Unit -1

Introduction to C# and .NET Framework

Course Contents [7 Hrs.]

Object Orientation; Type Safety; Memory Management; Platform Support; C# and CLR; CLR and .NET Framework; Framework Overview; .NET Standard 2.0; Applied Technologies

---

Introduction to C# Language

C# is a general-purpose, object-oriented programming language. **Anders Hejlsberg and his team developed C # during the development of .Net Framework.** It is used to create desktop, web and mobile applications.

C# is a hybrid of C and C++. It is a Microsoft programming language developed to compete with Sun's Java language. C# is an object-oriented programming language used with XML-based Web services on the .NET platform and designed for improving productivity in the development of Web applications.

C# is an elegant and type-safe object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework. You can use C# to create Windows client applications, XML Web services, distributed components, client-server applications, database applications, and much, much more. Visual C# provides an advanced code editor, convenient user interface designers, integrated debugger, and many other tools to make it easier to develop applications based on the C# language and the .NET Framework.

As an object-oriented language, C# supports the concepts of *encapsulation*, *inheritance*, and *polymorphism*. All variables and methods, including the Main method, the application's entry point, are encapsulated within class definitions.

The following reasons make C# a widely used professional language (**Features of C#**):

1. It is a modern, general-purpose programming language.
2. It is object oriented.
3. It is component oriented.
4. It is easy to learn.
5. It is a structured language.

6. It produces efficient programs.
7. It can be compiled on a variety of computer platforms.
8. It is a part of .Net Framework.

Object Orientation

C# is a rich implementation of the object-orientation paradigm, which includes encapsulation, abstraction, 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. In C#, methods are only one kind of function member, which also includes properties and events. 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* using 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 run- time.

Static typing eliminates a large class of errors before a program 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.

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 is 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.

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 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 start up 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 also type information (metadata). The presence of metadata allows assemblies to reference types in other assemblies without needing additional files. A program can query its own metadata (reflection), and even generate new IL at runtime (reflection.emit)