

Essential C# 7.0

 **Mark Michaelis**
with Eric Lippert, Technical Editor

◆ Addison-Wesley

Boston * Columbus * Indianapolis * New York * San Francisco * Amsterdam * Cape Town
Dubai * London * Madrid * Milan * Munich * Paris * Montreal * Toronto * Delhi * Mexico City
São Paulo * Sydney * Hong Kong * Seoul * Singapore * Taipei * Tokyo



Contents at a Glance

<i>Contents</i>	<i>ix</i>
<i>Figures</i>	<i>xv</i>
<i>Tables</i>	<i>xvii</i>
<i>Foreword</i>	<i>xix</i>
<i>Preface</i>	<i>xxi</i>
<i>Acknowledgments</i>	<i>xxxiii</i>
<i>About the Author</i>	<i>xxxv</i>

1	Introducing C#	1
2	Data Types	43
3	More with Data Types	77
4	Operators and Control Flow	109
5	Methods and Parameters	181
6	Classes	241
7	Inheritance	313
8	Interfaces	353
9	Value Types	379
10	Well-Formed Types	411
11	Exception Handling	465

12	Generics	487
13	Delegates and Lambda Expressions	537
14	Events	575
15	Collection Interfaces with Standard Query Operators	603
16	LINQ with Query Expressions	657
17	Building Custom Collections	679
18	Reflection, Attributes, and Dynamic Programming	721
19	Multithreading	771
20	Thread Synchronization	863
21	Platform Interoperability and Unsafe Code	897
22	The Common Language Infrastructure	923
	<i>Index</i>	<i>945</i>
	<i>Index of 7.0 Topics</i>	<i>995</i>
	<i>Index of 6.0 Topics</i>	<i>998</i>
	<i>Index of 5.0 Topics</i>	<i>1001</i>



Contents

Figures xv

Tables xvii

Foreword xix

Preface xxi

Acknowledgments xxxiii

About the Author xxxv

1 Introducing C# 1

Hello, World 2

C# Syntax Fundamentals 11

Working with Variables 20

Console Input and Output 24

Managed Execution and the Common Language Infrastructure 32

Multiple .NET Frameworks 37

2 Data Types 43

Fundamental Numeric Types 44

More Fundamental Types 53

null and void 67

Conversions between Data Types 69

3 More with Data Types 77

Categories of Types 77

Nullable Modifier 80

Tuples 83

Arrays 90



4 Operators and Control Flow 109

- Operators 110
- Introducing Flow Control 126
- Code Blocks ({ }) 132
- Code Blocks, Scopes, and Declaration Spaces 135
- Boolean Expressions 137
- Bitwise Operators (<<, >>, |, &, ^, ~) 147
- Control Flow Statements, Continued 153
- Jump Statements 165
- C# Preprocessor Directives 171

5 Methods and Parameters 181

- Calling a Method 182
- Declaring a Method 189
- The using Directive 195
- Returns and Parameters on Main() 200
- Advanced Method Parameters 203
- Recursion 215
- Method Overloading 217
- Optional Parameters 220
- Basic Error Handling with Exceptions 225

6 Classes 241

- Declaring and Instantiating a Class 245
- Instance Fields 249
- Instance Methods 251
- Using the this Keyword 252
- Access Modifiers 259
- Properties 261
- Constructors 278
- Static Members 289
- Extension Methods 299
- Encapsulating the Data 301

Nested Classes 304

Partial Classes 307

7 Inheritance 313

Derivation 314

Overriding the Base Class 326

Abstract Classes 338

All Classes Derive from `System.Object` 344Verifying the Underlying Type with the `is` Operator 345Pattern Matching with the `is` Operator 346Pattern Matching within a `switch` Statement 347Conversion Using the `as` Operator 349**8 Interfaces 353**

Introducing Interfaces 354

Polymorphism through Interfaces 355

Interface Implementation 360

Converting between the Implementing Class and Its Interfaces 366

Interface Inheritance 366

Multiple Interface Inheritance 369

Extension Methods on Interfaces 369

Implementing Multiple Inheritance via Interfaces 371

Versioning 374

Interfaces Compared with Classes 375

Interfaces Compared with Attributes 377

9 Value Types 379

Structs 383

Boxing 390

Enums 398

10 Well-Formed Types 411

Overriding object Members 411

Operator Overloading 424

Referencing Other Assemblies 432

- Defining Namespaces 442
- XML Comments 445
- Garbage Collection 449
- Resource Cleanup 452
- Lazy Initialization 461
- 11 Exception Handling 465**
 - Multiple Exception Types 465
 - Catching Exceptions 469
 - General Catch Block 473
 - Guidelines for Exception Handling 475
 - Defining Custom Exceptions 479
 - Rethrowing a Wrapped Exception 483
- 12 Generics 487**
 - C# without Generics 488
 - Introducing Generic Types 493
 - Constraints 506
 - Generic Methods 519
 - Covariance and Contravariance 524
 - Generic Internals 531
- 13 Delegates and Lambda Expressions 537**
 - Introducing Delegates 538
 - Declaring Delegate Types 542
 - Lambda Expressions 550
 - Anonymous Methods 556
- 14 Events 575**
 - Coding the Publish-Subscribe Pattern with Multicast Delegates 576
 - Understanding Events 591
- 15 Collection Interfaces with Standard Query Operators 603**
 - Collection Initializers 604
 - What Makes a Class a Collection: IEnumerable<T> 607

Standard Query Operators 613

Anonymous Types with LINQ 646

16 LINQ with Query Expressions 657

Introducing Query Expressions 658

Query Expressions Are Just Method Invocations 676

17 Building Custom Collections 679

More Collection Interfaces 680

Primary Collection Classes 683

Providing an Indexer 702

Returning Null or an Empty Collection 705

Iterators 705

18 Reflection, Attributes, and Dynamic Programming 721

Reflection 722

nameof Operator 733

Attributes 735

Programming with Dynamic Objects 759

19 Multithreading 771

Multithreading Basics 774

Working with System.Threading 781

Asynchronous Tasks 789

Canceling a Task 810

The Task-based Asynchronous Pattern 816

Executing Loop Iterations in Parallel 846

Running LINQ Queries in Parallel 856

20 Thread Synchronization 863

Why Synchronization? 864

Timers 893

21 Platform Interoperability and Unsafe Code 897

Platform Invoke 898

Pointers and Addresses 910

Executing Unsafe Code via a Delegate 920

22 The Common Language Infrastructure 923

Defining the Common Language Infrastructure 924

CLI Implementations 925

.NET Standard 928

Base Class Library 929

C# Compilation to Machine Code 929

Runtime 932

Assemblies, Manifests, and Modules 936

Common Intermediate Language 939

Common Type System 939

Common Language Specification 940

Metadata 941

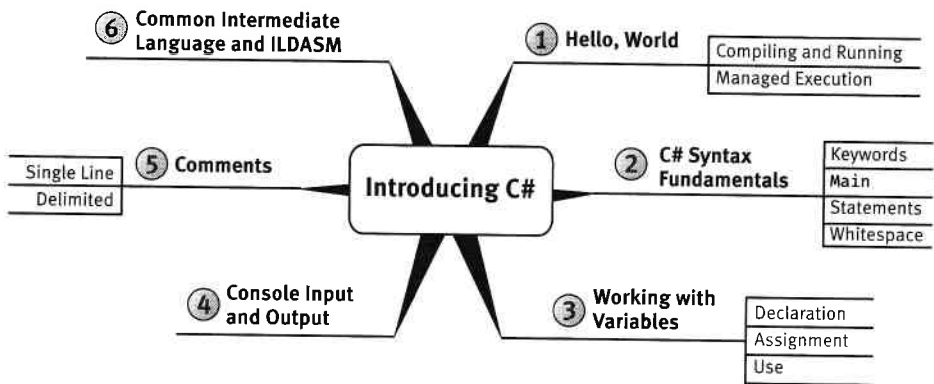
.NET Native and Ahead of Time Compilation 942

*Index 945**Index of 7.0 Topics 995**Index of 6.0 Topics 998**Index of 5.0 Topics 1001*

1

Introducing C#

C# IS NOW A WELL-ESTABLISHED LANGUAGE that builds on features found in its predecessor C-style languages (C, C++, and Java), making it immediately familiar to many experienced programmers.¹ Furthermore, the C# programming language can be used to build software components and applications that run on a wide variety of operating systems (platforms).



This chapter introduces C# using the traditional HelloWorld program. It focuses on C# syntax fundamentals, including defining an entry point into the C# program, which will familiarize you with the C# syntax style

1. The first C# design meeting took place in 1998.

and structure and enable you to produce the simplest of C# programs. Prior to the discussion of C# syntax fundamentals is a summary of managed execution context, which explains how a C# program executes at runtime. This chapter ends with a discussion of variable declaration, writing and retrieving data from the console, and the basics of commenting code in C#.

Hello, World

The best way to learn a new programming language is to write code. The first example is the classic HelloWorld program. In this program, you will display some text to the screen.

Listing 1.1 shows the complete HelloWorld program; in the following sections, you will compile and run the code.

LISTING 1.1: HelloWorld in C#²

```
class HelloWorld
{
    static void Main()
    {
        System.Console.WriteLine("Hello. My name is Inigo Montoya.");
    }
}
```

NOTE

C# is a case-sensitive language: Incorrect case prevents the code from compiling successfully.

Those experienced in programming with Java, C, or C++ will immediately see similarities. Like Java, C# inherits its basic syntax from C and C++.³ Syntactic punctuation (such as semicolons and curly braces), features (such as case sensitivity), and keywords (such as `class`, `public`, and `void`) are familiar to programmers experienced in these languages. Beginners and programmers from other languages will quickly find these constructs intuitive.

2. Refer to the movie *The Princess Bride* if you're confused about the Inigo Montoya references.
3. When creating C#, the language creators reviewed the specifications for C/C++, literally crossing out the features they didn't like and creating a list of the ones they did like. The group also included designers with strong backgrounds in other languages.

Creating, Editing, Compiling, and Running C# Source Code

Once you have written your C# code, it is time to compile and run it. You have a choice of which .NET implementation(s) to use—sometimes referred to as the **.NET framework(s)**. Generally, the implementation is packaged into a **software development kit (SDK)**. The SDK includes the compiler, the runtime execution engine, the framework of pragmatically accessible functionality that the runtime can access (see “Application Programming Interface” later in the chapter), and any additional tooling (such as a build engine for automating build steps) that might be bundled with the SDK. Given that C# has been publicly available since 2000, there are several different .NET frameworks to choose from (see “Multiple .NET Frameworks” later in the chapter).

For each .NET framework, the installation instructions vary depending on which operating system you are developing on and which .NET framework you select. For this reason, I recommend you visit <https://www.microsoft.com/net/download> for download and installation instructions, selecting first the .NET framework and then the package to download based on which operating system you will be developing for. While I could provide further details here, the .NET download site has the most updated instructions for each combination supported.

If you are unsure about which .NET framework to work with, choose .NET Core by default. It works on Linux, macOS, and Microsoft Windows and is the implementation where the .NET development teams are applying the majority of their investments. Furthermore, because of the cross-platform capabilities, I will favor .NET Core instructions inline within the chapters.

There are also numerous ways to edit your source code, including with the most rudimentary of tools, such as Notepad on Windows, TextEdit on Mac/macOS, or vi on Linux. However, you’re likely to want something more advanced so that at least your code is colorized. Any programming editor that supports C# will suffice. If you don’t already have a preference, I recommend you consider the open source editor Visual Studio Code (<https://code.visualstudio.com>). Or, if you are working on Windows or Mac, consider Microsoft Visual Studio 2017 (or later)—see <https://www.visualstudio.com>. Both are available free of charge.

In the next two sections, I provide instructions for both editors. For Visual Studio Code, we rely on the command line (Dotnet CLI) for creating the initial C# program scaffolding in addition to compiling and running the program. For Windows and Mac, we focus on using Visual Studio 2017.

With Dotnet CLI

Begin 7.0

The Dotnet command, `dotnet.exe`, is the Dotnet command-line interface, or Dotnet CLI, and it may be used to generate the initial code base for a C# program in addition to compiling and running the program. (To avoid ambiguity between CLI referring to the Common Language Infrastructure or the command-line interface, throughout the book I will prefix CLI with Dotnet when referring to the Dotnet CLI. CLI without the Dotnet prefix refers to Common Language Infrastructure.) Once you have completed the installation, verify that `dotnet` is an available command from the command prompt—thus verifying your installation.

Following are the instructions for creating, compiling, and executing the `HelloWorld` program from the command line on Windows, macOS, or Linux:

1. Open a command prompt on Microsoft Windows or the Terminal application on Mac/macOS. (Optionally, consider using the cross-platform command-line interface PowerShell.)⁴
2. Create a new directory where you would like to place the code. Consider a name such as `./HelloWorld` or `./EssentialCSharp/HelloWorld`. From the command line, use

```
mkdir ./HelloWorld
```

3. Navigate into the new directory so that it is the command prompt's current location.

```
cd ./HelloWorld
```

4. Execute `dotnet new console` from within the `HelloWorld` directory to generate the initial scaffolding for your program. While several files are generated, the two main files are `Program.cs` and the project file:

```
dotnet new console
```

5. Run the generated program. This will compile and run the default `Program.cs` created by the `dotnet new console` command. The content of `Program.cs` is similar to Listing 1.1, but it outputs "Hello World!" instead.

```
dotnet run
```

4. <https://github.com/PowerShell/PowerShell>

Even though we don't explicitly request the application to compile (or build), that step still occurs because it is invoked implicitly when the `dotnet run` command is executed.

6. Edit the `Program.cs` file and modify the code to match what is shown in Listing 1.1. If you use Visual Studio Code to open and edit `Program.cs`, you will see the advantage of a C#-aware editor, as the code will be colorized indicating the different types of constructs in your program. (Output 1.1 shows an approach using the command line that works for Bash and PowerShell.)
7. Rerun the program:

```
dotnet.exe run
```

Output 1.1 shows the output following the preceding steps.⁵

OUTPUT 1.1

```
1>
2> mkdir ./HelloWorld
3> cd ./HelloWorld/
4> dotnet new console
The template "Console Application" was created successfully.

Processing post-creation actions...
Running 'dotnet restore' on ...\\EssentialCSharp\\HelloWorld\\
HelloWorld.csproj...
Restoring packages for ...\\EssentialCSharp\\HelloWorld\\
HelloWorld.csproj...
Generating MSBuild file ...\\EssentialCSharp\\HelloWorld\\obj\\
HelloWorld.csproj.nuget.g.props.
Generating MSBuild file ...\\EssentialCSharp\\HelloWorld\\obj\\
HelloWorld.csproj.nuget.g.targets.
Restore completed in 184.46 ms for ...\\EssentialCSharp\\
HelloWorld\\HelloWorld.csproj.

Restore succeeded.
5> dotnet run
Hello World!
6> echo '
class HelloWorld
{
    static void Main()
    {
        System.Console.WriteLine("Hello. My name is Inigo Montoya.");
    }
}
' > Program.cs
7> dotnet run
Hello. My name is Inigo Montoya.
8>
```

5. The bold formatting in an Output indicates the user-entered content.

With Visual Studio 2017

With Visual Studio 2017, the procedure is similar, but instead of using the command line, you use an **integrated development environment (IDE)**, which has menus you can choose from rather than executing everything from the command line:

1. Launch Visual Studio 2017.
2. Open the New Project dialog using the **File->New Project (Ctrl+Shift+N)** menu.
3. From the **Search box (Ctrl+E)**, type *Console App* and select the **Console App (.NET Core)—Visual C#** item. For the **Name** text box, use *HelloWorld*, and for the **Location**, select a working directory of your choosing. See Figure 1.1.

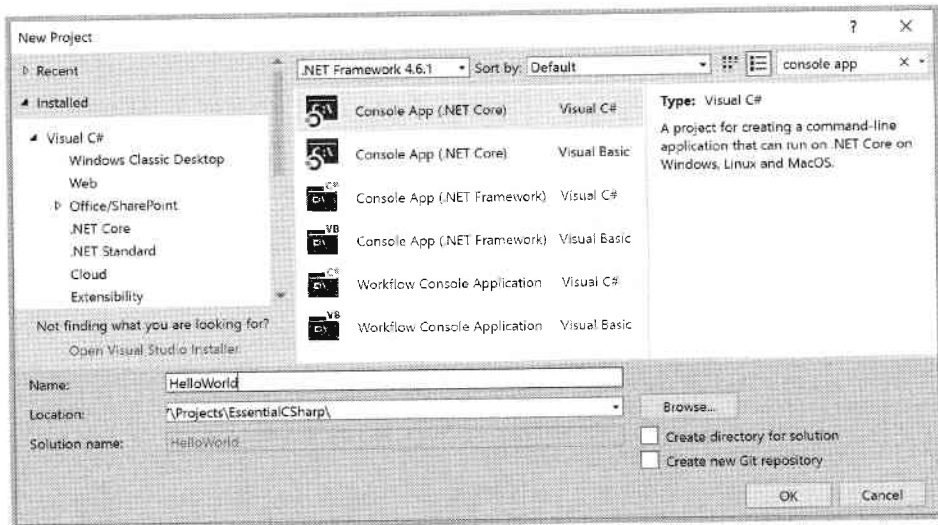


FIGURE 1.1: The New Project Dialog

4. Once the project is created, you should see a `Program.cs` file, as shown in Figure 1.2.
5. Run the generated program using the **Debug->Start Without Debugging (Ctrl+F5)** menu. This will display the command window with the text shown in Output 1.2 except the first line will display only "Hello World!".

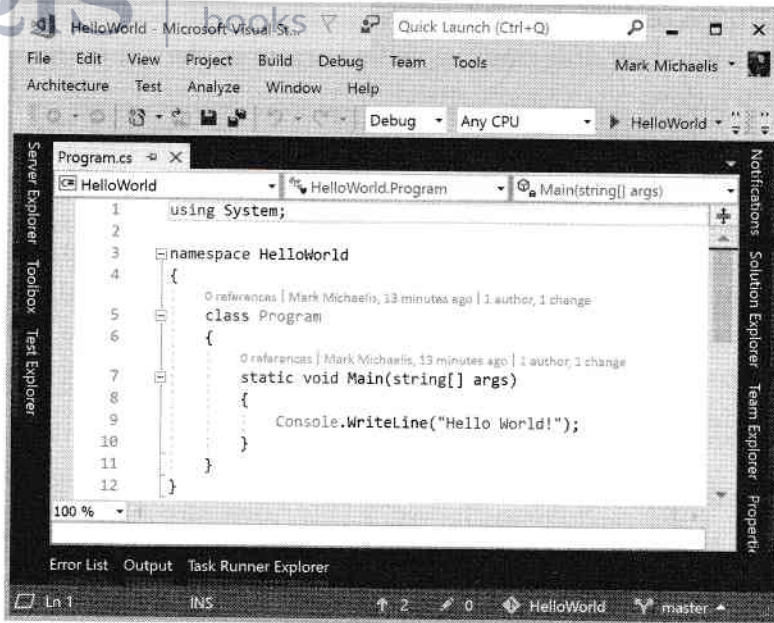


FIGURE 1.2: Dialog That Shows the Program.cs File

6. Modify Program.cs to match Listing 1.1.
7. Rerun the program to see the output shown in Output 1.2.

OUTPUT 1.2

```
> Hello. My name is Inigo Montoya.
Press any key to continue . . .
```

One significant feature of using an IDE is its support for debugging. To try it out, follow these additional steps:

8. Locate your cursor on the `System.Console.WriteLine` line and click the **Debug->Toggle Breakpoint** menu item to activate a breakpoint on that line.
9. Click the **Debug->Start Debugging** menu to relaunch the application but this time with debugging activated. Note that it will stop execution on the line where you set the breakpoint. You can then hover over a variable (e.g., `args`) to see its value. You can also move the current