Denis Panjuta · Jafar Jabbarzadeh

Learning C# Through Small Projects

Springer

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

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!

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

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

D. Panjuta, J. Jabbarzadeh, *Learning C# Through Small Projects*,
https://doi.org/10.1007/978-3-031-51914-7_1

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-safe[2] programming language is a general-purpose, multi-paradigm[3] 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.

[2] C# 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.

[3] A 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 Language[5] 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).

[6] An "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# proace 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).

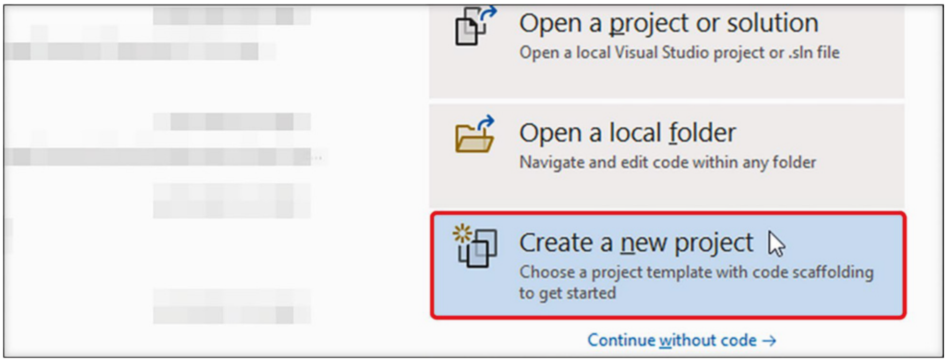

Open a project or solution
Open a local Visual Studio project or .sln file
Open a local folder
Navigate and edit code within any folder
Create a new project
Choose a project template with code scaffolding to get started
Continue without code →

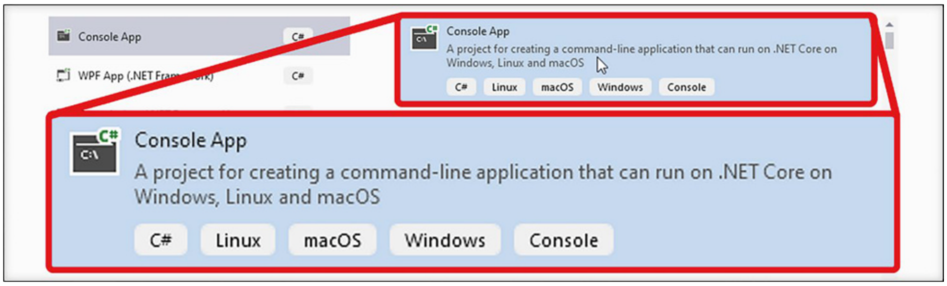
Fig. 1.1 We are starting by selecting the “Create a new project” option

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).



Console App
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

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

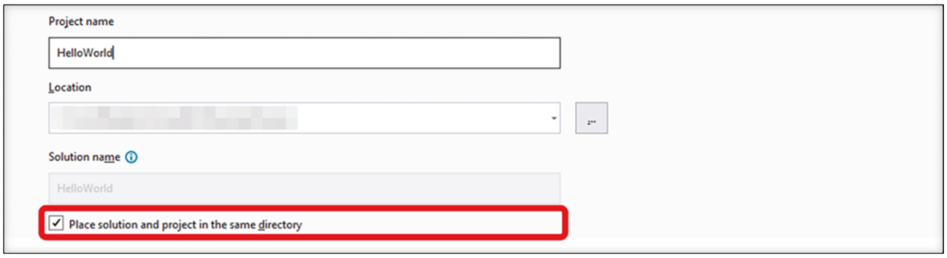
This template will give us a preconfigured project to create a command-line application on .NET Core on Windows, Linux, and macOS devices.

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).



Project name: HelloWorld

Location:

Solution name: HelloWorld

[✓] Place solution and project in the same directory

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).

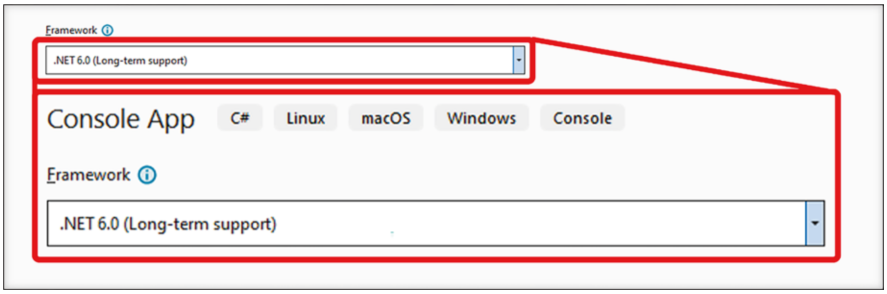
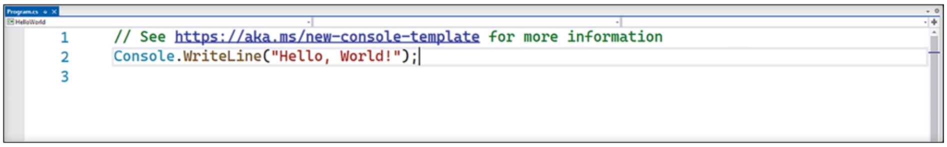


Fig. 1.4 We will be using .NET version 6.0 for our projects

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).



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

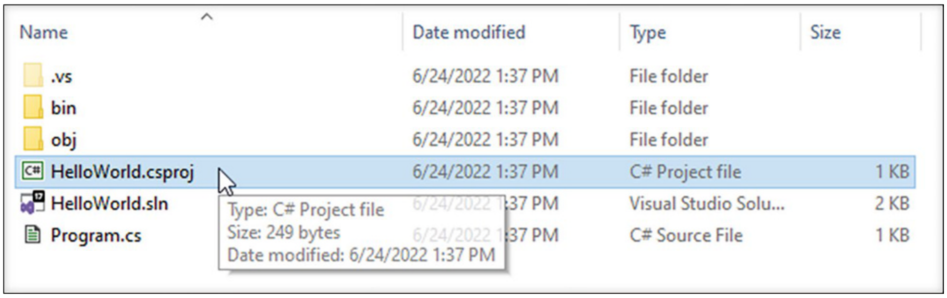
Fig. 1.5 We can see the base template given by the chosen console app template

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

Type: C# Project file
Size: 249 bytes
Date modified: 6/24/2022 1:37 PM

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.

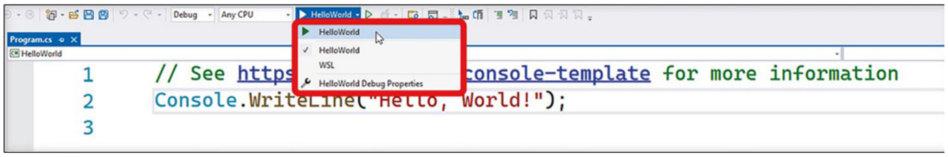
The Console.WriteLine method specifically is a method that will write the data given within its argument list and move the cursor to the following line, meaning that if two or more WriteLines are added, the second one will be displayed on the following line, as in the following example.

```
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).

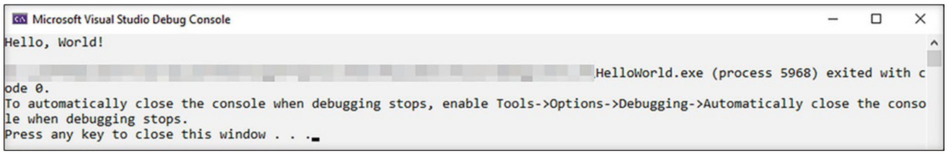


```
// See https                   console-template for more information
Console.WriteLine("Hello, World!");
```

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).



```
Hello, World!

[redacted]HelloWorld.exe (process 5968) exited with c
ode 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

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.

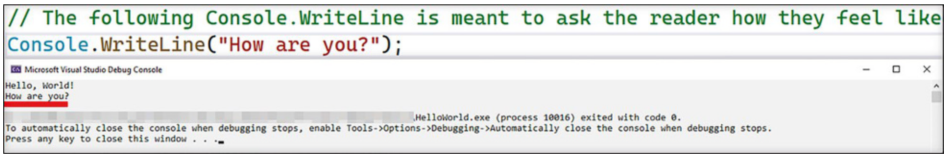
After a quick comment explaining this second Console.WriteLine, we will show the text "How are you?"

```
// 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?");
```

```
Hello, World!
How are you?

[redacted]\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 <br> 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();
```