Kickstart

C# Programming Fundamentals

Learn C# Programming, Object-Oriented Development, Windows Applications, Databases, and ASP.NET with Hands-On Projects

Dr. Edward D Lavieri Jr.

First Published: June 2026

Published by: Orange Education Pvt Ltd, AVA®

Address: 9, Daryaganj, Delhi, 110002, India

275 New North Road Islington Suite 1314 London,
N1 7AA, United Kingdom

ISBN (PBK): 978-93-49887-76-3

ISBN (E-BOOK): 978-93-49887-77-0

Scan the QR code to explore our entire catalogue

www.orangeava.com

Table of Contents

CHAPTER 1

Introduction to C# and the .NET Platform

Introduction

This chapter introduces the C# programming language and the broader .NET platform. It is intended to lay the groundwork for everything you will read and do in this book. It begins with a brief history of C#, highlighting the features that make it one of the most widely used languages in modern software development. You can gain an understanding of how C# fits into the .NET ecosystem, including its relationship to the Common Language Runtime (CLR) and the .NET Software Development Kit (SDK). The chapter walks you through setting up your development environment using Visual Studio Code, a free Integrated Development Environment (IDE). The chapter includes a hands-on tutorial to guide you through writing a simple C# program, helping you become familiar with the structure and syntax of a C# application. By the end of the chapter, you should have a solid foundation and a working development setup.

Structure

In this chapter, we will discuss the following topics:

- History and Features of C#
- Overview of the .NET Ecosystem
- Common Language Runtime and Compilation
- Installing .NET SDK and Visual Studio Code
- Writing your First C# Program

History and Features of C#

The **C# programming language** is an object-oriented language developed by Microsoft as part of the **.NET framework**. Initially released in 2000, C# was envisioned to be a language that would be easy to use (such as Visual Basic) and harness the power and flexibility of the C++ programming language.

The language, now 25 years old, has evolved into one of the most widely used languages. It is commonly ranked among the top five programming languages along with Python, Java, C, and C++. C# is commonly used for enterprise software development, cloud-based solutions, high-performance games, web applications, and mobile application development.

Microsoft, the creator of C#, developed the .NET framework with C# at its core. The primary purpose of this framework was to build and run applications on machines using the Microsoft Windows operating system. While .NET supports additional programming languages, our focus will be on C#.

***Note*:** *C# is pronounced C-sharp, and .NET is pronounced dot-net.*

Key Features

One of the primary features of C# is that it is tightly integrated with .NET's **CLR**, which enables cross-language support and manages code execution. This integration results in seamless interaction between our code and the libraries as well as the APIs that make up the .NET framework. Examples include the ability to work with files, access databases, implement machine learning, web development, and more.

Another key feature of C# is that it is a **strongly typed system**. This helps make our code more reliable by ensuring our variables are used appropriately. Strongly typed systems also reduce runtime errors.

A third C# feature is that it is a **statically typed programming language**. This means that the compiler process includes type checking instead of it occurring at runtime. This gives us the double benefit of improving application performance and helping us catch potential errors before we publish our code.

Since its initial launch, the C# language has continually evolved into the robust language it is today. Each release introduced several changes to the language.

Selected updates are provided in *Table 1.1* for the last seven versions.

Release	Release Date	Selected Update	Chapter in this Book
14	Nov. 2025	Developer ergonomics, performance, and productivity	N/A
13	Nov. 2024	`params` collections	13
12	Nov. 2023	Primary constructors	9
11	Nov. 2022	Generics math support	13
10	Nov. 2021	Lambda improvements	15

9	Nov. 2020	Records	13
8	Nov. 2019	Pattern matching enhancements	4

Table 1.1: *Major C# Release Summary*

In conclusion, C# is a mature, robust, and continually evolving language. It is ideal for many Windows-based development scenarios. When paired with the .NET framework, it represents an excellent choice for developers who want to build reliable, maintainable, and performant applications.

Overview of the .NET Ecosystem

The .NET Framework is the name of Microsoft's collection of libraries, tools, languages, and runtime environment collectively referred to as the .NET ecosystem. The ecosystem is continually evolving and supports the development of applications for a variety of platforms, including mobile, desktop, web, cloud, games, and even Internet of Things (IoT) devices. Microsoft's vision for .NET is to provide developers with a consistent programming model regardless of which platform or platforms they are writing applications for.

The .NET Framework is over two decades old and has come a long way from the Windows-specific original version. Today, the .NET ecosystem is cross-platform compatible, allowing us to develop and run applications on Windows, macOS, and Linux systems. .NET is a consolidation of Microsoft's previously disparate frameworks to include **.NET Core**, .**NET Framework**, and **Xamarin**.

The current .NET ecosystem, as illustrated in *Figure 1.1*, consists of eight key components, all contributing to a robust framework for cross-platform development.

Note: *.NET, although commonly capitalized, is not an acronym.*

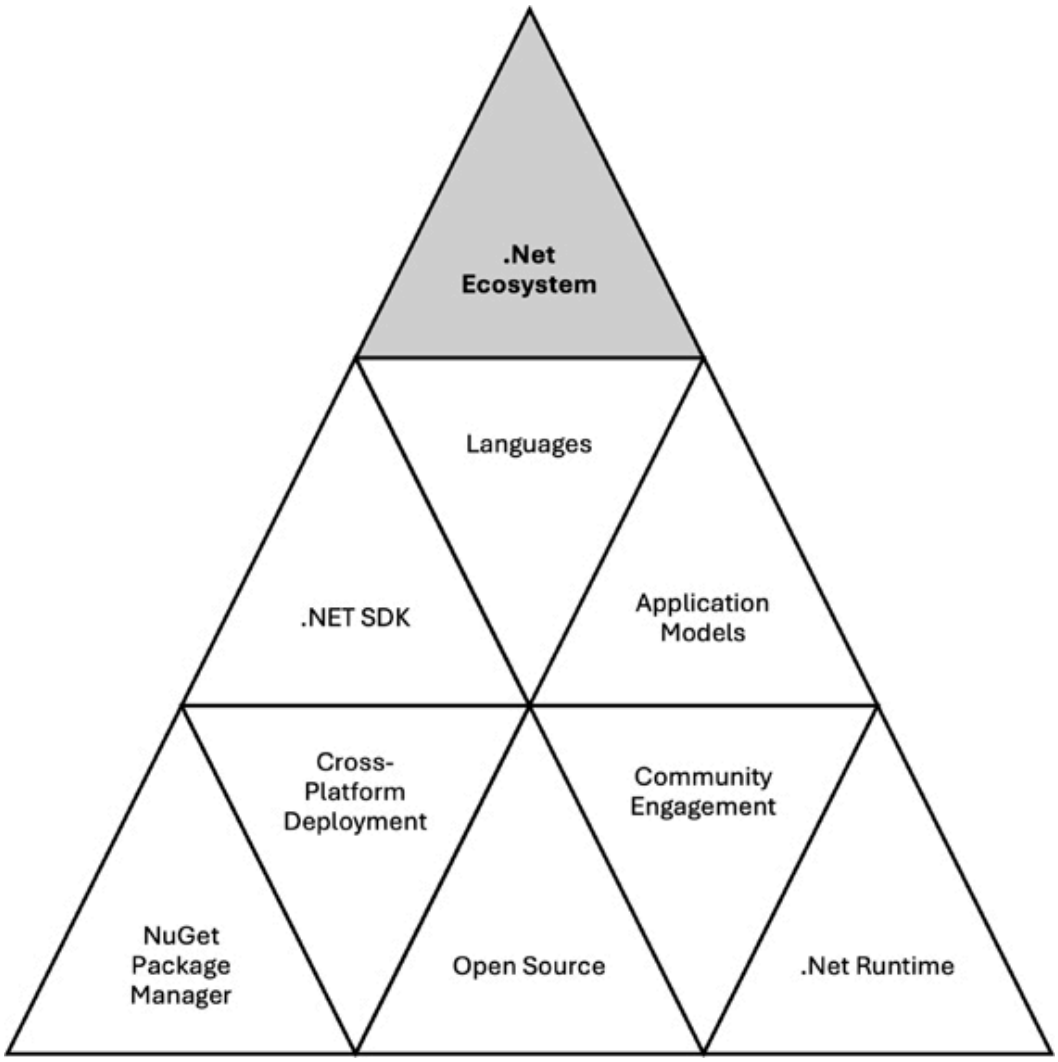

Figure 1.1: *.NET Ecosystem Major Components*

Let us explore each of these primary components, starting with the **.NET runtime.**

This is .NET's execution environment. Core to the runtime environment is the **CLR** for managing code execution. The runtime also includes the **Base Class Library (BCL)**, which is a robust set of **Application Programming Interfaces (APIs)**. These APIs make it easier for developers to implement common programming tasks such as working with databases, networks, and files. We will take a closer look at the CLR in the next section, *Common Language Runtime and Compilation*.

The **.NET SDK** is another core component of the .NET ecosystem. The SDK works with our **IDE** to allow us to develop .NET applications. The SDK includes command-line tools, compilers, and more. We will take a closer look at the .NET

SDK in a section later in this chapter, *Installing .NET SDK and Visual Studio Code*.

The .NET ecosystem supports C# as well as C++, F#, and Visual Basic .NET. C# is the most common language for .NET applications and is the focus of this book. Impressively, we can write components of a system in any of the supported languages, and they will interoperate with the .NET platform. This is possible because they all share the **Common Intermediate Language (CIL)**, which is what we compile our programs to, so that the CLR can execute them. We will examine CIL in the *Writing Your First C# Program* section later in this chapter.

When developing a .NET application, we select a model specific to the type of application we are developing. These models include those listed in *Table 1.2*.

Model	Purpose
ASP.NET Core	Web apps and APIs
Azure Integration	Cloud-native apps
Blazor	Interactive C# browser apps
Windows Forms and WPF	Desktop apps
Xamarin and .NET MAUI	Mobile apps

Table 1.2: *.NET Application Models and Their Purpose*

The .NET ecosystem also includes a package manager. **NuGet** is the official .NET package manager, helping us with the distribution and use of reusable code. NuGet accomplishes this task using packages.

Packages are collections of related libraries, modules, and codes.

Package management is an organized approach to adding tools and libraries to our projects and managing dependencies. It eases the task of code organization, improves code reusability, and facilitates distribution.

Development tools are an important part of the .NET ecosystem. Several **IDEs** can be used for developing .NET applications. The primary IDEs used with .NET are **Visual Studio** and **Visual Studio Code**. We will explore Visual Studio Code in the *Installing .NET SDK and Visual Studio Code* section of this chapter.

The final three components of the .NET ecosystem are:

- **Cross-Platform Development:** We can deploy across multiple operating systems from a single codebase. This significantly reduces our development effort.
- **Open Source:** The .NET platform is developed on GitHub.

- **Community Engagement:** Developers from the global community contribute to the .NET platform.

In summary, the .NET ecosystem is a robust set of tools, libraries, and frameworks that enable us to develop high-performance, readable, scalable, and maintainable systems.

Common Language Runtime and Compilation

The Common Language Runtime (CLR) is the .NET execution engine. It provides critical runtime services, including memory management, garbage collection, type safety, security, and more. This section explores the CLR's role and its behaviors.

The process of creating, compiling, and executing a C# program using the .NET model is shown in *Figure 1.2*. The process starts with our C# source code.

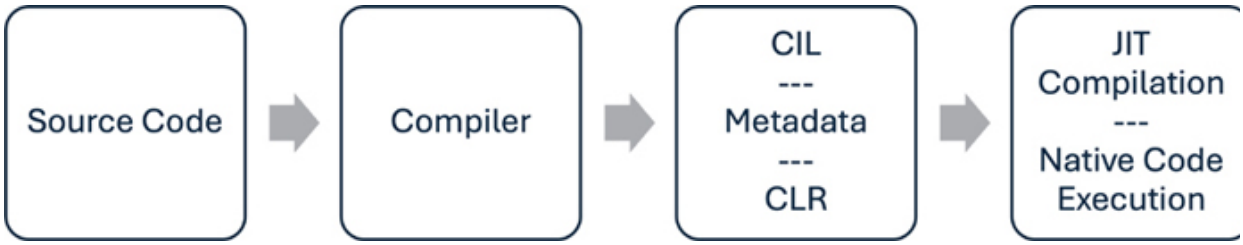

Figure 1.2: *C# Program Compilation and Execution Process*

As depicted in *Figure 1.2*, our source code is transformed using a multistep process before it can be executed (`run`) on a target device. The process starts with the source code being compiled into **Common Intermediate Language (CIL)**. This is an intermediate representation of our code and is not run directly on host machines. CIL is platform-agnostic which enhances portability, interoperability, and runtime optimization. CIL allows us to use the same compiled code on any system that supports the .NET runtime. Using this schema, we can write a cross-platform application and have it run on Windows, macOS, and Linux systems without having to make modifications for each platform.

The CLR, the .NET execution engine, uses a **Just-in-Time (JIT) compiler** when our application is launched to convert the CIL code into the host computer specific native machine code. This is a powerful capability that contributes to high performance.

The .NET JIT compilation process (Source Code-CILJIT-CLR-Native Code) is similar to Java's JIT process (Source Code-ByteCode-JIT-JVM-Native Code).

Once the native code is running, the CLR can monitor memory and release unused memory back to our application through a process called automatic

garbage collection. This process is managed by the CLR and frees us up from having to manually allocate and deallocate memory.

As previously stated, .NET supports C#, C++, F# and Visual Basic and they all compile to the same intermediate language (CIL). This allows CLR to execute code written in any of those languages within the same application. This feature allows us to leverage the existing codebases, libraries, and even the expertise of individual developers. As you would assume, this greatly increases code reuse and productivity.

We will further examine CLR and CIL in the *Writing Your First C# Program* section later in this chapter.

Installing .NET SDK and Visual Studio Code

There are two primary tools required to develop .NET applications, an IDE and the .NET SDK. We will use Visual Studio Code for our IDE, which is where we will write and edit our code. The .NET SDK will provide us with the compiler, tools, and libraries required to write and run C# applications. This section walks through installing both of those tools using a macOS system. While the screenshots and commands presented in this section reflect macOS, the installation processes are comparable on Windows and Linux systems.

Installing Visual Studio Code

Visual Studio Code (VS Code) is a Microsoft product, as is C# and the .NET framework. This helps ensure a streamlined development experience and is the reason VS Code is featured in this book. Follow the steps below to download and install VS Code.

1. Visit the **https://code.visualstudio.com/** site. As shown in *Figure 1.3*, the browser will detect your operating system and provide you with a download button specific to your system.

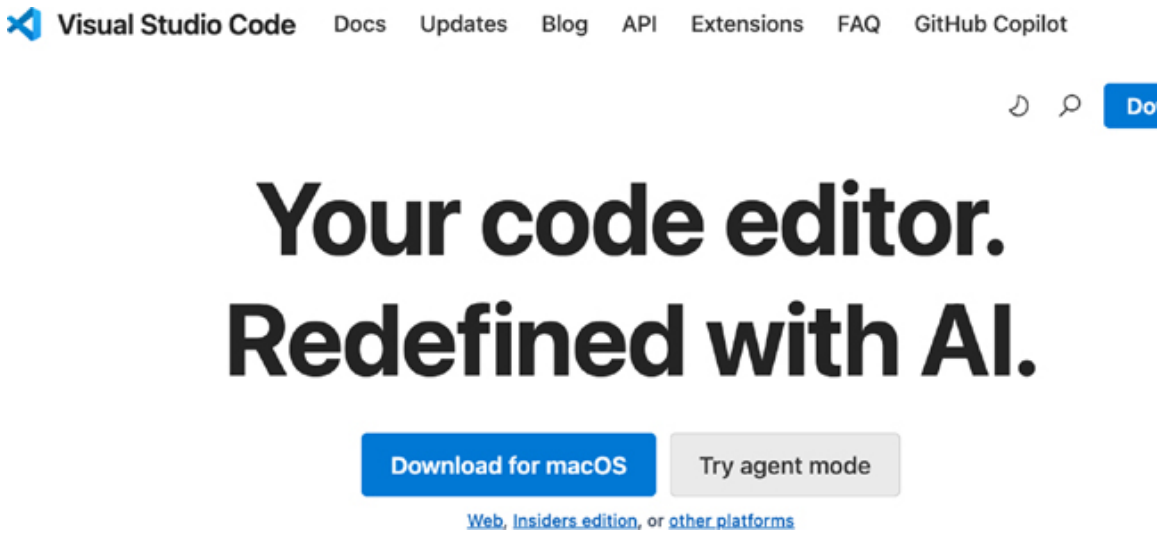

Figure 1.3: *VS Code Download Page*

2. Click the `Download` button and take note of the download location on your system.
3. Install VS Code using the download from Step 2. If you downloaded a .zip file, you will need to decompress/extract it first.
4. Launch VS Code.
5. With VS Code open, access the **Extensions** interface using the **Activity Bar** (see *Figure 1.4*) or using the *Command+Shift+X* keyboard combination.

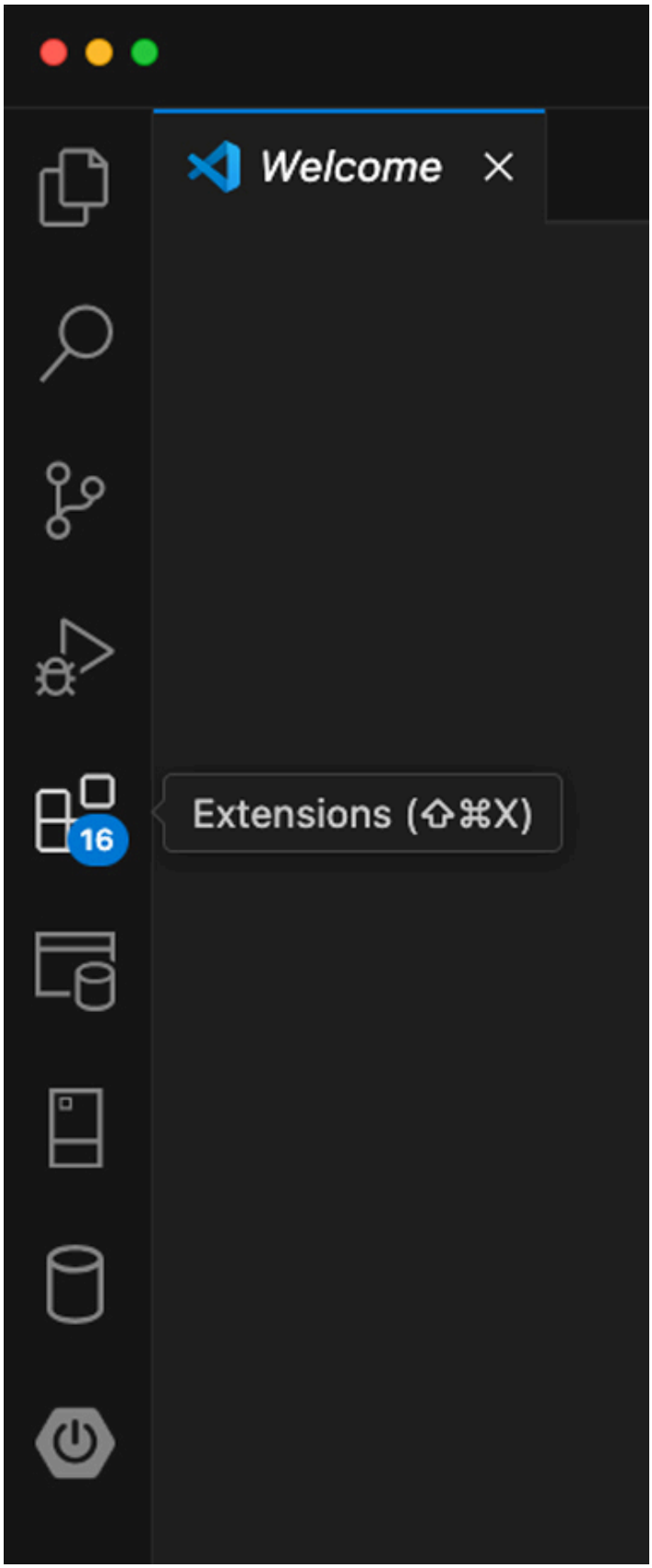
Welcome
16
Extensions (⇧⌘X)

Figure 1.4: *VS Code Activity Bar with Extensions Option Highlighted*

6. In the Extensions interface, search for **`C# Dev Kit`** by Microsoft (see *Figure 1.5*).

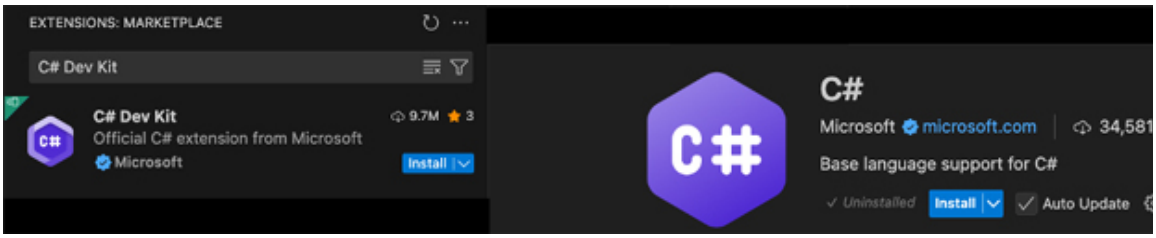

Figure 1.5: *VS Code C# Extension*

7. Click the **`Install`** button. You should now have VS Code installed on your computer with the C# extension by Microsoft. This extension provides us with tools to help us develop using C#. These tools will be explored throughout the remainder of this book.

Installing the .NET SDK

We are now ready to install the .NET SDK. This is the tool that provides us with source code compilers, command-line tools, and runtime libraries necessary for writing and running C# applications. Follow the steps below:

1. Visit the **https://dotnet.microsoft.com/download** page. As you can see in *Figure 1.6*, the page will detect your operating system and provide you with appropriate download options for your system.

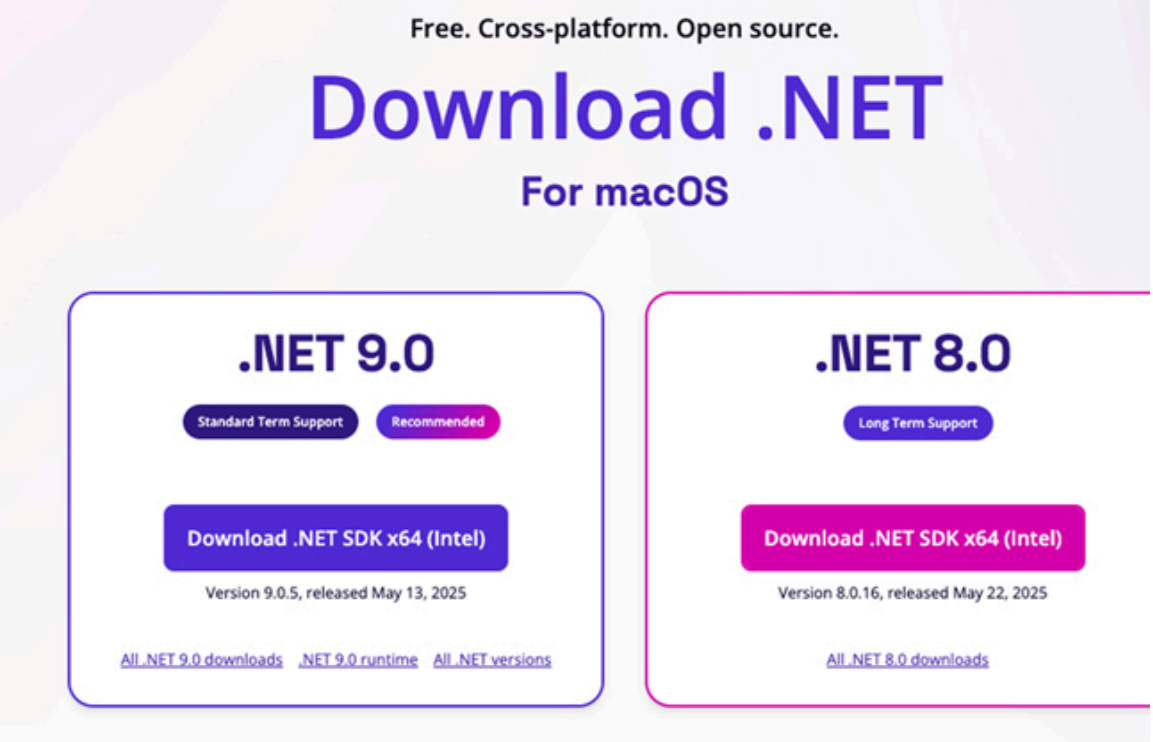

Figure 1.6: *.NET SDK Download Page*

2. Click the **`Download`** button and take note of the download location on your system.
3. Install the SDK using the download from Step 2. Follow the installation wizard's prompts to install the SDK system-wide. Note: You may be prompted for your system's password. Once the installation is complete, you should see a confirmation window as shown in *Figure 1.7*.

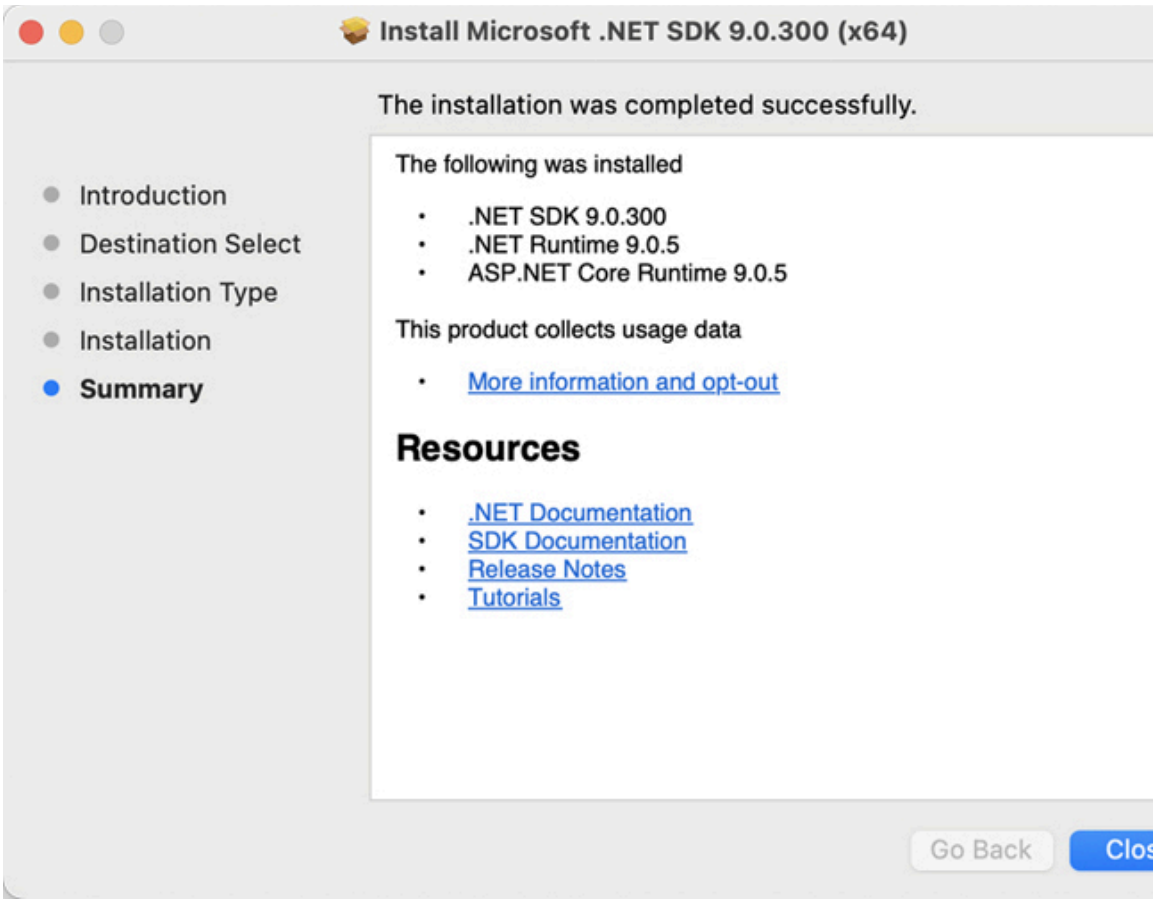

Figure 1.7: *.NET SDK Installation Confirmation Window*

You should now have everything ready to start developing and running C# programs with the .NET framework using VS Code. In the next section, we will walk through creating a simple C# program to test our installations.

Writing Your First C# Program

Now that we have VS Code, the C# extension, and the .NET SDK installed on our system, we are ready to write and execute a simple C# application. Our goals for this task are twofold: to test our installations and to quickly introduce the characteristics (for example, syntax and structure) of a C# application.

Follow the steps below to create a C# source file from scratch.

1. Create a project folder on your computer's file system. For example, you might create a `CH01` folder to coincide with the book's current chapter.
2. Open a terminal window and navigate to the new folder you created in Step 1. Enter the following command in the terminal window to set up the folder as a .NET project:

```
dotnet new console -name MyFirstProgram
```

3. Launch VS Code.
4. Open the folder you created in Step 1. You can do this via the Explorer, the Start window, or via the VS Code menu (**`File -> Open Folder`**). You will note that VS Code created two subfolders (**`.vscode`** and **`MyFirstProgram`**). (See *Figure 1.8*).

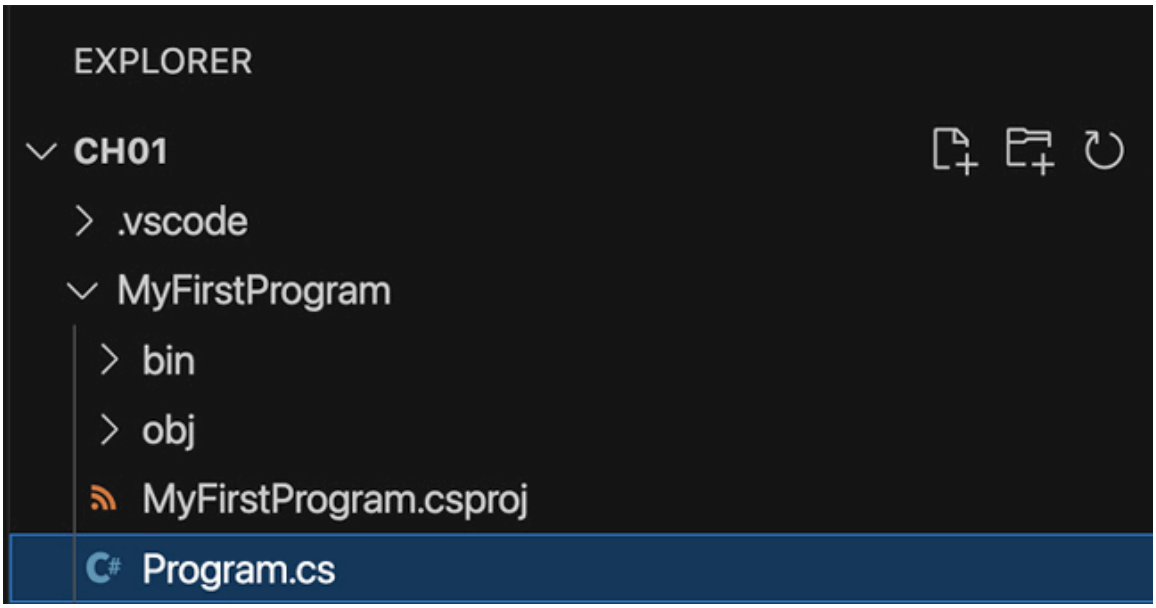

Figure 1.8: *Explorer New File Icon*

Note: *C# and .NET follows the PascalCase naming convention for programs. This schema capitalizes the first letter in each word such as with MyProgram.cs.*

5. Click the **`Program.cs`** file (see *Figure 1.8*) and replace the contents with the following code:

```
using System;
namespace MyFirstProgram
{
 class Program
 {
  static void Main(string[] args)
  {
   Console.WriteLine("Welcome to the world of C#
   programming!");
  }
 }
```

```
}
```

6. Save the file.
7. Next, we will build our project using the following command in the terminal window: **`dotnet build`**.
8. Now, build the project with this terminal command: **`dotnet build`**
9. Lastly, you can use the following terminal command to run your program: **`dotnet run`**. You should see the following in your terminal window:

```
Welcome to the world of C# programming!
```

Code Review

Now that we successfully created and ran our first C# program, refer to the line numbers in *Figure 1.9* for an explanation of each line of code.

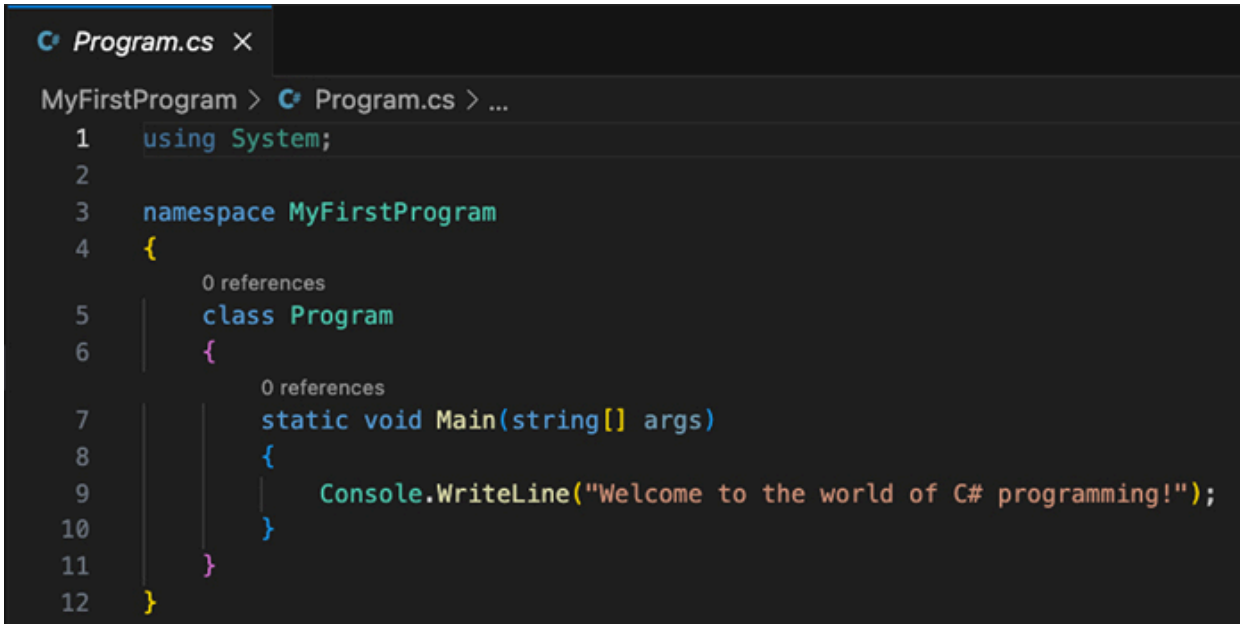

Figure 1.9: *Program.cs with Line Numbers*

- Line 1. This line imports the System namespace, which gives us access to basic classes and are necessary to use the console (our commands in the terminal window).
- Line 3. Here we set the namespace to group related classes. This helps us avoid class naming conflicts.
- Line 5. This line defines our class name.
- Line 7. The **`static void Main(string[] args)`** line of code sets up our **`Main`** method, which is the designated entry point of C# applications.

- Line 9. This statement outputs text to the console (terminal window).

Our simple program represents a complete and valid C# application.

Process Review

When we ran `dotnet run`, the .NET CLI performed several steps behind the scenes. First, our `Program.cs` file was compiled into Common Intermediate Language (CIL) code and packaged into a .NET assembly (typically a `.dll` file). Next, CLR loaded the compiled assembly and invoked the `Main` method, which produced the output via the `Console.WriteLine` method.

This process demonstrates .NET's execution model.

Conclusion

This introductory chapter was intended to help you obtain the foundational knowledge required so that you can develop applications with C# and the .NET platform. We explored the history and characteristics of the C# programming language, examined the .NET ecosystem, and covered the role the CLR has in the compilation process. The chapter provided walkthroughs for installing Visual Studio Code and the .NET SDK. We also had the opportunity to write our first C# program using the best practices for organization and execution.

The next chapter covers the foundational data elements of C# programming: variables, data types, and type conversion. It starts with explaining how to declare and initialize variables using different data types such as integers, floating-point numbers, characters, and Booleans. The chapter explains the significance of constants and introduces type inference. From there, the chapter explores how data is stored and manipulated in memory, along with the rules governing type compatibility. Important concepts like implicit and explicit type conversion, casting, and the difference between boxing and unboxing are also covered. The concepts of variable scope and lifetime, and their effect on how and where variables can be used, are explored. Practical examples are provided throughout the chapter to illustrate how these core concepts apply in real-world programming scenarios. By the end of the chapter, you should have the tools and confidence to handle basic data operations in C#.

Key Terms

- **C# (C-Sharp):** An object-oriented programming language developed by Microsoft for building a wide range of applications with the .NET platform.

- **Code Compilation:** The process of transforming source code into executable instructions, typically involving conversion to Common Intermediate Language (CIL) in .NET.
- **Common Intermediate Language (CIL):** A system-independent set of instructions to which .NET languages compile for execution by the Common Language Runtime.
- **Common Language Runtime (CLR):** The .NET execution environment responsible for code execution, memory management, type safety, and other runtime services.
- **Common Language Specification (CLS):** A set of rules and constraints that ensures interoperability among different .NET programming languages.
- **Common Type System (CTS):** A .NET specification that defines how types are declared and used across all languages targeting the platform.
- **Console Output:** Text output that is displayed in a terminal or command-line interface, often used for simple interactions or debugging.
- **Dotnet CLI:** The command-line interface for interacting with .NET used to create, build, and run projects.
- **Garbage Collection:** An automatic memory management feature of the CLR that reclaims memory occupied by objects no longer in use.
- **Just-In-Time (JIT) Compilation:** The process by which the CLR converts CIL into native machine code at runtime.
- **Namespace:** A container for classes and other components that organizes code and prevents naming conflicts.
- **.NET SDK:** A software development kit that includes compilers, tools, and libraries necessary for building and running .NET applications.
- **NuGet:** The .NET package manager used to distribute and install reusable code libraries and tools.
- **Visual Studio Code (VS Code):** A cross-platform code editor developed by Microsoft that supports C# through extensions.

Points to Remember

- A C# program must contain a Main method as its entry point.
- A namespace is used to group related classes and prevent naming conflicts.
- Adding the C# extension to Visual Studio Code enables advanced editing features such as IntelliSense and debugging.

- All .NET languages, including C#, compile to Common Intermediate Language (CIL), which is executed by the CLR.
- Automatic memory management in .NET is handled by garbage collection, which reclaims unused memory.
- C# is a statically typed language, meaning variable types are checked at compile time.
- CIL is platform-independent and enables cross-platform execution through the .NET runtime.
- Code inside the Main method is the first to execute when a console application starts.
- Console output is generated using `Console.WriteLine`, which prints text to the terminal.
- Creating a new C# project can be done using the `dotnet new` command.
- Cross-platform development is a key advantage of modern .NET, supported on Windows, macOS, and Linux.
- Each time a method is invoked for the first time, it is compiled to native code by the JIT compiler.
- Extensions in Visual Studio Code can significantly enhance the development experience by adding language-specific support.
- Garbage collection is performed automatically by the CLR and does not require explicit memory management by the developer.
- Installing the .NET SDK provides the compilers and tools needed to build and run C# programs.
- Just-In-Time (JIT) compilation occurs during program execution, converting CIL into machine code.
- `Program.cs` is the default file used in console applications and typically contains the `Main` method.
- The CLR provides services such as exception handling, type safety, and memory management.
- The CTS defines how data types are declared and managed across .NET languages.
- The dotnet CLI allows us to create, build, and run .NET applications from the command line.
- The .NET SDK must be installed to develop and execute .NET applications.

- The Visual Studio Code editor is lightweight, cross-platform, and supports C# through extensions.
- Typing **`dotnet run`** in the terminal builds and executes the current project.

Multiple Choice Questions

1. What is the primary role of the Common Language Runtime (CLR) in the .NET ecosystem?
    a. It is a code editor for writing C# applications.
    b. It manages memory, compiles C# code to native code, and enforces type safety.
    c. It provides hosting services for web applications.
    d. It converts native code into high-level source code.
2. Which of the following files typically contains the entry point for a basic C# console application?
    a. `Main.cs`
    b. `FirstProgram.csproj`
    c. `Program.cs`
    d. `Startup.cs`
3. What command initializes a new console application using the .NET CLI?
    a. dotnet init project
    b. dotnet create console
    c. dotnet new console
    d. dotnet start console
4. Why is Common Intermediate Language (CIL) used in the .NET compilation process?
    a. It enables cross-platform compatibility and delayed native compilation.
    b. It simplifies file compression.
    c. It allows code to be translated into other languages.
    d. It encrypts code for security purposes.
5. Which Visual Studio Code feature assists developers by suggesting method names and parameters as they type?

a. Code Complete
b. Build Tracker
c. Syntax Viewer
d. IntelliSense

Answers

1. b
2. c
3. c
4. a
5. d

Questions

1. What aspects of C# make it particularly well-suited for enterprise-level development, and how do these compare with other programming languages you have encountered?
2. In your own words, explain the relationship between the C# programming language, the .NET SDK, and the Common Language Runtime (CLR). Why is it important to understand how they work together?
3. Reflect on the decision to install Visual Studio Code before the .NET SDK in the development setup. What are the advantages of this order, and how might it affect your learning experience?
4. How does compiling C# into Common Intermediate Language (CIL) contribute to the cross-platform capabilities of .NET? Do you see this as an advantage or a limitation in certain contexts? Explain your position.
5. Think about your first impressions of the `Main` method in a C# program. Why do you think the language requires such a method, and how does it help structure an application?
6. Consider the role of IntelliSense in Visual Studio Code. How does this tool impact your coding efficiency, especially as you begin learning a new language like C#?
7. Describe the benefits and potential drawbacks of using a lightweight code editor like Visual Studio Code compared to a full-featured IDE such as Visual Studio Community Edition.

8. Discuss the significance of namespaces in C# applications. Why might this concept become increasingly important as software projects become more complex?
9. After manually writing your first C# program from scratch, what did you learn about the structure of C# code that might have been obscured by using auto-generated templates?
10. The .NET ecosystem is both open source and maintained by a large corporate entity (Microsoft). What are the implications of this dual stewardship for developers and the broader technology community?

Exercises

- **Exercise 1: Console Greeting**
  - **Objective:** Practice writing and running a basic C# program using the .NET CLI and Visual Studio Code.
  - **Challenge:** Create a new C# console application that prints a unique greeting.
- **Exercise 2: Console Output**
  - **Objective:** Additional practice writing and running a basic C# program using the .NET CLI and Visual Studio Code.
  - **Challenge:** Create a new C# console application that prints two lines of text.
- **Exercise 3: Take a C# Tour – Familiar Features**
  - **Objective:** Develop additional familiarity with C# language familiar features by exploring the official documentation.
  - **Challenge:** Visit the **https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview** page and study the *Familiar C# Features* section.
- **Exercise 4: Take a C# Tour – Distinct Features**
  - **Objective:** Develop additional familiarity with C# language distinct features by exploring the official documentation.
  - **Challenge:** Visit the **https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/overview** page and study the *Distinct C# Features* section.

- **Exercise 5: Manual Compilation and Execution**
    - **Objective:** Understand the structure of C# compilation without relying on project scaffolding.
    - **Challenge:** Without using `dotnet new`, create a folder and manually write a valid `Program.cs` file. Use the `dotnet build` and `dotnet run` commands to compile and execute your application. Submit both the code and a written description of the steps you followed to complete the task.