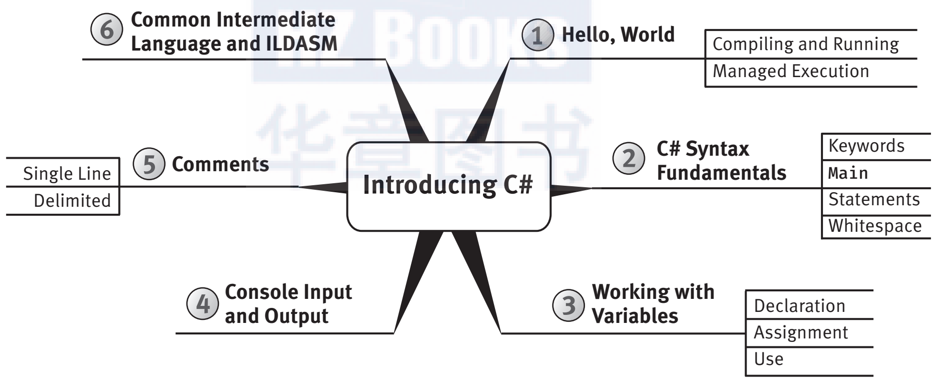
1 Introducing C#

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.

Begin 7.0

With Dotnet CLI

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>
```

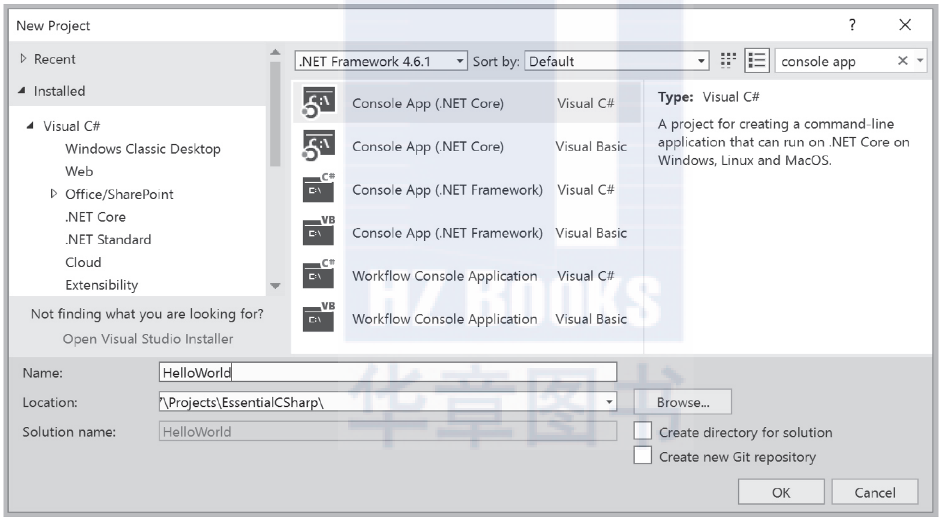
End 7.0

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.



.NET Framework 4.6.1 Sort by: Default console app

Console App (.NET Core)	Visual C#
Console App (.NET Core)	Visual Basic
Console App (.NET Framework)	Visual C#
Console App (.NET Framework)	Visual Basic
Workflow Console Application	Visual C#
Workflow Console Application	Visual Basic

Type: Visual C#

A project for creating a command-line application that can run on .NET Core on Windows, Linux and MacOS.

Name: HelloWorld

Location: "\Projects\EssentialCSharp\

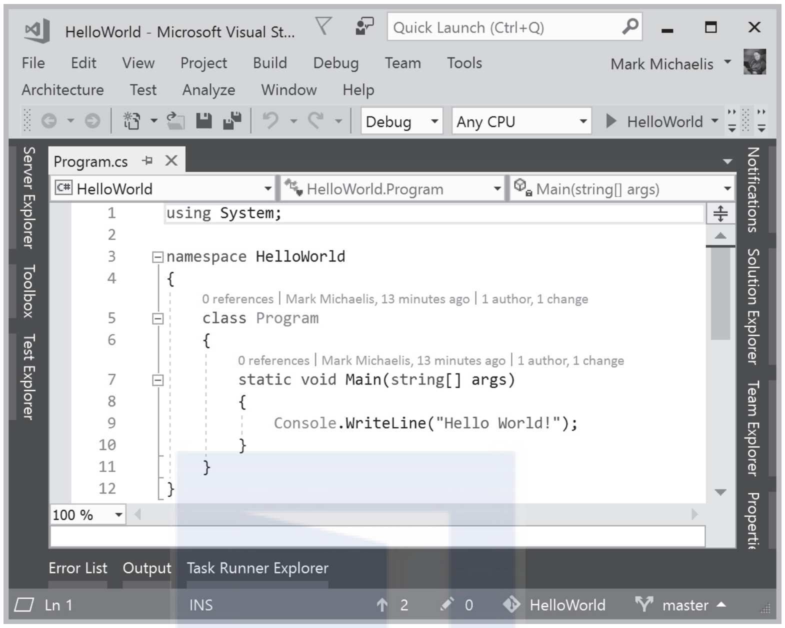
Solution name: HelloWorld

☐ Create directory for solution

☐ Create new Git repository

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!".



```
using System;

namespace HelloWorld
{
    0 references | Mark Michaelis, 13 minutes ago | 1 author, 1 change
    class Program
    {
        0 references | Mark Michaelis, 13 minutes ago | 1 author, 1 change
        static void Main(string[] args)
        {
            Console.WriteLine("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

execution of the program from the current line to another line within the method by dragging the yellow arrow in the left margin of the file window.

10. To continue the program execution, use the **Debug->Continue** (Ctrl+F5) menu (or the **Continue** button).

When debugging, you no longer see the “Press any key to continue…” text in the output window. Instead, the console window automatically closes.

Note that Visual Studio Code can also be used as an IDE, as described at https://code.visualstudio.com/docs/languages/csharp—which includes a link for debugging using Visual Studio Code.

Creating a Project

Whether using Dotnet CLI or Visual Studio, there are several files created. The first file is a C# file with the name `Program.cs` by convention. The name *Program* is commonly used as the starting point for a console program even though any name could be used. The `.cs` extension is the standard convention for all C# files and what the compiler expects to compile into the final program by default. To use the code shown in Listing 1.1, open the `Program.cs` file and replace its content with Listing 1.1. Before saving the updated file, observe that the only functional difference between Listing 1.1 and what was generated by default is the text within quotations and a semantic difference of where *System* appears.

While not necessarily required, typically a configuration file called a **project file** is included as part of the generated source code of your C# project. The project file content will vary from one application type and .NET framework to the next. However, at a minimum, it generally identifies what files to include in the compilation, what application type to build (console, Web, mobile, testing project, etc.), which .NET framework to support, potential settings needed to debug or launch the application, along with other dependencies the code may rely on (called *libraries*). For example, a simple .NET Core console application project file appears in Listing 1.2.

LISTING 1.2: Sample .NET Core Console Project File

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.0</TargetFramework>
  </PropertyGroup>
</Project>
```

In Listing 1.2, the application type is identified as a .NET Core version 2.0 (`netcoreapp2.0`) console application (`Exe`). All other settings (such as which C# files to compile) are identified by convention. For example, by default, all `*.cs` files in the same directory (or subdirectory) as the project file are included in the compile. We delve into more details on a project file in Chapter 10.

Compilation and Execution

Begin 7.0

The compiled output created by the `dotnet build` command is an **assembly** called `HelloWorld.dll`.[6] The extension stands for dynamic link library (DLL), and with .NET Core, all assemblies have a .dll extension even if they are console programs, as this one is. By default, the compiled output for a .NET Core application is placed into a subdirectory (`./bin/Debug/netcoreapp2.0/`). The `Debug` directory is used because the default configuration is debug. This configuration causes the output to be optimized for debugging rather than for optimal performance. The compiled output will not execute on its own. Rather, it needs the CLI to host the code. For .NET Core applications, this requires using the `Dotnet.exe` process as a host process for the application.

Instead of creating a console program that can be executed directly using `dotnet run`, developers can create a library of code that can be referenced by another, larger program. A library is also an assembly. In other words, the output from a successful C# compile is an assembly regardless of whether it is a program or a library.

End 7.0

Begin 2.0

Language Contrast: Java—Filename Must Match Class Name

In Java, the filename must follow the name of the class. In C#, this convention is frequently followed but is not required. In C#, it is possible to have two classes in one file, and starting with C# 2.0, it's possible to have a single class span multiple files with a feature called a partial class.

End 2.0

6. Note that if you use the Microsoft .NET Framework to create a console program, the compiled code is placed into a HelloWorld.exe file that you can execute directly assuming the Microsoft .NET Framework is installed on the computer.

Essential C# Source Code

The source code[7] associated with the book includes a solution file, `EssentialCSharp.sln`, which groups together the code from all the chapters. Building, running, and testing the source code is possible with either Visual Studio or `Dotnet.exe`.

Perhaps the simplest approach is to pass the source code into the `HelloWorld` program created earlier in the chapter and then execute the source code. Alternatively, the source code contains project files for each chapter and a menu from which it is possible to select which listing to run when executing a program, as shown in the next two sections.

Begin 7.0

With Dotnet CLI

To build and execute the code using the Dotnet CLI, open a command prompt with the current directory set to the same directory as the `EssentialCSharp.sln` file. From there, use the `dotnet build` command to compile all the projects.

To run the source code for a particular project, navigate to the directory containing the project file and run the `dotnet run` command. Alternatively, from any directory, use the `dotnet run -p <projectfile>` command where the project file is the path to the project file you are trying to execute (e.g., `dotnet run -p .\src\Chapter01\Chapter01.csproj`). Once executing, the program will prompt for which listing to execute and then proceed with that listing.

Many of the listings have corresponding unit tests in the `Chapter[??].Tests` directory where `[??]` indicates the chapter number. To execute the tests, use the `dotnet test` command from the project tests directory of the test you are looking execute (or the same command from the `EssentialCSharp.sln` file directory to execute them all).

End 7.0

With Visual Studio

After opening the solution file, use the **Build->Build Solution** menu to compile the code with Visual Studio. Before you can execute the source

7. The source code available for this book (along with some chapters related to earlier versions of C#) is available for download at https://IntelliTect.com/EssentialCSharp. You can also download the code from GitHub directly at https://github.com/IntelliTect/EssentialCSharp.

code, you need to select which project to execute by selecting the associated chapter's project as the startup project. For example, to execute the samples in Chapter 1, you would right-click on the **Chapter01** project and choose **Set As Startup Project**. Failure to choose the correct chapter will result in an exception with the message "Error, could not run the Listing..." when you specify the listing number at execution time.

Once you have selected the correct project, you can run the project from the **Debug->Start Without Debugging** menu. Alternatively, if you wish to debug the project, you can use **Debug->Start Debugging**. Once running, the program will prompt for the listing (e.g., 18.33) that you wish to execute. As mentioned earlier, you can enter only listings from the project that was set to start up.

Many of the listings have corresponding unit tests. To execute a test, open the test project and navigate to the test corresponding to the listing you wish to execute. From there, right-click on the test method and choose either **Run Tests** (Ctrl+R, T) or **Debug Tests** (Ctrl+R, Ctrl+T).

C# Syntax Fundamentals

Once you successfully compile and run the `HelloWorld` program, you are ready to start dissecting the code to learn its individual parts. First, consider the C# keywords along with the identifiers that the developer chooses.

BEGINNER TOPIC

Keywords

To enable the compiler to interpret the code, certain words within C# have special status and meaning. Known as **keywords**, they provide the concrete syntax that the compiler uses to interpret the expressions the programmer writes. In the `HelloWorld` program, `class`, `static`, and `void` are examples of keywords.

The compiler uses the keywords to identify the structure and organization of the code. Because the compiler interprets these words with elevated significance, C# requires that developers place keywords only in certain locations. When programmers violate these rules, the compiler will issue errors.

Begin 2.0

Begin 3.0

Begin 4.0

Begin 5.0

Begin 6.0

C# Keywords

Table 1.1 shows the C# keywords.

Table 1.1: C# Keywords

abstract	enum	long	static
add* (1)	equals* (3)	nameof* (6)	string
alias* (2)	event	namespace	struct
as	explicit	new	switch
ascending* (3)	extern	null	this
async* (5)	false	object	throw
await* (5)	finally	on* (3)	true
base	fixed	operator	try
bool	float	orderby* (3)	typeof
break	for	out	uint
by* (3)	foreach	override	ulong
byte	from* (3)	params	unchecked
case	get* (1)	partial* (2)	unsafe
catch	global* (2)	private	ushort
char	goto	protected	using
checked	group* (3)	public	value* (1)
class	if	readonly	var* (3)
const	implicit	ref	virtual
continue	in	remove* (1)	void
decimal	int	return	volatile
default	interface	sbyte	where* (2)
delegate	internal	sealed	when* (6)
descending* (3)	into* (3)	select* (3)	while
do	is	set* (1)	yield* (2)
double	join* (3)	short	
dynamic* (4)	let* (3)	sizeof	
else	lock	stackalloc	

End 2.0

End 3.0

End 4.0

End 5.0

End 6.0

* Contextual keyword. Numbers in parentheses (*n*) identify in which version the contextual keyword was added.

Begin 2.0

After C# 1.0, no new **reserved keywords** were introduced to C#. However, some constructs in later versions use **contextual keywords**, which are significant only in specific locations. Outside these designated locations, contextual keywords have no special significance.[8] By this method, most C# 1.0 code is compatible with the later standards.[9]

End 2.0

Identifiers

Like other languages, C# includes **identifiers** to identify constructs that the programmer codes. In Listing 1.1, `HelloWorld` and `Main` are examples of identifiers. The identifiers assigned to a construct are used to refer to the construct later, so it is important that the names the developer assigns are meaningful rather than arbitrary.

A keen ability to select succinct and indicative names is an important characteristic of a strong programmer because it means the resultant code will be easier to understand and reuse. Clarity coupled with consistency is important enough that the Framework Design Guidelines (http://bit.ly/dotnetguidelines) advise against the use of abbreviations or contractions in identifier names and even recommend avoiding acronyms that are not widely accepted. If an acronym is sufficiently well established (e.g., HTML), you should use it consistently: Avoid spelling out the accepted acronym in some cases but not in others. Generally, adding the constraint that all acronyms be included in a glossary of terms places enough overhead on the use of acronyms that they are not used flippantly. Ultimately, select clear, possibly even verbose names—especially when working on a team or when developing a library against which others will program.

8. For example, early in the design of C# 2.0, the language designers designated `yield` as a keyword, and Microsoft released alpha versions of the C# 2.0 compiler, with `yield` as a designated keyword, to thousands of developers. However, the language designers eventually determined that by using `yield return` rather than `yield`, they could ultimately avoid adding `yield` as a keyword because it would have no special significance outside its proximity to `return`.

9. There are some rare and unfortunate incompatibilities, such as the following:
   - C# 2.0 requires implementation of `IDisposable` with the `using` statement rather than simply using a `Dispose()` method.
   - Some rare generic expressions are different between versions. For example, F(G<A,B>(7)) means F((G<A), (B>7)) in C# 1.0, but in C# 2.0, it means to call generic method G<A,B> with argument 7 and pass the result to F.

There are two basic casing formats for an identifier. **Pascal case** (henceforth PascalCase), as the .NET framework creators refer to it because of its popularity in the Pascal programming language, capitalizes the first letter of each word in an identifier name; examples include `ComponentModel`, `Configuration`, and `HttpFileCollection`. As `HttpFileCollection` demonstrates with HTTP, when using acronyms that are more than two letters long, only the first letter is capitalized. The second format, camel case (henceforth camelCase), follows the same convention except that the first letter is lowercase; examples include `quotient`, `firstName`, `httpFileCollection`, `ioStream`, and `theDreadPirateRoberts`.

Guidelines

DO favor clarity over brevity when naming identifiers.

DO NOT use abbreviations or contractions within identifier names.

DO NOT use any acronyms unless they are widely accepted, and even then, only when necessary.

Notice that although underscores are legal, generally there are no underscores, hyphens, or other nonalphanumeric characters in identifier names. Furthermore, C# doesn't follow its predecessors in that Hungarian notation (prefixing a name with a data type abbreviation) is not used. This convention avoids the variable rename that is necessary when data types change or the inconsistency introduced due to failure to adjust the data type prefix when using Hungarian notation.

In rare cases, some identifiers, such as `Main`, can have a special meaning in the C# language.

Guidelines

DO capitalize both characters in two-character acronyms, except for the first word of a camelCased identifier.

DO capitalize only the first character in acronyms with three or more characters, except for the first word of a camelCased identifier.

DO NOT capitalize any of the characters in acronyms at the beginning of a camelCased identifier.

DO NOT use Hungarian notation (that is, do not encode the type of a variable in its name).

ADVANCED TOPIC

Keywords

Although it is rare, keywords may be used as identifiers if they include `@` as a prefix. For example, you could name a local variable `@return`. Similarly (although it doesn't conform to the casing standards of C# coding standards), it is possible to name a method `@throw()`.

There are also four undocumented reserved keywords in the Microsoft implementation: `__arglist`, `__makeref`, `__reftype`, and `__refvalue`. These are required only in rare interop scenarios, and you can ignore them for all practical purposes. Note that these four special keywords begin with two underscores. The designers of C# reserve the right to make any identifier that begins with two underscores into a keyword in a future version; for safety, avoid ever creating such an identifier yourself.

Type Definition

All executable code in C# appears within a type definition, and the most common type definition begins with the keyword `class`. A **class definition** is the section of code that generally begins with `class identifier { ... }`, as shown in Listing 1.3.

LISTING 1.3: Basic Class Declaration

```
class HelloWorld
{
  //...
}
```

The name used for the type (in this case, `HelloWorld`) can vary, but by convention, it must be PascalCased. For this particular example, therefore, other possible names are `Greetings`, `HelloInigoMontoya`, `Hello`, or simply `Program`. (`Program` is a good convention to follow when the class contains the `Main()` method, described next.)

Guidelines

DO name classes with nouns or noun phrases.

DO use PascalCasing for all class names.

Generally, programs contain multiple types, each containing multiple methods.

Main Method

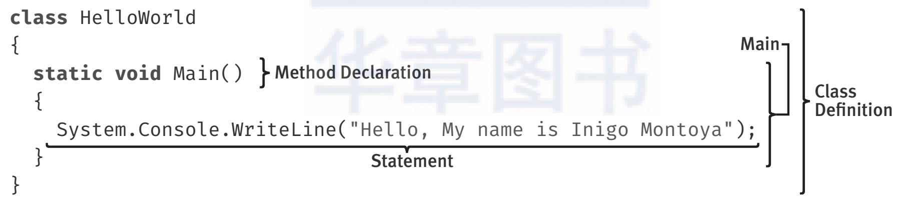
BEGINNER TOPIC

What Is a Method?

Syntactically, a **method** in C# is a named block of code introduced by a method declaration (e.g., `static void Main()`) and (usually) followed by zero or more statements within curly braces. Methods perform computations and/or actions. Similar to paragraphs in written languages, methods provide a means of structuring and organizing code so that it is more readable. More important, methods can be reused and called from multiple places and so avoid the need to duplicate code. The method declaration introduces the method and defines the method name along with the data passed to and from the method. In Listing 1.4, `Main()` followed by `{ ... }` is an example of a C# method.

The location where C# programs begin execution is the `Main` **method**, which begins with `static void Main()`. When you execute the program by typing `HelloWorld.exe` at the command console, the program starts up, resolves the location of `Main`, and begins executing the first statement within Listing 1.4.

LISTING 1.4: Breaking Apart `HelloWorld`



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

Although the `Main` method declaration can vary to some degree, `static` and the method name, `Main`, are always required for a program.

ADVANCED TOPIC

Declaration of the Main Method

C# requires that the `Main` method return either `void` or `int` and that it take either no parameters or a single array of strings. Listing 1.5 shows the full declaration of the `Main` method.

Listing 1.5: The Main Method with Parameters and a Return

```
static int Main(string[] args)
{
    //...
}
```

The `args` parameter is an array of strings corresponding to the command-line arguments. However, the first element of the array is not the program name but the first command-line parameter to appear after the executable name, unlike in C and C++. To retrieve the full command used to execute the program, use `System.Environment.CommandLine`.

The `int` returned from `Main` is the status code, and it indicates the success of the program's execution. A return of a nonzero value generally indicates an error.

Language Contrast: C++/Java—`main()` Is All Lowercase

Unlike its C-style predecessors, C# uses an uppercase *M* for the `Main` method to be consistent with the PascalCased naming conventions of C#.

The designation of the `Main` method as `static` indicates that other methods may call it directly off the class definition. Without the `static` designation, the command console that started the program would need to perform additional work (known as **instantiation**) before calling the method. (Chapter 6 contains an entire section devoted to the topic of static members.)

Placing `void` prior to `Main()` indicates that this method does not return any data. (This is explained further in Chapter 2.)

One distinctive C/C++-style characteristic followed by C# is the use of curly braces for the body of a construct, such as the class or the method. For example, the `Main` method contains curly braces that surround its implementation; in this case, only one statement appears in the method.

Statements and Statement Delimiters

The `Main` method includes a single statement, `System.Console.WriteLine()`, which is used to write a line of text to the console. C# generally uses a semicolon to indicate the end of a **statement**, where a statement comprises one or more actions that the code will perform. Declaring a variable, controlling the program flow, and calling a method are typical uses of statements.

Language Contrast: Visual Basic—Line-Based Statements

Some languages are line based, meaning that without a special annotation, statements cannot span a line. Until Visual Basic 2010, Visual Basic was an example of a line-based language. It required an underscore at the end of a line to indicate that a statement spans multiple lines. Starting with Visual Basic 2010, many cases were introduced where the line continuation character was optional.

ADVANCED TOPIC

Statements without Semicolons

Many programming elements in C# end with a semicolon. One example that does not include the semicolon is a `switch` statement. Because curly braces are always included in a `switch` statement, C# does not require a semicolon following the statement. In fact, code blocks themselves are considered statements (they are also composed of statements), and they don't require closure using a semicolon. Similarly, there are cases, such as the `using` declarative, in which a semicolon occurs at the end but it is not a statement.

Since creation of a newline does not separate statements, you can place multiple statements on the same line and the C# compiler will interpret the line as having multiple instructions. For example, Listing 1.6 contains two statements on a single line that, in combination, display `Up` and `Down` on two separate lines.

Listing 1.6: Multiple Statements on One Line

```
System.Console.WriteLine("Up");System.Console.WriteLine("Down");
```

C# also allows the splitting of a statement across multiple lines. Again, the C# compiler looks for a semicolon to indicate the end of a statement (see Listing 1.7).

Listing 1.7: Splitting a Single Statement across Multiple Lines

```
System.Console.WriteLine(
  "Hello. My name is Inigo Montoya.");
```

In Listing 1.7, the original `WriteLine()` statement from the `HelloWorld` program is split across multiple lines.

BEGINNER TOPIC

What Is Whitespace?

Whitespace is the combination of one or more consecutive formatting characters such as tab, space, and newline characters. Eliminating all whitespace between words is obviously significant, as is including whitespace within a quoted string.

Whitespace

The semicolon makes it possible for the C# compiler to ignore whitespace in code. Apart from a few exceptions, C# allows developers to insert whitespace throughout the code without altering its semantic meaning. In Listings 1.6 and 1.7, it didn't matter whether a newline was inserted within a statement or between statements, and doing so had no effect on the resultant executable created by the compiler.

Frequently, programmers use whitespace to indent code for greater readability. Consider the two variations on `HelloWorld` shown in Listings 1.8 and 1.9.

LISTING 1.8: No Indentation Formatting

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

LISTING 1.9: Removing Whitespace

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

Although these two examples look significantly different from the original program, the C# compiler sees them as identical.

BEGINNER TOPIC

Formatting Code with Whitespace

Indenting the code using whitespace is important for greater readability. As you begin writing code, you need to follow established coding standards and conventions to enhance code readability.

The convention used in this book is to place curly braces on their own line and to indent the code contained between the curly brace pair. If another curly brace pair appears within the first pair, all the code within the second set of braces is also indented.

This is not a uniform C# standard but a stylistic preference.

Working with Variables

Now that you've been introduced to the most basic C# program, it's time to declare a local variable. Once a variable is declared, you can assign it a value, replace that value with a new value, and use it in calculations, output, and so on. However, you cannot change the data type of the variable. In Listing 1.10, `string max` is a variable declaration.

Listing 1.10: Declaring and Assigning a Variable

```
class miracleMax
{
  static void Main()
  {
        Data Type
        string max;
                Variable
      max = "Have fun storming the castle!";
      System.Console.WriteLine(max);
  }
}
```

BEGINNER TOPIC

Local Variables

A **variable** is a name that refers to a value that can change over time. *Local* indicates that the programmer **declared** the variable within a method.

To declare a variable is to define it, which you do by

- Specifying the type of data which the variable will contain
- Assigning it an identifier (name)

Data Types

Listing 1.10 declares a variable with the data type `string`. Other common data types used in this chapter are `int` and `char`.

- `int` is the C# designation of an integer type that is 32 bits in size.
- `char` is used for a character type. It is 16 bits, large enough for (non-surrogate) Unicode characters.

The next chapter looks at these and other common data types in more detail.

BEGINNER TOPIC

What Is a Data Type?

The type of data that a variable declaration specifies is called a **data type** (or object type). A data type, or simply **type**, is a classification of things that share similar characteristics and behavior. For example, *animal* is a type. It classifies all things (monkeys, warthogs, and platypuses) that have animal characteristics (multicellular, capacity for locomotion, and so on). Similarly, in programming languages, a type is a definition for several items endowed with similar qualities.

Declaring a Variable

In Listing 1.10, `string max` is a variable declaration of a string type whose name is `max`. It is possible to declare multiple variables within the same statement by specifying the data type once and separating each identifier with a comma. Listing 1.11 demonstrates such a declaration.

Listing 1.11: Declaring Two Variables within One Statement

```
string message1, message2;
```

Because a multivariable declaration statement allows developers to provide the data type only once within a declaration, all variables will be of the same type.

In C#, the name of the variable may begin with any letter or an underscore (_), followed by any number of letters, numbers, and/or underscores. By convention, however, local variable names are camelCased (the first letter in each word is capitalized, except for the first word) and do not include underscores.

Guidelines

DO use camelCasing for local variable names.

Assigning a Variable

After declaring a local variable, you must assign it a value before reading from it. One way to do this is to use the = **operator**, also known as the **simple assignment operator**. Operators are symbols used to identify the function the code is to perform. Listing 1.12 demonstrates how to use the assignment operator to designate the string values to which the variables `miracleMax` and `valerie` will point.

LISTING 1.12: Changing the Value of a Variable

```
class StormingTheCastle
{
  static void Main()
  {
      string valerie;
      string miracleMax = "Have fun storming the castle!";

      valerie = "Think it will work?";

      System.Console.WriteLine(miracleMax);
      System.Console.WriteLine(valerie);

      miracleMax = "It would take a miracle.";
      System.Console.WriteLine(miracleMax);
  }
}
```

From this listing, observe that it is possible to assign a variable as part of the variable declaration (as it was for `miracleMax`) or afterward in a separate statement (as with the variable `valerie`). The value assigned must always be on the right side of the declaration.

Running the compiled program produces the code shown in Output 1.3.

OUTPUT 1.3

```
>dotnet run
Have fun storming the castle!
Think it will work?
It would take a miracle.
```

In this example, we show the command `dotnet run` explicitly. In future output listings, we will omit this line unless there is something special about the command used to execute the program.

C# requires that local variables be determined by the compiler to be "definitely assigned" before they are read. Additionally, an assignment results in a value. Therefore, C# allows two assignments within the same statement, as demonstrated in Listing 1.13.

LISTING 1.13: Assignment Returning a Value That Can Be Assigned Again

```
class StormingTheCastle
{
  static void Main()
  {
    // ...
    string requirements, miracleMax;
    requirements = miracleMax = "It would take a miracle.";
    // ...
  }
}
```

Using a Variable

The result of the assignment, of course, is that you can then refer to the value using the variable identifier. Therefore, when you use the variable `miracleMax` within the `System.Console.WriteLine(miracleMax)` statement, the program displays `Have fun storming the castle!`, the value of `miracleMax`, on the console. Changing the value of `miracleMax` and executing the same `System.Console.WriteLine(miracleMax)` statement causes the new `miracleMax` value, `It would take a miracle.`, to be displayed.

ADVANCED TOPIC

Strings Are Immutable

All values of type `string`, whether string literals or otherwise, are immutable (or unmodifiable). For example, it is not possible to change the string `Come As You Are.` to `Come As You Age.` A change such as this requires that you reassign the variable instead of modifying the data to which the variable originally referred.

Console Input and Output

This chapter already used `System.Console.WriteLine` repeatedly for writing out text to the command console. In addition to being able to write out data, a program needs to be able to accept data that a user may enter.

Getting Input from the Console

One way to retrieve text that is entered at the console is to use `System.Console.ReadLine()`. This method stops the program execution so that the user can enter characters. When the user presses the Enter key, creating a newline, the program continues. The output, also known as the **return**, from the `System.Console.ReadLine()` method is the string of text that was entered. Consider Listing 1.14 and the corresponding output shown in Output 1.4.

LISTING 1.14: Using System.Console.ReadLine()

```
class HeyYou
{
  static void Main()
  {
      string firstName;
      string lastName;

      System.Console.WriteLine("Hey you!");

      System.Console.Write("Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write("Enter your last name: ");
      lastName = System.Console.ReadLine();
  }
}
```

Output 1.4

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
```

After each prompt, this program uses the `System.Console.ReadLine()` method to retrieve the text the user entered and assign it to an appropriate variable. By the time the second `System.Console.ReadLine()` assignment completes, `firstName` refers to the value `Inigo` and `lastName` refers to the value `Montoya`.

ADVANCED TOPIC

System.Console.Read()

In addition to the `System.Console.ReadLine()` method, there is a `System.Console.Read()` method. However, the data type returned by the `System.Console.Read()` method is an integer corresponding to the character value read, or –1 if no more characters are available. To retrieve the actual character, it is necessary to first cast the integer to a character, as shown in Listing 1.15.

Listing 1.15: Using System.Console.Read()

```
int readValue;
char character;
readValue = System.Console.Read();
character = (char) readValue;
System.Console.Write(character);
```

The `System.Console.Read()` method does not return the input until the user presses the Enter key; no processing of characters will begin, even if the user types multiple characters before pressing the Enter key.

Begin 2.0

In C# 2.0 and above, you can use `System.Console.ReadKey()`, which, in contrast to `System.Console.Read()`, returns the input after a single keystroke. It allows the developer to intercept the keystroke and perform actions such as key validation, restricting the characters to numerics.

End 2.0

Writing Output to the Console

In Listing 1.14, you prompt the user for his or her first and last names using the method `System.Console.Write()` rather than `System.Console.WriteLine()`. Instead of placing a newline character after displaying the text, the `System.Console.Write()` method leaves the current position on the same line. In this way, any text the user enters will be on the same line as the prompt for input. The output from Listing 1.14 demonstrates the effect of `System.Console.Write()`.

Begin 6.0

The next step is to write the values retrieved using `System.Console.ReadLine()` back to the console. In the case of Listing 1.16, the program writes out the user's full name. However, instead of using `System.Console.WriteLine()` as before, this code uses a slight variation that leverages a C# 6.0 feature known as **string interpolation**. Notice the dollar sign preceding the string literal in the call to `Console.WriteLine`; it indicates that string interpolation will be used. Output 1.5 shows the corresponding output.

Listing 1.16: Formatting Using String Interpolation

```
class HeyYou
{
  static void Main()
  {
      string firstName;
      string lastName;

      System.Console.WriteLine("Hey you!");

      System.Console.Write("Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write("Enter your last name: ");
      lastName = System.Console.ReadLine();

      System.Console.WriteLine(
        $"Your full name is { firstName } { lastName }.");
  }
}
```

Output 1.5

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Inigo Montoya.
```

Instead of writing out "Your full name is" followed by another `Write` statement for `firstName`, a third `Write` statement for the space, and finally a `WriteLine` statement for `lastName`, Listing 1.16 writes out the entire output using C# 6.0's string interpolation. With string interpolation, the compiler interprets the interior of the curly brackets within the string as regions in which you can embed code (expressions) that the compiler will evaluate and convert to strings. Rather than executing lots of code snippets individually and combining the results as a string at the end, string interpolation allows you to do this in a single step. This makes the code easier to understand.

End 6.0

Prior to C# 6.0, C# used a different approach, that of **composite formatting**. With composite formatting, the code first supplies a **format string** to define the output format—see Listing 1.17.

LISTING 1.17: Formatting Using `System.Console.WriteLine()`'s Composite Formatting

```
class HeyYou
{
  static void Main()
  {
      string firstName;
      string lastName;

      System.Console.WriteLine("Hey you!");

      System.Console.Write("Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write("Enter your last name: ");
      lastName = System.Console.ReadLine();

      System.Console.WriteLine(
        "Your full name is {0} {1}.", firstName, lastName);
  }
}
```

In this example, the format string is `Your full name is {0} {1}.` It identifies two indexed placeholders for data insertion in the string. Each placeholder corresponds to the order of the arguments that appear after the format string.

Note that the index value begins at zero. Each inserted argument (known as a **format item**) appears after the format string in the order corresponding to the index value. In this example, since `firstName` is the first argument to follow immediately after the format string, it corresponds to index value `0`. Similarly, `lastName` corresponds to index value `1`.

Note that the placeholders within the format string need not appear in order. For example, Listing 1.18 switches the order of the indexed placeholders and adds a comma, which changes the way the name is displayed (see Output 1.6).

Listing 1.18: Swapping the Indexed Placeholders and Corresponding Variables

```
System.Console.WriteLine("Your full name is {1}, {0}",
  firstName, lastName);
```

Output 1.6

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Montoya, Inigo
```

In addition to not having the placeholders appear consecutively within the format string, it is possible to use the same placeholder multiple times within a format string. Furthermore, it is possible to omit a placeholder. It is not possible, however, to have placeholders that do not have a corresponding argument.

NOTE

Since C# 6.0-style string interpolation is almost always easier to understand than the alternative composite string approach, throughout the remainder of the book we use string interpolation by default.

Comments

In this section, we modify the program in Listing 1.17 by adding comments. In no way does this modification change the execution of the program; rather, providing comments within the code can simply make the code more understandable in areas where it isn't inherently clear. Listing 1.19 shows the new code, and Output 1.7 shows the corresponding output.

Listing 1.19: Commenting Your Code

```
class Comment Samples
{
  static void Main()
  {
                    Single-Line Comment
      string firstName; //Variable for storing the first name
      string lastName;  //Variable for storing the last name

      System.Console.WriteLine("Hey you!");
                 Delimited Comment Inside Statement
      System.Console.Write /* No new line */ (
          "Enter your first name: ");
      firstName = System.Console.ReadLine();

      System.Console.Write /* No new line */ (
          "Enter your last name: ");
      lastName = System.Console.ReadLine();

      /* Display a greeting to the console  } Delimited Comment
         using composite formatting. */     }

      System.Console.WriteLine("Your full name is {0} {1}.",
          firstName, lastName);
      // This is the end
      // of the program listing
  }
}
```

Output 1.7

```
Hey you!
Enter your first name: Inigo
Enter your last name: Montoya
Your full name is Inigo Montoya.
```

In spite of the inserted comments, compiling and executing the new program produces the same output as before.

Programmers use comments to describe and explain the code they are writing, especially where the syntax itself is difficult to understand, or perhaps a particular algorithm implementation is surprising. Since comments are pertinent only to the programmer reviewing the code, the compiler ignores comments and generates an assembly that is devoid of any trace that comments were part of the original source code.

Table 1.2 shows four different C# comment types. The program in Listing 1.18 includes two of these.

Begin 2.0

Table 1.2: C# Comment Types

Comment Type	Description	Example
Delimited comments	A forward slash followed by an asterisk, `/*`, identifies the beginning of a delimited comment. To end the comment, use an asterisk followed by a forward slash: `*/`. Comments of this form may span multiple lines in the code file or appear embedded within a line of code. The asterisks that appear at the beginning of the lines but within the delimiters are simply for formatting.	`/*comment*/`
Single-line comments	Comments may be declared with a delimiter comprising two consecutive forward slash characters: `//`. The compiler treats all text from the delimiter to the end of the line as a comment. Comments of this form are considered a single line. It is possible, however, to place sequential single-line comments one after another, as is the case with the last comment in Listing 1.18.	`//comment`
XML delimited comments	Comments that begin with `/**` and end with `**/` are called *XML delimited comments*. They have the same characteristics as regular delimited comments, except that instead of ignoring XML comments entirely, the compiler can place them into a separate text file.†	`/**comment**/`
XML single-line comments	XML single-line comments begin with `///` and continue to the end of the line. In addition, the compiler can save single-line comments into a separate file with the XML delimited comments.	`///comment`

End 2.0

† XML delimited comments were explicitly added only in C# 2.0, but the syntax is compatible with C# 1.0.

A more comprehensive discussion of the XML comments appears in Chapter 10, where we further discuss the various XML tags.

There was a period in programming history when a prolific set of comments implied a disciplined and experienced programmer. This is no longer the case. Instead, code that is readable without comments is more valuable than that which requires comments to clarify what it does. If developers find it necessary to enter comments to clarify what a particular code block is doing, they should favor rewriting the code more clearly over commenting it. Writing comments that simply repeat what the code clearly shows serves only to clutter the code, decrease its readability, and increase the likelihood of the comments going out of date because the code changes without the comments getting updated.

Guidelines

DO NOT use comments unless they describe something that is not obvious to someone other than the developer who wrote the code.

DO favor writing clearer code over entering comments to clarify a complicated algorithm.

BEGINNER TOPIC

Extensible Markup Language

The Extensible Markup Language (XML) is a simple and flexible text format frequently used within Web applications and for exchanging data between applications. XML is extensible because included within an XML document is information that describes the data, known as **metadata**. Here is a sample XML file:

```
<?xml version="1.0" encoding="utf-8" ?>
<body>
  <book title="Essential C# 7.0">
      <chapters>
          <chapter title="Introducing C#"/>
          <chapter title="Data Types"/>
          ...
      </chapters>
  </book>
</body>
```

The file starts with a header indicating the version and character encoding of the XML file. After that appears one main “book” element. Elements begin with a word in angle brackets, such as `<body>`. To end an element, place the same word in angle brackets and add a forward slash to prefix the word, as in `</body>`. In addition to elements, XML supports attributes. `title="Essential C# 7.0"` is an example of an XML attribute. Note that the metadata (book title, chapter, and so on) describing the data (“Essential C# 7.0,” “Data Types”) is included in the XML file. This can result in rather bloated files, but it offers the advantage that the data includes a description to aid in interpreting it.

Managed Execution and the Common Language Infrastructure

The processor cannot directly interpret an assembly. Assemblies consist mainly of a second language known as **Common Intermediate Language (CIL)**, or **IL** for short.[10] The C# compiler transforms the C# source file into this intermediate language. An additional step, usually performed at execution time, is required to change the CIL code into **machine code** that the processor can understand. This involves an important element in the execution of a C# program: the **Virtual Execution System** (**VES**). The VES, also casually referred to as the **runtime,** compiles CIL code as needed (a process known as **just-in-time** compilation or **jitting**). The code that executes under the context of an agent such as the runtime is termed **managed code**, and the process of executing under control of the runtime is called **managed execution**. The code is "managed" because the runtime controls significant portions of the program's behavior by managing aspects such as memory allocation, security, and just-in-time compilation. Code that does not require the runtime to execute is called **native code** (or **unmanaged code**).

The specification for a runtime is included in a broader specification known as the **Common Language Infrastructure (CLI)** specification.[11] An international standard, the CLI includes specifications for the following:

- The VES or runtime
- The CIL
- A type system that supports language interoperability, known as the **Common Type System** (**CTS**)

10. A third term for CIL is Microsoft IL (MSIL). This book uses the term CIL because it is the term adopted by the CLI standard. IL is prevalent in conversation among people writing C# code because they assume that IL refers to CIL rather than other types of intermediate languages.

11. Miller, J., and S. Ragsdale. 2004. *The Common Language Infrastructure Annotated Standard*. Boston: Addison-Wesley.

- Guidance on how to write libraries that are accessible from CLI-compatible languages (available in the **Common Language Specification [CLS]**)
- Metadata that enables many of the services identified by the CLI (including specifications for the layout or file format of assemblies)

NOTE

The term *runtime* can refer to either execution time or the VES. To help clarify the intended meaning, this book uses the term *execution time* to indicate when the program is executing, and it uses the term *runtime* when discussing the agent responsible for managing the execution of a C# program while it executes.

Running within the context of a runtime execution engine enables support for several services and features that programmers do not need to code for directly, including the following:

- *Language interoperability:* Interoperability between different source languages. This is possible because the language compilers translate each source language to the same intermediate language (CIL).
- *Type safety:* Checks for conversion between types, ensuring that only conversions between compatible types will occur. This helps prevent the occurrence of buffer overruns, a leading cause of security vulnerabilities.
- *Code access security:* Certification that the assembly developer's code has permission to execute on the computer.
- *Garbage collection:* Memory management that automatically de-allocates memory previously allocated by the runtime.
- *Platform portability:* Support for potentially running the same assembly on a variety of operating systems. One obvious restriction is that no platform-dependent libraries are used; therefore, platform-dependent idiosyncrasies need to be worked out separately.
- *Base Class Library (BCL):* Provides a foundation of code that developers can depend on (in all .NET frameworks) so that they do not have to develop the code themselves.

NOTE

This section gives a brief synopsis of the CLI to familiarize you with the context in which a C# program executes. It also provides a summary of some of the terms that appear throughout this book. Chapter 22 is devoted to the topic of the CLI and its relevance to C# developers. Although the chapter appears last in the book, it does not depend on any earlier chapters, so if you are eager to become more familiar with the CLI, you can jump to it at any time.

Common Intermediate Language and ILDASM

As mentioned in the introduction of this section, the C# compiler converts C# code to CIL code and not to machine code. The processor can directly understand machine code; therefore, CIL code needs to be converted before the processor can execute it. Given an assembly, it is possible to view the CIL code using a CIL disassembler utility to deconstruct the assembly into its CIL representation. (The CIL disassembler is affectionately referred to by its Microsoft .NET Framework specific filename, ILDASM, which stands for IL Disassembler.) ILDASM will disassemble an assembly and extract the CIL generated by the C# compiler into text.

The output that results from disassembling a .NET assembly is significantly easier to understand than machine code. For many developers, this may raise a concern because it is easier for programs to be decompiled and algorithms understood without explicitly redistributing the source code. As with any program, CLI based or not, the only foolproof way of preventing disassembly is to disallow access to the compiled program altogether (e.g., hosting a program only on a website instead of distributing it out to a user's machine). However, if decreased accessibility to the source code is all that is required, there are several obfuscators available. Obfuscators open up the IL code and transform it so that it does the same thing but in a way that is much more difficult to understand. This technique prevents the casual developer from accessing the code and creates assemblies that are much more difficult and tedious to decompile into comprehensible code. Unless a program requires a high degree of algorithm security, obfuscators are generally sufficient.

ADVANCED TOPIC

CIL Output for `HelloWorld.exe`

The exact command used for the CIL disassembler depends on which implementation of the CLI is used. For .NET Core, instructions are available at http://itl.tc/ildasm. Listing 1.20 shows the CIL code created from running ILDASM.

LISTING 1.20: Sample CIL Output

```
  .assembly extern System.Runtime
  {
    .publickeytoken = ( B0 3F 5F 7F 11 D5 0A 3A )
    .ver 4:2:0:0
  }

  .assembly extern System.Console
  {
    .publickeytoken = ( B0 3F 5F 7F 11 D5 0A 3A )
    .ver 4:1:0:0
  }

  .assembly 'HelloWorld'
  {
    .custom instance void [System.Runtime]System.Runtime.
↳CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01 00 08
↳00 00 00 00 00 )
    .custom instance void [System.Runtime]System.Runtime.
↳CompilerServices.RuntimeCompatibilityAttribute::.ctor() = ( 01 00 01 00 54
↳02 16 57 72 61 70 4E 6F 6E 45 78 63 65 70 74 69 6F 6E 54 68 72 6F 77 73 01 )
    .custom instance void [System.Runtime]System.Runtime.
↳Versioning.TargetFrameworkAttribute::.ctor(string) = ( 01 00 18 2E 4E 45 54
↳43 6F 72 65 41 70 70 2C 56 65 72 73 69 6F 6E 3D 76 32 2E 30 01 00 54 0E 14
↳46 72 61 6D 65 77 6F 72 6B 44 69 73 70 6C 61 79 4E 61 6D 65 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyCompanyAttribute::.ctor(string) = ( 01 00 0A 48 65 6C 6C
↳6F 57 6F 72 6C 64 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyConfigurationAttribute::.ctor(string) = ( 01 00 05 44 65
↳62 75 67 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyDescriptionAttribute::.ctor(string) = ( 01 00 13 50 61 63
↳6B 61 67 65 20 44 65 73 63 72 69 70 74 69 6F 6E 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyFileVersionAttribute::.ctor(string) = ( 01 00 07 31 2E 30
↳2E 30 2E 30 00 00 )
    .custom instance void [System.Runtime]System.
↳Reflection.AssemblyInformationalVersionAttribute::.ctor(string) = ( 01 00 05
↳31 2E 30 2E 30 00 00 )
```

```
    .custom instance void [System.Runtime]System.
↳ Reflection.AssemblyProductAttribute::.ctor(string) = ( 01 00 0A 48 65 6C 6C
↳ 6F 57 6F 72 6C 64 00 00 )
    .custom instance void [System.Runtime]System.
↳ Reflection.AssemblyTitleAttribute::.ctor(string) = ( 01 00 0A 48 65 6C 6C 6F
↳ 57 6F 72 6C 64 00 00 )
    .hash algorithm 0x00008004
    .ver 1:0:0:0
  }

  .module 'HelloWorld.dll'
  // MVID: {c0fe557b-4474-4563-94e1-95c9ead4e3c9}
  .imagebase 0x00400000
  .file alignment 0x00000200
  .stackreserve 0x00100000
  .subsystem 0x0003  // WindowsCui
  .corflags 0x00000001  // ILOnly

  .class private auto ansi beforefieldinit HelloWorld extends [System.
↳ Runtime]System.Object
  {

    .method private hidebysig static void Main() cil managed
    {
      .entrypoint
      // Code size 13
      .maxstack 8
      IL_0000: nop
      IL_0001: ldstr "Hello. My name is Inigo Montoya."
      IL_0006: call void [System.Console]System.Console::WriteLine(string)
      IL_000b: nop
      IL_000c: ret
  } // End of method System.Void HelloWorld::Main()

    .method public hidebysig specialname rtspecialname instance void .ctor()
↳ cil managed
    {
      // Code size 8
      .maxstack 8
      IL_0000: ldarg.0
      IL_0001: call instance void [System.Runtime]System.Object::.ctor()
      IL_0006: nop
      IL_0007: ret
  } // End of method System.Void HelloWorld::.ctor()
  } // End of class HelloWorld
```

The beginning of the listing is the manifest information. It includes not only the full name of the disassembled module (`HelloWorld.exe`) but also all the modules and assemblies it depends on, along with their version information.

Perhaps the most interesting thing that you can glean from such a listing is how relatively easy it is to follow what the program is doing compared to trying to read and understand machine code (assembler). In the listing, an explicit reference to `System.Console.WriteLine()` appears. There is a lot of peripheral information to the CIL code listing, but if a developer wanted to understand the inner workings of a C# module (or any CLI-based program) without having access to the original source code, it would be relatively easy unless an obfuscator is used. In fact, several free tools are available (such as Red Gate's Reflector, ILSpy, JustDecompile, dotPeek, and CodeReflect) that can decompile from CIL to C# automatically.

Multiple .NET Frameworks

As briefly mentioned earlier in the chapter, there are multiple .NET frameworks. The large number of offerings is driven mainly by the desire to provide .NET implementations across multiple operating systems and, potentially, even different hardware platforms. Table 1.3 shows those that are predominant.

TABLE 1.3: Predominant .NET Framework Implementations

Comment Type	Description
.NET Core	A truly cross-platform and open source .NET framework that supports a highly modularized set of APIs for both the server and command line applications.
Microsoft .NET Framework	The first, largest, and most widely deployed of the .NET frameworks.
Xamarin	A mobile platform implementation of .NET that works with both iOS and Android and enables the development of mobile applications from a single code base while still enabling access to native platform APIs.
Mono	The oldest open source implementation of .NET that formed the foundation upon which Xamarin and Unity were built. Mono has been replaced by .NET Core for new development.
Unity	A cross-platform game engine used to develop video games for game consoles, PCs, mobile devices, and even websites. (The Unity engine is the first public implementation to support projections into the Microsoft Hololens augmented reality realm.)

All the samples in the book will work for both .NET Core and Microsoft .NET Framework, at a minimum, unless they specifically indicate otherwise. However, because .NET Core is where most of the .NET future investment is occurring, the sample source code that accompanies the book (available from https://IntelliTect.com/EssentialCSharp), will be configured to work with .NET Core by default.

NOTE

Throughout the book, *.NET framework* (lowercase) to refers to the framework supported by .NET implementations in general. In contrast, *Microsoft .NET Framework* refers to the specific .NET framework implementation that runs only on Microsoft Windows and was *first* released by Microsoft back in 2001.

Application Programming Interface

All the methods (or more generically, the members) found on a data type such as `System.Console` are what define the `System.Console`'s **application programming interface (API)**. The API defines how a software program interacts with a component. As such, it is found not just with a single data type, but more generically; the combination of all the APIs for a set of data types are said to create an API for the collective set of components. In .NET, for example, all the types (and the members within those types) in an assembly are said to form the assembly's API. Likewise, given a combination of assemblies, such as those found in .NET Core or the Microsoft .NET Framework, the collective group of assemblies form a larger API. Often, this larger group of APIs is referred to as the **framework**—hence the term .NET *framework* in reference to the APIs exposed by all the assemblies included with the Microsoft .NET Framework. Generically, the API comprises the set of interfaces and protocols (or instructions) for programming against a set of components. In fact, with .NET, the protocols themselves are the rules for how .NET assemblies execute.

C# and .NET Versioning

Since the development life cycle of .NET frameworks is different from that of the C# language, the version of the underlying .NET framework and the corresponding version of the C# language end up with different numbers.

This means that if you compile with the C# 5.0 compiler, it will, by default, compile against the Microsoft .NET Framework version 4.6, for example. Table 1.4 is a brief overview of the C# and .NET releases for the Microsoft .NET Framework and .NET Core.

Table 1.4: C# and .NET Versions

Comment Type	Description
C# 1.0 with Microsoft .NET Framework 1.0/1.1 (Visual Studio 2002 and 2003)	The initial release of C#. A language built from the ground up to support .NET programming.
C# 2.0 with Microsoft .NET Framework 2.0 (Visual Studio 2005)	Added generics to the C# language and libraries that supported generics to the Microsoft .NET Framework 2.0.
Microsoft .NET Framework 3.0	An additional set of APIs for distributed communications (Windows Communication Foundation [WCF]), rich client presentation (Windows Presentation Foundation [WPF]), workflow (Windows Workflow [WF]), and Web authentication (Cardspaces).
C# 3.0 with Microsoft .NET Framework 3.5 (Visual Studio 2008)	Added support for LINQ, a significant improvement to the APIs used for programming collections. The Microsoft .NET Framework 3.5 provided libraries that extended existing APIs to make LINQ possible.
C# 4.0 with Microsoft .NET Framework 4 (Visual Studio 2010)	Added support for dynamic typing along with significant improvements in the API for writing multithreaded programs that capitalized on multiple processors and cores within those processors.
C# 5.0 with Microsoft .NET Framework 4.5 (Visual Studio 2012) and WinRT integration	Added support for asynchronous method invocation without the explicit registration of a delegate callback. An additional change in the framework was support for interoperability with the Windows Runtime (WinRT).
C# 6.0 with Microsoft .NET Framework 4.6 and .NET Core 1.X (Visual Studio 2015)	Added string interpolation, null propagating member access, exception filters, dictionary initializers, and numerous other features.
C# 7.0 with Microsoft .NET Framework 4.7 and .NET Core 1.1 or 2.0 (Visual Studio 2017)	Added tuples, deconstructors, pattern matching, local functions, return by reference, and more.

Perhaps the most important framework feature added alongside C# 6.0 was support for cross-platform compilation. In other words, not only would the Microsoft .NET Framework run on Windows, but Microsoft also provided the .NET Core implementation that would run on Linux and macOS. Although the .NET Core is not an equivalent feature set to the full Microsoft .NET Framework, it includes enough functionality that entire (ASP.NET) websites can be hosted on operating systems other than Windows and its Internet Information Server (IIS). This means that with the same code base it is possible to compile and execute applications that run across multiple platforms. .NET Core is an entire SDK with everything from the .NET Compiler Platform ("Roslyn"), which itself executes on Linux and macOS, to the .NET Core runtime, along with tools such as the Dotnet command-line utility, `dotnet.exe` (which was introduced around the time of C# 7.0).

End 6.0

End 7.0

.NET Standard

With so many different implementations of .NET, including multiple versions of each individual .NET framework, the frameworks became furcated as each implementation supported a different set of only somewhat overlapping APIs. As a result, writing code that was reusable across multiple .NET frameworks became difficult because it was necessary to litter the code with conditional checks as to whether a particular API was available. To reduce this complexity, the .NET Standard emerged as a means of defining what APIs were supported by which version of the standard. The **.NET Standards**, therefore define what a .NET framework needs to support in order to be compliant with each .NET Standard version number. However, since many of the implementations were already released, the decision tree of identifying which API went into which standard was to some extent based on the existing implementations and the association of those existing implementations with a .NET Standard version number.

The latest release at the time of this writing is .NET Standard 2.0. The advantage of this particular version is that all the base frameworks have implemented (or are working toward implementing) this standard so that, in fact, .NET Standard 2.0 represents a reunification of the furcated APIs found in older versions of each framework.

SUMMARY

This chapter served as a rudimentary introduction to C#. It provided a means of familiarizing you with basic C# syntax. Because of C#'s similarity to C++-style languages, much of this chapter's content might not have been new material to you. However, C# and managed code do have some distinct characteristics, such as compilation down to CIL. Although it is not unique, another key characteristic of C# is its full support for object-oriented programming. Even tasks such as reading and writing data to the console are object oriented. Object orientation is foundational to C#, as you will see throughout this book.

The next chapter examines the fundamental data types that are part of the C# language and discusses how you can use these data types with operands to form expressions.