5 s, with something like this:

```
Console.WriteLine("Messages are read each 5 seconds.");
```

Then simply await a `Delay()` method set to 5 s.

```
await Task.Delay(5 * 1000);
```

Next, create an empty `IEnumerable` object called `messagesInTheChannel`, which will be used to store all the messages that are read from the channel.

```
IEnumerable<UserMessage> messagesInTheChannel = await  
channel.ReadMessages();
```

Output a message to the console indicating that the messages have been read from the channel.

```
Console.WriteLine("Messages read from the channel");
```

Iterate through each message in the `messagesInTheChannel` sequence using a `foreach` loop. For each message, we output a message to the console that indicates that a message has been found, and displays the content of the message using string interpolation. And the `message.Content` property represents the actual text content of the message.

```
foreach (var message in messagesInTheChannel)
{
    Console.WriteLine($"\\nFound a message, the message is:
{message.Content}");
}
```

Finally, this line outputs a message to the console indicating that there are no more messages left to read in the `messagesInTheChannel` sequence.

```
Console.WriteLine("\\nNo more queries have been found");
```

So, overall, this code reads messages from a channel asynchronously, outputs a message indicating that the messages have been read, displays each message's content, and then outputs a message indicating that there are no more messages left to read.

DISCLAIMER This will read every message on the channel and write them on the console. So make sure to ideally not use an old channel with too many messages, as it may slow down the execution of the program greatly. Although its asynchronous nature will not allow it to freeze, having to wait an eternity for a response is also not ideal.

Now we can simply call it in our `Main()` method.

```
Task task = MessageProcessor(channel);
task.Wait();
```

This should already be enough for us to make our first test! So, make sure there is something written on that channel and hit run (Fig. 11.7)!



The screenshot shows the Microsoft Visual Studio Debug Console window. The output text is as follows:

```
Microsoft Visual Studio Debug Console
Welcome to the Transcendental ECommerce Discord Online Bot Advanced Console app, or TECDOBACA in short
To start the bot press any key
Messages are read each 5 seconds.
Messages read from the channel

Found a message, the message is: I am a message from the Discord channel!
No more queries have been found
```

Fig. 11.7 Result after reading all messages from a newly created Discord channel

As we can see, we had great success! However, this program simply stops now after it has read all messages once. Well, that is because we haven't set it to repeat the process each 5 s yet. So let us do so.

Let us start by placing our code within a while loop. This will ensure that the code gets repeated indefinitely once every 5 s.

```
while (true)
{
    await Task.Delay(5 * 1000);
    IEnumerable<UserMessage> messagesInTheChannel = await
channel.ReadMessages();
    Console.WriteLine("Messages read from the channel");

    foreach (var message in messagesInTheChannel)
    {
        Console.WriteLine($"\\nFound a message, the message is:
${message.Content}");
    }
    Console.WriteLine("\\nNo more queries have been found");
}
```

This will work already, but we for sure cannot just leave this repeating forever; we need a way to stop the process at some point. To keep it simple, we can just check if the user pressed the Esc key, and if yes, stop the bot, like so:

```
while (true)
{
    if (Console.KeyAvailable)
    {
        ConsoleKeyInfo keyInfo = Console.ReadKey(true);
        if (keyInfo.Key == ConsoleKey.Escape)
        {
            break;
        }
    }
}
```

Now, we can inform the user about this feature once at the beginning, when starting the application. We can use our "Messages are read each 5 seconds." print for that.

```
Console.WriteLine("Messages are read each 5 seconds.
To exit the bot, press Esc.");
```

And, since this message will be left behind after a while, let us also repeat this message on each iteration.

```
while (true)
{
    Console.WriteLine("Waiting... Press Esc to quit.");

    if (Console.KeyAvailable)
    {
```

And now, just for fun, let us add another two messages letting the user know that the application is shutting down. So right after where we called our `MessageProcessor()` method, we can print the following.

```
Task task = MessageProcessor(channel);
task.Wait();

Console.WriteLine("Process finalized");
Console.WriteLine("Shutting down...");
```

And that is it! We can run again and should have our application looping and checking for messages every 5 s. And exit as soon as we press Esc (Fig. 11.8).

Welcome to the Transcendental ECommerce Discord Online Bot Advanced Console app, or TECDOBACA in short
To start the bot press any key
Messages are read each 5 seconds. To exit the bot, press Esc.
Waiting... Press Esc to quit.
Messages read from the channel
Found a message, the message is: I am a message from the Discord channel!
No more queries have been found
Waiting... Press Esc to quit.
Messages read from the channel
Found a message, the message is: I am a message from the Discord channel!
No more queries have been found
Waiting... Press Esc to quit.
Process finalized
Shutting down... **ESC KEY WAS PRESSED**

Fig. 11.8 The process repeats every 5 s and scans the channel for messages to read. As soon as the ESC key is pressed, the process exits and stops the application

Perfect execution! We have our bot running and reading messages! So what do we want to do next? As this bot currently can only read messages over and over again, we should now get to implementing the methods responsible for sending, deleting, and filtering messages, as well as our direct messaging functionality. We also still need a JSON file that contains our products. Remember that this is going to be an E-commerce bot! Or at least the closest we can get to one!

So let us start by sending a message at the beginning that shows what we offer to our potential customers. These methods will be implemented in the `DiscordChannel` class so let's get back in there to continue our work.

So in our `DiscordChannel` class, let us create a new method called `SendMessage`. This method has to be a public `async` method that returns a string type `Task` and takes a

string as a parameter. That parameter represents the content of a message that will be sent to the bot.

```
public async Task<string> SendMessage(string text)
{ }
```

The first thing that happens in the method is that the “text” parameter is used to create a new message object which is represented as a JSON object with a single property called “content” whose value is the text passed to the method.

```
var message = new { content = text };
```

Next, an HTTP POST request is made to the bot’s server using the RequestAsync method of the bot object. This method takes three arguments—the HTTP method to use (POST), the resource address to use, and the message object created earlier. The resource address specifies where the message should be sent on the server, in this case the base address of the server with “/messages” appended to it to specify that this is a message being sent.

```
HttpResponseMessage httpResponse = await bot.RequestAsync(HttpMethod.Post, resourceAddress + "messages", message);
```

After the message is sent to the server, the response is obtained and read as a string using the ReadAsStringAsync method of the Content property of the HTTP response object. This response should be a JSON object containing information about the message that was sent.

```
string response = await httpResponse.Content.ReadAsStringAsync();
```

The response is then parsed as a JSON document and the root element of the document is obtained. From this root element, the ID property of the message that was sent is obtained and stored in a variable called “messageId.”

```
JsonDocument document = JsonDocument.Parse(response);
JsonElement root = document.RootElement;
string? messageId = root.GetProperty("id").GetString();
```

Finally, the method returns the messageId if it is not null. If the messageId is null, an exception is thrown indicating that no message ids were returned from the server.

```
return messageId ?? throw new Exception("No message ids returned from the server!");
```

That is it for our SendMessage() method. Next, we need to build ourselves a list of products that we want to use to list on the channel. For that, if you want to keep it simple for now, just create a new JSON file in the “ProjectDirectory/bin/Debug/net6.0” folder called “products.json.”

Now, populate this file with the following products.

Listing 11.3 Contents of the Products.json File Used for Our Discord Bot

```
{  
    "Health Tracker": "$85",  
    "Keyboard": "$39.48",  
    "AirPods": "$129",  
    "Monitor": "$109",  
    "Mouse": "$22.14",  
    "8 Core CPU": "$169.57"  
}
```

We can obviously change any product within this file to any other product. Simply make sure to maintain the correct formatting.

Now that we have our JSON file, we can use it in our code. Since we already know how to get the contents of a file, let us quickly go over the steps.

So right before we call the MessageProcessor() method within the Program class Main() method, write the following.

```
string json = File.ReadAllText("products.json");  
Dictionary<string, string> products =  
JsonSerializer.Deserialize<Dictionary<string, string>>(json);
```

This will naturally throw an error as the System.Text.Json namespace was not yet added to this file. So make sure to do so.

```
using Discord;  
using System.Text.Json;  
  
class Program  
{
```

Now, as we need to send this information to the MessageProcessor() method, add a new parameter to it. We will be needing a dictionary for all of our products.

```
public async static Task MessageProcessor(DiscordChannel channel,  
Dictionary<string, string> products)  
{
```

Now, pass the new products dictionary we just created into our method call as a parameter, like so:

```
string json = File.ReadAllText("products.json");
Dictionary<string, string> products =
JsonSerializer.Deserialize<Dictionary<string, string>>(json);

Task task = MessageProcessor(channel, products);
task.Wait();
```

And right before we print for the first time in the `MessageProcessor()` method, use the `SendMessage` method to send out our product list!

```
await channel.SendMessage(MessageTemplates.ProductList(products));
Console.WriteLine("The message containing the product
list has been sent out");
Console.WriteLine("Messages are read each 5 seconds.
To exit the bot, press Esc.");
```

As we can see, we are using a new class for our `SendMessage` method. A class we also have not implemented yet. That is, simply, as the name suggests, we decided to separate the template into a new class to improve readability. In essence, this class is fairly simple. Let us check this out.

Let us, as always, create a new class and call it `MessageTemplate`.

This class will provide a template for both the first product list message and our direct message. So let us see how we will do that.

Starting with our current need, the product list message. Inside the `MessageTemplates` static class, we will need a static `ProductList()` method that returns `string` and takes a dictionary as a parameter. As we remember from before, that dictionary is the one we created from the JSON products file.

```
static class MessageTemplates
{
    public static string ProductList(Dictionary<string, string> products)
    {
    }
```

Within this method, we first need to turn our dictionary into a simpler `IEnumerable` list that would contain the names of our products.

```
IEnumerable<string> productNames = products.Select((product, i) => $"{i +
1}. {product.Key}");
```

This can then be converted into a simple list stored in a string variable.

```
IEnumerable<string> productNames = products.Select((product, i) => $"{i + 1}. {product.Key}");
string strProductNames = string.Join("\n", productNames);
```

This variable can then be set into a final formatted message that will look something like this:

```
IEnumerable<string> productNames = products.Select((product, i) => $"{i + 1}. {product.Key}");
string strProductNames = string.Join("\n", productNames);
string template = $"Select a product by it's id
from the following list:\n{strProductNames}";
```

Our final result is then returned to the caller.

```
public static string ProductList(Dictionary<string, string> products)
{
    IEnumerable<string> productNames = products.Select((product, i) => $"{i + 1}. {product.Key}");
    string strProductNames = string.Join("\n", productNames);
    string template = $"Select a product by it's id
    $from the following list:\n{strProductNames}";
    return template;
}
```

And that is our message template. If we now check back to our Program.cs file, we'll see that our error is gone and that we can make our first test sending a product list to our Discord Channel. Let us try that out (Fig. 11.9)!

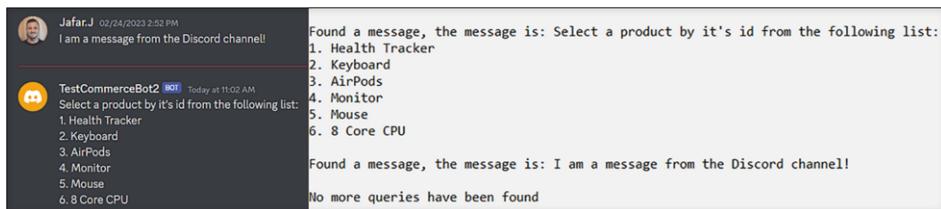


Fig. 11.9 Launching the project after the implementation of the messaging functionality we see that the code correctly turns our list of products into a Discord message that gets sent to the assigned Discord channel

Working as intended. We take the JSON file, break it down, turn it into a message using the template, and send it right before commencing our read cycle to find potential customers.

However, after this victory, we do see that our code currently is picking up both the previous message sent by us and its own message. Although this is normal and expected, we do not have a use for our own message. As a matter of fact, we would only have a use for messages that correspond to our list of products. So yes, it is time to filter out the things we do not need to find what user wants to learn more about what specific product. As we already have our products nicely laid out in a dictionary, doing so could not be easier. Let us get into filtering out the messages that we are interested in.

If we check our current code, we are simply checking for messages in our channel, then going through the resulting `IEnumerable` and printing them out onto our console.

```
IEnumerable<UserMessage> messagesInTheChannel = await
channel.ReadMessages();
Console.WriteLine("Messages read from the channel");
foreach (var message in messagesInTheChannel)
{
    Console.WriteLine($"\\nFound a message, the message is:
{message.Content}");
}
```

If we were to think of a way to improve this code to contain some way of filtering the list of messages, the first thing that might come to mind is running a quick check within our `foreach` loop. As we are getting the complete message within that loop, we could read what it says, and compare it to our product list to see if the message matches any of our product names, right? Well, let us try that idea out.

As we are using a dictionary, we get access to a handy little method called `TryGetValue()`. This will attempt to find within the message any of the products that we have in our dictionary. And the best part is, we can then directly get a variable with the price of the value we have gotten as an output. That will come in handy right now when we attempt to send the user a direct message!

```
foreach (var message in messagesInTheChannel)
{
    if (products.TryGetValue(message.Content, out string? price))
    {
        Console.WriteLine($"\\nFound a price query,
the response from the bot is:
{message.Content} costs {price}");
    }
}
```

This will now change the print to only show messages that have a product query in them, and while we are at it, we can also list out the price so we can check if we are doing everything right.

Now, if we run this, we see the following result (Fig. 11.10).

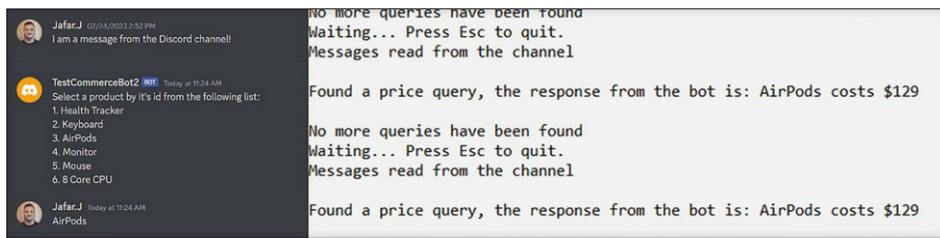


Fig. 11.10 After implementing filtering we are only seeing the message that asks for a specific product. Although all messages are still read, only those that we need are kept for further use

For a simple code solution we got the exact result we were after! Naturally this is not perfect by any means, but for a simple bot like ours, it is already a great step forward. Also, we are getting the correct price! Meaning that the product is being found correctly within the dictionary! So then, we have everything we need to direct message the user the current price for that specific product, right? Well, yes and no.

As for data, yes, we got the message, the correct product, the price, and the user. However, we need to still implement the direct message sending functionality. Well then let us not wait any longer, this being pretty much the final feature: the end of this program is already on the horizon!

First, we need to set up another template for our direct message. Back in our `MessageTemplates` class, we can create a new method for our new message.

This is a C# method named `PriceReport()` that takes two string parameters, “messageContent” and “price,” and returns a string.

The method is a one-liner that uses string interpolation to concatenate the `messageContent` and `price` parameters into a single string with the format “`{messageContent}` costs `{price}`.” Simple enough, starting with the method signature, which declares that the method is public, static, returns a string, and takes two string parameters named `messageContent` and `price`, as mentioned.

```
public static string PriceReport(string messageContent, string price)
```

Then, we need the method body, which uses an arrow (`=>`) to indicate that this is a “lambda expression,” a shorthand way of writing a method that consists of a single expression. Then the expression is a string interpolation, which uses curly braces (`{}`) to insert the values of the `messageContent` and `price` variables into the string.

```
public static string PriceReport(string messageContent, string price)
=> $"{messageContent} costs {price}";
```

There we go. We just needed this for this second template. And as we can imagine, this will create a message that could look like, for example, this: “AirPods costs 129\$.”

Perfect. Now, we need a way to send this message as a direct message. For that, we need our final class, the `DirectMessage` class. Let us get to it.

As before, we will start off this class with a namespace `Discord`; at the top and by declaring a public class called `DirectMessage`.

```
namespace Discord;

public class DirectMessage
{}
```

Now, first, we will start preparing the class with its constructor. For that, we need a `DiscordBot` object.

```
public class DirectMessage
{
    readonly DiscordBot bot;
```

Next, we declare a public constructor for the `DirectMessage` class. This constructor takes a `DiscordBot` object as a parameter and assigns it to the `bot` field.

```
public DirectMessage(DiscordBot bot)
=> this.bot = bot;
```

Alright, now that we got this ready, we will need to set up our direct message sender methods, starting with the `Send()` method. We declare a public `async` method called `Send()`. This method takes two string parameters: `recipientId`, which is the ID of the user who will receive the direct message, and `message`, which is the content of the direct message.

```
public async Task Send(string recipientId, string message)
{
    // Method code will be defined here...
}
```

Here, we will call two other methods, the `async InitDM()` method and the `async SendDM()` method.

```
public async Task Send(string recipientId, string message)
{
    string dmId = await InitDm(recipientId);
    await SendDm(dmId, message);
}
```

The `string dmId = await InitDm(recipientId);` statement calls a private method called `InitDm()` with the `recipientId` parameter and awaits its result. The result of the method call is stored in a variable called `dmId`.

```
string dmId = await InitDm(recipientId);
```

The `await SendDm(dmId, message);` statement calls a private method called `SendDm` with the `dmId` and `message` parameters and awaits its result.

```
await SendDm(dmId, message);
```

Great, but we haven't yet implemented these two methods. So then let us do so.

First, for the `InitDM()` method, we need to declare a private async method and call it `InitDm`. This method takes a string parameter called `recipientId`, which is the ID of the user who will receive the direct message. The method returns a `Task<string>`.

```
private async Task<string> InitDm(string recipientId)
{
    // Method code will be defined here...
}
```

Here, we need to make an HTTP POST request to the Discord API using the `RequestAsync` method of the `bot` object. The request is sent to the `/users/@me/channels` endpoint with a JSON payload that contains the `recipientId` parameter. The result of the HTTP request is stored in a variable called `httpResponse`.

```
private async Task<string> InitDm(string recipientId)
{
    var httpResponse = await bot.RequestAsync(
        HttpMethod.Post,
        "/users/@me/channels",
        new { recipient_id = recipientId });
}
```

Then, we read the response content of the HTTP response and store it in a variable called `response`.

```
private async Task<string> InitDm(string recipientId)
{
    var httpResponse = await bot.RequestAsync(
        HttpMethod.Post,
        "/users/@me/channels",
        new { recipient_id = recipientId });
    string response = await httpResponse.Content.ReadAsStringAsync();
```

Then, we parse the response string as a JSON document using the `JsonDocument.Parse` method of the `System.Text.Json` namespace. The resulting JSON document is stored in a variable called `document`.

```
private async Task<string> InitDm(string recipientId)
{
    var httpResponse = await bot.RequestAsync(
        HttpMethod.Post,
        "/users/@me/channels",
        new { recipient_id = recipientId });
    string response = await httpResponse.Content.ReadAsStringAsync();
    JsonDocument document = JsonDocument.Parse(response);
```

Also make sure to add the System.Text.Json using statement in the document.

```
using System.Text.Json;
namespace Discord;
public class DirectMessage
{
```

And finally, we get the id property of the root element of the JSON document using the GetProperty method of the JsonElement class. The value of the id property is returned as a string using the GetString method. If the id property is not found, an empty string is returned.

```
private async Task<string> InitDm(string recipientId)
{
    var httpResponse = await bot.RequestAsync(
        HttpMethod.Post,
        "/users/@me/channels",
        new { recipient_id = recipientId });
    string response = await httpResponse.Content.ReadAsStringAsync();
    JsonDocument document = JsonDocument.Parse(response);
    return document.RootElement.GetProperty("id").GetString() ?? "";
}
```

Now that we have the InitDM() method, we need the method that actually uses this to send the DM, the SendDM() method.

For that, we simply declare a private async method called SendDm. This method takes two string parameters: dmId, which is the ID of the direct message channel to send the message to, and message, which is the content of the direct message. The method returns a Task.

```
private Task SendDm(string dmId, string message)
=> bot.RequestAsync(HttpMethod.Post, $"{"/channels/{dmId}/messages"}"
    , new
    {
        content = message
    });
```

And that is it! We got the class ready to send some DMs. Now, how do we use this class in our main program? Let's find out.

Go back to the Program.cs file. And right under our DiscordChannel object creation, create a directMessage object and pass in our bot.

```
DiscordBot bot = new DiscordBot("YourDiscordToken");
DiscordChannel channel = new DiscordChannel(bot, channelId: "YourChannelID");
DirectMessage directMessage = new DirectMessage(bot);
```

Now, int our MessageProcessor() method and add a new parameter, a DirectMessage parameter.

```
public async static Task MessageProcessor(DiscordChannel channel,
Dictionary<string, string> products, DirectMessage directMessage)
{
```

And to its call, add the new DirectMessage object we just created.

```
Task task = MessageProcessor(channel, products, directMessage);
task.Wait();
```

Great, now that we got the object in our method, we can start using it, starting with using our new template.

```
if (products.TryGetValue(message.Content, out string? price))
{
    Console.WriteLine($"\\nFound a price query, the response
        $from the bot is: {message.Content} costs {price}");
    string priceReport = MessageTemplates.PriceReport
(message.Content, price);
}
```

Just as before with our product list message, this will be used to build a string variable that contains the message we want to send. This time, the message that contains the product name that was asked for with the price of that product.

Next, we need to get the user's ID, and send him this DM. Using our DirectMessage class and the Send() method, we can do just that with this simple line.

```
if (products.TryGetValue(message.Content, out string? price))
{
    Console.WriteLine($"\\nFound a price query, the response from
        $the bot is: {message.Content} costs {price}");
    string priceReport = MessageTemplates.PriceReport(message.Content, price);
    await directMessage.Send(message.UserId, priceReport);
}
```

Done! If we try our app out now, we will see that, if there is a message of us requesting a specific product in that chat, we will get a direct message by our bot giving us the pricing of that product (Fig. 11.11).



Fig. 11.11 After the bot sent out the message with the product list, we responded with the product we wanted to know more about. Then the bot read that message, constructed a response, and sent a direct message to us with that response

Seems to be working great! We send out a list and users can read that and send in a request. Then the bot, every 5 s, reads the chat for requests, finds them, and direct messages them the price of their desired product! There is just one more thing we can do to finish this application. We need to delete the messages of the requesters, simply because if we don't, we will indefinitely keep sending them a DM every 5 s until the end of time or until they manually delete their message, which is, clearly, less than ideal.

Luckily doing so is not so difficult. So let us do that. Go back to the `DiscordChannel` class. This is where we will be implementing that method. To be specific, the `DeleteMessage()` method. This should be an `async` method that returns `Task` and takes a string as a parameter.

```
public async Task<float> DeleteMessage(string messageId)
{
    // Method code will be defined here...
}
```

Nice, now, what does it do? In short, it sends out an HTTP request to `DELETE` a certain message given its ID. To do that, we first must start building the request itself. So an HTTP `DELETE` request is created using the specified bot object and resource address, along with the message ID that needs to be deleted.

```
public async Task<float> DeleteMessage(string messageId)
{
    HttpResponseMessage response = await
bot.RequestAsync(HttpMethod.Delete, resourceAddress
+ $"messages/{messageId}");
```

Then, the response from the messaging service is read as a string using the `ReadAsStringAsync` method.

```
HttpResponseMessage response = await bot.RequestAsync(HttpMethod.Delete,  
resourceAddress + $"messages/{messageId}");  
string json = await response.Content.ReadAsStringAsync();
```

If the response content is empty or null, the method returns 0. This step is important to prevent errors when trying to parse the response content.

```
HttpResponseMessage response = await bot.RequestAsync(HttpMethod.Delete,  
resourceAddress + $"messages/{messageId}");  
string json = await response.Content.ReadAsStringAsync();  
if (string.IsNullOrEmpty(json))  
{  
    return 0;  
}
```

The response content is parsed as a JSON document using the `JsonDocument.Parse` method. The root element of the document is obtained using the `RootElement` property.

```
HttpResponseMessage response = await bot.RequestAsync(HttpMethod.Delete,  
resourceAddress + $"messages/{messageId}");  
string json = await response.Content.ReadAsStringAsync();  
if (string.IsNullOrEmpty(json))  
{  
    return 0;  
}  
JsonDocument document = JsonDocument.Parse(json);  
JsonElement root = document.RootElement;
```

The “`retry_after`” property is obtained from the root element using the `GetProperty` method.

```
HttpResponseMessage response = await bot.RequestAsync(HttpMethod.Delete,  
resourceAddress + $"messages/{messageId}");  
string json = await response.Content.ReadAsStringAsync();  
if (string.IsNullOrEmpty(json))  
{  
    return 0;  
}  
JsonDocument document = JsonDocument.Parse(json);  
JsonElement root = document.RootElement;  
JsonElement retryAfterElement = root.GetProperty("retry_after");
```

If the “`retry_after`” property is null, the `retryAfter` variable is set to null. Otherwise, the value of the “`retry_after`” property is obtained as a string.

```

HttpResponseMessage response = await bot.RequestAsync(HttpMethod.Delete,
resourceAddress + $"messages/{messageId}");
string json = await response.Content.ReadAsStringAsync();
if (string.IsNullOrEmpty(json))
{
    return 0;
}
JsonDocument document = JsonDocument.Parse(json);
JsonElement root = document.RootElement;
JsonElement retryAfterElement = root.GetProperty("retry_after");
string? retryAfter = retryAfterElement.ValueKind == JsonValueKind.Null ?
    null : retryAfterElement.GetString();

```

Then, the `retryAfter` string is parsed as a float using the `float.Parse` method. If the `retryAfter` string is null, the default value of 0 is returned. And we have completed the method.

```

public async Task<float> DeleteMessage(string messageId)
{
    HttpResponseMessage response = await
bot.RequestAsync(HttpMethod.Delete, resourceAddress +
$"messages/{messageId}");
    string json = await response.Content.ReadAsStringAsync();
    if (string.IsNullOrEmpty(json))
    {
        return 0;
    }
    JsonDocument document = JsonDocument.Parse(json);
    JsonElement root = document.RootElement;
    JsonElement retryAfterElement = root.GetProperty("retry_after");
    string? retryAfter = retryAfterElement.ValueKind == JsonValueKind.Null ?
        null : retryAfterElement.GetString();
    return float.Parse(retryAfter ?? "0");
}

```

Great, next, we go back to our `Program` class and simply make use of this new method. Inside our `foreach` loop where we get the individual requests and send the DMs, we can then delete that message that requested the product, like this:

```

foreach (var message in messagesInTheChannel)
{
    if (products.TryGetValue(message.Content, out string? price))
    {
        Console.WriteLine($"\\nFound a price query, the response from
the bot is: {message.Content} costs {price}");
        string priceReport = MessageTemplates.PriceReport(message.
Content, price);
    }
}

```

```
        await directMessage.Send(message.UserId, priceReport);
        await channel.DeleteMessage(message.MessageId);
    }
}
```

And if we now run the program, we should see messages that requested a product disappear, leaving just the product list messages behind (Fig. 11.12)!

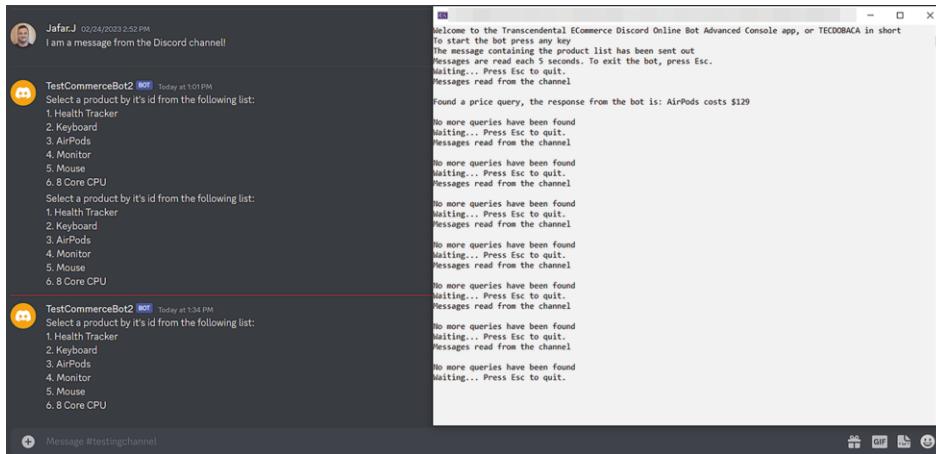


Fig. 11.12 Our bot successfully deleted messages that contained requests so that these messages are not read multiple times over

As we can see, only unrelated messages and the bot's own message is left. And also, check your inbox as you should also now have a direct message from our bot with the price of the requested product.

Would you look at that, we have finished this book's final project! Now this project, as well as any other project in this book, is still not a finished perfect product for sure, but that is where you come in.

We have attempted with this book to give you as much important knowledge as possible and projects that will serve you as great portfolio pieces. Then also, if you want, you could even take them and expand them further to become fully fledged products you could sell to companies and customers alike.

We sincerely hope that each project taught you a clear path to follow and that this book has provided you with a lot of value!

And with that, we only can wish you all the luck on anything you will move on to and the best of luck on your professional C# career!

Thank you from the entire TutorialsEU team and the Springer team for choosing this book!

Let's get ready for OUR next chapter.

11.4.3 Source Code

Link to the project on [Github.com](#):

https://github.com/tutorialseu/TutorialsEU_TinyDiscordBot

11.5 Summary

This chapter taught us:

- APIs, or Application Programming Interfaces, are tools that help different software components communicate with each other by following a set of rules. For example, the weather app on our phone uses APIs to get daily weather updates from a weather bureau's software system
- APIs can provide businesses access to a broad range of resources, enabling cross-platform communication and improving workflow efficiency. Generally, these benefits include:
 - Real-Time Quoting. Businesses can provide potential clients with quick policy quotes by enabling agile API data quality and delivery speed.
 - Efficiency. Third-party APIs can handle many background tasks in Web-centric information gathering and computation tasks, reducing the need for extensive IT department labor.
 - Simplicity. Using APIs allows data to be automatically distributed externally and internally, allowing businesses to eliminate redundant work.
 - Customization. Enterprise developers can deploy their central platform alongside APIs, allowing customization to fit each customer's needs.
- An API acts as an intermediary between a client (e.g., a Web or mobile application) and a server. The client requests the API, specifying what data or functionality it needs. The API processes the request and responds with the requested data or functionality.
- APIs and REST are technologies used for building software applications and Web services, while an SDK is a collection of development tools and resources for building said software applications.
- The HttpClient class, a component of the System.Net.Http namespace, is a tool utilized in C# for making HTTP requests and receiving HTTP responses in a modern and asynchronous manner.
- An instance of the HttpClient class can be utilized to make various types of HTTP requests, such as GET, POST, PUT, and DELETE. The class offers a wide range of methods and options for the manipulation of HTTP requests and responses.
- The HttpClient class is designed to be employed as a singleton object, meaning that a single instance should be created and reused throughout an application. This approach improves performance and reduces the risk of socket exhaustion.

- A constructor is a special method that is automatically called when an object of a class is created. Constructors are used to initialize the object's state (i.e., its fields and properties) to some initial values.
 - The => syntax is a shorthand way of writing a constructor with a single assignment statement.
-

References

- Amazon. (n.d.). What Is An API (Application Programming Interface)? Retrieved January 10, 2023 from [aws.amazon.com: https://aws.amazon.com/what-is/api/](https://aws.amazon.com/what-is/api/)
- Davis, N. (n.d.). WHAT ARE APIs & WHY DO WE NEED THEM? Retrieved January 10, 2023 from openinsurance.io: <https://openinsurance.io/tech-lab/what-are-apis-why-do-we-need-them/>
- Red Hat, Inc. (2022, June 2). What is an API? Retrieved January 10, 2023 from [www.redhat.com: https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces](https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces)