

DOT NET PROGRAMMING

CHAPTER -1 :Introduction to C# & .NET platform and Building C# Applications

SESSION -1: Introduction to C# and .NET platform

C# is a general-purpose, type-safe, object-oriented programming language. The goal of the language is programmer productivity. To this end, C# balances simplicity, expressiveness, and performance. The chief architect of the language since its first version is Anders Hejlsberg (creator of Turbo Pascal and architect of Delphi). The C# language is platform-neutral and works with a range of platform-specific compilers and frameworks, most notably the Microsoft .NET Framework for Windows.

Applications of C#

C# is used to develop a wide range of applications. Following are some of the applications developed using C# :

- **Windows Client Applications** → It is used to build Windows client applications using Windows Forms and WPF, which are application templates provided by Microsoft Visual Studio.
- **Web Applications** → It is used to build modern web applications using ASP.NET combined with JavaScript and other libraries and APIs. ASP.NET is one of the most widely used technology for building web applications and is supported by templates developed by Microsoft Visual Studio.
- **Console Applications** → It is used to build applications that run in a command-line interface.
- **Azure Cloud Applications** → It is used to build cloud-based applications for Windows Azure, which is an operating system of Microsoft for cloud computing and hosting.
- **iOS and Android Mobile Apps** → It is used for cross-platform native mobile app development via Xamarin, which can be used to create platform-specific UI code layer.
- **Game Development** → It is a quite popular language to build games because it is fast and has much control over memory management. It has a rich library for designing graphics. C# is used in many game engines like Unity Engine and Unreal.
- **Internet of Things Devices** → It is used for building IoT devices because it doesn't need much processing power.
- **Artificial Intelligence** → C# has extensive libraries for basic deep learning and embedded systems. As a result, it is used in the fields of Computer Vision and Artificial Intelligence.

Object Orientation

C# is a rich implementation of the object-orientation paradigm, which includes *encapsulation*, *inheritance*, and *polymorphism*. Encapsulation means creating a boundary around an *object*, to separate its external (public) behavior from its internal (private) implementation details. The distinctive features of C# from an object-oriented perspective are:

Unified type system

The fundamental building block in C# is an encapsulated unit of data and functions called a *type*. C# has a *unified type system*, where all types ultimately share a common base type. This means that all types, whether they represent business objects or are primitive types such as numbers, share the same basic functionality. For example, an instance of any type can be converted to a string by calling its `Tostring` method.

Classes and interfaces

In a traditional object-oriented paradigm, the only kind of type is a class. In C#, there are several other kinds of types, one of which is an *interface*. An interface is like a class, except that it only *describes* members. The implementation for those members comes from types that *implement* the interface. Interfaces are particularly useful in scenarios where multiple inheritance is required (unlike languages such as C++ and Eiffel, C# does not support multiple inheritance of classes).

Properties, methods, and events

In the pure object-oriented paradigm, all functions are *methods* (this is the case in Smalltalk). In C#, methods are only one kind of *function member*, which also includes *properties* and *events* (there are others, too). Properties are function members that encapsulate a piece of an object's state, such as a button's color or a label's text. Events are function members that simplify acting on object state changes.

While C# is primarily an object-oriented language, it also borrows from the *functional programming* paradigm. Specifically:

Functions can be treated as values

Through the use of *delegates*, C# allows functions to be passed as values to and from other functions.

C# supports patterns for purity

Core to functional programming is avoiding the use of variables whose values change, in favor of declarative patterns. C# has key features to help with those patterns, including the ability to write unnamed functions on the fly that “capture” variables (*lambda expressions*), and the ability to perform list or reactive programming via *query expressions*. C# also makes it easy to define read-only fields and properties for writing *immutable* (read-only) types.

Type Safety

C# is primarily a *type-safe* language, meaning that instances of types can interact only through protocols they define, thereby ensuring each type's internal consistency. For instance, C# prevents you from interacting with a *string* type as though it were an *integer* type.

More specifically, C# supports *static typing*, meaning that the language enforces type safety at *compile time*. This is in addition to type safety being enforced at *runtime*.

Static typing eliminates a large class of errors before a program is even run. It shifts the burden away from runtime unit tests onto the compiler to verify that all the types in a program fit together correctly. This makes large programs much easier to manage, more predictable, and more robust. Furthermore, static typing allows tools such as IntelliSense in Visual Studio to help you write a program, since it knows for a given variable what type it is, and hence what methods you can call on that variable.

Note

C# also allows parts of your code to be dynamically typed via the `dynamic` keyword. However, C# remains a predominantly statically typed language.

C# is also called a *strongly typed language* because its type rules (whether enforced statically or at runtime) are very strict. For instance, you cannot call a function that's designed to accept an integer with a floating-point number, unless you first *explicitly* convert the floating-point number to an integer. This helps prevent mistakes.

Strong typing also plays a role in enabling C# code to run in a *sandbox*—an environment where every aspect of security is controlled by the host. In a sandbox, it is important that you cannot arbitrarily corrupt the state of an object by bypassing its type rules.

Memory Management

C# relies on the runtime to perform automatic memory management. The Common Language Runtime has a garbage collector that executes as part of your program, reclaiming memory for objects that are no longer referenced. This frees programmers from explicitly deallocating the memory for an object, eliminating the problem of incorrect pointers encountered in languages such as C++.

C# does not eliminate pointers: it merely makes them unnecessary for most programming tasks. For performance-critical hotspots and interoperability, pointers and explicit memory allocation is permitted in blocks that are marked `unsafe`.

Platform Support

Historically, C# was used almost entirely for writing code to run on Windows platforms. Recently, however, Microsoft and other companies have invested in other platforms, including Linux, macOS, iOS, and Android. Xamarin™ allows cross-platform C# development for mobile applications, and Portable Class Libraries are becoming increasingly widespread. Microsoft's ASP.NET Core is a cross-platform lightweight web hosting framework that can run either on the .NET Framework or on .NET Core, an open source cross-platform runtime.

C# and the CLR

C# depends on a runtime equipped with a host of features such as automatic memory management and exception handling. At the core of the Microsoft .NET Framework is the *Common Language Runtime* (CLR), which provides these runtime features. (The .NET Core and Xamarin frameworks provide similar runtimes.) The CLR is language-neutral, allowing developers to build applications in multiple languages (e.g., C#, F#, Visual Basic .NET, and Managed C++).

C# is one of several *managed languages* that get compiled into managed code. Managed code is represented in *Intermediate Language* or *IL*. The CLR converts the IL into the native code of the machine, such as X86 or X64, usually just prior to execution. This is referred to as Just-In-Time (JIT) compilation. Ahead-of-time compilation is also available to improve startup time with large assemblies or resource-constrained devices (and to satisfy iOS app store rules when developing with Xamarin).

The container for managed code is called an *assembly* or *portable executable*. An assembly can be an executable file (*.exe*) or a library (*.dll*), and contains not only IL, but type information (*metadata*). The presence of metadata allows assemblies to reference types in other assemblies without needing additional files.

Note

You can examine and disassemble the contents of an IL assembly with Microsoft's *ildasm* tool. And with tools such as ILSpy, dotPeek (JetBrains), or Reflector (Red Gate), you can go further and decompile the IL to C#. Because IL is higher-level than native machine code, the decompiler can do quite a good job of reconstructing the original C#.

A program can query its own metadata (*reflection*), and even generate new IL at runtime (*reflection.emit*).

The CLR and .NET Framework

The .NET Framework consists of the CLR plus a vast set of libraries. The libraries consist of core libraries (which this book is concerned with) and applied libraries, which depend on the core libraries. Figure 1-1 is a visual overview of those libraries (and also serves as a navigational aid to the book).

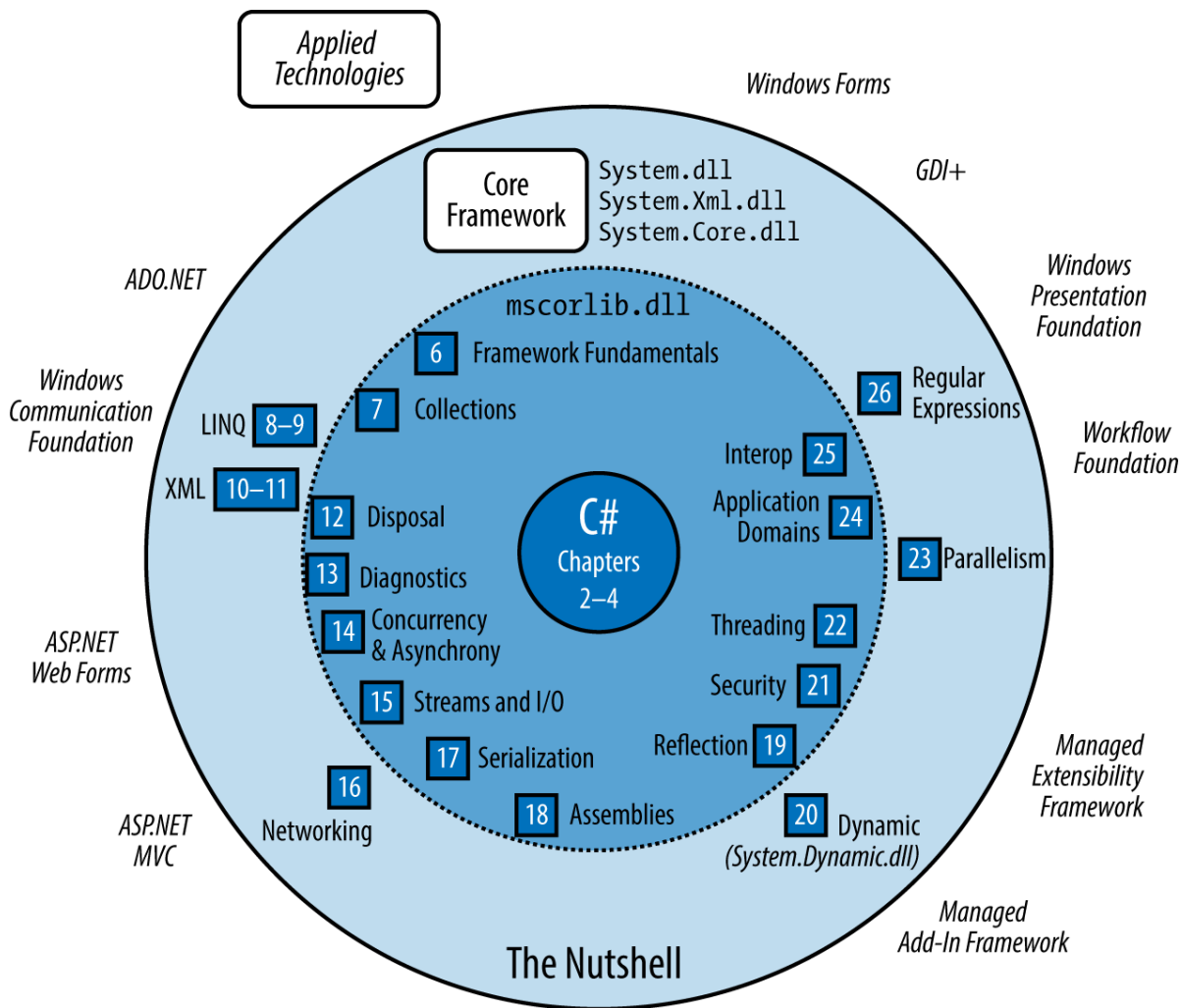


Figure 1-1. Topics covered in this book and the chapters in which they are found. Topics not covered are shown outside the large circle.

Note

The core libraries are sometimes collectively called the *Base Class Library* (BCL). The entire framework is called the *Framework Class Library* (FCL).

Other Frameworks

The Microsoft .NET Framework is the most expansive and mature framework, but runs only on Microsoft Windows (desktop/server). Over the years, other frameworks have emerged to support other platforms. There are currently three major players besides the .NET Framework, all of which are currently owned by Microsoft:

Universal Windows Platform (UWP)

For writing Windows 10 Store Apps and for targeting Windows 10 devices (mobile, XBox, Surface Hub, Hololens). Your app runs in a sandbox to lessen the threat of malware, prohibiting operations such as reading or writing arbitrary files.

.NET Core with ASP.NET Core

An open source framework (originally based on a cut-down version of the .NET Framework) for writing easily deployable Internet apps and micro-services that run on Windows, macOS, and Linux. Unlike the .NET Framework, .NET Core can be packaged with the web application and xcopy-deployed (self-contained deployment).

Xamarin

For writing mobile apps that target iOS, Android, and Windows Mobile. The Xamarin company was purchased by Microsoft in 2016.

Table 1-1 compares the current platform support for each of the major frameworks.

Table 1-1. Platform support for the popular frameworks

A nuance not shown in Table 1-1 is that UWP uses .NET Core under the covers, so technically .NET Core does run on Windows 10 devices (albeit not for the purpose of providing a framework for ASP.NET Core). It's likely that we'll see more uses for .NET Core 2 in the future.

Legacy and Niche Frameworks

The following frameworks are still available to support older platforms:

- Windows Runtime for Windows 8/8.1 (now UWP)
- Windows Phone 7/8 (now UWP)
- Microsoft XNA for game development (now UWP)
- Silverlight (no longer actively developed since the rise of HTML5 and JavaScript)
- .NET Core 1.x (the predecessor to .NET Core 2.0, with significantly reduced functionality)

There are also a couple of niche frameworks worth mentioning:

- The .NET Micro Framework is for running .NET code on highly resource-constrained embedded devices (under 1 MB).
- Mono, the open source framework upon which Xamarin sits, also has libraries to develop cross-platform desktop applications on Linux, macOS, and Windows. Not all features are supported, or work fully.

It's also possible to run managed code inside SQL Server. With SQL Server CLR integration, you can write custom functions, stored procedures, and aggregations in C# and then call them from SQL. This works in conjunction with the standard .NET Framework, but with a special "hosted" CLR that enforces a sandbox to protect the integrity of the SQL Server process.

Windows Runtime

C# also interoperates with *Windows Runtime* (WinRT) technology. WinRT means two things:

- A language-neutral object-oriented execution interface supported in Windows 8 and above

- A set of libraries baked into Windows 8 and above that support the preceding interface

By *execution interface*, we mean a protocol for calling code that's (potentially) written in another language. Microsoft Windows has historically provided a primitive execution interface in the form of low-level C-style function calls comprising the Win32 API.

WinRT is much richer. In part, it is an enhanced version of COM (Component Object Model) that supports .NET, C++, and JavaScript. Unlike Win32, it's object-oriented and has a relatively rich type system. This means that referencing a WinRT library from C# feels much like referencing a .NET library—you may not even be aware that you're using WinRT.

The WinRT libraries in Windows 10 form an essential part of the UWP platform (UWP relies on both WinRT and .NET Core libraries). If you're targeting the standard .NET Framework platform, referencing the Windows 10 WinRT libraries is optional, and can be useful if you need to access Windows 10-specific features not otherwise covered in the .NET Framework.

What distinguishes WinRT from ordinary COM is that WinRT *projects* its libraries into a multitude of languages, namely C#, VB, C++, and JavaScript, so that each language sees WinRT types (almost) as though they were written especially for it. For example, WinRT will adapt capitalization rules to suit the standards of the target language, and will even remap some functions and interfaces. WinRT assemblies also ship with rich *metadata* in *.winmd* files, which have the same format as .NET assembly files, allowing transparent consumption without special ritual; this is why you might be unaware that you're using WinRT rather than .NET types, aside from namespace differences. Another clue is that WinRT types are subject to COM-style restrictions; for instance, they offer limited support for inheritance and generics.

In C#, you can not only consume WinRT libraries, but also write your own (and call them from a JavaScript application).

The .NET solution,

1.Full interoperability with existing code:This is (of course) a good thing. Existing COM binaries can commingle (i.e., interop) with newer .NET binaries and vice versa. Also, Platform Invocation Services (PInvoke) allows you to call C-based libraries from .NET code

2.Complete and total language integration:Unlike COM, .NET supports cross-language inheritance, cross-language exception handling, and cross-language debugging.

3.A common runtime engine shared by all .NET-aware languages:One aspect of this engine is a well-defined set of types that each .NET-aware language “understands.”

4.A base class library:This library provides shelter from the complexities of raw API calls and offers a consistent object model used by all .NET-aware languages.

5.No more COM plumbing:IClassFactory, IUnknown, IDispatch, IDL code, and the evil VARIANTcompliant data types (BSTR,SAFEARRAY, and so forth) have no place in a native .NET binary.

6.A truly simplified deployment model:Under .NET, there is no need to register a binary unit into the system registry. Furthermore, .NET allows multiple versions of the same *.dllto exist in harmony on a single machine

.NET Framework

The .NET Framework is a new and revolutionary platform created by Microsoft for developing applications

- It is a platform for application developers
- It is a Framework that supports Multiple Language and Cross languageintegration.
- IT has IDE (Integrated Development Environment).
- Framework is a set of utilities or can say building blocks of your application system.
- .NET Framework provides GUI in a GUI manner.
- .NET is a platform independent but with help of Mono Compilation System (MCS). MCS is a middle level interface.
- .NET Framework provides interoperability between languages i.e. Common Type System (CTS) .
- .NET Framework also includes the .NET Common Language Runtime (CLR), which is responsible for maintaining the execution of all applications developed using the .NET library.
- The .NET Framework consists primarily of a gigantic library of code.

Definition: A programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies, such as desktop applications and Web services.

Cross Language integration

You can use a utility of a language in another language (It uses Class Language Integration).

.NET Framework includes no restriction on the type of applications that are possible. The .NET Framework allows the creation of Windows applications, Web applications, Web services, and lot more.

The .NET Framework has been designed so that it can be used from any language, including C#, C+, Visual Basic, JScript, and even older languages such as COBOL.

Difference between Visual Studio and Visual Studio .NET

Visual Studio	Visual Studio .Net
It is object based	It is object oriented
Internet based application - Web Application - Web services - Internet enable application - Third party API	All developing facilities in internet based application

- Peer to peer Application	
Poor error handling Exception/Error	Advance error handler and debugger
Memory Management System Level Task	Memory Management Application Domain with help of GC (Garbage Collector)
DLL HELL	VS .NET has solved DLL HELL Problem

Simple explanation of definition used in the above comparison:

Web Application

All websites are example of web application. They use a web server.

Internet Enabled Application

They are desktop application. Yahoo messenger is an example of desktop application.

Peer to Peer

Communication through computers through some system.

Web Services

It doesn't use web-based server. Internet payment systems are example of web services.

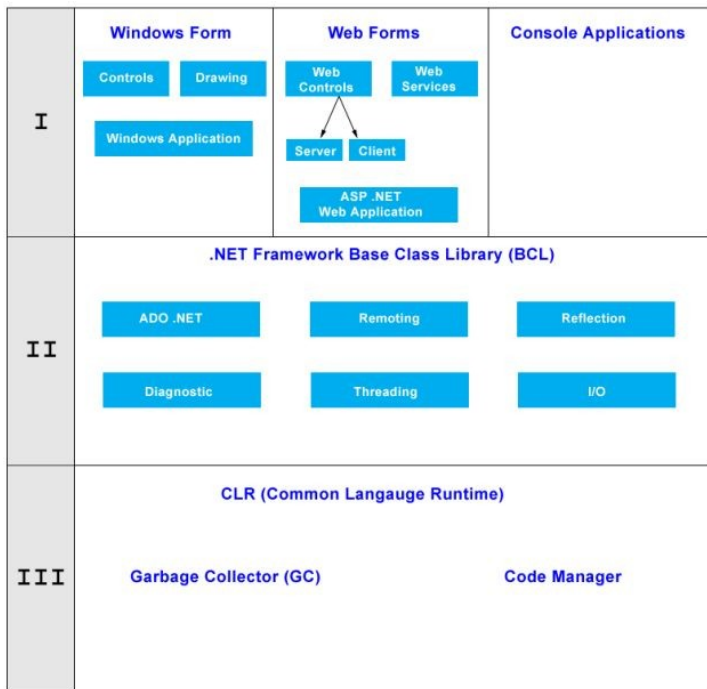
DLL Hell

"**DLL Hell**" refers to the set of problems caused when multiple applications attempt to share a common component like a dynamic link library (DLL) or a Component Object Model (COM) class.

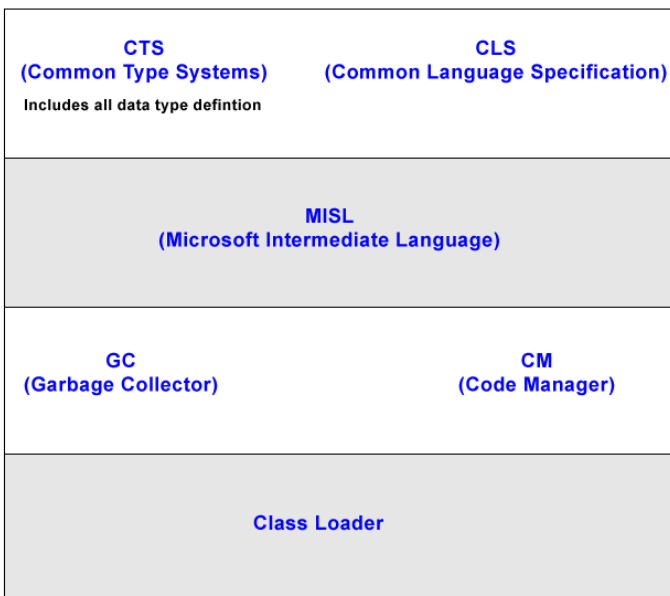
The reason for this issue was that the version information about the different components of an application was not recorded by the system. (Windows Registry cannot support the multiple versions of same COM component this is called the dll hell problem.)

.Net Framework provides operating systems with a Global Assembly Cache (GAC). This Cache is a repository for all the .Net components that are shared globally on a particular machine. When a .Net component is installed onto the machine, the Global Assembly Cache looks at its version, its public key, and its language information and creates a strong name for the component. The component is then registered in the repository and indexed by its strong name, so there is no confusion between different versions of the same component, or DLL.

Architecture of .NET Framework



Architecture of CLR



CLS (Common Language Specification)

It is a subset of CTS. All instruction is in CLS i.e. instruction of CTS is written in CLS.

Code Manager

Code manager invokes class loader for execution.

.NET supports two kind of coding

1. Managed Code

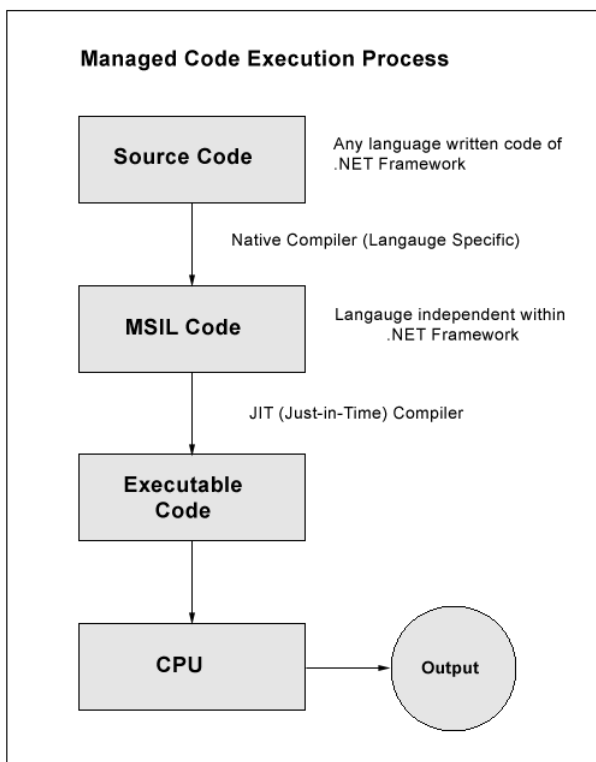
2. Unmanaged Code

Managed Code

The resource, which is within your application domain is, managed code. The resources that are within domain are faster.

The code, which is developed in .NET framework, is known as managed code. This code is directly executed by CLR with help of managed code execution. Any language that is written in .NET Framework is managed code.

Managed code uses CLR which in turn looks after your applications by managing memory, handling security, allowing cross - language debugging, and so on.



Unmanaged Code

The code, which is developed outside .NET, Framework is known as unmanaged code.

Applications that do not run under the control of the CLR are said to be unmanaged, and certain languages such as C++ can be used to write such applications, which, for example, access low - level functions of the operating system. Background compatibility with code of VB, ASP and COM are examples of unmanaged code.

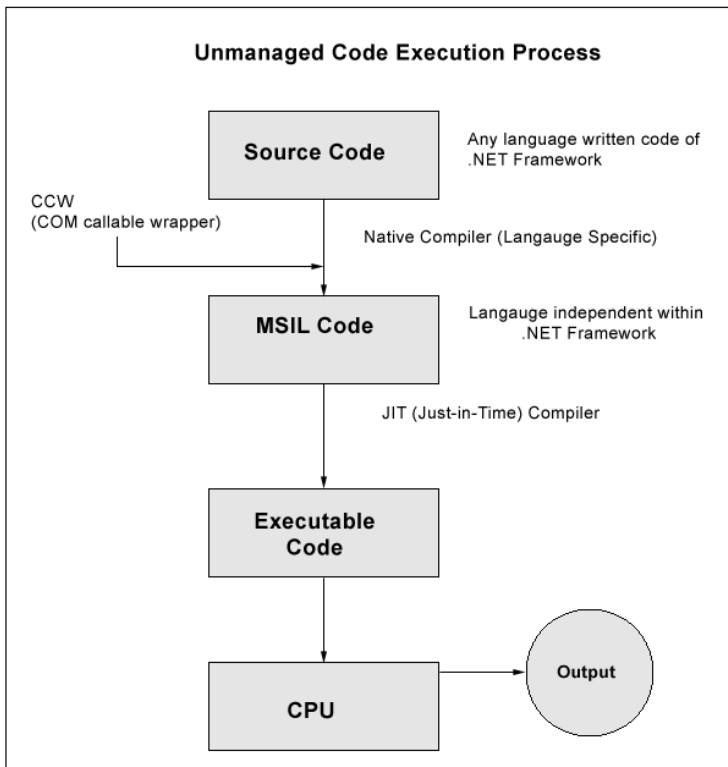
Unmanaged code can be unmanaged source code and unmanaged compile code.

Unmanaged code is executed with help of wrapper classes.

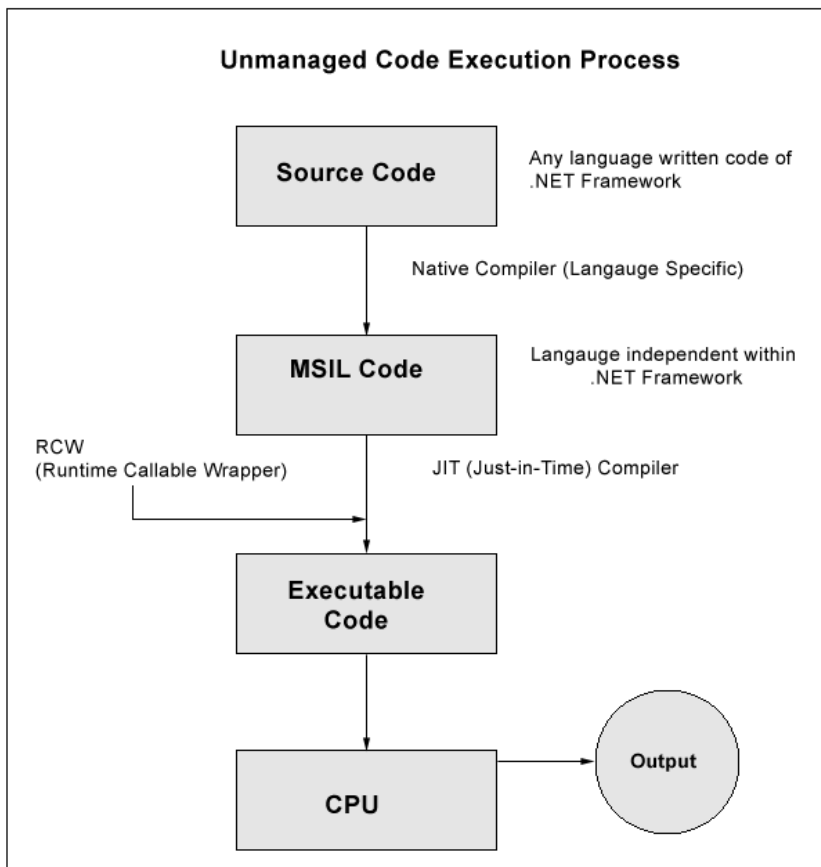
Wrapper classes are of two types: CCW (COM callable wrapper) and RCW (Runtime Callable Wrapper).

Wrapper is used to cover difference with the help of CCW and RCW.

COM callable wrapper unmanaged code



Runtime Callable Wrapper unmanaged code



Native Code

The code to be executed must be converted into a language that the target operating system understands, known as native code. This conversion is called **compiling** code, an act that is performed by a compiler.

Under the .NET Framework, however, this is a two - stage process. With help of MSIL and JIT.

MSIL (Microsoft Intermediate Language)

It is language independent code. When you compile code that uses the .NET Framework library, you don't immediately create operating system - specific native code.

Instead, you compile your code into Microsoft Intermediate Language (MSIL) code. The MSIL code is not specific to any operating system or to any language.

JIT (Just-in-Time)

Just - in - Time (JIT) compiler, which compiles MSIL into native code that is specific to the OS and machine architecture being targeted. Only at this point can the OS execute the application. The just - in - time part of the name reflects the fact that MSIL code is only compiled as, and when, it is needed.

In the past, it was often necessary to compile your code into several applications, each of which targeted a specific operating system and CPU architecture. Often, this was a form of optimization.

This is now unnecessary, because JIT compilers (as their name suggests) use MSIL code, which is independent of the machine, operating system, and CPU. Several JIT compilers exist, each targeting a different architecture, and the appropriate one will be used to create the native code required.

The beauty of all this is that it requires a lot less work on your part - in fact, you can forget about system - dependent details and concentrate on the more interesting functionality of your code.

JIT are of three types:

1. Pre JIT
2. Econo JIT
3. Normal JIT

Pre JIT

It converts all the code in executable code and it is slow

Econo JIT

It will convert the called executable code only. But it will convert code every time when a code is called again.

Normal JIT

It will only convert the called code and will store in cache so that it will not require converting code again. Normal JIT is fast.

Assemblies

When you compile an application, the MSIL code created is stored in an assembly. Assemblies include both executable application files that you can run directly from Windows without the need

for any other programs (these have a .exe file extension), and libraries (which have a .dll extension) for use by other applications.

In addition to containing MSIL, assemblies also include meta information (that is, information about the information contained in the assembly, also known as metadata) and optional resources (additional data used by the MSIL, such as sound files and pictures).

The meta information enables assemblies to be fully self - descriptive. You need no other information to use an assembly, meaning you avoid situations such as failing to add required data to the system registry and so on, which was often a problem when developing with other platforms.

This means that deploying applications is often as simple as copying the files into a directory on a remote computer. Because no additional information is required on the target systems, you can just run an executable file from this directory and (assuming the .NET CLR is installed) you're good to go.

Of course, you won't necessarily want to include everything required to run an application in one place. You might write some code that performs tasks required by multiple applications. In situations like that, it is often useful to place the reusable code in a place accessible to all applications. In the .NET Framework, this is the Global Assembly Cache (GAC). Placing code in the GAC is simple - you just place the assembly containing the code in the directory containing this cache.

Base Class Libraries

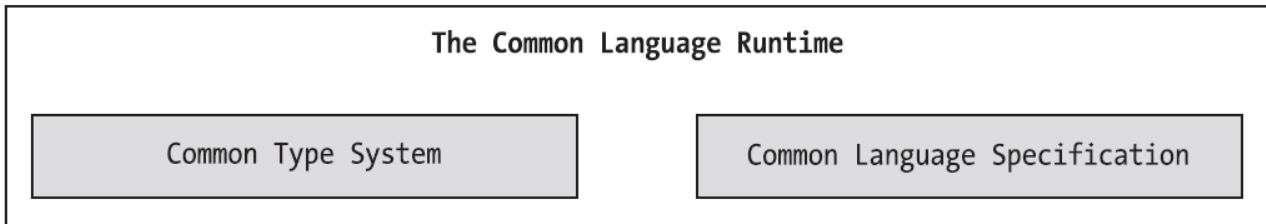
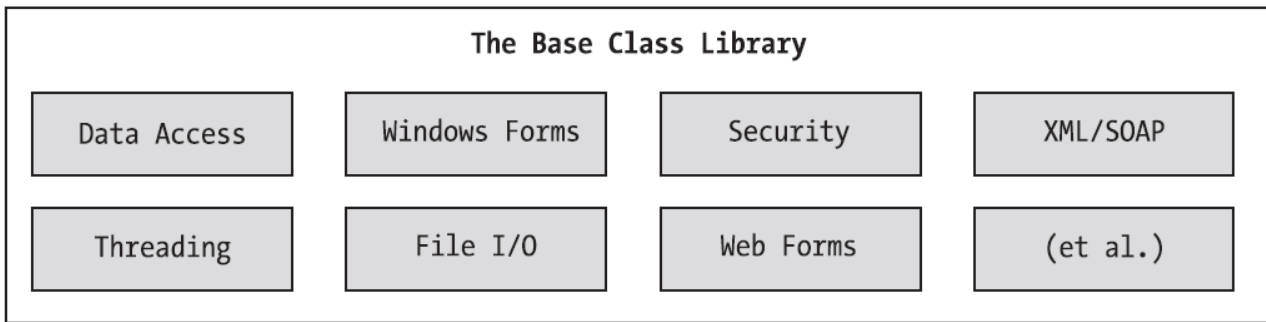
The BCL provides the most foundational types and utility functionality and are the base of all other .NET class libraries. They aim to provide very general implementations without any bias to any workload. Performance is always an important consideration, since apps might prefer a particular policy, such as low-latency to high-throughput or low-memory to low-CPU usage. These libraries are intended to be high-performance generally, and take a middle-ground approach according to these various performance concerns. For most apps, this approach has been quite successful.

The .NET platform provides a base class library that is available to all .NET programming languages.

Base class library encapsulate various primitives such as threads, fileinput/output (I/O), graphical rendering, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications.

For example, the base class libraries define types that facilitate database access,XML manipulation, programmatic security, and the construction of web-enabled front ends.

The relationship between the CLR, CTS, CLS, and the base class library, as shown in Figure.



The CLR, CTS, CLS, and base class library relationship

The .NET Framework class library is a library of classes, interfaces, and valuetypes that are included in the Windows Software Development Kit (SDK). This library provides access to system functionality and is designed to be the foundation on which .NET Framework applications, components, and controls are built.

System :-

The System namespace contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.

System.Data :-

The System.Data namespaces contain classes for accessing and managing data from diverse sources. The top-level namespace and a number of the child namespaces together form the ADO.NET architecture and ADO.NET data providers. For example, providers are available for SQL Server, Oracle, ODBC, and OleDb. Other child namespaces contain classes used by the ADO.NET Entity Data Model (EDM) and by WCF Data Services.

System.IO :-

The System.IO namespaces contain types that support input and output, including the ability to read and write data to streams either synchronously or asynchronously, to compress data in streams, to create and use isolated stores, to map files to an application's logical address space, to store multiple

data objects in a single container, to communicate using anonymous or named pipes, to implement custom logging, and to handle the flow of data to and from serial ports.

System.Net :-

The System.Net namespaces contain classes that provide a simple programming interface for a number of network protocols, programmatically access and update configuration settings for the System.Net namespaces, define cache policies for web resources, compose and send e-mail, represent Multipurpose Internet Mail Exchange (MIME) headers, access network traffic data and network address information, and access peer-to-peer networking functionality. Additional child namespaces provide a managed implementation of the Windows Sockets (Winsock) interface and provide access to network streams for secure communications between hosts.

System.Web :-

The System.Web namespaces contain types that enable browser/server communication. Child namespaces include types that support ASP.NET forms authentication, application services, data caching on the server, ASP.NET application configuration, dynamic data, HTTP handlers, JSON serialization, incorporating AJAX functionality into ASP.NET, ASP.NET security, and webservice.

System.Windows :-

The System.Windows namespaces contain types used in Windows Presentation Foundation (WPF) applications, including animation clients, user interface controls, data binding, and type conversion. System.Windows.Forms and its child namespaces are used for developing Windows Forms applications.

Primitive Types

.NET includes a set of primitive types, which are used (to varying degrees) in all programs. These types contain data, such as numbers, strings, bytes and arbitrary objects. The C# language includes keywords for these types. A sample set of these types is listed below, with the matching C# keywords.

- System.Object (object) - The ultimate base class in the CLR type system. It is the root of the type hierarchy.
- System.Int16 (short) - A 16-bit signed integer type. The unsigned UInt16 also exists.
- System.Int32 (int) - A 32-bit signed integer type. The unsigned UInt32 also exists.
- System.Single (float) - A 32-bit floating-point type.
- System.Decimal (decimal) - A 128-bit decimal type.
- System.Byte (byte) - An unsigned 8-bit integer that represents a byte of memory.
- System.Boolean (bool) - A Boolean type that represents true or false.
- System.Char (char) - A 16-bit numeric type that represents a Unicode character.
- System.String (string) - Represents a series of characters. Different than a char[], but enables indexing into each individual char in the string.

.NET APIs include classes, interfaces, delegates, and value types that expedite and optimize the development process and provide access to system functionality. To facilitate interoperability between languages, most .NET types are CLS-compliant and can therefore be used from any programming language whose compiler conforms to the common language specification (CLS).

.NET types are the foundation on which .NET applications, components, and controls are built. .NET includes types that perform the following functions:

- Represent base data types and exceptions.
- Encapsulate data structures.
- Perform I/O.
- Access information about loaded types.
- Invoke .NET security checks.
- Provide data access, rich client-side GUI, and server-controlled, client-side GUI.

.NET provides a rich set of interfaces, as well as abstract and concrete (non-abstract) classes. You can use the concrete classes as-is or, in many cases, derive your own classes from them. To use the functionality of an interface, you can either create a class that implements the interface or derive a class from one of the .NET classes that implements the interface.

Type of .NET Languages

To help create languages for the .NET Framework, Microsoft created the Common Language Infrastructure specification (CLI). The CLI describes the features that each language must provide in order to use the .NET Framework and common language runtime and to interoperate with components written in other languages. If a language implements the necessary functionality, it is said to be .NET-compliant.

Every .NET-compliant language supports the same data types, uses the same .NET Framework classes, compiles to the same MSIL, and uses a single common language runtime to manage execution. Because of this, every .NET-compliant language is a first-class Microsoft .NET citizen.-Developers are free to choose the best language for a particular component without losing any of the power and freedom of the platform. In addition, components written in one language can easily interoperate with components written in another language. For example, you can write a class in C# that inherits from a base class written in Visual Basic.

The .NET Framework was developed so that it could support a theoretically infinite number of development languages. Currently, more than 20 development languages work with the .NET Framework. C# is the programming language specifically designed for the .NET platform, but C++ and Visual Basic have also been upgraded to fully support the .NET framework. The following are the commonly used languages provided by the Microsoft:

- **VC++**
- **VB.NET**
- **C#**
- **J#**
- **JScript .NET**

Many third parties are writing compilers for other languages with .NET support. With CLR, Microsoft has adopted a much liberal policy. Microsoft has them selves evolved/ developed/ modified many of their programming languages which compliant with .NET CLR.

We'll be covering the following topics in this tutorial:

- [VC++](#)
- [VB.NET](#)
- [C#](#)
- [J#](#)
- [JScript.NET](#)
- [Third-party languages](#)

VC++

Although Visual C++ (VC++) , has undergone changes to incorporate .NET; yet VC++ also maintains its status being a platform dependent programming. Many new MFC classes have been added a programmer can choose between using MFC and compiling the program into a platform specific executable file; or using .NET framework classes and compile into platform independent MISL file. A programmer can also specify (via directives) when ever he uses "unsafe" (the code that by passes CLR, e.g. the use of pointers) code.

VB.NET

Out of ALL .NET languages, Visual Basic.NET (VB.NET) is one language that has probably undergone into the most of changes. Now VB.NET may be considered a

complete Object- Oriented Language (as opposed to its previous “Half Object Based and Half Object Oriented” status).

Visual Basic .NET provides substantial language innovations over previous versions of visual basic. Visual Basic .NET supports inheritance, constructors, polymorphism, constructor overloading, structured exceptions, stricter type checking, free threading, and many other features. There is only one form of assignment: noLet of set methods. New rapid application development (RAD) features, such as XML Designer, Server Explorer, and Web Forms designer, are available in Visual Basic from Visual Studio .NET. With this release, Visual Basic Scripting Edition provides full Visual Basic functionality.

C#

Microsoft has also developed a brand new programming language C# (C Sharp). This language makes full use of .NET. It is a pure object oriented language. A [Java](#) programmer may find most aspects of this language which is identical to [Java](#). If you are a new comer to Microsoft Technologies – this language is the easiest way to get on the .NET band wagon. While VC++ and VB enthusiast would stick to VC.NET and VB.NET; they would probably increase their productivity by switching to C#. C# is developed to make full use of all the intricacies of .NET. The learning curve of C# for a Java programmer is minimal. Microsoft has also come up with a The Microsoft Java Language Conversion Assistant- which is a tool that automatically converts existing Java-language source code into C# for developers who want to move their existing applications to the Microsoft .NET Framework.

J#

Microsoft has also developed J# (Java Sharp). C# is similar to Java, but it is not entirely' identical. It is for this reason that Microsoft has developed J# – the syntax of J# is identical to Visual J++. Microsoft's growing legal battle with Sun, over Visual J++ – forced Microsoft to discontinue Visual J++. So J# is Microsoft's indirect continuation of Visual J++. It has been reported that porting a medium sized Visual J++ project, entirely to J# takes only a few days of effort.

JScript.NET

Jscript.NET is rewritten to be fully .NET aware. It includes support for classes, inheritance, types and compilation, and it provides improved performance and productivity features. JScript.NET is also integrated with visual Studio .NET. You can take advantage of any .NET Framework class in JScript .NET.

Third-party languages

Microsoft encourages third party vendors to make use of Visual Studio. Net. Third, party vendors can write compilers for different languages ~ that compile the language to MSIL

(Microsoft Intermediate Language). These vendors need not develop their own development environment. They can easily use Visual Studio.NET as an IDE for their .NET compliant language. A vendor has already produced COBOL.NET that integrates with Visual Studio.NET and compiles into MSIL. Theoretically it would then be possible to come up with Java compiler that compiles into MSIL, instead of Java Byte code; and uses CLR

instead of JVM. However Microsoft has not pursued this due to possible legal action by Sun.

Several third party languages are supporting the .NET platform. These languages include APL, COBOL, Pascal, Eiffel, Haskell, ML, Oberon, Perl, [Python](#), Scheme and Smalltalk.

Assemblies in .NET

Assemblies form the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations.

In .NET Core and .NET Framework, you can build an assembly from one or more source code files. In .NET Framework, assemblies can contain one or more modules. This allows larger projects to be planned so that several developers can work on separate source code files or modules, which are combined to create a single assembly. For more information about modules, see [How to: Build a multifile assembly](#).

Assemblies have the following properties:

- Assemblies are implemented as .exe or .dll files.
- For libraries that target the .NET Framework, you can share assemblies between applications by putting them in the global assembly cache (GAC). You must strong-name assemblies before you can include them in the GAC.
- Assemblies are only loaded into memory if they are required. If they aren't used, they aren't loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
- it can programmatically obtain information about an assembly by using reflection.

Assemblies in the common language runtime

Assemblies provide the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

An assembly defines the following information:

- Code that the common language runtime executes. Note that each assembly can have only one entry point: `DllMain`, `WinMain`, or `Main`.
- Security boundary. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries in assemblies, see [Assembly security considerations](#).
- Type boundary. Every type's identity includes the name of the assembly in which it resides. A type called `MyType` that is loaded in the scope of one assembly is not the same as a type called `MyType` that is loaded in the scope of another assembly.
- Reference scope boundary. The assembly manifest has metadata that is used for resolving types and satisfying resource requests. The manifest specifies the types and resources to expose outside the assembly, and enumerates other assemblies on which it depends. Microsoft intermediate language (MSIL) code in a portable executable (PE) file won't be executed unless it has an associated assembly manifest.
- Version boundary. The assembly is the smallest versionable unit in the common language runtime. All types and resources in the same assembly are versioned as a unit. The assembly manifest describes the version dependencies you specify for any dependent assemblies. For more information about versioning, see [Assembly versioning](#).
- Deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as assemblies containing localization resources or utility classes, can be retrieved on demand. This allows apps to be simple and thin when first downloaded. For more information about deploying assemblies, see [Deploy applications](#).
- Side-by-side execution unit. For more information about running multiple versions of an assembly, see [Assemblies and side-by-side execution](#).

Assemblies, Manifests, Metadata, and Modules

Different .NET-compliant language compilers produce MSIL (Microsoft Intermediate Language) along with *metadata*, which describes the types in your code. Following are the functions of metadata:

- Describes and references the datatypes defined by the VOS (Virtual Object System) type system
- Lays out instances of classes in memory
- Resolves method invocation
- Solves versioning problems (DLL hell)

The Microsoft Intermediate Language and metadata are packed into an executable format called a *portable executable file* (PE file), which extends PE and the Common Object File Format (COFF). COFF enables the operating system to recognize Common Language Runtime (CLR). One great advantage of packing metadata along with the MSIL is that it enables your code to describe itself, thus eliminating the need for type libraries or Interface Definition Language (IDL).

Assemblies, modules, and manifests are grouping constructs, each playing a different role in the CLR environment.

An *assembly* is more of a logical entity than a physical entity. Hence, it can be used in multiple files. It contains the CLR targeted code compiled by different language compilers. An assembly is a completely self-describing unit, which can be thought of as .dll or .exe. If the Portable executable doesn't contain the associated assembly manifest, it doesn't get executed and moreover there will be one and only one entry point for an assembly. Following are the some of the functions of the assemblies:

- It acts as a boundary for all sort of security, types, reference library, and versioning requests.
- It's the deployment unit. On the initialization of an application, assemblies of an application must be present and others could be called on demand.
- It makes side-by-side deployment possible.

A *manifest* is basically an area of metadata occupied in an assembly that allows checks to be made on the version of the assembly. It also checks for the integrity of an assembly.

A *module* contains the file format of an executable. If the module contains a manifest, it becomes an assembly as well as a module. There will be one and only one manifest in all of the files. The relationship between assembly, module, and files can be visualized from the diagram in [Figure 2](#).

Figure 2 Assembly-Module-File relationship.

Having seen how assembly, module, and files communicate with each other, let's look into the sequence of the declaration followed while writing the IL programs:

```
.assembly  
.assembly extern  
.class  
.class extern  
.corflags  
.custom  
.data  
.field  
.file  
.mresource  
.method  
.module  
.module extern  
.subsystem  
<externSourceDecl>  
<securityDecl>
```

I hope that this overview gives you the initial confidence required to start writing intermediate language programs. The subject is not as simple as it seems. ECMA documents contain complete information about this language. We'll try to probe more into this subject in coming articles.

4.Naming conventions

.NET types use a dot syntax naming scheme that connotes a hierarchy. This technique groups related types into namespaces so they can be searched and referenced more easily. The first part of the full name — up to the rightmost dot — is the namespace name. The last part of the name is the type name. For example, `System.Collections.Generic.List<T>` represents the `List<T>` type, which belongs to the `System.Collections.Generic` namespace. The types in `System.Collections.Generic` can be used to work with generic collections.

This naming scheme makes it easy for library developers extending .NET to create hierarchical groups of types and name them in a consistent, informative manner. It also allows types to be unambiguously identified by their full name (that is, by their namespace and type name), which prevents type name collisions. Library developers are expected to use the following convention when creating names for their namespaces:

CompanyName.TechnologyName

For example, the namespace `Microsoft.Word` conforms to this guideline.

The use of naming patterns to group related types into namespaces is a useful way to build and document class libraries. However, this naming scheme has no effect on visibility, member access, inheritance, security, or binding. A namespace can be partitioned across multiple assemblies and a single assembly can contain types from multiple namespaces. The assembly provides the formal structure for versioning, deployment, security, loading, and visibility in the common language runtime.

For more information on namespaces and type names, see [Common Type System](#).

Namespace

A namespace is a grouping of related types contained in an assembly. Keep all the types within the base class libraries well organized, the .NET platform makes possible by of the namespace concept.

For example,

- o The `System.IO` namespace contains file I/O related types;
- o The `System.Data` namespace defines basic database types

Very important single assembly (such as `mscorlib.dll`) can contain any number of namespaces, each of which can contain any number of types.

The key difference between this approach and a language-specific library such as MFC is that any language targeting the .NET runtime makes use of the same namespaces and same types.

System namespace

The System namespace is the root namespace for fundamental types in .NET. This namespace includes classes that represent the base data types used by all applications: Object (the root of the inheritance hierarchy), Byte, Char, Array, Int32, String, and so on. Many of these types correspond to the primitive data types that your programming language uses. When you write code using .NET types, you can use your language's corresponding keyword when a .NET base data type is expected.

The following table lists the base types that .NET supplies, briefly describes each type, and indicates the corresponding type in Visual Basic, C#, C++, and F#.

System namespace

Category	Class name	Description	Visual Basic data type	C# data type	C++/CLI data type	F# data type
Integer	Byte	An 8-bit unsigned integer.	Byte	byte	unsigned char	byte
	SByte	An 8-bit signed integer. Not CLS-compliant.	SByte	sbyte	char -or- signed char	sbyte
	Int16	A 16-bit signed integer.	Short	short	short	int16
	Int32	A 32-bit signed integer.	Integer	int	-or- long	int
	Int64	A 64-bit signed integer.	Long	long	__int64	int64
	UInt16	A 16-bit unsigned integer. Not CLS-compliant.	UShort	ushort	unsigned short	uint16
	UInt32	A 32-bit unsigned integer. Not CLS-compliant.	UInteger	uint	unsigned int -or- unsigned long	uint32
	UInt64	A 64-bit unsigned integer. Not CLS-compliant.	ULong	ulong	unsigned __int64	uint64
Floating point	Single	A single-precision (32-bit) floating-point number.	Single	float	float	float32 or single
	Double	A double-precision (64-bit) floating-point number.	Double	double	double	float or double
Logical	Boolean	A Boolean value (true or false).	Boolean	bool	bool	bool
Other	Char	A Unicode (16-bit) character.	Char	char	wchar_t	char
	Decimal	A decimal (128-bit) value.	Decimal	decimal	Decimal	decimal

IntPtr	A signed integer whose size depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform).	IntPtr No built-in type.	IntPtr No built-in in type.	IntPtr No built-in type.	unativeint
UIntPtr	An unsigned integer whose size depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform).	UIntPtr No built-in type.	UIntPtr No built-in in type.	UIntPtr No built-in type.	unativeint
Object	Not CLS-compliant. The root of the object hierarchy.	Object	object	Object^	obj
String	An immutable, fixed-length string of Unicode characters.	String	string	String^	string

In addition to the base data types, the System namespace contains over 100 classes, ranging from classes that handle exceptions to classes that deal with core runtime concepts, such as application domains and the garbage collector. The System namespace also contains many second-level namespaces.

For more information about namespaces, use the .NET API Browser to browse the .NET Class Library. The API reference documentation provides documentation on each namespace, its types, and each of their members.

A **namespace** is designed for providing a way to keep one set of names separate from another. The class names declared in one namespace does not conflict with the same class names declared in another.

Defining a Namespace

A namespace definition begins with the keyword **namespace** followed by the namespace name as follows –

```
namespace namespace_name {
    // code declarations
}
```

To call the namespace-enabled version of either function or variable, prepend the namespace name as follows –

```
namespace_name.item_name;
```

The following program demonstrates use of namespaces –

```
using System;
namespace first_space {
    class namespace_cl {
        public void func() {
```

```

        Console.WriteLine("Inside first_space");
    }
}
namespace second_space {
    class namespace_cl {
        public void func() {
            Console.WriteLine("Inside second_space");
        }
    }
}
class TestClass {
    static void Main(string[] args) {
        first_space.namespace_cl fc = new first_space.namespace_cl();
        second_space.namespace_cl sc = new second_space.namespace_cl();
        fc.func();
        sc.func();
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

```

Inside first_space
Inside second_space

```

The *using* Keyword

The **using** keyword states that the program is using the names in the given namespace. For example, we are using the **System** namespace in our programs. The class Console is defined there. We just write –

```

Console.WriteLine ("Hello there");

```

We could have written the fully qualified name as –

```

System.Console.WriteLine("Hello there");

```

You can also avoid prepending of namespaces with the **using** namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code –

Let us rewrite our preceding example, with using directive –

```

using System;
using first_space;
using second_space;

namespace first_space {
    class abc {
        public void func() {
            Console.WriteLine("Inside first_space");
        }
    }
}

```

```

    }
}
namespace second_space {
    class efg {
        public void func() {
            Console.WriteLine("Inside second_space");
        }
    }
}
class TestClass {
    static void Main(string[] args) {
        abc fc = new abc();
        efg sc = new efg();
        fc.func();
        sc.func();
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

```

Inside first_space
Inside second_space

```

Nested Namespaces

You can define one namespace inside another namespace as follows –

```

namespace namespace_name1 {

    // code declarations
    namespace namespace_name2 {
        // code declarations
    }
}

```

You can access members of nested namespace by using the dot (.) operator as follows –

```

using System;
using first_space;
using first_space.second_space;

namespace first_space {
    class abc {
        public void func() {
            Console.WriteLine("Inside first_space");
        }
    }
}
namespace second_space {
    class efg {
        public void func() {

```

```

        Console.WriteLine("Inside second_space");
    }
}
}
}
class TestClass {
    static void Main(string[] args) {
        abc fc = new abc();
        efg sc = new efg();
        fc.func();
        sc.func();
        Console.ReadKey();
    }
}

```

When the above code is compiled and executed, it produces the following result –

```

Inside first_space
Inside second_space

```

Role of the command line compiler(csc.exe)

There are a number of techniques one may use to compile C# source code. In addition to Visual Studio 2005 or 2010 (as well as various third-party .NET IDEs), we are able to create .NET assemblies using the C# command-line compiler, csc.exe (where csc stands for C-Sharp Compiler). This tool is included with the .NET Framework 2.0 SDK. Few reasons one should get a grip on the process:

- The simple fact that one might not have a copy of Visual Studio 2005 or 2010.
- One plan to make use of automated build tools such as MSBuild or NAnt.
- One wants to deepen his understanding of C#. When one uses graphical IDEs to build applications, we are ultimately instructing csc.exe how to manipulate our C# input files. In this light, it's enriching to see what takes place behind the scenes. Another nice by-product of working with csc.exe in the raw is that one become that much more comfortable manipulating other command-line tools included with the .NET Framework 2.0 SDK.

Building a C# application using csc.exe

CSC stands for the *C-sharp compiler*, to illustrate how to build a .NET application IDE-free; we will build a simple executable assembly named TestApp.exe using the C# command-line compiler and Notepad. First, you need some source code.

Open Notepad (using the Start ->Programs ->Accessories menu option) and enter the following trivial C# code:

// A simple C# application.

```
using System;
class TestApp
{
static void Main()
{
Console.WriteLine("Testing! 1, 2, 3");
}
}
```

Once you have finished, save the file in a convenient location (e.g., C:\CscExample) as TestApp.cs. Now, let's get to know the core options of the C# compiler.

Specifying Input and Output Targets

The first point of interest is to understand how to specify the name and type of assembly to create (e.g., a console application named MyShell.exe, a code library named MathLib.dll, a Windows Forms application named Hello.exe, and so forth). Each possibility is represented by a specific flag passed into csc.exe as a command-line parameter

Table 2-1. Output Options of the C# Compiler

Option	Meaning in Life
/out	This option is used to specify the name of the assembly to be created. By default, the assembly name is the same as the name of the initial input *.cs file.
/target:exe	This option builds an executable console application. This is the default assembly output type, and thus may be omitted when building this type of application.
/target:library	This option builds a single-file *.dll assembly.
/target:module	This option builds a <i>module</i> . Modules are elements of multifile assemblies (fully described in Chapter 15).
/target:winexe	Although you are free to build graphical user interface-based applications using the /target:exe option, /target:winexe prevents a console window from appearing in the background.

To compile TestApp.cs into a console application named TestApp.exe, change to the directory containing your source code file:

```
cd C:\CscExample
```

And enter the following command set (note that command-line flags must come before the name of the input files, not after):

```
csc /target:exe TestApp.cs
```

Here I did not explicitly specify an /out flag, therefore the executable will be named TestApp.exe, given that TestApp is the name of the input file. Also be aware that most of the C# compiler flags support an abbreviated version, such as /t rather than /target (you can view all abbreviations by entering csc -? at the command prompt).

```
csc /t:exe TestApp.cs
```

Furthermore, given that the /t:exe flag is the default output used by the C# compiler, you could also compile TestApp.cs simply by typing csc TestApp.cs

Referencing External Assemblies

Next, let's examine how to compile an application that makes use of types defined in a separate .NET assembly. Speaking of which, just in case you are wondering how the C# compiler understood your reference to the System.Console type.

Let's update the TestApp application to display a Windows Forms message box. Open your TestApp.cs file and modify it as follows:

```
using System;
```

```
// Add this!
```

```
using System.Windows.Forms;
```

```
class TestApp
```

```
{
```

```
static void Main()
```

```
{  
Console.WriteLine("Testing! 1, 2, 3");  
// Add this!  
MessageBox.Show("Hello...");  
}  
}
```

Notice you are importing the System.Windows.Forms namespace via the C# using keyword (introduced in Chapter 1). Recall that when you explicitly list the namespaces used within a given *.cs file, you avoid the need to make use of fully qualified names of a type.

At the command line, you must inform csc.exe which assembly contains the namespaces you are using. Given that you have made use of the System.Windows.Forms.MessageBox class, you must specify the System.Windows.Forms.dll assembly using the /reference flag (which can be abbreviated to /r):

```
csc /r:System.Windows.Forms.dll TestApp.cs
```

Referencing Multiple External Assemblies

On a related note, what if you need to reference numerous external assemblies using csc.exe? Simply list each assembly using a semicolon-delimited list. You don't need to specify multiple external assemblies for the current example, but some sample usage follows:

```
csc /r:System.Windows.Forms.dll;System.Drawing.dll *.cs
```

Compiling Multiple Source Files

The current incarnation of the TestApp.exe application was created using a single *.cs source code file. While it is perfectly permissible to have all of your .NET types defined in a single *.cs file, most projects are composed of multiple *.cs files to keep your code base a bit more flexible. Assume you have authored an additional class contained in a new file named HelloMsg.cs:

```
// The HelloMessage class  
using System;  
using System.Windows.Forms;  
class HelloMessage  
{  
public void Speak()  
{  
MessageBox.Show("Hello...");  
}  
}
```

Now, update your initial TestApp class to make use of this new class type, and comment out the previous Windows Forms logic:

```
using System;

// Don't need this anymore.
// using System.Windows.Forms;

class TestApp
{
    static void Main()
    {
        Console.WriteLine("Testing! 1, 2, 3");
        // Don't need this anymore either.
        // MessageBox.Show("Hello...");
        // Use the HelloMessage class!
        HelloMessage h = new HelloMessage();
        h.Speak();
    }
}
```

You can compile your C# files by listing each input file explicitly:

```
csc /r:System.Windows.Forms.dll TestApp.cs HelloMsg.cs
```

As an alternative, the C# compiler allows you to make use of the wildcard character (*) to inform csc.exe to include all *.cs files contained in the project directory as part of the current build:

```
csc /r:System.Windows.Forms.dll *.cs
```

When you run the program again, the output is identical. The only difference between the two applications is the fact that the current logic has been split among multiple files.

The Command Line Debugger (cordbg.exe)

The .NET SDK does provide a command line debugger named cordbg.exe. This tool provides many number of options that allow us to run our .NET assemblies under debug mode. We may view them by specifying the -? flag

cordbg -?

Table shows some of the command line flags recognized by cordbg.exe (with the alternative shorthand notation).

<i>Command Line Flag of cordbg.exe</i>	<i>Meaning in Life</i>
b[reak]	Set or display current breakpoints
del[ete]	Remove one or more breakpoints
ex[it]	Exit the debugger

g[o]	Continue debugging the current process until hitting next breakpoint
si	Step into the next line
o[ut]	Step out of the current function
so	Step over of the next line
p[rint]	Print all loaded variables (local, arguments, etc.)

Debugging at the Command Line : Before we can debug our application using cordbg.exe, the first step is to generate symbolic debugging symbols for our current application by specifying the /debug flag of csc.exe.

For example:

```
csc @testapp.rsp /debug
```

This generates a new file named testapp.pdb . If one do not have an associated *.pdb file, it is still possible to make use of cordbg.exe; however, one will not be able to view our C# source code during the process.

Once we have a valid *.pdb file, open a session with cordbg.exe by specifying our .NET assembly as a command line argument (the *.pdb file will be loaded automatically):

```
cordbg.exe testapp.exe
```

At this point, we are in debugging mode, and may apply any number of `cordbg.exe` flags at the "(cordbg)" command prompt. When we are finished debugging our application and wish to exit debugging mode, simply type `exit` (or the `sh`

.NET has an expansive standard set of class libraries, referred to as either the base class libraries (core set) or framework class libraries (complete set). These libraries provide implementations for many general and app-specific types, algorithms and utility functionality. Both commercial and community libraries build on top of the framework class libraries, providing easy to use off-the-shelf libraries for a wide set of computing tasks.

A subset of these libraries are provided with each .NET implementation. Base Class Library (BCL) APIs are expected with any .NET implementation, both because developers will want them and because popular libraries will need them to run. App-specific libraries above the BCL, such as ASP.NET, will not be available on all .NET implementations.

Base Class Libraries

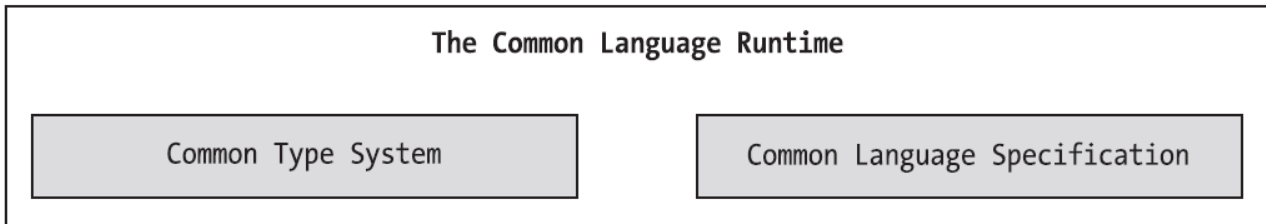
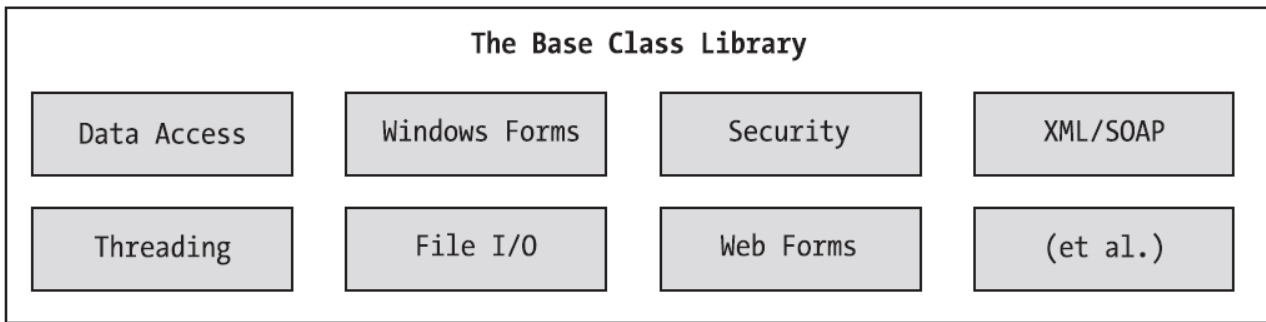
The BCL provides the most foundational types and utility functionality and are the base of all other .NET class libraries. They aim to provide very general implementations without any bias to any workload. Performance is always an important consideration, since apps might prefer a particular policy, such as low-latency to high-throughput or low-memory to low-CPU usage. These libraries are intended to be high-performance generally, and take a middle-ground approach according to these various performance concerns. For most apps, this approach has been quite successful.

The .NET platform provides a base class library that is available to all .NET programming languages.

Base class library encapsulate various primitives such as threads, fileinput/output (I/O), graphical rendering, and interaction with various external hardware devices, but it also provides support for a number of services required by most real-world applications.

For example, the base class libraries define types that facilitate database access, XML manipulation, programmatic security, and the construction of web-enabled front ends.

The relationship between the CLR, CTS, CLS, and the base class library, as shown in Figure.



The CLR, CTS, CLS, and base class library relationship

The .NET Framework class library is a library of classes, interfaces, and valuetypes that are included in the Windows Software Development Kit (SDK). This library provides access to system functionality and is designed to be the foundation on which .NET Framework applications, components, and controls are built.

System :-

The System namespace contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.

System.Data :-

The System.Data namespaces contain classes for accessing and managing data from diverse sources. The top-level namespace and a number of the child namespaces together form the ADO.NET architecture and ADO.NET data providers. For example, providers are available for SQL Server, Oracle, ODBC, and OleDb. Other child namespaces contain classes used by the ADO.NET Entity Data Model (EDM) and by WCF Data Services.

System.IO :-

The System.IO namespaces contain types that support input and output, including the ability to read and write data to streams either synchronously or asynchronously, to compress data in streams, to create and use isolated stores, to map files to an application's logical address space, to store multiple

data objects in a single container, to communicate using anonymous or named pipes, to implement custom logging, and to handle the flow of data to and from serial ports.

System.Net :-

The System.Net namespaces contain classes that provide a simple programming interface for a number of network protocols, programmatically access and update configuration settings for the System.Net namespaces, define cache policies for web resources, compose and send e-mail, represent Multipurpose Internet Mail Exchange (MIME) headers, access network traffic data and network address information, and access peer-to-peer networking functionality. Additional child namespaces provide a managed implementation of the Windows Sockets (Winsock) interface and provide access to network streams for secure communications between hosts.

System.Web :-

The System.Web namespaces contain types that enable browser/server communication. Child namespaces include types that support ASP.NET forms authentication, application services, data caching on the server, ASP.NET application configuration, dynamic data, HTTP handlers, JSON serialization, incorporating AJAX functionality into ASP.NET, ASP.NET security, and webservice.

System.Windows :-

The System.Windows namespaces contain types used in Windows Presentation Foundation (WPF) applications, including animation clients, user interface controls, data binding, and type conversion. System.Windows.Forms and its child namespaces are used for developing Windows Forms applications.

Primitive Types

.NET includes a set of primitive types, which are used (to varying degrees) in all programs. These types contain data, such as numbers, strings, bytes and arbitrary objects. The C# language includes keywords for these types. A sample set of these types is listed below, with the matching C# keywords.

- System.Object (object) - The ultimate base class in the CLR type system. It is the root of the type hierarchy.
- System.Int16 (short) - A 16-bit signed integer type. The unsigned UInt16 also exists.
- System.Int32 (int) - A 32-bit signed integer type. The unsigned UInt32 also exists.
- System.Single (float) - A 32-bit floating-point type.
- System.Decimal (decimal) - A 128-bit decimal type.
- System.Byte (byte) - An unsigned 8-bit integer that represents a byte of memory.
- System.Boolean (bool) - A Boolean type that represents true or false.
- System.Char (char) - A 16-bit numeric type that represents a Unicode character.
- System.String (string) - Represents a series of characters. Different than a char[], but enables indexing into each individual char in the string.

.NET APIs include classes, interfaces, delegates, and value types that expedite and optimize the development process and provide access to system functionality. To facilitate interoperability between languages, most .NET types are CLS-compliant and can therefore be used from any programming language whose compiler conforms to the common language specification (CLS).

.NET types are the foundation on which .NET applications, components, and controls are built. .NET includes types that perform the following functions:

- Represent base data types and exceptions.
- Encapsulate data structures.
- Perform I/O.
- Access information about loaded types.
- Invoke .NET security checks.
- Provide data access, rich client-side GUI, and server-controlled, client-side GUI.

.NET provides a rich set of interfaces, as well as abstract and concrete (non-abstract) classes. You can use the concrete classes as-is or, in many cases, derive your own classes from them. To use the functionality of an interface, you can either create a class that implements the interface or derive a class from one of the .NET classes that implements the interface.

Type of .NET Languages

By Dinesh Thakur

To help create languages for the .NET Framework, Microsoft created the Common Language Infrastructure specification (CLI). The CLI describes the features that each language must provide in order to use the .NET Framework and common language runtime and to interoperate with components written in other languages. If a language implements the necessary functionality, it is said to be .NET-compliant.

Every .NET-compliant language supports the same data types, uses the same .NET Framework classes, compiles to the same MSIL, and uses a single common language runtime to manage execution. Because of this, every .NET-compliant language is a first-class Microsoft .NET citizen.-Developers are free to choose the best language for a particular component without losing any of the power and freedom of the platform. In addition, components written in one language can easily interoperate with components written in another language. For example, you can write a class in C# that inherits from a base class written in Visual Basic.

The .NET Framework was developed so that it could support a theoretically infinite number of development languages. Currently, more than 20 development languages work with the .NET Framework. C# is the programming language specifically designed for the .NET platform, but C++ and Visual Basic have also been upgraded to fully support the .NET framework. The following are the commonly used languages provided by the Microsoft:

- **VC++**
- **VB.NET**
- **C#**
- **J#**
- **JScript .NET**

Many third parties are writing compilers for other languages with .NET support. With CLR, Microsoft has adopted a much liberal policy. Microsoft has them selves evolved/ developed/ modified many of their programming languages which compliant with .NET CLR.

We'll be covering the following topics in this tutorial:

- [VC++](#)
- [VB.NET](#)
- [C#](#)
- [J#](#)
- [JScript.NET](#)
- [Third-party languages](#)

VC++

Although Visual C++ (VC++) , has undergone changes to incorporate .NET; yet VC++ also maintains its status being a platform dependent programming. Many new MFC classes have been added a programmer can choose between using MFC and compiling the program into a platform specific executable file; or using .NET framework classes and compile into platform independent MISL file. A programmer can also specify (via directives) when ever he uses "unsafe" (the code that by passes CLR, e.g. the use of pointers) code.

VB.NET

Out of ALL .NET languages, Visual Basic.NET (VB.NET) is one language that has probably undergone into the most of changes. Now VB.NET may be considered a

complete Object- Oriented Language (as opposed to its previous “Half Object Based and Half Object Oriented” status).

Visual Basic .NET provides substantial language innovations over previous versions of visual basic. Visual Basic .NET supports inheritance, constructors, polymorphism, constructor overloading, structured exceptions, stricter type checking, free threading, and many other features. There is only one form of assignment: noLet of set methods. New rapid application development (RAD) features, such as XML Designer, Server Explorer, and Web Forms designer, are available in Visual Basic from Visual Studio .NET. With this release, Visual Basic Scripting Edition provides full Visual Basic functionality.

C#

Microsoft has also developed a brand new programming language C# (C Sharp). This language makes full use of .NET. It is a pure object oriented language. A [Java](#) programmer may find most aspects of this language which is identical to [Java](#). If you are a new comer to Microsoft Technologies – this language is the easiest way to get on the .NET band wagon. While VC++ and VB enthusiast would stick to VC.NET and VB.NET; they would probably increase their productivity by switching to C#. C# is developed to make full use of all the intricacies of .NET. The learning curve of C# for a Java programmer is minimal. Microsoft has also come up with a The Microsoft Java Language Conversion Assistant- which is a tool that automatically converts existing Java-language source code into C# for developers who want to move their existing applications to the Microsoft .NET Framework.

J#

Microsoft has also developed J# (Java Sharp). C# is similar to Java, but it is not entirely' identical. It is for this reason that Microsoft has developed J# – the syntax of J# is identical to Visual J++. Microsoft's growing legal battle with Sun, over Visual J++ – forced Microsoft to discontinue Visual J++. So J# is Microsoft's indirect continuation of Visual J++. It has been reported that porting a medium sized Visual J++ project, entirely to J# takes only a few days of effort.

JScript.NET

Jscript.NET is rewritten to be fully .NET aware. It includes support for classes, inheritance, types and compilation, and it provides improved performance and productivity features. JScript.NET is also integrated with visual Studio .NET. You can take advantage of any .NET Framework class in JScript .NET.

Third-party languages

Microsoft encourages third party vendors to make use of Visual Studio. Net. Third, party vendors can write compilers for different languages ~ that compile the language to MSIL

(Microsoft Intermediate Language). These vendors need not develop their own development environment. They can easily use Visual Studio.NET as an IDE for their .NET compliant language. A vendor has already produced COBOL.NET that integrates with Visual Studio.NET and compiles into MSIL. Theoretically it would then be possible to come up with Java compiler that compiles into MSIL, instead of Java Byte code; and uses CLR

instead of JVM. However Microsoft has not pursued this due to possible legal action by Sun.

Several third party languages are supporting the .NET platform. These languages include APL, COBOL, Pascal, Eiffel, Haskell, ML, Oberon, Perl, [Python](#), Scheme and Smalltalk.

3.Assemblies in .NET

Assemblies form the fundamental units of deployment, version control, reuse, activation scoping, and security permissions for .NET-based applications. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. Assemblies take the form of executable (.exe) or dynamic link library (.dll) files, and are the building blocks of .NET applications. They provide the common language runtime with the information it needs to be aware of type implementations.

In .NET Core and .NET Framework, you can build an assembly from one or more source code files. In .NET Framework, assemblies can contain one or more modules. This allows larger projects to be planned so that several developers can work on separate source code files or modules, which are combined to create a single assembly. For more information about modules, see [How to: Build a multifile assembly](#).

Assemblies have the following properties:

- Assemblies are implemented as .exe or .dll files.
- For libraries that target the .NET Framework, you can share assemblies between applications by putting them in the global assembly cache (GAC). You must strong-name assemblies before you can include them in the GAC. For more information, see [Strong-named assemblies](#).
- Assemblies are only loaded into memory if they are required. If they aren't used, they aren't loaded. This means that assemblies can be an efficient way to manage resources in larger projects.
- You can programmatically obtain information about an assembly by using reflection. For more information, see [Reflection \(C#\)](#) or [Reflection \(Visual Basic\)](#).
- You can load an assembly just to inspect it by using the `MetadataLoadContext` class in .NET Core and the `Assembly.ReflectionOnlyLoad` or `Assembly.ReflectionOnlyLoadFrom` methods in .NET Core and .NET Framework.

Assemblies in the common language runtime

Assemblies provide the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

An assembly defines the following information:

- Code that the common language runtime executes. Note that each assembly can have only one entry point: `DllMain`, `WinMain`, or `Main`.
- Security boundary. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries in assemblies, see [Assembly security considerations](#).
- Type boundary. Every type's identity includes the name of the assembly in which it resides. A type called `MyType` that is loaded in the scope of one assembly is not the same as a type called `MyType` that is loaded in the scope of another assembly.
- Reference scope boundary. The assembly manifest has metadata that is used for resolving types and satisfying resource requests. The manifest specifies the types and resources to expose outside the assembly, and enumerates other assemblies on which it depends. Microsoft intermediate language (MSIL) code in a portable executable (PE) file won't be executed unless it has an associated assembly manifest.
- Version boundary. The assembly is the smallest versionable unit in the common language runtime. All types and resources in the same assembly are versioned as a unit. The assembly manifest describes the version dependencies you specify for any dependent assemblies. For more information about versioning, see [Assembly versioning](#).
- Deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as assemblies containing localization resources or utility classes, can be retrieved on demand. This allows apps to be simple and thin when first downloaded. For more information about deploying assemblies, see [Deploy applications](#).
- Side-by-side execution unit. For more information about running multiple versions of an assembly, see [Assemblies and side-by-side execution](#).

Assemblies, Manifests, Metadata, and Modules

Different .NET-compliant language compilers produce MSIL (Microsoft Intermediate Language) along with *metadata*, which describes the types in your code. Following are the functions of metadata:

- Describes and references the datatypes defined by the VOS (Virtual Object System) type system
- Lays out instances of classes in memory
- Resolves method invocation
- Solves versioning problems (DLL hell)

The Microsoft Intermediate Language and metadata are packed into an executable format called a *portable executable file* (PE file), which extends PE and the Common Object File Format (COFF). COFF enables the operating system to recognize Common Language Runtime (CLR). One great

advantage of packing metadata along with the MSIL is that it enables your code to describe itself, thus eliminating the need for type libraries or Interface Definition Language (IDL).

Assemblies, modules, and manifests are grouping constructs, each playing a different role in the CLR environment.

An *assembly* is more of a logical entity than a physical entity. Hence, it can be used in multiple files. It contains the CLR targeted code compiled by different language compilers. An assembly is a completely self-describing unit, which can be thought of as .dll or .exe. If the Portable executable doesn't contain the associated assembly manifest, it doesn't get executed and moreover there will be one and only one entry point for an assembly. Following are the some of the functions of the assemblies:

- It acts as a boundary for all sort of security, types, reference library, and versioning requests.
- It's the deployment unit. On the initialization of an application, assemblies of an application must be present and others could be called on demand.
- It makes side-by-side deployment possible.

A *manifest* is basically an area of metadata occupied in an assembly that allows checks to be made on the version of the assembly. It also checks for the integrity of an assembly.

A *module* contains the file format of an executable. If the module contains a manifest, it becomes an assembly as well as a module. There will be one and only one manifest in all of the files. The relationship between assembly, module, and files can be visualized from the diagram in [Figure 2](#).

Figure 2 Assembly-Module-File relationship.

Having seen how assembly, module, and files communicate with each other, let's look into the sequence of the declaration followed while writing the IL programs:

```
.assembly
.assembly extern
.class
.class extern
.corflags
.custom
.data
.field
.file
.mresource
.method
.module
```

4.Naming conventions

.NET types use a dot syntax naming scheme that connotes a hierarchy. This technique groups related types into namespaces so they can be searched and referenced more easily. The first part of

4.Naming conventions

.NET types use a dot syntax naming scheme that connotes a hierarchy. This technique groups related types into namespaces so they can be searched and referenced more easily. The first part of the full name — up to the rightmost dot — is the namespace name. The last part of the name is the type name. For example, `System.Collections.Generic.List<T>` represents the `List<T>` type, which belongs to the `System.Collections.Generic` namespace. The types in `System.Collections.Generic` can be used to work with generic collections.

This naming scheme makes it easy for library developers extending .NET to create hierarchical groups of types and name them in a consistent, informative manner. It also allows types to be unambiguously identified by their full name (that is, by their namespace and type name), which prevents type name collisions. Library developers are expected to use the following convention when creating names for their namespaces:

CompanyName.TechnologyName

For example, the namespace `Microsoft.Word` conforms to this guideline.

The use of naming patterns to group related types into namespaces is a useful way to build and document class libraries. However, this naming scheme has no effect on visibility, member access, inheritance, security, or binding. A namespace can be partitioned across multiple assemblies and a single assembly can contain types from multiple namespaces. The assembly provides the formal structure for versioning, deployment, security, loading, and visibility in the common language runtime.

For more information on namespaces and type names, see [Common Type System](#).

System namespace

The `System` namespace is the root namespace for fundamental types in .NET. This namespace includes classes that represent the base data types used by all applications: `Object` (the root of the inheritance hierarchy), `Byte`, `Char`, `Array`, `Int32`, `String`, and so on. Many of these types correspond to the primitive data types that your programming language uses. When you write code using .NET types, you can use your language's corresponding keyword when a .NET base data type is expected.

The following table lists the base types that .NET supplies, briefly describes each type, and indicates the corresponding type in Visual Basic, C#, C++, and F#.

System namespace

Category	Class name	Description	Visual Basic data type	C# data type	C++/CLI data type	F# data type
Integer	Byte	An 8-bit unsigned integer.	Byte	byte	unsigned char	byte
	SByte	An 8-bit signed integer.	SByte	sbyte	char -or- signed	sbyte
		Not CLS-compliant.				

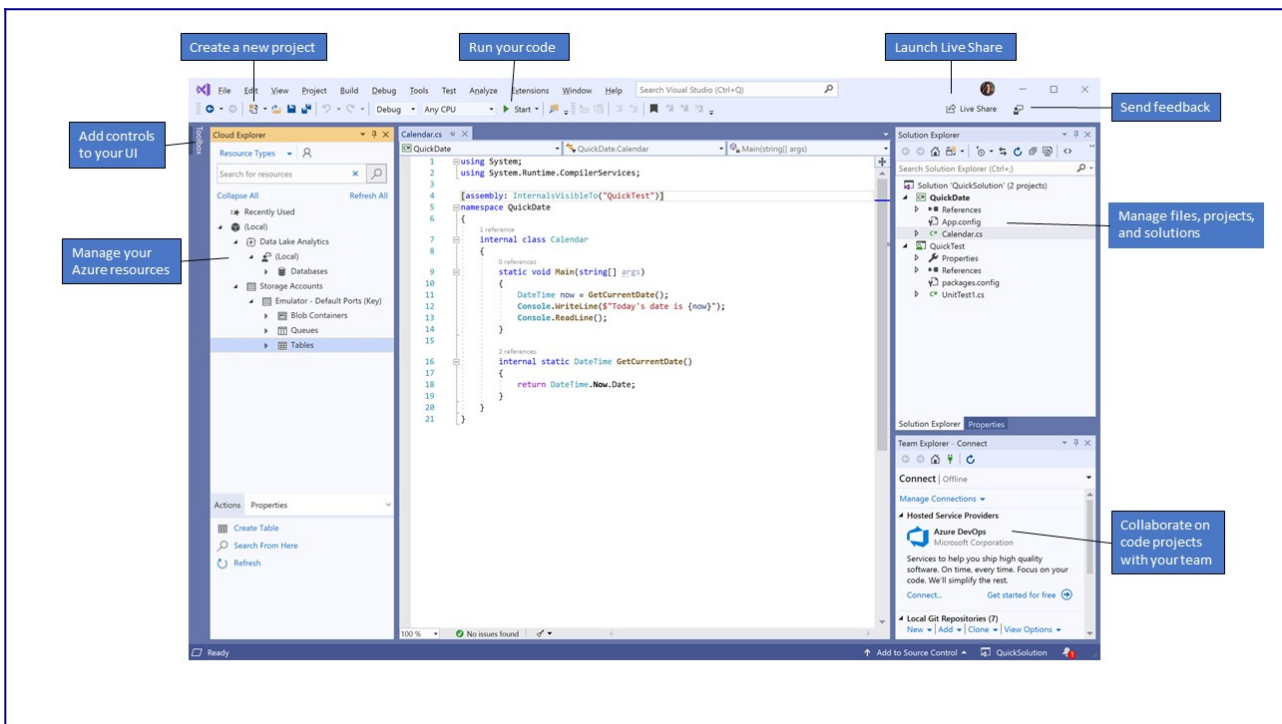
	Int16	A 16-bit signed integer.	Short	short	char short int	int16
	Int32	A 32-bit signed integer.	Integer	int	-or- long	int
	Int64	A 64-bit signed integer.	Long	long	__int64	int64
	UInt16	A 16-bit unsigned integer. Not CLS-compliant.	UShort	ushort	unsigned short	uint16
	UInt32	A 32-bit unsigned integer. Not CLS-compliant.	UInteger	uint	unsigned int -or- unsigned long	uint32
	UInt64	A 64-bit unsigned integer. Not CLS-compliant.	ULong	ulong	unsigned __int64	uint64
Floating point	Single	A single-precision (32-bit) floating-point number.	Single	float	float	float32 or single
	Double	A double-precision (64-bit) floating-point number.	Double	double	double	float or double
Logical	Boolean	A Boolean value (true or false).	Boolean	bool	bool	bool
Other	Char	A Unicode (16-bit) character.	Char	char	wchar_t	char
	Decimal	A decimal (128-bit) value.	Decimal	decimal	Decimal	decimal
	IntPtr	A signed integer whose size depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform).	IntPtr	IntPtr	IntPtr	unativeint
	UIntPtr	An unsigned integer whose size depends on the underlying platform (a 32-bit value on a 32-bit platform and a 64-bit value on a 64-bit platform).	UIntPtr	UIntPtr	UIntPtr	unativeint
	Object	Not CLS-compliant. The root of the object hierarchy.	Object	object	Object^	obj
	String	An immutable, fixed-length string of Unicode characters.	String	string	String^	string

In addition to the base data types, the System namespace contains over 100 classes, ranging from classes that handle exceptions to classes that deal with core runtime concepts, such as application domains and the garbage collector. The System namespace also contains many second-level namespaces.

For more information about namespaces, use the .NET API Browser to browse the .NET Class Library. The API reference documentation provides documentation on each namespace, its types, and each of their members.

using the visual studio .NET IDE

The Visual Studio *integrated development environment* is a creative launching pad that you can use to edit, debug, and build code, and then publish an app. An integrated development environment (IDE) is a feature-rich program that can be used for many aspects of software development. Over and above the standard editor and debugger that most IDEs provide, Visual Studio includes compilers, code completion tools, graphical designers, and many more features to ease the software development process.



This image shows Visual Studio with an open project and several key tool windows you'll likely use:

- **Solution Explorer** (top right) lets you view, navigate, and manage your code files. **Solution Explorer** can help organize your code by grouping the files into solutions and projects.
- The editor window (center), where you'll likely spend a majority of your time, displays file contents. This is where you can edit code or design a user interface such as a window with buttons and text boxes.
- **Team Explorer** (bottom right) lets you track work items and share code with others using version control technologies such as Git and Team Foundation Version Control (TFVC).

Editions

Visual Studio is available for Windows and Mac. Visual Studio for Mac has many of the same features as Visual Studio 2019, and is optimized for developing cross-platform and mobile apps. This article focuses on the Windows version of Visual Studio 2019.

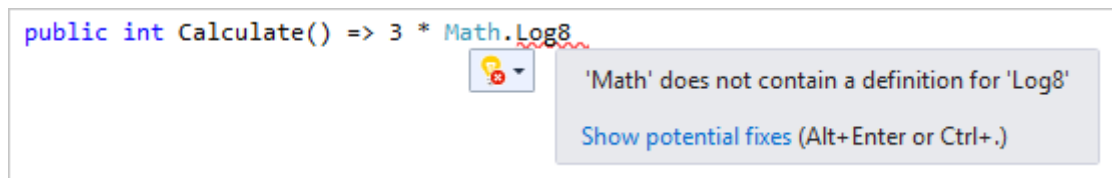
There are three editions of Visual Studio 2019: Community, Professional, and Enterprise. See [Compare Visual Studio editions](#) to learn about which features are supported in each edition.

Popular productivity features

Some of the popular features in Visual Studio that help you to be more productive as you develop software include:

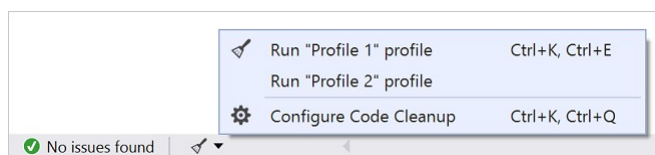
- Squiggles and Quick Actions

Squiggles are wavy underlines that alert you to errors or potential problems in your code as you type. These visual clues enable you to fix problems immediately without waiting for the error to be discovered during build or when you run the program. If you hover over a squiggle, you see additional information about the error. A light bulb may also appear in the left margin with actions, known as Quick Actions, to fix the error.



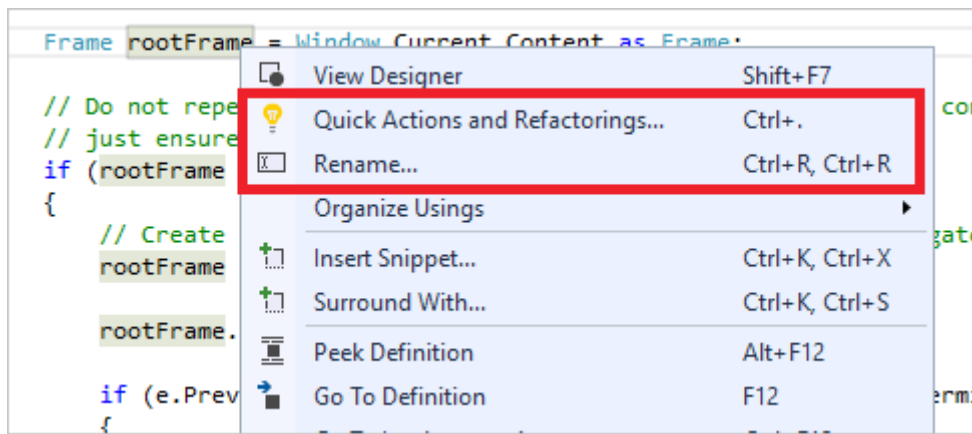
- Code Cleanup

With the click of a button, format your code and apply any code fixes suggested by your code style settings, .editorconfig conventions, and Roslyn analyzers. **Code Cleanup** helps you resolve issues in your code before it goes to code review. (Currently available for C# code only.)



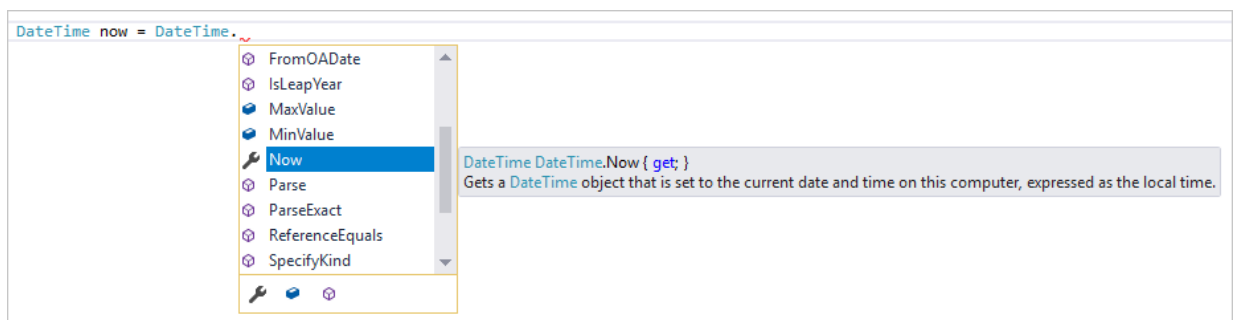
- Refactoring

Refactoring includes operations such as intelligent renaming of variables, extracting one or more lines of code into a new method, changing the order of method parameters, and more.



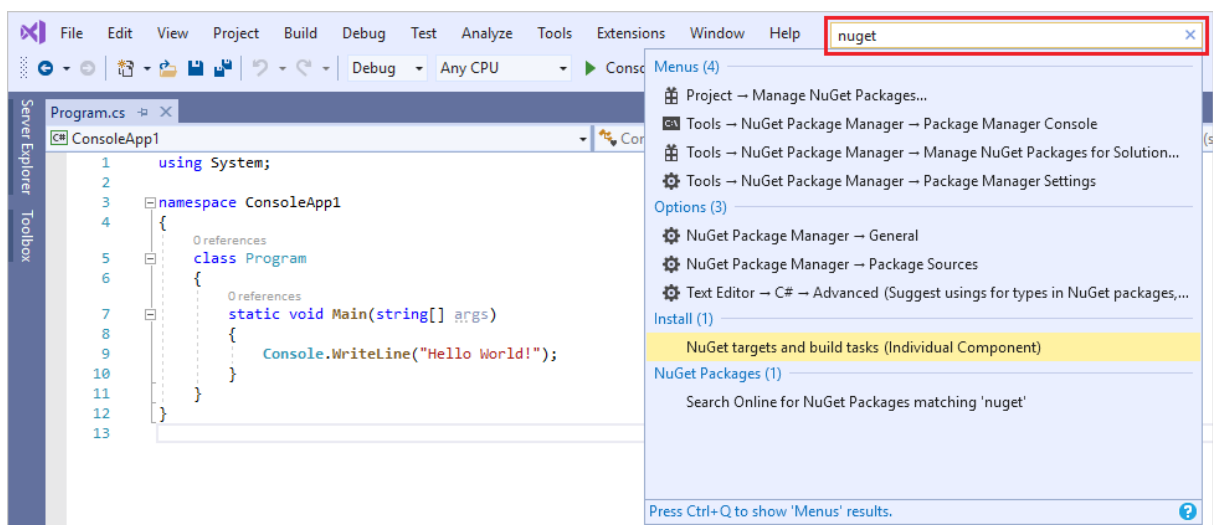
- IntelliSense

IntelliSense is a term for a set of features that displays information about your code directly in the editor and, in some cases, write small bits of code for you. It's like having basic documentation inline in the editor, which saves you from having to look up type information elsewhere. IntelliSense features vary by language. For more information, see [C# IntelliSense](#), [Visual C++ IntelliSense](#), [JavaScript IntelliSense](#), and [Visual Basic IntelliSense](#). The following illustration shows how IntelliSense displays a member list for a type:



- Visual Studio search

Visual Studio can seem overwhelming at times with so many menus, options, and properties. Visual Studio search (**Ctrl+Q**) is a great way to rapidly find IDE features and code in one place.



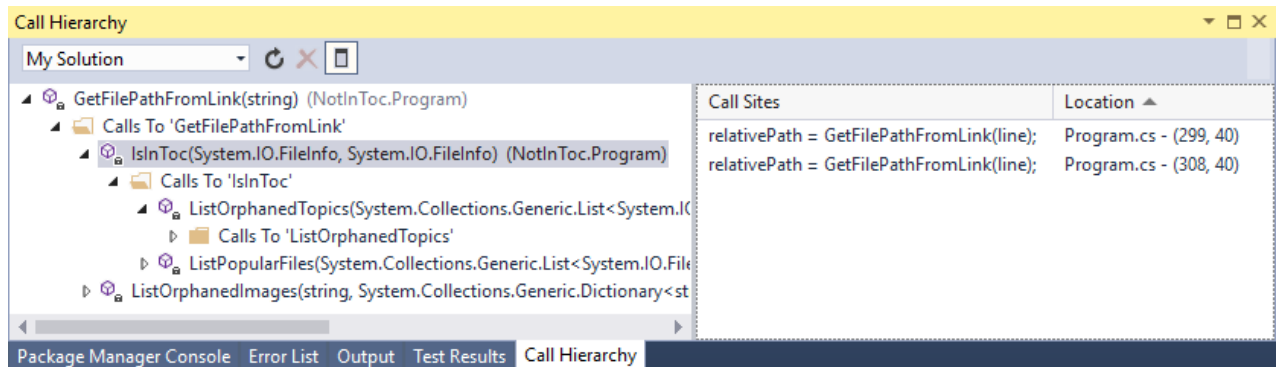
For information and productivity tips, see [How to use Visual Studio search](#).

- Live Share

Collaboratively edit and debug with others in real time, regardless of what your app type or programming language. You can instantly and securely share your project and, as needed, debugging sessions, terminal instances, localhost web apps, voice calls, and more.

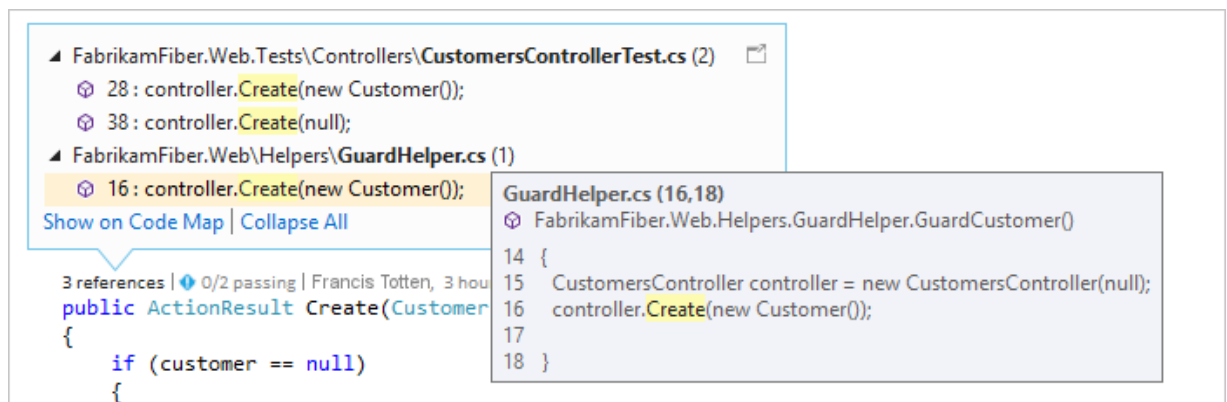
- Call Hierarchy

The **Call Hierarchy** window shows the methods that call a selected method. This can be useful information when you're thinking about changing or removing the method, or when you're trying to track down a bug.



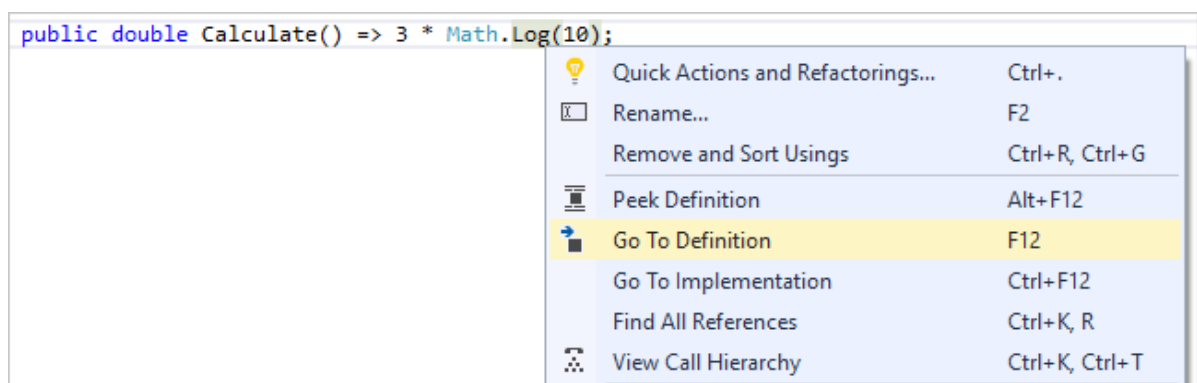
- CodeLens

CodeLens helps you find references to your code, changes to your code, linked bugs, work items, code reviews, and unit tests, all without leaving the editor.



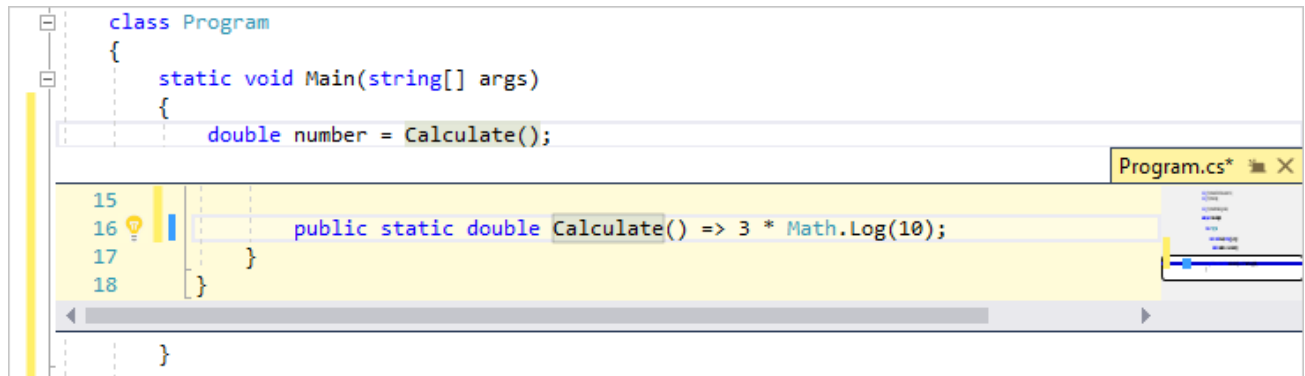
- Go To Definition

The Go To Definition feature takes you directly to the location where a function or type is defined.



- Peek Definition

The **Peek Definition** window shows the definition of a method or type without actually opening a separate file.



The screenshot shows a Visual Studio code editor with a C# file named Program.cs. The code is as follows:

```
class Program
{
    static void Main(string[] args)
    {
        double number = Calculate();
    }
}

public static double Calculate() => 3 * Math.Log(10);
```

The Peek Definition window is open, showing the definition of the `Calculate()` method: `public static double Calculate() => 3 * Math.Log(10);`. The window title is `Program.cs*`.

Build cross-platform apps and games

You can use Visual Studio to build apps and games for macOS, Linux, and Windows, as well as for Android, iOS, and other mobile devices.

- Build .NET Core apps that run on Windows, macOS, and Linux.
- Build mobile apps for iOS, Android, and Windows in C# and F# by using Xamarin.
- Use standard web technologies—HTML, CSS, and JavaScript—to build mobile apps for iOS, Android, and Windows by using Apache Cordova.
- Build 2D and 3D games in C# by using Visual Studio Tools for Unity.
- Build native C++ apps for iOS, Android, and Windows devices. Share common code in libraries built for iOS, Android, and Windows, by using C++ for cross-platform development.
- Deploy, test, and debug Android apps with the Android emulator.

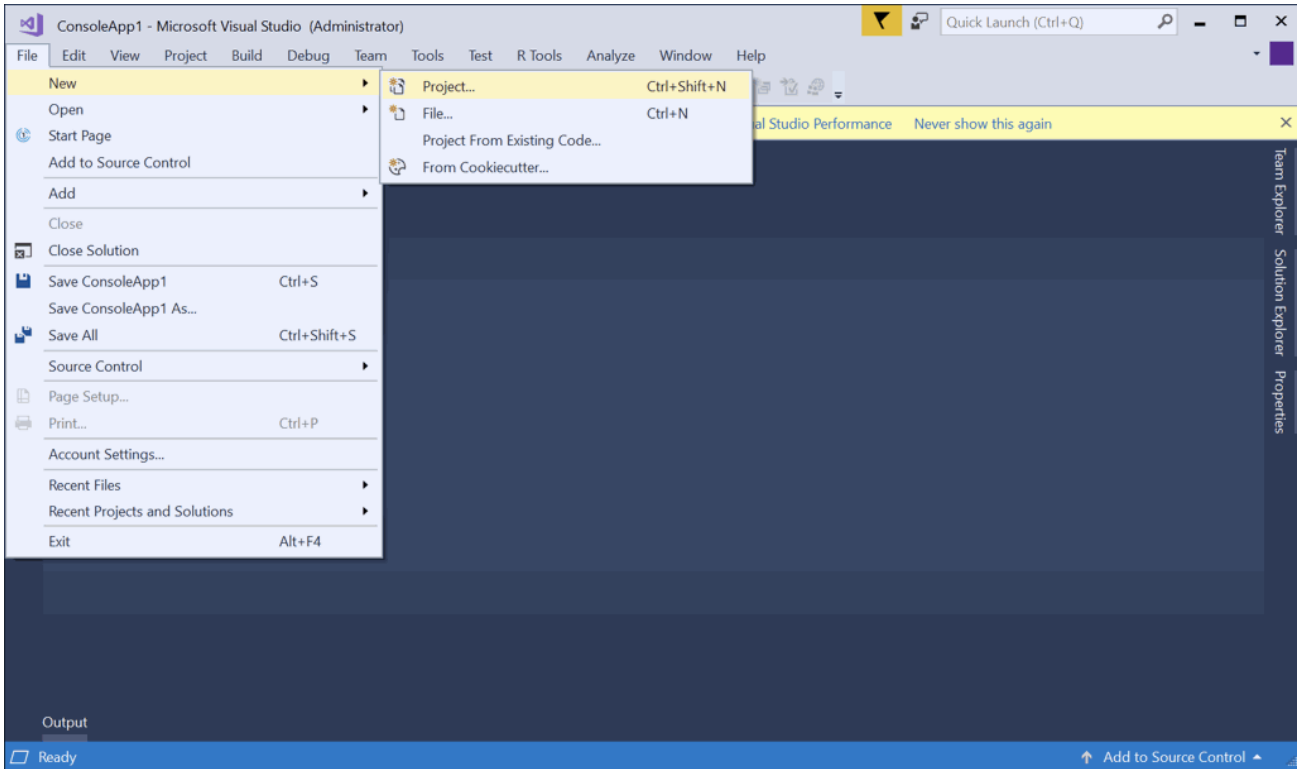
There are many IDEs available for editing and compiling C# programs in Windows like Microsoft Visual Studio and NetBeans. We will see how to run a C# code using Visual Studio and Command Prompt.

Visual Studio

We can edit and compile programs using Visual Studio which allows us to write, compile and run our program. Let's see how to do it.

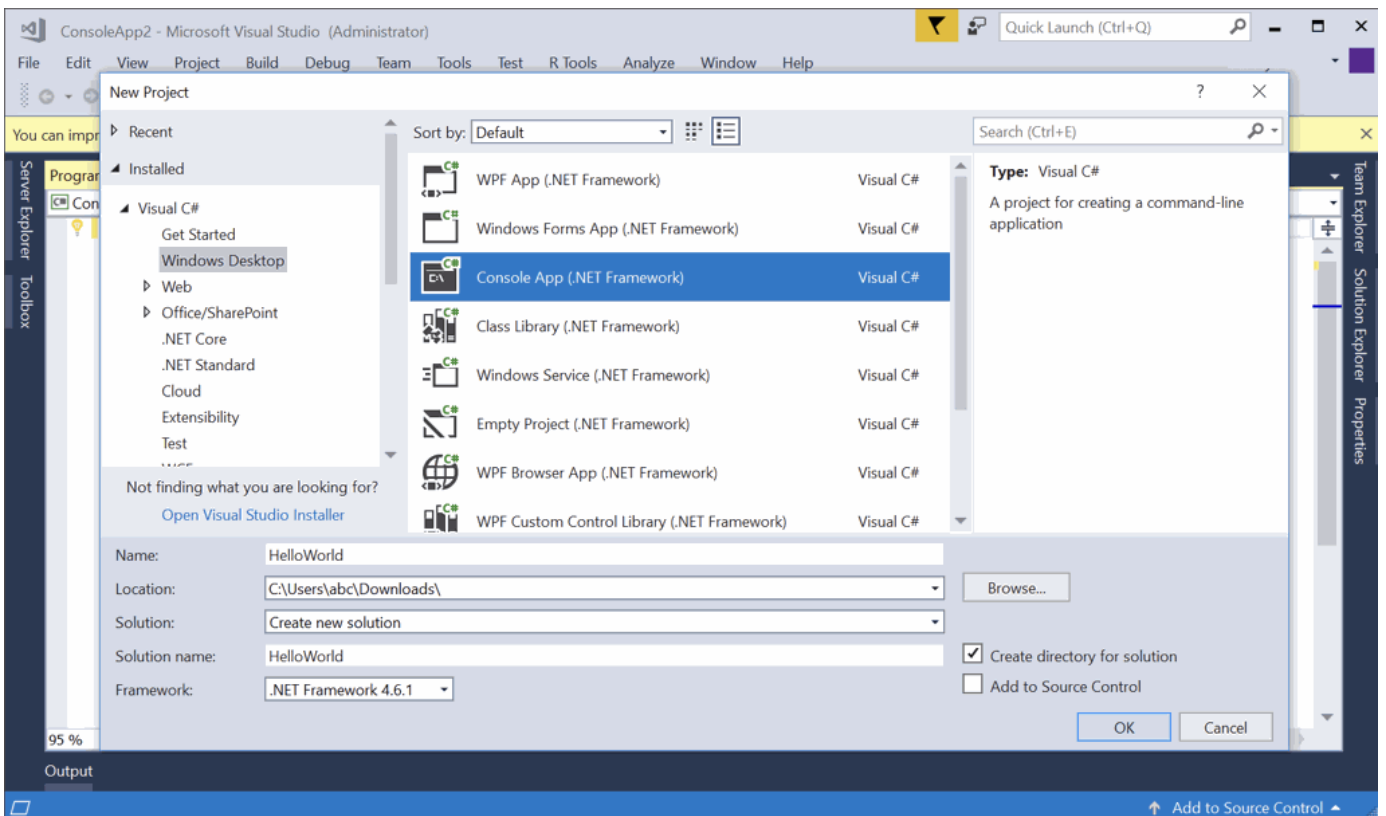
Before learning to write a program in Visual Studio, first, make sure it is installed in your computer. If not, then download and install it from [here](#).

1. On opening Visual Studio, you will get a window. Click on File → New → Project.



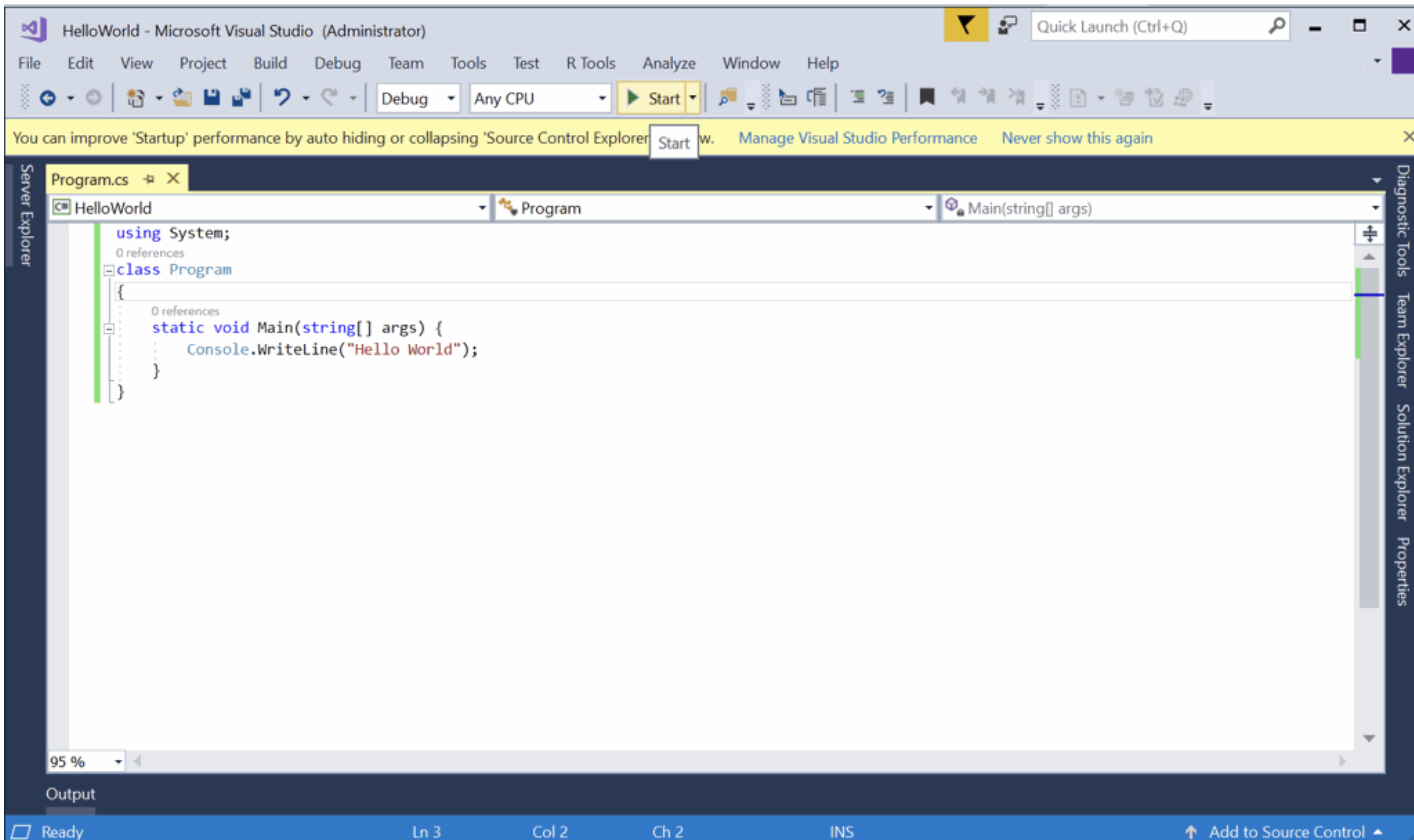
2. The New Project dialog box will appear. Expand **Installed**, then expand **Visual C#**, then select **Windows Desktop**, and then select **Console Application**.

3. Give the project name in the **Name** text box, and then click on OK button.

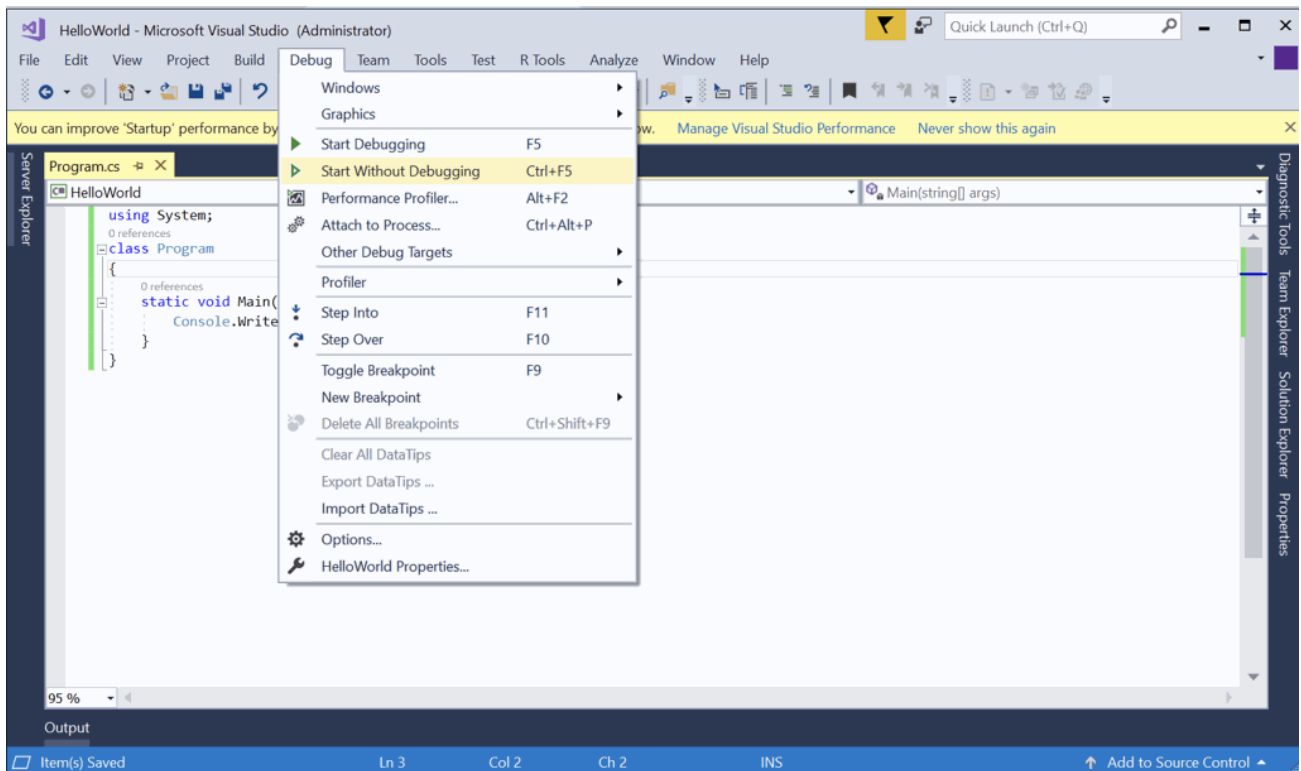


3. A file named *Program.cs* will get opened in the Code Editor. Replace its content with your C# program given below.

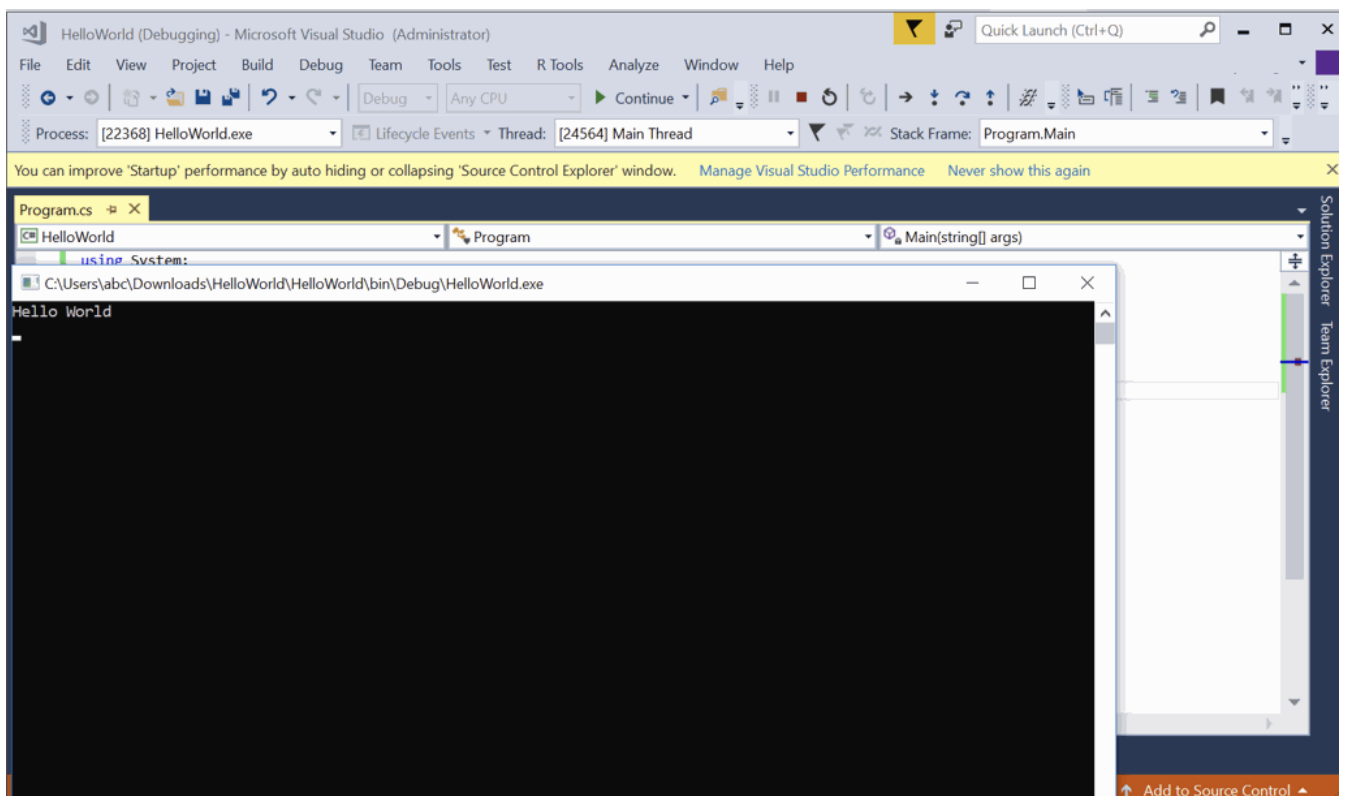
```
using System;
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World");
    }
}
```



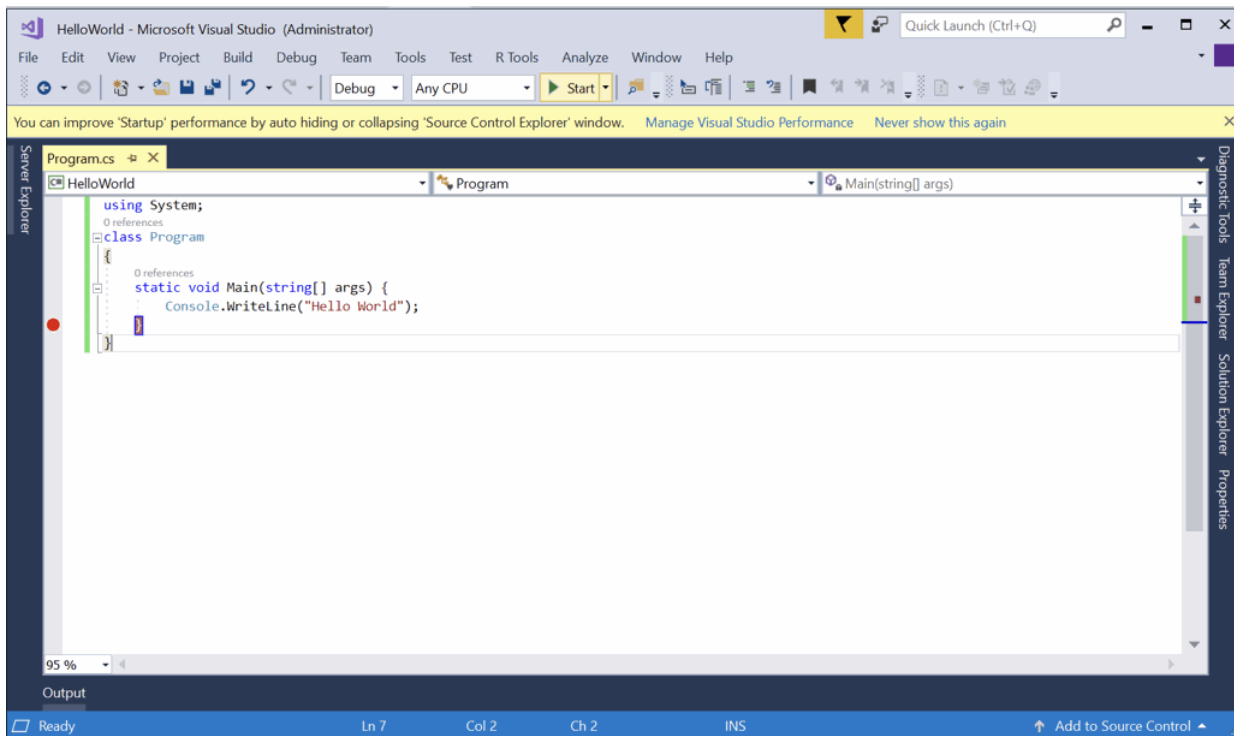
4. To run the project, click on Debug → Start Without Debugging or press Ctrl+F5 keys.



5. A Command Prompt window will appear with "Hello World" written.



If you want to use the debugger, then put the breakpoint on the last line as shown below. Then to run the project, click on the **Start** button or press the **F5** key.



using the visual studio .NET IDE

Create a program

Let's dive in and create a simple program.

1. Open Visual Studio.

The start window appears with various options for cloning a repo, opening a recent project, or creating a brand new project.

2. Choose **Create a new project**.

visual studio .NET IDE debugging, C# "pre-processor" directives.

Use Debug build configuration

Debug and *Release* are Visual Studio's built-in build configurations. You use the Debug build configuration for debugging and the Release configuration for the final release distribution.

In the Debug configuration, a program compiles with full symbolic debug information and no optimization. Optimization complicates debugging, because the relationship between source code and generated instructions is more complex. The release configuration of a program has no symbolic debug information and is fully optimized.

By default, Visual Studio uses the Debug build configuration, so you don't need to change it before debugging.

1. Start Visual Studio.
2. Open the project that you created in Create a .NET console application using Visual Studio.

The current build configuration is shown on the toolbar. The following toolbar image shows that Visual Studio is configured to compile the Debug version of the app:

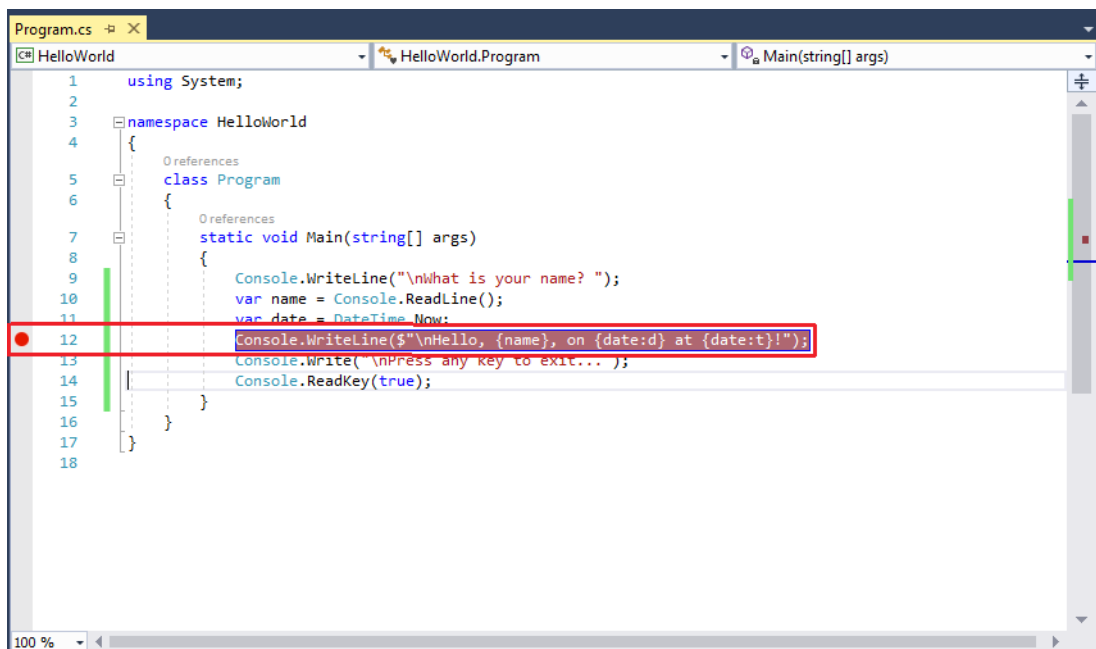


Set a breakpoint

A *breakpoint* temporarily interrupts the execution of the application before the line with the breakpoint is executed.

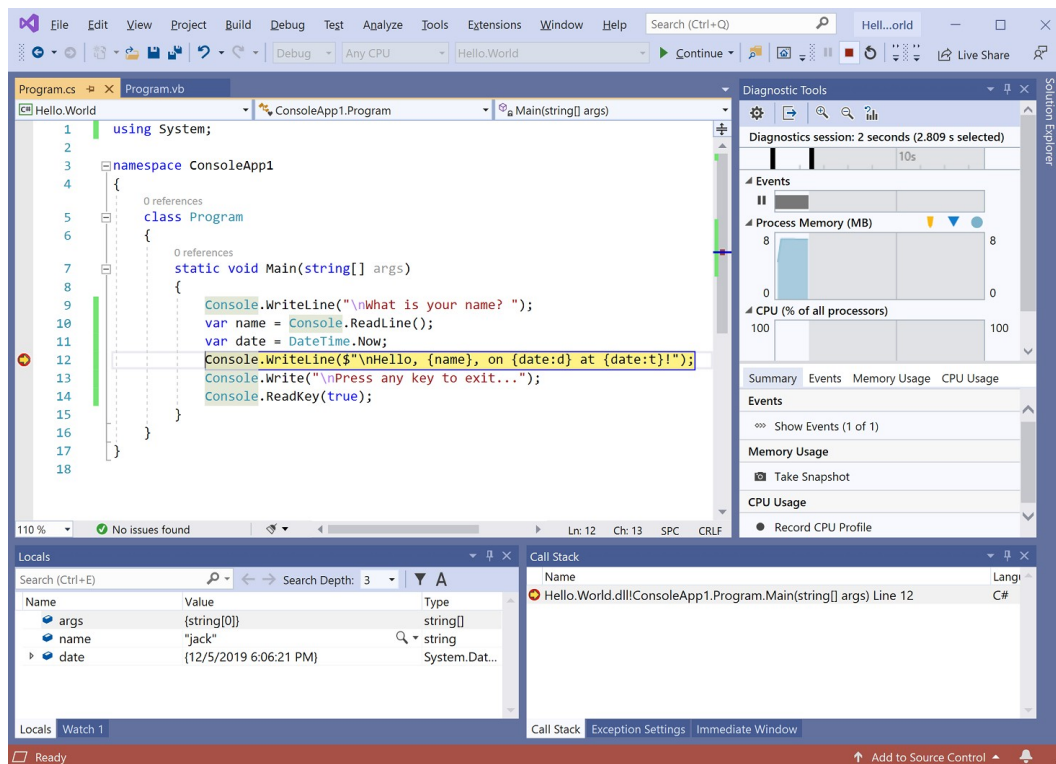
1. Set a *breakpoint* on the line that displays the name, date, and time, by clicking in the left margin of the code window on that line. The left margin is to the left of the line numbers. Other ways to set a breakpoint are by placing the cursor in the line of code and then pressing F9 or choosing **Debug > Toggle Breakpoint** from the menu bar.

As the following image shows, Visual Studio indicates the line on which the breakpoint is set by highlighting it and displaying a red dot in the left margin.



2. Press F5 to run the program in Debug mode. Another way to start debugging is by choosing **Debug > Start Debugging** from the menu.
3. Enter a string in the console window when the program prompts for a name, and then press Enter.

4. Program execution stops when it reaches the breakpoint and before the `Console.WriteLine` method executes. The **Locals** window displays the values of variables that are defined in the currently executing method.



Use the Immediate window

The **Immediate** window lets you interact with the application you're debugging. You can interactively change the value of variables to see how it affects your program.

1. If the **Immediate** window is not visible, display it by choosing **Debug > Windows > Immediate**.
2. Enter `name = "Gracie"` in the **Immediate** window and press the Enter key.
3. Enter `date = DateTime.Parse("2019-11-16T17:25:00Z").ToUniversalTime()` in the **Immediate** window and press the Enter key.

The **Immediate** window displays the value of the string variable and the properties of the `DateTime` value. In addition, the values of the variables are updated in the **Locals** window.

4. Press F5 to continue program execution. Another way to continue is by choosing **Debug > Continue** from the menu.

The values displayed in the console window correspond to the changes you made in the **Immediate** window.

```
C:\Users\WDAGUtilityAccount\source\repos\HelloWorld\HelloWorld\bin\Debug\net5.0\HelloWorld.exe
What is your name?
jack
Hello, Gracie, on 11/16/2019 at 5:25 PM!
Press any key to exit...
```

5. Press any key to exit the application and stop debugging.

C# "pre-processor" directives

C# compiler does not have a separate preprocessor; however, the directives are processed as if there was one. In C# the preprocessor directives are used to help in conditional compilation.

The preprocessor directives give instruction to the compiler to preprocess the information before actual compilation starts.

The following are the preprocessor directives in C# –

Sr.No.	Preprocessor Directive & Description
1	#define It defines a sequence of characters, called symbol.
2	#undef It allows you to undefine a symbol.
3	#if It allows testing a symbol or symbols to see if they evaluate to true.
4	#else It allows to create a compound conditional directive, along with #if.
5	#elif It allows creating a compound conditional directive.
6	#endif Specifies the end of a conditional directive.
7	#line It lets you modify the compiler's line number and (optionally) the file name output for errors and warnings.
8	#error It allows generating an error from a specific location in your code.
9	#warning It allows generating a level one warning from a specific location in your code.
10	#region It lets you specify a block of code that you can expand or collapse when using the outlining feature of the Visual Studio Code Editor.
11	#endregion It marks the end of a #region block.

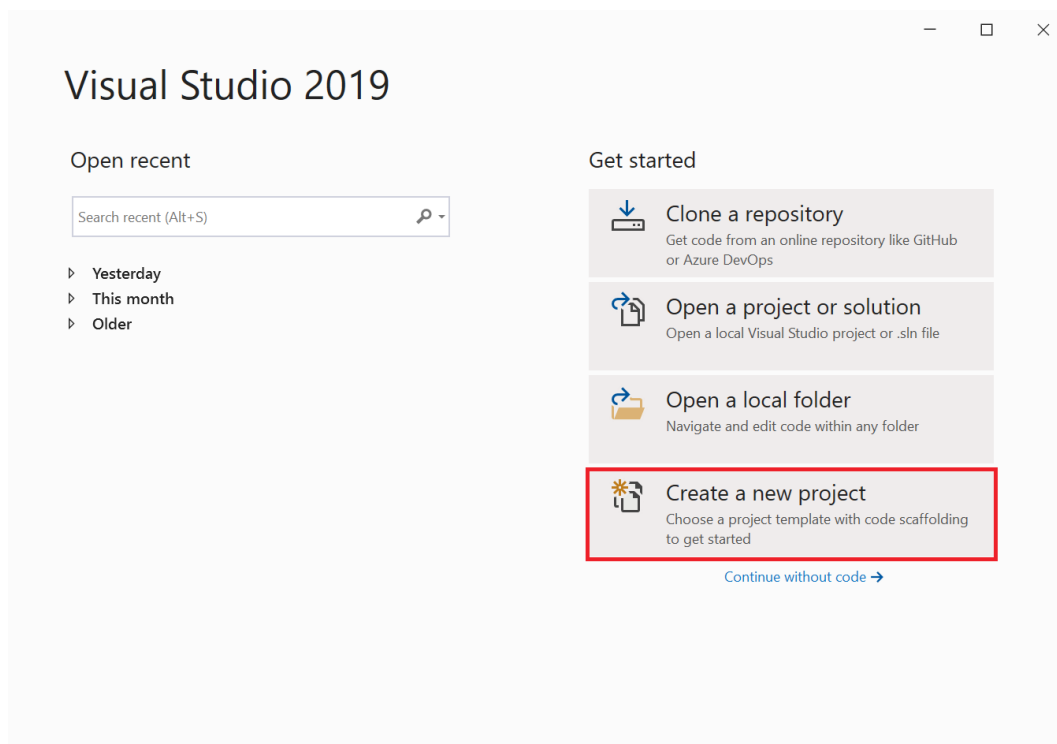
Let us see an example to learn about the usage of pre-processor directive in C# –

Example

#define PI 3.12

```
using System;

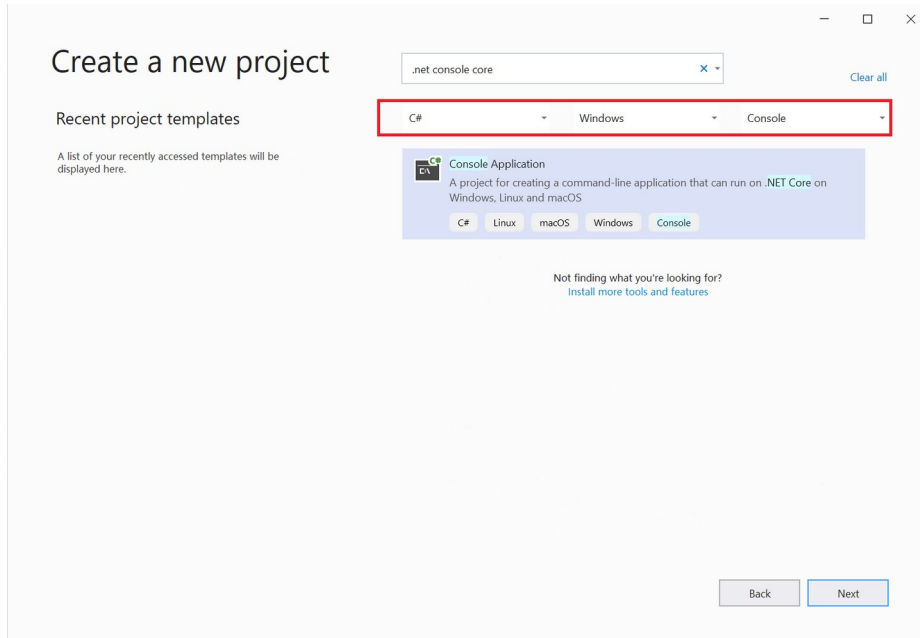
namespace Demo {
    class Program {
        static void Main(string[] args) {
            #if (PI)
                Console.WriteLine("PI is defined");
            #else
                Console.WriteLine("PI is not defined");
            #endif
            Console.ReadKey();
        }
    }
}
```



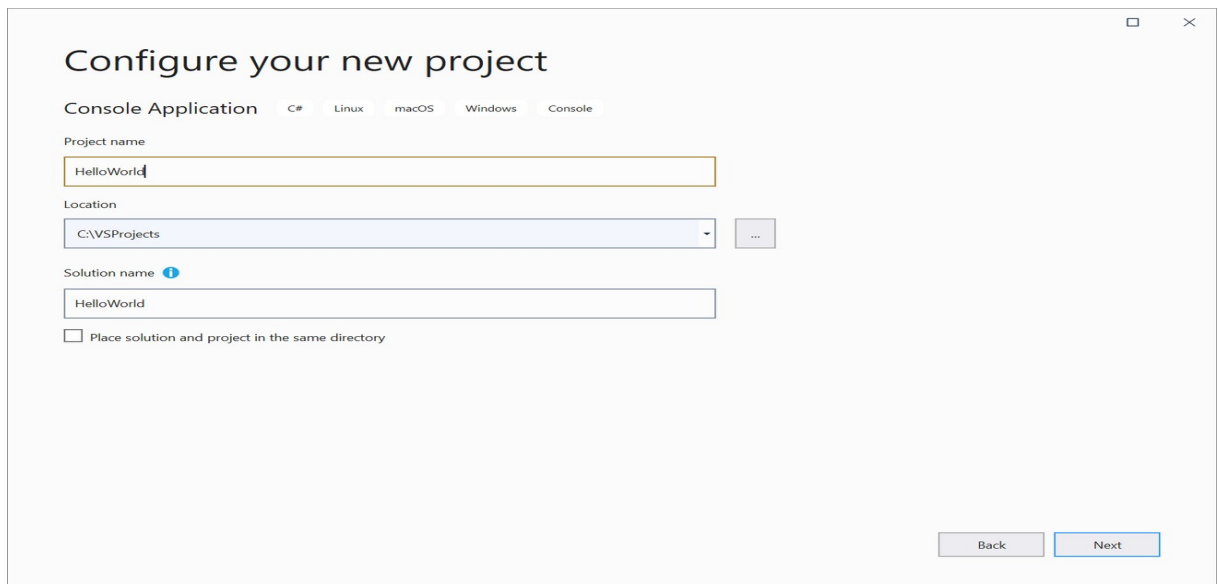
The **Create a new project** window opens and shows several project *templates*. A template contains the basic files and settings required for a given project type.

- To find the template we want, type or enter **.net core console** in the search box. The list of available templates is automatically filtered based on the keywords you entered. You can further filter the template results by choosing **C#** from the **All language** drop-down list, **Windows** from the **All platforms** list, and **Console** from the **All project types** list .

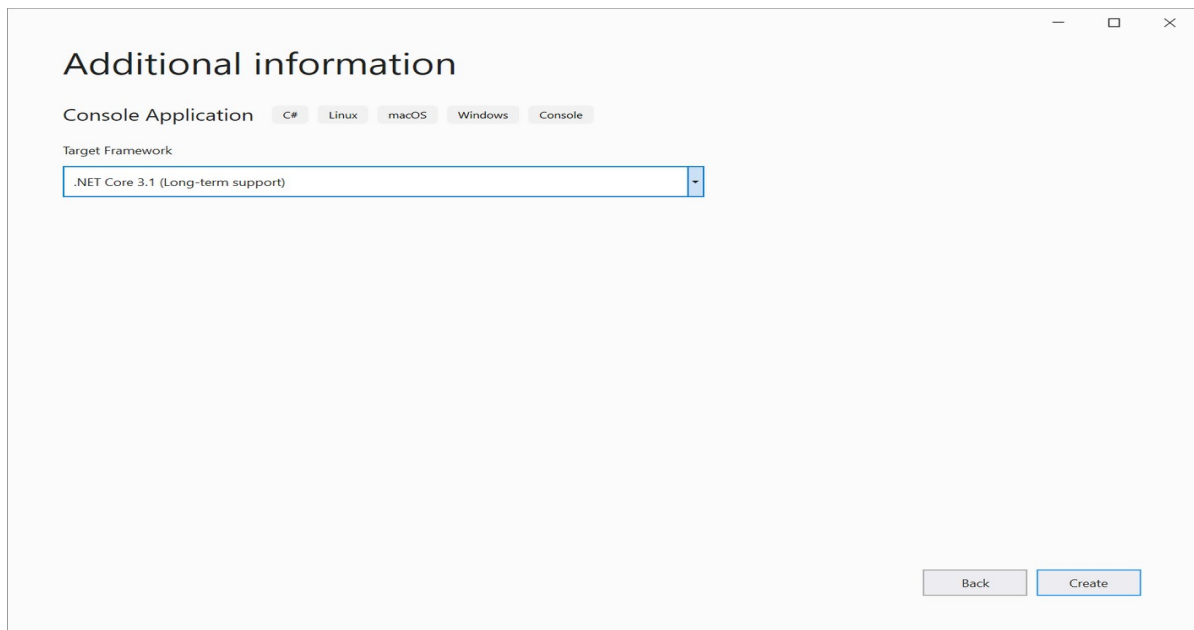
Select the **Console Application** template, and then click **Next**.



- In the **Configure your new project** window, enter **HelloWorld** in the **Project name** box, optionally change the directory location for your project files (the default locale is `C:\Users\\source\repos`), and then click **Next**.

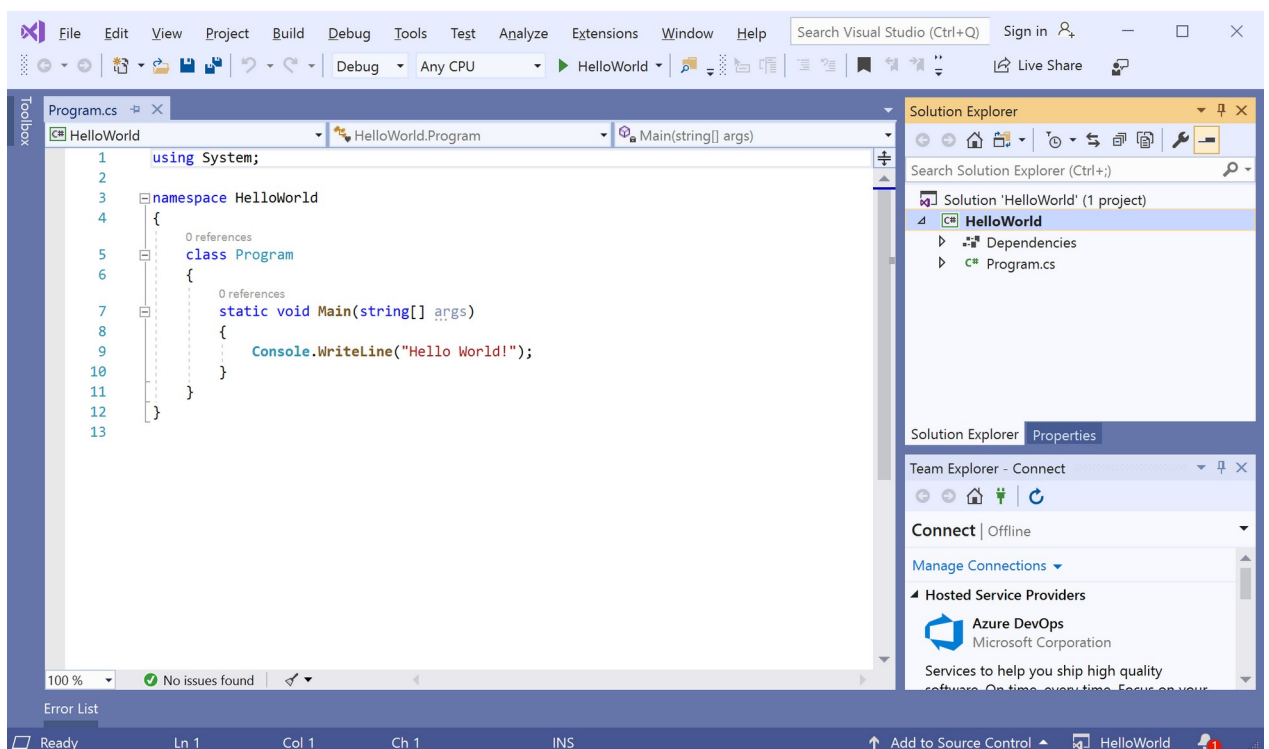


- In the **Additional information** window, verify that **.NET Core 3.1** appears in the **Target Framework** drop-down menu, and then click **Create**.



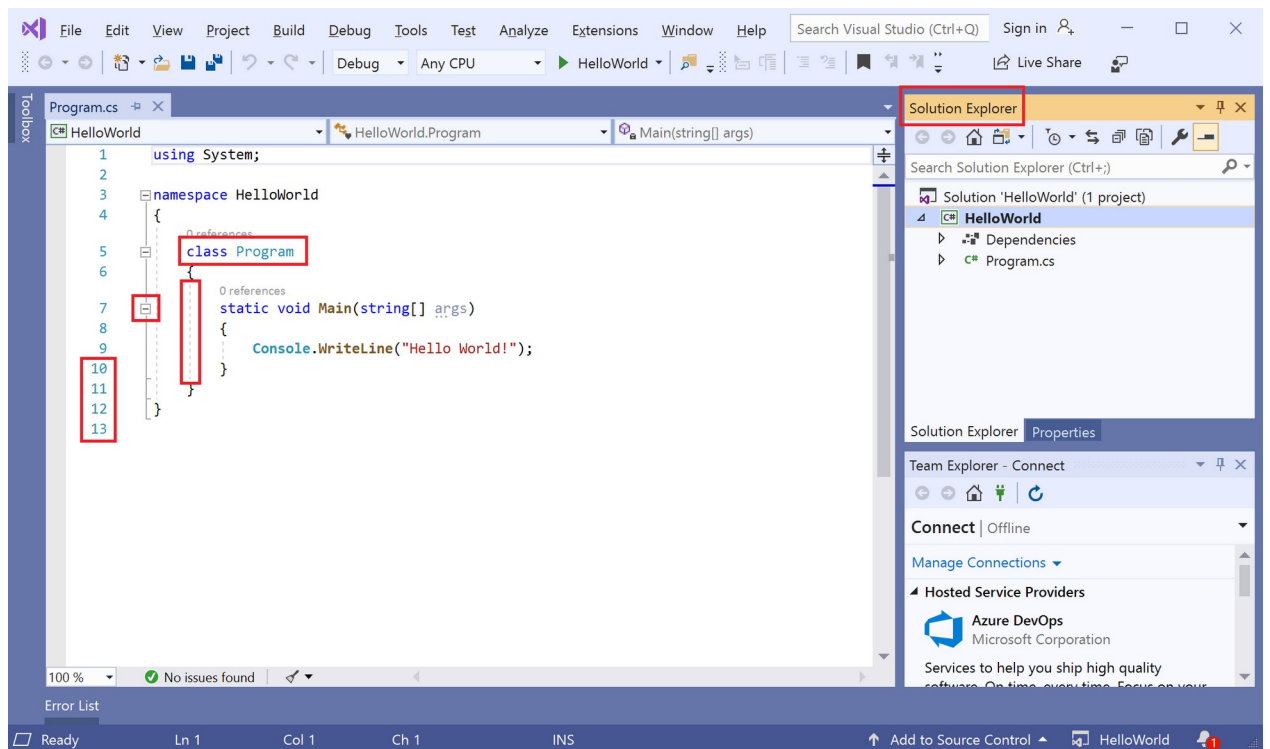
Visual Studio creates the project. It's a simple "Hello World" application that calls the `Console.WriteLine()` method to display the literal string "Hello World!" in the console (program output) window.

Shortly, you should see something like the following:



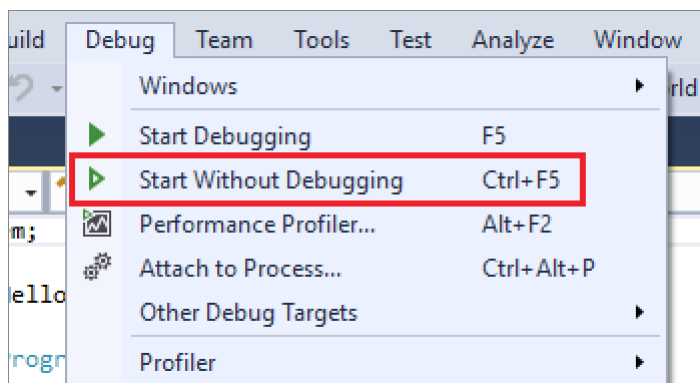
The C# code for your application shows in the editor window, which takes up most of the space. Notice that the text is automatically colorized to indicate different parts of the code, such as keywords and types. In addition, small, vertical dashed lines in the code indicate which braces match one another, and line numbers help you locate code later. You can choose the small, boxed minus signs to collapse or expand blocks of code. This code outlining feature lets you hide code you don't need, helping to minimize onscreen clutter.

The project files are listed on the right side in a window called **Solution Explorer**.

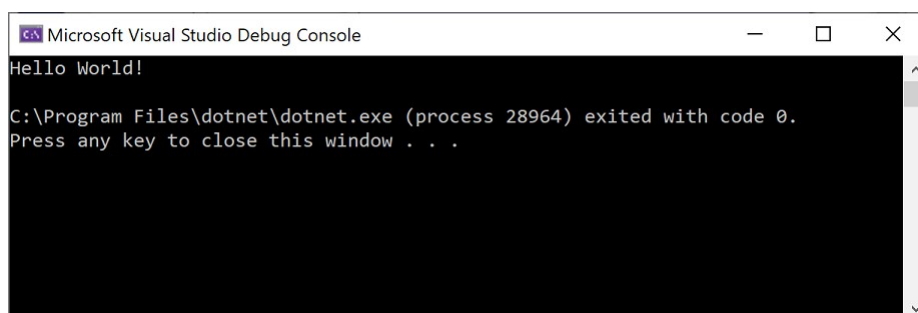


There are other menus and tool windows available, but let's move on for now.

6. Now, start the app. You can do this by choosing **Start Without Debugging** from the **Debug** menu on the menu bar. You can also press **Ctrl+F5**.



Visual Studio builds the app, and a console window opens with the message **Hello World!**. You now have a running app!



- To close the console window, press any key on your keyboard.
- Let's add some additional code to the app. Add the following C# code before the line that says `Console.WriteLine("Hello World!");`:

C#

```
• Console.WriteLine("\nWhat is your name?");  
var name = Console.ReadLine();
```

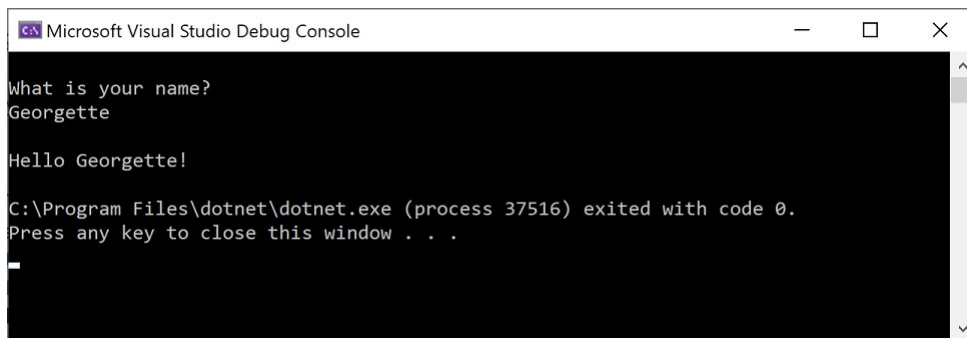
This code displays **What is your name?** in the console window, and then waits until the user enters some text followed by the **Enter** key.

- Change the line that says `Console.WriteLine("Hello World!");` to the following code:

C#

```
9. Console.WriteLine($"Hello {name}!");
```

- Run the app again by selecting **Debug > Start Without Debugging** or by pressing **Ctrl+F5**. Visual Studio rebuilds the app, and a console window opens and prompts you for your name.
- Enter your name in the console window and press **Enter**.



- Press any key to close the console window and stop the running program.

Use refactoring and IntelliSense

Let's look at a couple of the ways that refactoring and IntelliSense can help you code more efficiently.

First, let's rename the `name` variable:

- Double-click the `name` variable to select it.
- Type in the new name for the variable, **username**.

Notice that a gray box appears around the variable, and a light bulb appears in the margin.

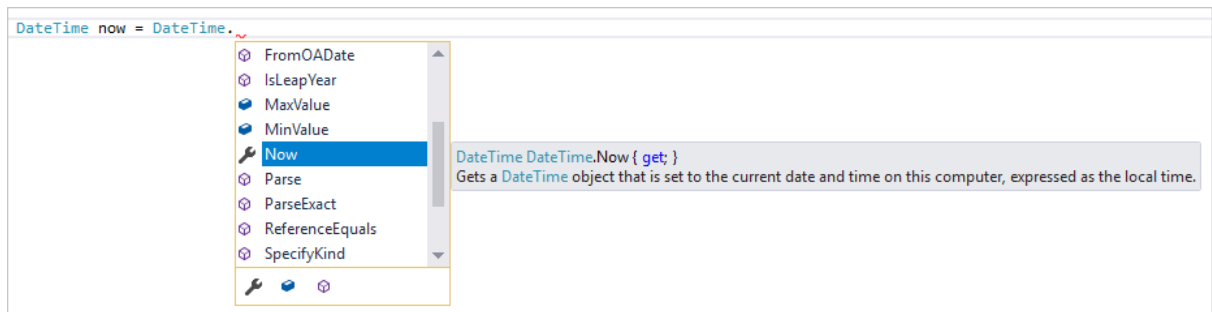
- Select the light bulb icon to show the available Quick Actions. Select **Rename 'name' to 'username'**.



The variable is renamed across the project, which in our case is only two places.

- Now let's take a look at IntelliSense. Below the line that says `Console.WriteLine($" \nHello {username}!");`, type `DateTime now = DateTime..`

A box displays the members of the `DateTime` class. In addition, the description of the currently selected member displays in a separate box.



- Select the member named **Now**, which is a property of the class, by double-clicking on it or pressing **Tab**. Complete the line of code by adding a semi-colon to the end.
- Below that, type in or paste the following lines of code:

C#

- `int dayOfYear = now.DayOfYear;`

```
Console.Write("Day of year: ");
Console.WriteLine(dayOfYear);
```

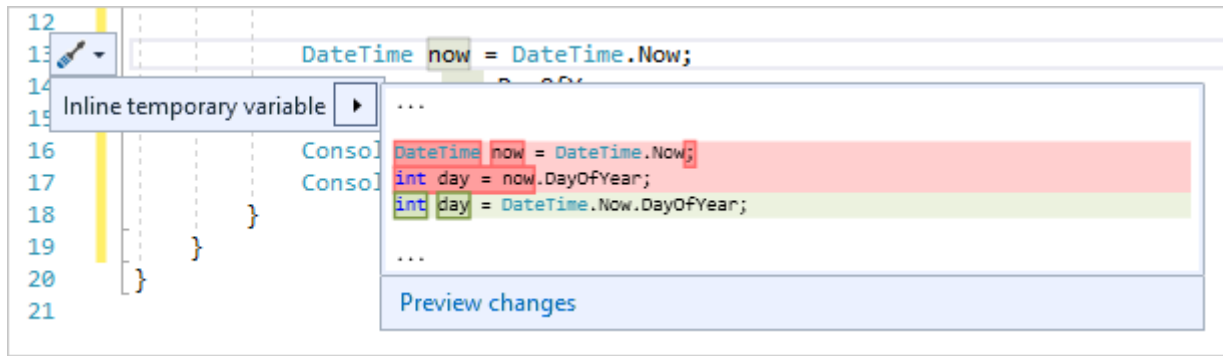
Tip

`Console.Write` is a little different to `Console.WriteLine` in that it doesn't add a line terminator after it prints. That means that the next piece of text that's sent to the output will print on the same line. You can hover over each of these methods in your code to see their description.

- Next, we'll use refactoring again to make the code a little more concise. Click on the variable `now` in the line `DateTime now = DateTime.Now;`

Notice that a little screwdriver icon appears in the margin on that line.

- Click the screwdriver icon to see what suggestions Visual Studio has available. In this case, it's showing the Inline temporary variable refactoring to remove a line of code without changing the overall behavior of the code:



6. Click **Inline temporary variable** to refactor the code.

10. Run the program again by pressing **Ctrl+F5**. The output looks something like this:

