Introduction to .NET

What is Application Software?

Ans: Application software is commonly defined as any program or number of programs designed for end-users. In that sense, any end user program can be called an “application.” People often use the term “application software” to talk about bundles or groups of individual software applications, using a different term, “application program” to refer to individual applications. Examples of application software include items like Notepad, WordPad, Microsoft Word, Microsoft Excel, or any of the Web Browsers used to navigate the Internet, etc.

Another way to understand application software is, in a very basic sense, every program that you use on your computer is a piece of application software. The operating system, on the other hand, is system software. Historically, the application was generally born as computers evolved into systems where you could run a particular codebase on a given operating system. Even social media platforms have come to resemble applications, especially on our mobile phone devices, where individual applications are given the nickname “apps.” So while the term “application software” can be used broadly, it’s an important term in describing the rise of sophisticated computing environments.

How to develop Application software?

Ans: There are two basic camps of software development: Applications Development and Systems Development. Applications Development is focused on creating programs that meet the users' needs. These can range from mobile phone apps, video games, enterprise-level accounting software. Systems Development is focused on creating and maintaining operating systems and to do this we need to familiar with some Programming Language. Thousands of different programming languages have been created, and more are being created every year. Many programming languages are written in an imperative form (i.e., as a sequence of operations to perform) while other languages use the declarative form (i.e. the desired result is specified, not how to achieve it).

What is a Programming Language?

Ans: A programming language is a formal language comprising a set of instructions that produce various kinds of output. Programming languages are used in computer programming to implement algorithms. Most programming languages consist of instructions for computers. Since the early 1800s, programs have been used to direct the behavior of machines such as Jacquard looms, music boxes and player pianos.

“A computer programming language is a language used to write computer programs, which involves a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on.”

Anyone can come up with ideas, but a developer will be able to turn those ideas into something concrete. Even if you only want to work on the design aspects of software, you should have some familiarity with coding and be able to create basic prototypes. There are a huge variety of programming languages that we can learn.

Very early computers, such as Colossus is thus regarded as the world's first programmable, electronic, digital computer, although it was programmed by switches and plugs and not by a stored program.

Slightly later, programs could be written in machine language, where the programmer writes each instruction in a numeric form the hardware can execute directly. For example, the instruction to add the value in two memory location might consist of 3 numbers: an “opcode” that selects the “add” operation, and two memory locations. The programs, in decimal or binary form, were read in from punched cards, paper tape, and magnetic tape or toggled in on switches on the front panel of the computer. Machine languages were later termed first-generation programming languages (1GL).

The next step was development of so-called second-generation programming languages (2GL) or assembly languages, which were still closely tied to the instruction set architecture of the specific computer. These served to make the program much more human-readable and relieved the programmer of tedious and error-prone address calculations.

The first high-level programming languages, or third-generation programming languages (3GL), were written in the 1950s. John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer. Unlike machine code, Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.

At the University of Manchester, Alick Glennie developed Autocode in the early 1950s. As a programming language, it used a compiler to automatically convert the language into machine code. The first code and compiler was developed in 1952 for the Mark 1 computer at the University of Manchester and is considered to be the first compiled high-level programming language.

In 1954, FORTRAN was invented at IBM by John Backus. It was the first widely used high-level general purpose programming language to have a functional implementation, as opposed to just a design on paper. It is still a popular language for high-performance computing and is used for programs that benchmark and rank the world's fastest supercomputers.

Another early programming language was devised by Grace Hopper in the US, called FLOW-MATIC. It was developed for the UNIVAC I at Remington Rand during the period from 1955 until 1959. Hopper found that business data processing customers were uncomfortable with mathematical notation, and in early 1955, she and her team wrote a specification for an English programming language and implemented a prototype. The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959. FLOW-MATIC was a major influence in the design of COBOL.

COBOL an acronym for “common business-oriented language” is a compiled English-like computer programming language designed for business use. It is imperative, procedural and, since 2002, object-oriented. COBOL is primarily used in business, finance, and administrative systems for companies and governments. COBOL is still widely used in applications deployed on mainframe computers, such as large-scale batch and transaction processing jobs. But due to its declining popularity and the retirement of experienced COBOL programmers, programs are being migrated to new platforms, rewritten in modern languages. Most programming in COBOL is now purely to maintain existing applications.

Pascal is an imperative and procedural programming language, designed by Niklaus Wirth as a small, efficient language intended to encourage good programming practices using structured programming and data structuring. It is named in honor of the French mathematician, philosopher and physicist Blaise Pascal. Pascal enabled defining complex data types and building dynamic and recursive data structures such as lists, trees and graphs. Pascal has strong typing on all objects, which means that one type of data cannot be converted or interpreted as another without explicit conversions.

C is a general-purpose, imperative procedural computer programming language supporting structured programming, lexical variable scope, and recursion, with a static type system. By design, C provides constructs that map efficiently to typical machine instructions. It has found lasting use in applications previously coded in assembly language. Such applications include operating systems, various application software for computers that range from super computers to PLCs and embedded systems. A successor to the programming language B, C was originally developed at Bell Labs by Dennis Ritchie between 1972 and 1973 to construct utilities running on UNIX. It was applied to re-implementing the kernel of the UNIX operating system. During the 1980s, C gradually gained popularity. It has become one of the most widely used programming languages, with C compilers from various vendors available for the majority of existing computer architectures and operating systems. C has been standardized by the ANSI since 1989 (ANSI C) and by the International Organization for Standardization (ISO).

C++ is a general-purpose programming language developed by Danish computer scientist Bjarne Stroustrup at Bell Labs since 1979 as an extension of the C programming language, or "C with Classes" as he wanted an efficient and flexible language similar to C that also provided high-level features for program organization. The language has expanded significantly over time, and modern C++ now has object-oriented, generic, and functional features in addition to facilities for low-level memory manipulation. It is almost always implemented as a compiled language, and many vendors provide C++ compilers, including the Free Software Foundation, LLVM, Microsoft, Intel, Oracle, and IBM, so it is available on many platforms. C++ has also been found useful in many contexts, with key strengths being software infrastructure and resource-constrained applications, including desktop applications, video games, servers (e.g. e-commerce, Web search, or SQL Servers), and performance-critical applications (e.g. telephone switches or space probes).

Objective-C is a general-purpose, object-oriented programming language that adds Smalltalk-style messaging to the C programming language. It was the main programming language supported by Apple for macOS, iOS, and their respective application programming interfaces (APIs). The language was originally developed in the early 1980s. It was later selected as the main language used by NeXT for its NeXTSTEP operating system, from which macOS and iOS are derived. Objective-C source code 'implementation' program files usually have .m filename extensions, while Objective-C 'header/interface' files have .h extensions, the same as C header files. Objective-C++ files are denoted with a .mm file extension.

Python is an interpreted, high-level, general-purpose programming language. Created by Guido van Rossum and first released in 1991. Its language constructs and object-oriented approach aim to help programmers write clear, logical code for small and large-scale projects. Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly, procedural), object-oriented, and functional programming. Python was conceived in the late 1980s as a successor to the ABC language. Python 2.0 released in 2000 and Python 3.0, released in 2008, was a major revision of the language that is not completely backward-compatible, i.e. Python 2 code does not run unmodified on Python 3. The Python 2 language was officially discontinued in 2020 (first planned for 2015) and now only Python 3.5.x and later are supported.

Java is a general-purpose programming language that is class-based and object-oriented, and designed to have as few implementation dependencies as possible. It is intended to let application developers write once, run anywhere (WORA), meaning that compiled Java code can run on all platforms that support Java without the need of recompilation. Java applications are typically compiled to byte code that can run on any Java virtual machine (JVM) regardless of the underlying computer architecture. The syntax of Java is similar to C and C++. Java was originally developed by James Gosling at Sun Microsystems (which has since been acquired by Oracle) and released in 1995 as a core component of Sun Microsystems' Java platform.

C# (pronounced see sharp, like the musical note C♯, but written with the number sign) is a general-purpose, multi-paradigm programming language encompassing strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed around 2000 by Microsoft as part of its .NET initiative and later approved as an international standard by ECMA in 2002 and ISO in 2003. C# was designed by Anders Hejlsberg, and its development team is currently led by “Mads Torgersen”. The most recent version is 9.0, which was released on November 2020 alongside Visual Studio 2019.

What is .NET?

Ans: .NET is a free, cross-platform, open source developer platform for building many different types of applications like Desktop, Web, Mobile, Games and IOT by using multiple languages, editors, and libraries.

What is a Platform?

Ans: It is the environment in which a piece of software is executed. A platform can also be called as the stage on which computer programs can run. Platform can refer to the type of processor (CPU) on which a given operating system runs, the type of operating system on a computer or the combination of the type of hardware and the type of operating system running on it. An example of a common platform is Microsoft Windows running on x86 architecture. Other well-known desktop computer platforms include Linux/Unix and macOS

What is Cross-platform?

Ans: In computing, cross-platform software (also multi-platform software or platform-independent software) is computer software that is implemented to run on multiple platforms. For example, a cross-platform application may run on Microsoft Windows, Linux, and macOS. Cross-platform programs may run on as many as all existing platforms, or on few platforms.

What is meant by developing applications using multiple languages?

Ans: .NET languages are programming languages that are used to produce libraries and programs that conform to the Common Language Infrastructure (CLI) specifications. Most of the CLI languages compile entirely to the Common Intermediate Language (CIL), an intermediate language that can be executed using the Common Language Runtime, implemented by .NET Framework, .NET Core, and Mono. As the program is being executed, the CIL code is just-in-time compiled to the machine code appropriate for the architecture on which the program is running. While there are currently over 30+ languages in .NET, but only a small number of them are widely used and supported by Microsoft. List of .NET languages include: C#, F#, Visual Basic, C++, Iron Python, etc and the most popular and widely used language as a developer choice is C#. Visit the following link to view the list of .NET Languages: https://microsoft.fandom.com/wiki/Microsoft_.NET_Languages

What is CLI (Common Language Infrastructure)?

Ans: The Common Language Infrastructure (CLI) is an open specification (technical standard) developed by Microsoft and standardized by ISO (International Organization for Standardization) and ECMA (European Computer Manufacturers Association) that describes about executable code and a runtime environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures. This implies it is platform independent. The .NET Framework, .NET Core and Mono are implementations of the CLI.

CLI specification describes the following four aspects:

1. **The Common Language Specification (CLS):** A set of base rules to which any language targeting the CLI should conform in order to interoperate to other CLS-compliant languages. The CLS rules define a subset of the Common Type System.

2. **The Common Type System (CTS):** A set of data types and operations that are shared by all CTS-compliant programming languages. According to this all .NET Languages has to adopt the rule "Uniform Data Type Structure" i.e. similar data types must be same in size in all Languages of .NET.

3. **The Metadata:** Information about program structure is language - independent, so that it can be referenced between languages and tools', making it easy to work with code written in a language the developer is not aware.

4. **The Virtual Execution System (VES):** The VES loads and executes CLI-compatible programs. All compatible .NET languages compile to Common Intermediate Language (CIL), which is an intermediate code that is abstracted from the platform hardware. When the code is executed, the platform-specific VES will compile the CIL to the machine language according to the specific hardware and operating system.

What is .NET Framework and .NET Core?

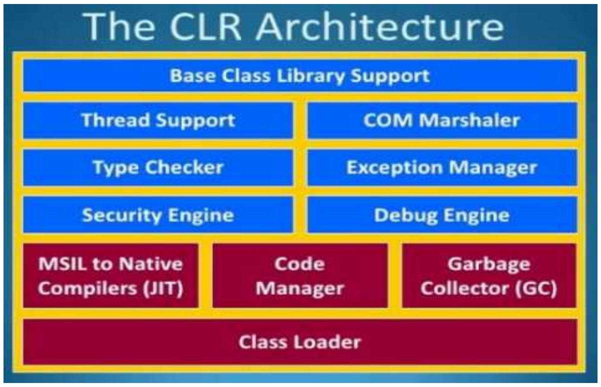
Ans: .NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications. There are various implementations of .NET and each implementation allows .NET code to execute in different places - Linux, macOS, Windows, iOS, Android, and many more. Various implementations of the .NET include:

1. **.NET Framework:** it is the original implementation of .NET and it supports running websites, services, desktop apps, and more on Windows.
2. **.NET Core:** it is a cross-platform implementation for running websites, services, and console apps on Windows, Linux, and macOS.
3. **Xamarin/Mono:** it is a .NET implementation for running apps on all the major mobile operating systems, including iOS and Android.

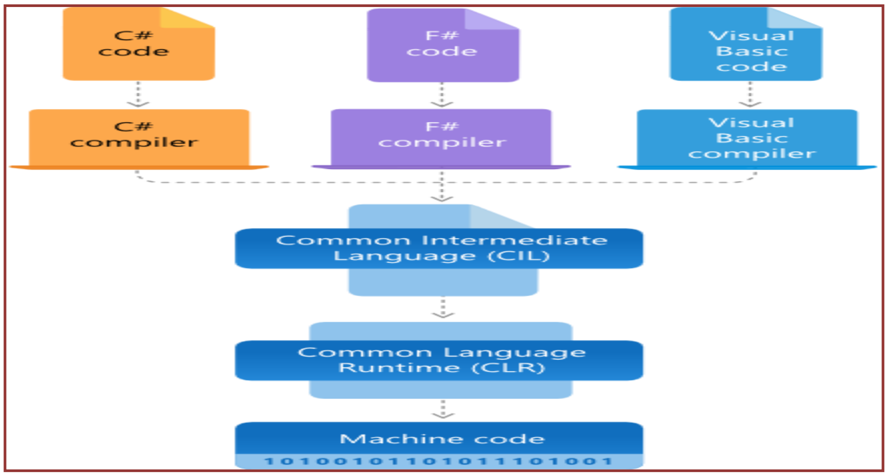
Architecture of .NET Framework: The two major components of .NET Framework are the .NET Framework Class Library and the Common Language Runtime.

1. The Class Library provides a set of APIs and types for common functionality. It provides types for strings, dates, numbers, etc. The Class Library includes APIs for reading and writing files, connecting to databases, drawing, and more.
2. The Common Language Runtime (CLR) is the heart of .NET Framework and the execution engine that handles running applications. It provides services like thread management, garbage collection, type-safety, exception handling, and more.

Architecture of .NET Framework CLR:



.NET applications can be written in any .NET Language like C#, F#, or Visual Basic. Source Code we write by using some .NET Language is compiled into a language-agnostic Common Intermediate Language (CIL) and the compiled code is stored as assemblies (files with a ".dll" or ".exe" extension). When we run the applications, CLR takes the assemblies and uses a just-in-time compiler (JIT) to turn it into machine code that can execute on the specific architecture of the computer it is running on.



.NET Framework FAQ's

What is .NET Framework used for?

Ans: .NET Framework is used to create and run software applications. .NET apps can run on many operating systems, using different implementations of .NET. .NET Framework is used for running .NET apps on Windows.

Who uses .NET Framework?

Ans: Software developers and the users of their applications both use .NET Framework:

- Users need to install .NET Framework to run application built with the .NET Framework. In most cases, .NET Framework is already installed with Windows. If needed, you can download .NET Framework.
- Software developers use .NET Framework to build many different types of applications - websites, services, desktop apps, and more with Visual Studio. Visual Studio is an integrated development environment (IDE) that provides development productivity tools and debugging capabilities. See the .NET customer showcase for examples of what people are building with .NET.

Why do I need .NET Framework?

Ans: You need .NET Framework installed in order to run applications on Windows that were created using .NET Framework. It is already included in many versions of Windows. You only need to download and install .NET Framework if prompted to do so.

How does .NET Framework work?

Ans: .NET Framework applications can be written in many languages like C#, F#, or Visual Basic and compiled to Common Intermediate Language (CIL). The Common Language Runtime (CLR) runs .NET applications on a given machine, converting the CIL to machine code. See Architecture of .NET Framework for more info.

What are the main components/features of .NET Framework?

Ans: The two major components of .NET Framework are the Common Language Runtime (CLR) and the .NET Framework Class Library. The CLR is the execution engine that handles running applications. The Class Library provides a set of APIs and types for common functionality.

How many versions do we have for .NET Framework?

Ans: There are multiple versions of .NET Framework but each new version adds new features but retains features from previous versions. List of .NET Framework Versions:

.NET Framework 1.0	.NET Framework 1.1	.NET Framework 2.0	.NET Framework 3.0
.NET Framework 3.5	.NET Framework 4	.NET Framework 4.5	.NET Framework 4.5.1
.NET Framework 4.5.2	.NET Framework 4.6	.NET Framework 4.6.1	.NET Framework 4.6.2
.NET Framework 4.7	.NET Framework 4.7.1	.NET Framework 4.7.2	.NET Framework 4.8

Can you have multiple .NET Frameworks installed?

Ans: Some versions of .NET Framework are installed side-by-side, while others will upgrade an existing version (known as an in-place update). In-place updates occur when two .NET Framework versions share the same CLR version. For example, installing .NET Framework 4.8 on a machine with .NET Framework 4.7.2 and 3.5 installed will perform an in-place update of the 4.7.2 installation and leave 3.5 installed separately.

.NET Framework Version	CLR Version
.NET Framework 4.x	4.0
.NET Framework 2.x and 3.x	2.0
.NET Framework 1.1	1.1
.NET Framework 1.0	1.0

How much does .NET Framework cost?

Ans: .NET Framework is free, like the rest of the .NET platform. There are no fees or licensing costs, including for commercial use.

Which version of .NET Framework should I use?

Ans: In most cases, you should use the latest stable release and currently, that's .NET Framework 4.8. Applications that were created with any 4.x version of .NET Framework will run on .NET Framework 4.8. To run an application that was created for an earlier version (for example, .NET Framework 3.5), you should install that version.

What is the support policy for .NET Framework?

Ans: .NET Framework 4.8 is the latest version of .NET Framework and will continue to be distributed with future releases of Windows. As long as it is installed on a supported version of Windows, .NET Framework 4.8 will continue to also be supported.

Can customers continue using the .NET Framework and get support?

Ans: Yes. Many products both within and outside Microsoft rely on .NET Framework. The .NET Framework is a component of Windows and receives the same support as Windows version which it ships with or on which it is installed. .NET Framework 4.8 is the latest version of .NET Framework and will continue to be distributed with future releases of Windows. As long as it is installed on a supported version of Windows, .NET Framework 4.8 will continue to also be supported.

Architecture of .NET Core: The two main components of .NET Core are CoreCLR and CoreFX, respectively, which are comparable to the Common Language Runtime (CLR) and the Framework Class Library (FCL) of the .NET Framework's Common Language Infrastructure (CLI) implementation.

1. CoreFX is the foundational class libraries for .NET Core. It includes types for collections, file systems, console, JSON, XML and many others.
2. CoreCLR is the .NET execution engine in .NET Core, performing functions such as garbage collection and compilation to machine code. As a CLI implementation of Virtual Execution System (VES), CoreCLR is a complete runtime and virtual machine for managed execution of .NET programs and includes a just-in-time compiler called RyuJIT.

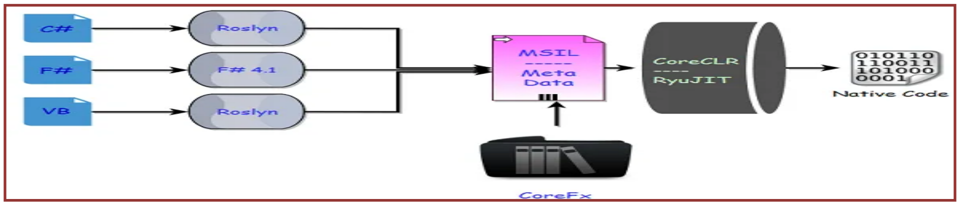
Note: .NET Core releases have a single product version, that is, there is no separate CLR version.

What is CoreFX?

Ans: CoreFX, also referred to as the Unified Base Class Library, consists of the basic and fundamental classed that form the core of the .Net Core platform. These set of libraries comprise the System.* (and to a limited extent Microsoft.*) namespaces. Majority of the .NET Core APIs are also available in the .NET Framework, so you can think of CoreFX as an extension of the .NET Framework Class Library.

What is CoreCLR?

Ans: CoreCLR is the .NET execution engine in .NET Core which is a complete runtime and virtual machine for managed execution of .NET programs and includes a just-in-time compiler called RyuJIT, performing functions such as garbage collection and compilation to machine code. CoreCLR is built from the same code base of the Framework CLR.



What is Roslyn?

Ans: Roslyn is the codename-that-stuck for the open-source compiler for C# and Visual Basic.NET. It is an open source, cross-platform, public language engine for C# and VB. The conversations about Roslyn were already ongoing when “Mads Torgersen” joined Microsoft in 2005 - just before .NET 2.0 would ship. That conversation was about rewriting C# in C# which is a normal practice for programming languages. But there was a more practical and important motivation: the creators of C# were not programming in C# themselves, they were coding in C++.

.NET CORE FAQ’s

What is .NET Core?

Ans: The .NET Core platform is a new .NET stack that is optimized for open source development. .NET Core has two major components. It includes a runtime that is built from the same codebase as the .NET Framework CLR. The .NET Core runtime includes the same GC and JIT (RyuJIT), but doesn’t include features like Application Domains or Code Access Security. .NET Core also includes the base class libraries. These libraries are the same code as the .NET Framework class libraries, but have been factored to enable to ship as smaller set of libraries. .NET Core refers to several technologies including ASP.NET Core, Entity Framework Core, and more.

What are the characteristics of .NET Core?

Ans: .NET Core has the following characteristics:

- **Cross platform:** Runs on Windows, macOS, and Linux operating systems.
- **Open source:** The .NET Core framework is open source, using MIT and Apache 2 licenses. .NET Core is a .NET Foundation project.

- **Modern:** It implements modern paradigms like asynchronous programming, no-copy patterns using struts', and resource governance for containers.
- **Performance:** Delivers high performance with features like hardware intrinsics, tiered compilation, and Span<T>.
- **Consistent across environments:** Runs your code with the same behavior on multiple operating systems and architectures, including x64, x86, and ARM.
- **Command-line tools:** Includes easy-to-use command-line tools that can be used for local development and for continuous integration.
- **Flexible deployment:** You can include .NET Core in your app or install it side-by-side (user-wide or system-wide installations). Can be used with Docker containers.

What is the composition of .NET Core?

Ans: NET Core is composed of the following parts:

- The .NET Core runtime, which provides a type system, assembly loading, a garbage collector, native interop, and other basic services. .NET Core framework libraries provide primitive data types, app composition types, and fundamental utilities.
- The ASP.NET Core runtime, which provides a framework for building modern, cloud-based, internet-connected apps, such as web apps, IOT apps, and mobile backend.
- The .NET Core SDK and language compilers (Roslyn and F#) that enable the .NET Core developer experience.
- The dotnet command, which is used to launch .NET Core apps and CLI commands. It selects and hosts the runtime, provides an assembly loading policy, and launches apps and tools.

What is .NET Core SDK?

Ans: The .NET Core SDK (Software Development Kit) includes everything you need to build and run .NET Core applications using command line tools or any editor like Visual Studio. It also contains a set of libraries and tools that allow developers to create .NET Core applications and libraries. It contains the following components that are used to build and run applications:

1. The .NET Core CLI.
2. .NET Core libraries and runtime.
3. The dotnet driver.

What is .NET Core Runtime?

Ans: This includes everything you need to run a .NET Core Application. The runtime is also included in the SDK. When an app author publishes an app, they can include the runtime with their app. If they don't include the runtime, it's up to the user to install the runtime. There are three different runtimes you can install on Windows:

- ASP.NET Core runtime: Runs ASP.NET Core apps. Includes the .NET Core runtime.
- Desktop runtime: Runs .NET Core WPF and .NET Core Windows Forms desktop apps for Windows. Includes the .NET Core runtime.
- .NET Core runtime: This runtime is the simplest runtime and doesn't include any other runtime. It's highly recommended that you install both ASP.NET Core runtime and Desktop runtime for the best compatibility with .NET Core apps.

What's the difference between SDK and Runtime in .NET Core?

Ans: The SDK is all of the stuff that is needed for developing a .NET Core application easier, such as the CLI and a compiler. The runtime is the “virtual machine” that hosts/runs the application and abstracts all the interaction with the base operating system.

What is the difference between .NET Core and .NET Framework?

Ans: .NET Core and .NET Framework share many of the same components and you can share code across the two. Some key differences include:

- .NET Core is cross-platform and runs on Linux, macOS, and Windows. .NET Framework only runs on Windows.
- .NET Core is open-source and accepts contributions from the community. The .NET Framework source code is available, but does not take direct contributions.
- The majority of .NET innovation happens in .NET Core.
- .NET Framework is included in Windows and automatically updated machine-wide by Windows Update. .NET Core is shipped independently.

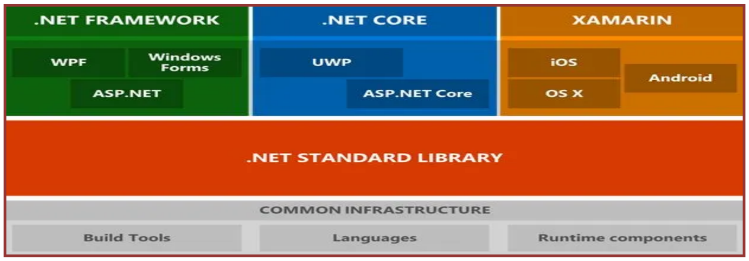
What is the difference between .NET Core and Mono?

Ans: To be simple, Mono is third party implementation of .Net Framework for Linux/Android/iOS and .Net Core is Microsoft's own implementation for same.

What's the difference between .NET Core, .NET Framework, and Xamarin?

Ans: difference between .NET Core, .NET Framework and Xamarin are:

- .NET Framework is the “traditional” flavor of .NET that's distributed with Windows. Use this when you are building a desktop Windows or UWP app, or working with older ASP.NET 4.8.
- .NET Core is cross-platform .NET that runs on Windows, Mac, and Linux. Use this when you want to build console or web apps that can run on any platform, including inside Docker containers.
- Xamarin is used for building mobile apps that can run on iOS, Android, or Windows Phone devices.



What is the support policy to .NET Core?

Ans: .NET Core is supported by Microsoft on Windows, macOS, and Linux. It's updated for security and quality regularly (the second Tuesday of each month). .NET Core binary distributions from Microsoft are built and tested on Microsoft-maintained servers in Azure and follow Microsoft engineering and security practices.

Red Hat supports .NET Core on Red Hat Enterprise Linux (RHEL). Red Hat builds .NET Core from source and makes it available in the Red Hat Software Collections. Red Hat and Microsoft collaborate to ensure that .NET Core works well on RHEL (Red Hat Enterprise Linux).

Tizen (developed by Samsung) supports .NET Core on Tizen platforms.

How much does .NET Core cost?

Ans: .NET Core is an open-source and cross-platform version of .NET that is maintained by Microsoft and the .NET community on GitHub. All aspects of .NET Core are open-source including class libraries, runtime, compilers, languages, ASP.NET Core web framework, Windows desktop frameworks, and Entity Framework Core data access library. There are no licensing costs, including for commercial use.

What is GitHub?

Ans: GitHub is a code hosting platform for collaboration and version control. It is a repository (usually abbreviated to "repo") is a location where all the files for a particular project are stored which lets you (and others) work together on projects. Each project has its own repo, and you can access it with a unique URL. Git is an open-source version control system that was started by "Linus Torvalds" - the same person who created Linux. Git is similar to other version control systems—Subversion, CVS, and Mercurial to name a few.

What is the release schedule for .NET Core?

Ans: .NET Core 2.1 and .NET Core 3.1 are the current LTS releases made available on August 2018 and December 2019, respectively. After .NET Core 3.1, the product will be renamed to .NET and LTS releases will be made available every other year in November. So, the next LTS release will be .NET 6, which will ship in November 2021. This will help customers plan upgrades more effectively.

How many versions do we have for .NET Core?

Ans: This table tracks release dates and end of support dates for .NET Core versions.

Version	Original Release Date	Support Level	End of Support
.NET Core 3.1	December 3, 2019	LTS	December 3, 2022
.NET Core 3.0	September 23, 2019	EOL	March 3, 2020
.NET Core 2.2	December 4, 2018	EOL	December 23, 2019
.NET Core 2.1	May 30, 2018	LTS	August 21, 2021
.NET Core 2.0	August 14, 2017	EOL	October 1, 2018
.NET Core 1.1	November 16, 2016	EOL	June 27, 2019
.NET Core 1.0	June 27, 2016	EOL	June 27, 2019

EOL (end of life) releases have reached end of life, meaning it is not longer supported and recommended moving to a supported version.

LTS (long-term support) releases have an extended support period. Use this if you need to stay supported on the same version of .NET Core for longer.

.NET 5 (.NET Core vNext)

.NET 5 is the next step forward with .NET Core. This new project and direction are a game-changer for .NET. With .NET 5, your code and project files will look and feel the same no matter which type of app you're building. You'll have access to the same runtime, API and language capabilities with each app. The project aims to improve .NET in a few key ways:

- Produce a single .NET runtime and framework that can be used everywhere and that has uniform runtime behaviors and developer experiences.
- Expand the capabilities of .NET by taking the best of .NET Core, .NET Framework, Xamarin and Mono.
- Build that product out of a single code-base that developers (Microsoft and the community) can work on and expand together and that improves all scenarios.

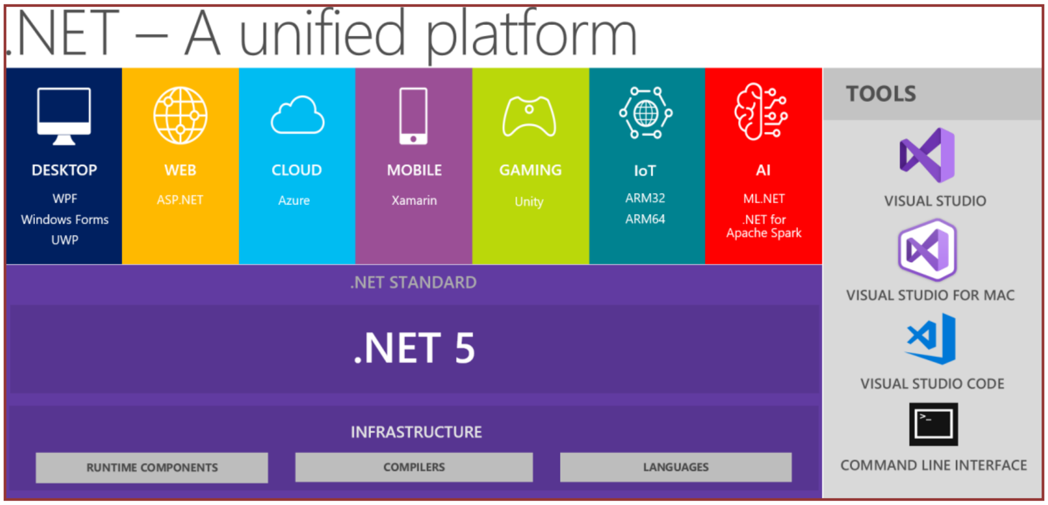
Microsoft skipped the version 4 because it would confuse users that are familiar with the .NET Framework, which has been using the 4.x series for a long time. Additionally, they wanted to clearly communicate that .NET 5 is the future for the .NET platform. They are also taking the opportunity to simplify naming. They thought that if there is only one .NET going forward, they don't need a clarifying term like "Core". The shorter name is a simplification and also communicates that .NET 5 has uniform capabilities and behaviors. Feel free to continue to use the ".NET Core" name if you prefer it.

Runtime experiences:

Mono is the original cross-platform implementation of .NET. It started out as an open-source alternative to .NET Framework and transitioned to targeting mobile devices as iOS and Android devices became popular. Mono is the runtime used as part of Xamarin.

CoreCLR is the runtime used as part of .NET Core. It has been primarily targeted at supporting cloud applications, including the largest services at Microsoft, and now is also being used for Windows desktop, IoT and machine learning applications.

Taken together, the .NET Core and Mono runtimes have a lot of similarities (they are both .NET runtimes after all) but also valuable unique capabilities. It makes sense to make it possible to pick the runtime experience you want. They are in the process of making CoreCLR and Mono drop-in replacements for one another and will make it as simple as a build switch to choose between the different runtime options.



.NET – A unified platform

DESKTOP
WPF
Windows Forms
UWP

WEB
ASP.NET

CLOUD
Azure

MOBILE
Xamarin

GAMING
Unity

IoT
ARM32
ARM64

AI
ML.NET
.NET for Apache Spark

.NET STANDARD

.NET 5

INFRASTRUCTURE

RUNTIME COMPONENTS

COMPILERS

LANGUAGES

TOOLS

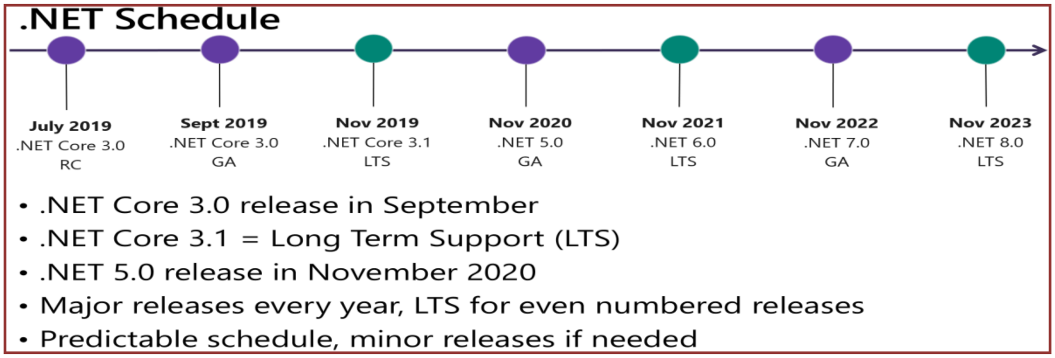
VISUAL STUDIO

VISUAL STUDIO FOR MAC

VISUAL STUDIO CODE

COMMAND LINE INTERFACE

.NET Schedule: .NET 5 is shipped in November 2020, and then they intend to ship a major version of .NET once a year, every November:



.NET Schedule

July 2019
.NET Core 3.0
RC

Sept 2019
.NET Core 3.0
GA

Nov 2019
.NET Core 3.1
LTS

Nov 2020
.NET 5.0
GA

Nov 2021
.NET 6.0
LTS

Nov 2022
.NET 7.0
GA

Nov 2023
.NET 8.0
LTS

- .NET Core 3.0 release in September
- .NET Core 3.1 = Long Term Support (LTS)
- .NET 5.0 release in November 2020
- Major releases every year, LTS for even numbered releases
- Predictable schedule, minor releases if needed

The .NET 5 project is an important and exciting new direction for .NET. You will see .NET become simpler but also have broader and more expansive capability and utility. All new development and feature capabilities will be part of .NET 5, including new C# versions. We see a bright future ahead in which you can use the same .NET APIs and languages to target a broad range of application types, operating systems, and chip architectures. It will be easy to make changes to your build configuration to build your applications differently, in Visual Studio, Visual Studio for Mac, Visual Studio Code, and Azure DevOps or at the command line.

.NET 5.0 is the next major release of .NET Core following 3.1. They named this new release .NET 5.0 instead of .NET Core 4.0 for two reasons:

1. They skipped version numbers 4.x to avoid confusion with .NET Framework 4.x.
2. They dropped "Core" from the name to emphasize that this is the main implementation of .NET going forward. .NET 5.0 supports more types of apps and more platforms than .NET Core or .NET Framework.

Note: ASP.NET Core 5.0 is based on .NET 5.0 but retains the name "Core" to avoid confusing with ASP.NET MVC 5. Likewise, Entity Framework Core 5.0 retains the name "Core" to avoid confusing it with Entity Framework 5 and 6.

The .NET 5 project is an important and exciting new direction for .NET. You will see .NET become simpler but also have broader and more expansive capability and utility. All new development and feature capabilities will be part of .NET 5, including new C# versions.

We see a bright future ahead in which you can use the same .NET APIs and languages to target a broad range of application types, operating systems, and chip architectures. It will be easy to make changes to your build configuration to build your applications differently, in Visual Studio, Visual Studio for Mac, Visual Studio Code, Azure DevOps or at the command line.

C# Programming Langauge

C# (pronounced see sharp, like the musical note ♯, but written with the number sign is a general-purpose, programming language encompassing strong typing, lexically scoped, imperative, declarative, functional, generic, object-oriented (class-based), and component-oriented programming disciplines. It was developed around 2000 by Microsoft as part of its .NET initiative and later approved as an international standard by ECMA (European Computers Manufacturing Association) in 2002 and ISO (International Organization for Standardization) in 2003.

The name "C Sharp" was inspired by the musical notation where a sharp indicates that the written note should be made a semitone i.e. higher in pitch. This is similar to the language name of C++, where "++" indicates that a variable should be incremented by 1 after being evaluated. The sharp symbol also resembles a ligature of four "+" symbols (in a two-by-two grid), further implying that the language is an increment of C++. Due to technical limitations of display and the fact that the sharp symbol is not present on most keyboard layouts, the number sign "#" was choosen to approximate the sharp symbol in the written name of the programming language.

C# was designed by Anders Hejlsberg, and its development team is currently led by Mads Torgersen. C# has Procedural, Object Oriented syntax based on C++ and includes influences from several programming languages, most importantly Delphi and Java with a particular emphasis on simplification. The most recent stable version is 9.0, which was released in November 10, 2020.

History: During the development of the .NET, the libraries were originally written using a managed code compiler system called "Simple Managed C" (SMC). In January 1999, Anders Hejlsberg formed a team to build a new language at the time called "COOL", which stood for "C-like Object Oriented Language". Microsoft had considered keeping the name "COOL" as the final name of the language, but chose not to do so for trademark reasons. By the time .NET project was publicly announced at the July 2000 in Professional Developers Conference, the language had been renamed "C#", and the libraries and ASP.NET runtime had been ported to "C#". Anders Hejlsberg is C#'s principal designer and lead architect at Microsoft, and was previously involved with the design of Turbo Pascal, Borland Delphi and Visual J++. In interviews and technical papers he has stated that flaws in most major programming languages (e.g. C++, Java, Delphi, and Smalltalk) drove the design of the C# language.

Design Goals: The ECMA standard lists these design goals for C#.

- The language is intended to be a simple, modern, general-purpose and object-oriented programming language.
- The language, and implementations thereof, should provide support for software engineering principles such as strong type checking, array bounds checking, detection of attempts to use uninitialized variables, and automatic garbage collection.
- Support for internationalization is very important.
- The language is intended for use in developing software components suitable for deployment in distributed environments.
- Portability is very important for programmers, especially those already familiar with C and C++.
- C# is intended to be suitable for writing applications for both hosted and embedded systems, ranging from the very large that use sophisticated operating systems, down to the very small having dedicated functions.

Versions of the language: 1.0, 1.2, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0.

New features in C# 2.0:

- Generics
- Partial types
- Anonymous methods
- Iterators
- Nullable value types
- Getter/setter separate accessibility
- Static classes
- Delegate inference
- Null coalescing operator

New features in C# 3.0:

- Implicitly typed local variables
- Object initializers
- Collection initializers
- Auto-Implemented properties
- Anonymous types
- Extension methods
- Query expressions
- Lambda expressions
- Expression trees
- Partial methods

New features in C# 4.0:

- Dynamic binding
- Named and optional arguments
- Generic covariant and contravariant

- Embedded interop types

New features in C# 5.0:

- Asynchronous methods
- Caller info attributes

New features in C# 6.0:

- Static imports
- Exception filters
- Auto-property initializers
- Default values for getter-only properties
- Expression bodied members
- Null propagator
- String interpolation
- nameof operator
- Index initializers
- Await in catch/finally blocks

New features in C# 7.0:

- Out variables
- Tuples and deconstruction
- Pattern matching
- Local functions
- Expanded expression bodied members
- Ref locals and returns
- Discards
- Binary Literals and Digit Separators
- Throw expressions

New features in C# 7.1:

- Async main method
- Default literal expressions
- Inferred tuple element names
- Pattern matching on generic type parameters

New features in C# 7.2:

- Techniques for writing safe efficient code
- Non-trailing named arguments
- Leading underscores in numeric literals
- private protected access modifier
- Conditional ref expressions

New features in C# 7.3:

- Accessing fixed fields without pinning
- Reassigning ref local variables
- Using initializers on stackalloc arrays

- Using fixed statements with any type that supports a pattern
- Using additional generic constraints

New features in C# 8.0:

- Readonly members
- Default interface methods
- Pattern matching enhancements:
    - Switch expressions
    - Property patterns
    - Tuple patterns
    - Positional patterns
- Using declarations
- Static local functions
- Disposable ref structs
- Nullable reference types
- Asynchronous streams
- Asynchronous disposable
- Indices and ranges
- Null-coalescing assignment
- Unmanaged constructed types
- Stackalloc in nested expressions
- Enhancement of interpolated verbatim strings

New features in C# 9.0 (Supported on .NET 5 only):

- Records
- Init only setters
- Top-level statements
- Pattern matching enhancements
- Native sized integers
- Function pointers
- Suppress emitting localsinit flag
- Target-typed new expressions
- static anonymous functions
- Target-typed conditional expressions
- Covariant return types
- Extension GetEnumerator support for foreach loops
- Lambda discard parameters
- Attributes on local functions
- Module initializers
- New features for partial methods

.NET Framework, .NET Core and .NET 5 support for C# language versions:

Target framework	version	C# language version
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0

.NET Core	2.x	C# 7.3
.NET Framework	all	C# 7.3

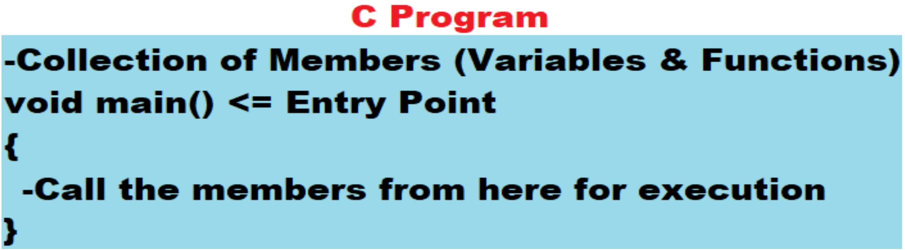
Writing a program by using different Programming Approaches

To write a program we generally follow 2 different approaches in the industry:

1. Procedural Programming Approach
2. Object Oriented Programming Approach

Procedural Programming Approach: This is a very traditional approach followed by the industry to develop applications till 70's. E.g.: COBOL, Pascal, FORTRAN, C, etc.

In this approach a program is a collection of members like variables and functions, and the members that are defined inside the program should be explicitly called for execution and we do that calling from “main” function because it is the entry point of any program that is developed by using any programming language.



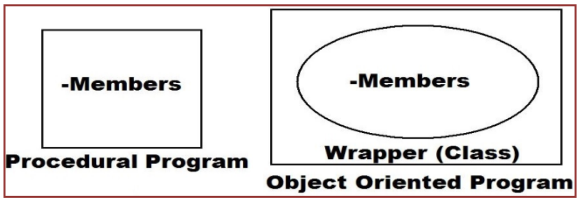
```
C Program
-Collection of Members (Variables & Functions)
void main() <= Entry Point
{
  -Call the members from here for execution
}
```

Note: the drawbacks of procedural programming languages are they don't provide security and re-usability.

Object Oriented Programming Approach: This came into existence in late 70's to overcome the drawbacks of Procedural Programming Language’s by providing Security and Re-usability.

E.g.: CPP, Python, Java, C#, etc.

In an Object Oriented Programming approach also a program is a Collection of members like variables and functions only, but the main difference between Object Oriented Languages and Procedural Languages is, here to protect the members of a program we put them under a container or wrapper known as a "class".



What is a class?

Ans: it is a **user-defined type** very much similar to structures we have learnt in C language, i.e. by using these we can define new types, whereas the difference between the two are, structure in “C” language can contain only variables in it but class of Object Oriented languages can contain both variables and functions also.

Syntax to define Class and Structure:

```
struct <Name>                   class <Name>
{                               {
  -Variables                      -Variables
};                              -Functions
                               };
```

Example:

```
struct Student                 class Employee
{                              {
 int Id;                        int Id;
 char Name[25];                 string Name, Job;
 float Marks, Fees;             float Salary;
};                              -Can be defined with functions also
                               };
```

In the above case int, float and char are pre-defined structures whereas string is a pre-defined class which we are calling them as types, same as that Student and Employee are also types (user-defined). The other difference between int, float, char and string types, as well as Student and Employee types is the 1st 4 are scalar types which can hold 1 and only 1 value under them whereas the next 2 are complex types which can hold more than 1 value under them.

How to consume a type?

Ans: types can't be consumed directly because they do not have any memory allocation.

```
int = 100; //Invalid
```

So to consume a type first we need to create a copy of that type:

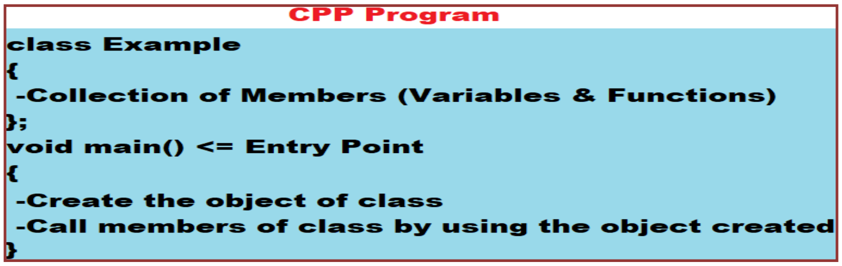
```
int i = 100; //Valid
```

Note: In the above case "i" is a copy of pre-defined type int for which memory gets allocated and the above rule of types can't be consumed directly, applies both to pre-defined and user-defined types also.

```
int i;                  //i is a copy of pre-defined type int
string s;               //s is a copy of pre-defined type string
Student ss;             //ss is a copy of user-defined type Student
Employee emp;           //emp is a copy of user-defined type Employee
```

Note: Generally copies of scalar types like int, float, char, bool, string, etc. are known as variables, whereas copies of complex types which we have defined like Student and Employee are known as Objects or Instances.

Conclusion: After defining a class or structure if we want to consume them, first we need to create a copy of them and then only the memory which is required for execution gets allocated and by using that copy (Object or Instance) only we can call members that are defined under them.



CPP Program

```
class Example
{
 -Collection of Members (Variables & Functions)
};
void main() <= Entry Point
{
 -Create the object of class
 -Call members of class by using the object created
}
```

Note: CPP is the first Object Oriented Programming Language which came into existence, but still it suffers from a criticism that it is not fully Object Oriented Language; because in CPP Language we can't write main function inside of the class and according to the standards of Object Oriented Programming each and every Member of the Program should be inside of the class.

The reason why we write main function outside of class is, if it is defined inside of the class then it becomes a member of that class and members of a class can be called only by using object of that class, but unfortunately we create object of class inside main function only, so until and unless object of class is created main function can't be called and at the same time until and unless main function starts it's execution, object creation will not take place and this is called as "Circular Dependency" and to avoid this problem, in CPP Language we write main function outside of the class.

<u>Object Oriented Programming in Java:</u> Java language came into existence in the year 1995 and here also a class is a collection of members like variables and methods. While designing the language, designers have taken it as a challenge that their language should not suffer from the criticism that it is not fully Object Oriented so they want “main” method of the class to be present inside of the class only and still execute without the need of class object and to do that they have divided members of a class into 2 categories, like:

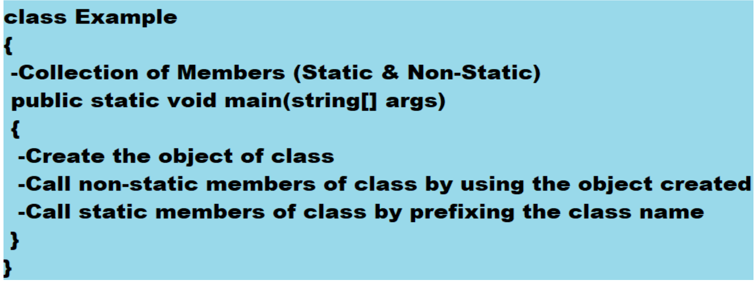
- Non-static Members
- Static Members

Every member of a class is by default a non-static member only and what we have learnt till now in C or CPP Language is also about non-static members only, whereas if we prefix any of those members with static keyword, we call them as Static Members.

```
class Test
{
  int x = 100;                    //Non-Static Member
  static int y = 200;             //Static Member
}
```

<u>Note:</u> Static members of the class doesn't require object of that class for both initialization and execution also, whereas non-static members require it, so in Java Language "main method" is defined inside of the class only but declared as static, so even if it is inside of the class also it can start the execution without the need of class object.

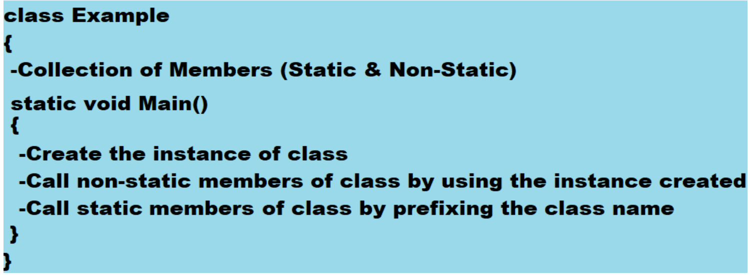
Java Program



```
class Example
{
 -Collection of Members (Static & Non-Static)
 public static void main(string[] args)
 {
  -Create the object of class
  -Call non-static members of class by using the object created
  -Call static members of class by prefixing the class name
 }
}
```

<u>Object Oriented Programming in C#:</u> C# Language came into existence after Java and was influenced by Java, so in C# Language also the programming style will be same as Java i.e. defining Main method inside the class by declaring it as "static".

C# Program



```
class Example
{
 -Collection of Members (Static & Non-Static)
 static void Main()
 {
  -Create the instance of class
  -Call non-static members of class by using the instance created
  -Call static members of class by prefixing the class name
 }
}
```

Note: In Java or C# Languages if at all the class contains only Main method in it we don't require to create object or instance of that class to run the class.

Writing programs by using C# Language: C# language has lot of standards to be followed while writing code, as following:

1. It's a case sensitive language so we need to follow the below rules and conventions:
   I. All keywords in the language must be in lower case (rule).
   II. While consuming the libraries, names will be in Pascal Case (rule). E.g.: WriteLine, ReadLine
   III. While defining our own classes and members to name them we can follow any casing pattern but Pascal case is suggested (convention).
2. A C# program should be saved with ".cs" extension.
3. We can use any name as a file name under which we write the program but class name is suggested to be used as file name also.
4. To write programs in C# we use an IDE (Integrated Development Environment) known as Visual Studio but we can also write them by using any text editor like Notepad also.

Syntax to define a class:

```
[<modifiers>] class <Name>
{
  -Define Members here                  [] => Optional
}                                       <> => Any
```

- modifiers are some special keywords that can be used on a class like public, internal, static, abstract, partial, sealed, etc.
- class is a keyword to tell that we are defining a class just like we used struct keyword to define a structure in C Language.
- <Name> refers to name of the class for identification.
- Members refer to contents of the class like fields, methods, etc.

Syntax to define Main Method in the class:

```
static void Main( [string[] args] )
{
  -Stmt's
}
```

- static is a keyword we use to declare a member as static member and if a member is declared as static, instance of the class is not required to call or execute it. In C# Main method should be declared static to start the execution from there.
- void is a keyword to specify that the method is non-value returning.
- Main is name of the method, which can't be changed and more over it should be in **Pascal Case** only.
- If required (optional) we can pass parameters to Main method but it should be of type string array only.
- Stmt's refers to the logic we want to implement.

Writing the first program in C# using Notepad:

Step 1: Open Notepad and write the following code in it:

```
class First
{
 static void Main()
 {
  System.Console.WriteLine("My first C# program using Notepad.");
  }
}
```

Step 2: Saving the program.

Create a folder on any drive of your computer with the name “**CSharp”** and save the above file into that folder naming it as **“First.cs”**.

Step 3: Compilation of the program.

We need to compile our C# program by using C# Compiler at “Developer Command Prompt”, provided along with the Visual Studio software, and to do that go to Windows Search and search for “Developer Command Prompt for VS”, click on it to open. Once it is open it will be pointing to the location where Visual Studio software is installed, so change to the location where you have created the folder and compile the program as following:

Syntax: csc <File Name>

*E.g.: **<drive>:\CSharp>** csc First.cs*

Once the program is compiled successfully it generates an output file with the name First.exe that contains “CIL (Common Intermediate Language) or MSIL (Microsoft Intermediate Language) Code” in it which we need to execute.

Step 4: Execution of the program.

Now at the same Command Prompt we can run our First.exe file as below:

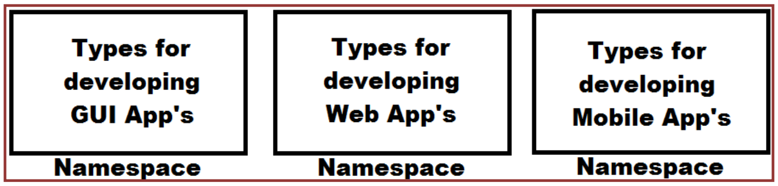
*E.g.: **<drive>:\CSharp>** First*

System.Console.WriteLine:

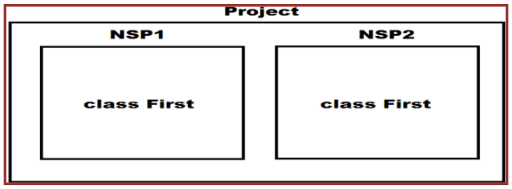
Console is a pre-defined class under the Libraries of our language which provides with a set of static members using which we can perform IO operations on the standard IO devices. WriteLine is a method in the class Console to display output on the monitor, and apart from WriteLine method there are many other methods present in the class Console like: Write, Read, ReadLine, ReadKey, Clear, etc and all these methods are also static, so we can call them directly by prefixing the class name.

System is a namespace and a namespace is a logical container for types like: Class, Structure, Interface, Enum and Delegate, and we use these namespaces in a language for 2 reasons:

1. Grouping related types i.e. types that are designed for developing similar kind of App's are grouped together under a namespace for easy access and identification as following:


Types for developing GUI App's
Namespace
Types for developing Web App's
Namespace
Types for developing Mobile App's
Namespace

2. To overcome the naming collision i.e. if a project contains multiple types with the same name we can overcome conflict between names by putting them under separate namespaces, as following:



Note: Every pre-defined type in our Libraries is defined under some namespace and we can also define types under namespaces and we will learn this process while working with Visual Studio.

If a type is defined under any namespace then, when ever and where ever we want to consume the type we need to prefix namespace name to type name and this is the reason why in our previous program we have referred to “Console” class as “System.Console”. To overcome the problem of prefixing namespace name every time before the type, we are provided with an option of “importing a namespace” which is done by “**using directive**” as following:

Syntax: using <namespace>;
using System;
using Microsoft.VisualBasic;

Note: We can import any no. of namespaces as above but each import should be a separate statement.

What is a directive?

Ans: directive in our language is an instruction that is given to the compiler which it has to follow, by importing the namespace we are telling the C# compiler that types consumed in the program are from the imported namespace.

To test the process of importing a namespace write the below code in Notepad and execute:

```
using System;
class Second
{
 static void Main()
 {
  Console.Clear();
  Console.WriteLine("Importing a namespace.");
 }
}
```

Note: If there are multiple namespaces containing a type with same name then it's not possible to consume those types by importing the namespace, and in such cases it's mandatory to refer to each type by prefixing the namespace name to them as following:

E.g.: **NSP1.First** **NSP2.First**

---

using static directive:

This is a new feature introduced in “C# 6.0” which allows us to import a type and then consume all the static members of that type without a type name prefix.

Syntax: using static <namespace.type>;
using static System.Console;

To test the process of importing a class write below code in Notepad and execute:

```
using static System.Console;
class Third
{
 static void Main()
 {
  Clear(); WriteLine("Importing a type.");
 }
}
```

Data Types in C#

C# Types	**CIL Types**	**Size/Capacity**	**Default Value**
		Integer Types	
byte	System.Byte	1 byte (0 - 255)	0
short	System.Int16	2 bytes (-2 ^ 15 to 2 ^ 15 - 1)	0
int	System.Int32	4 bytes (-2 ^ 31 to 2 ^ 31 - 1)	0
long	System.Int64	8 bytes (-2 ^ 63 to 2 ^ 63 - 1)	0
sbyte	System.SByte	1 byte (-128 to 127)	0
ushort	System.UInt16	2 bytes (0 to 2 ^ 16 - 1)	0
uint	System.UInt32	4 bytes (0 to 2 ^ 32 - 1)	0
ulong	System.UInt64	8 bytes (0 to 2 ^ 64 - 1)	0
		Decimal Types	
float	System.Single	4 bytes	0
double	System.Double	8 bytes	0
decimal	System.Decimal	16 bytes	0
		Boolean Type	
bool	System.Boolean	1 byte	False
		DateTime Type	
DateTime	System.DateTime	8 bytes	01/01/0001 00:00:00
		Unique Identifier Type	
Guid	System.Guid	32 bytes	00000000-0000-0000-000000000000
		Character Types	
char	System.Char	2 bytes	\0
string	System.String		Null
		Base Type	
object	System.Object		Null

- All the above types are known as primitive/pre-defined types i.e. they are defined under the libraries of our language which can be consumed from anywhere.

- All C# Types after compilation of source code gets converted into CIL Types and in CIL Format these types are either classes or structures defined under the "System" namespace. String and Object types are classes, whereas rest of the other 15 types, are structures.

- short, int, long & sbyte types can store signed integer (Positive or Negative) values where as ushort, uint, ulong & byte types can store un-signed integer (Pure Positive) values only.

- Guid is a type used for storing Unique Identifier values that are loaded from SQL Server DB, which is a 32 byte alpha-numeric string holding a Global Unique Identifier value and it will be in the following format: 00000000-0000-0000-0000-000000000000.

- The size of char type has been increased to 2 bytes for giving support to Unicode characters i.e. characters of languages other than English.
- We are aware that every English language character has a numeric value representation known as ASCII; characters of languages other than English also have that numeric value representation and we call it as Unicode.

  char ch = 'A'; => ASCII => Binary

  char ch = 'अ'; => Unicode => Binary

- Just like ASCII values converts into binary for storing by a computer; Unicode values also converts into binary, but the difference is ASCII requires 1 byte of memory for storing its value whereas Unicode requires 2 bytes of memory for storing its value.

- String is a variable length type i.e. it doesn't have any fixed size and its size varies based on the value that is assigned to it.

- Object is a parent of all the types, so capable of storing any type of value in it and more over it is also a variable length type.

Syntax to declare fields and variables in a class:

[<modifiers>] [const] [readonly] <type> <name> [=default value] [,...n]

```
class Test
{
  int x;                        //Field (Global Scope)
  static void Main()
  {
    int y = 100;                //Variable (Local Scope)
  }
}
```

- "<type>" refers to the type of field or variable we want to declare and it can be any of the 17 types we discussed above.
- "<name>" refers to the name of the field or variable and it should be unique within the location.

  E.g.: int i; float f; bool b; char c; string s; object o; DateTime dt; Guid id;
- Fields and variables can be initialized with any value at the time of their declaration and if they are not initialized then every **field** has a default value which is "0" for all numeric types, "false" for bool type, "\0" for char type, "00000000-0000-0000-0000-000000000000" for Guid type, "01/01/0001 00:00:00" for DateTime type and "null" for string and object types.

Note: Variables doesn't have any default value so it's must to initialize them while declaration or before consumption. E.g.: int x = 100;

- Modifiers are generally used to define the scope of a field i.e. from where it can be accessed, and the default scope for every member of a class in our language is **private** which can be either changed to public or internal or protected.
- “const” is a keyword to declare a constant and those constants values can't be modified once after their declaration:

const float pi = 3.14f; //Declaration and Initialization

- “readonly” is a keyword to declare a field as readonly and these readonly field values also can't be modified, but after initialization:

  readonly float pi;   //Declaration
  pi = 3.14f;   //Initialization

Note: decimal values are by default treated as double by the compiler, so if we want to use them as float the value should be suffixed with character “f” and “m” to use the value as decimal.

float pi = 3.14f;   double pi = 3.14;   decimal pi = 3.14m

```
using System;
class TypesDemo {
 static int x;                  //Field
 static void Main() {
  Console.Clear();
  Console.WriteLine("Field x value is: " + x + " and it's type is: "  + x.GetType());
  int y = 10;                   //Variable
  Console.WriteLine("Variable y value is: " + y + " and it's type is: "  + y.GetType());
  float f = 3.14f;              //Variable
  Console.WriteLine("Variable f value is: " + f + " and it's type is: "  + f.GetType());
  double d = 3.14;              //Variable
  Console.WriteLine("Variable d value is: " + d + " and it's type is: "  + d.GetType());
  decimal de = 3.14m;           //Variable
  Console.WriteLine("Variable de value is: " + de + " and it's type is: "  + de.GetType());
 }
}
```

Note: GetType() is a pre-defined method which returns the type (CIL Format) of a variable or field or instance on which it is called.

Data Types are divided into 2 categories:

1. Value Types   2. Reference Types

Value Types:

- All fixed length types comes under the category of value types. E.g.: integer types, decimal types, bool type, char type, DateTime type and Guid type.

- Value types will store their values on "Stack" and stack is a Data Structure that works on a principal “First in Last out (FILO)” or “Last in First out (LIFO)”.

- Each and every program when it starts the execution, a Stack will be created and given to that program for storing its values and in the end of program's execution Stack is destroyed.

- Every program will be having its own stack for storing values that are associated with the program and no 2 programs can share the same stack.

- Stack is under the control of O.S. and memory allocation is performed only in fixed length i.e. once allocated that is final which can't either be increased or decreased also.

Reference Types:

- All variable length types comes under the category of reference types and these types will store their values on “Heap” memory and their address or reference is stored on “Stack”. E.g.: String and Object.

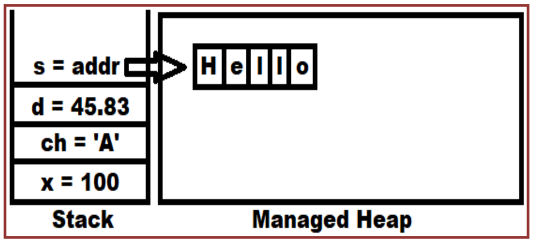
- Heap memory doesn’t have any limitations like stack and it provides a beautiful feature like, Dynamic Memory Management and because of that, all programs in execution can share the same heap.

- In older programming languages like C and CPP, Heap Memory is under developer’s control, whereas in modern programming languages like Java and .NET, Heap memory is under control of special component known as “Garbage Collector”, so we call Heap memory in these languages as “Managed Heap”.

Suppose if we declare fields in a program as following:

```
int i = 100;    char ch = 'A';    double d = 45.83;    string s = "Hello";
```

Then memory is allocated for them as following:



Nullable Value Types: These are introduced in C# 2.0 for storing null values under value types because, by default value types can't store null values under them where as reference types can store null values under them.

```
string str = null;          //Valid
object obj = null;          //Valid
int i = null;               //Invalid
decimal d = null;           //Invalid
```

To overcome the above problem nullable value types came into picture and if we want a value type as nullable we need to suffix the type with "?" and declare it as following:

```
int? i = null;              //Valid
decimal? d = null;          //Valid
```

Implicitly typed variables: This is a new feature introduced in C# 3.0, which allows declaring variables by using "var" keyword, so that the "type" of that variable is identified based on the value that is assigned to it, for example:

```
var i = 100;                //i is of type int
var f = 3.14f;              //f is of type float
var b = true;               //b is of type bool
var s = "Hello";            //s is of type string
```

Note: While using implicitly typed variables we have 2 restrictions:

1. We can't declare these variables with-out initialization. **E.g.: var x;** **//Invalid**
2. We can use "var" only on variables but not on fields.

Dynamic Type: This is a new type introduced in C# 4.0, which is very similar to implicitly typed variables we discussed above, but here in place of "var" keyword we use dynamic.

Differences between "var" and "dynamic"

Var	Dynamic
Type identification is performed at compilation time.	Type identification is performed at runtime.

Once the type is identified can't be changed to a new type again.
var v = 100; //v is of type int
v = 34.56; //Invalid
	We can change the type of dynamic with a new value in every statement.
	dynamic d = 100; //d is of type int
	d = 34.56; //d is of type double (Valid)

Can't be declared with-out initialization.
var v; //Invalid
	Declaration time initialization is only optional.
	dynamic d; //Valid
	d = 100; //d is of type int
	d = false; //d is of type bool
	d = "Hello"; //d is of type string
	d = 34.56; //d is of type double

Can be used for declaring variables only.	Can be used for declaring variables and fields also.

```
using System;
class VarDynamic
{
 static void Main()
 {
  var i = 100;
  Console.WriteLine(i.GetType());
  var c = 'A';
  Console.WriteLine(c.GetType());
  var f = 45.67f;
  Console.WriteLine(f.GetType());
  var b = true;
  Console.WriteLine(b.GetType());
  var s = "Hello";
  Console.WriteLine(s.GetType());
  Console.WriteLine("----------------------------------------------------------------------------------------------------------");

  dynamic d;
  d = 100;
  Console.WriteLine(d.GetType());
  d = 'Z';
  Console.WriteLine(d.GetType());
  d = 34.56;
  Console.WriteLine(d.GetType());
  d = false;
  Console.WriteLine(d.GetType());
  d = "Hello";
  Console.WriteLine(d.GetType());
 }
}
```

Boxing and Un-Boxing:

Boxing is a process of converting values types into reference types:

```
int i = 100;
object obj = i;                         //Boxing
```

Unboxing is a process of converting a reference type which is created from a value type back into value type, but un-boxing requires an explicit conversion:

```
int j = Convert.ToInt32(obj);           //Un-Boxing
```

Value Type	=> Reference Type		//Boxing
Value Type	=> Reference Type	=> Value Type	//UnBoxing
Reference Type	=> Value Type		//Invalid

Note: "Convert" is a predefined class in "System" namespace and "ToInt32" is a static method under that class, and this class also provides other methods for conversion like "ToDouble", "ToSingle", "ToDecimal", "ToBoolean", etc, to convert into different types.

Taking input from end user's, into a program:

```
using System;
class AddNums {
 static void Main() {
  Console.Clear();

  Console.Write("Enter 1st number: ");
  string s1 = Console.ReadLine();
  double d1 = Convert.ToDouble(s1);

  Console.Write("Enter 2nd number: ");
  string s2 = Console.ReadLine();
  double d2 = double.Parse(s2);

  double d3 = d1 + d2;

  Console.WriteLine("Sum of " + d1 + " & " + d2 + " is: " + d3);
  Console.WriteLine("Sum of {0} & {1} is: {2}", d1, d2, d3);
  Console.WriteLine($"Sum of {d1} & {d2} is: {d3}");
 }
}
```

ReadLine method of the Console class is used for reading the input from end user's into our programs and this method will perform 3 actions when used in the program, those are:

1. Waits at the command prompt for the user to enter a value.
2. Once the user finishes entering his value, immediately the value will be read into the program.
3. Returns the value as string by performing boxing because return type of the method is string.

public static string ReadLine()

Note: after reading the value as string in our program we need to convert it back into it's original type by performing an un-boxing which can be done in either of the ways:

```
        string s1 = Console.ReadLine();                         string s2 = Console.ReadLine();
        double d1 = Convert.ToDouble(s1);                       double d2 = double.Parse(s2);
                        or                                                      or
double d1 = Convert.ToDouble(Console.ReadLine());       double d2 = double.Parse(Console.ReadLine());
```

Parse(String): this method is used to convert the string representation of a value to its equivalent type on which the method is called.

```
string s1 = "100"  ;        int i = int.Parse(s1);
string s2 = "34.56";        double d = double.Parse(s2);
string s3 = "true";         bool b = bool.Parse(s3);
```

String Interpolation: String interpolation provides a more readable and convenient syntax to create formatted strings than a string composite formatting feature. An interpolated string is a string literal that might contain interpolation expressions. When an interpolated string is resolved to a result string, items with interpolation expressions are replaced by the string representations of the expression results. This feature is available starting with C# 6.0.

Operators in C#: An operator is a special symbol that tells the compiler to perform a specific mathematical or logical operation when used between a set of operands. C# has a rich set of built-in operators as following:

Arithmetic Operators	**=>**	**+, -, *, /, %**
Assignment Operators	**=>**	**=, +=, -=, *=, /=, %=**
Relational Operators	**=>**	**==, !=, <, <=, >, >=**
Logical Operators	**=>**	**&&, ||, !**
Unary Operators	**=>**	**++, --**
Miscellaneous Operators	**=>**	**sizeof(), typeof(), is, as, ?:, ??,**

```
using System;
class OperatorsDemo {
 static void Main() {
  Console.WriteLine(sizeof(double));
  Console.WriteLine(typeof(float));

  double d = 34.56;
  object obj1 = d;
  if(obj1 is double)
    Console.WriteLine("d is of type System.Double");

  string str1 = "Hello World";
  object obj2 = str1;
  string str2 = (string)obj2;
  string str3 = obj2 as string;

  int i = 100;
  Console.WriteLine(i == 100 ? "Hello India" : "Hello World");

  string Country1 = null;
  string Country2 = null;
  Console.WriteLine(Country1 ?? Country2);
  Country2 = "India";
  Console.WriteLine(Country1 ?? Country2);
  Country1 = "America";
  Console.WriteLine(Country1 ?? Country2);
 }
}
```

Conditional Statements in C#: it's a block of code that executes based on a conditional and they are divided into 2 categories.

1. **Conditional Branching**
2. **Conditional Looping**

Conditional Branching: these statements allows us to branch the code depending on whether certain conditions are met or not. C# has 2 constructs for branching code, the "if" statement which allow us to test whether a specific condition is met or not, and the switch statement which allows us to compare an expression with a number of different values.

Syntax of "if" Condition:

```
if (<condition>)
  [{] <statement(s)>; [}]
else if (<condition>)
  [{] <statement(s)>; [}]
[<multiple else if's>]
else
  [{] <statement(s)>; [}]
```

Note: Curly braces are optional if the conditional block contains single statement in it or else it's mandatory.

```
using System;
class IfDemo
{
  static void Main()
  {
    Console.Write("Enter 1st number: ");
    double d1 = double.Parse(Console.ReadLine());
    Console.Write("Enter 2nd number: ");
    double d2 = double.Parse(Console.ReadLine());

    if(d1 > d2)
      Console.WriteLine("1st number is greater than 2nd number.");
    else if(d1 < d2)
      Console.WriteLine("2nd number is greater than 1st number.");
    else
      Console.WriteLine("Both the given numbers are equal.");
  }
}
```

Syntax of "switch case" Condition:

```
switch (<expression>)
{
  case <value>:
    <stmts>;
    break;
  [<multiple case blocks>]
  default:
    <stmts>;
    break;
}
```

Note: In C and CPP languages using a break statement after each "case block" is only optional whereas it is mandatory in case of C# language, which should be used after "default block" also.

```
using System;
class SwitchDemo {
  static void Main() {
    Console.Write("Enter Student Id. (1-3): ");
    int Id = int.Parse(Console.ReadLine());

    switch(Id)
    {
      case 1:
        Console.WriteLine("Student 1");
        break;
      case 2:
        Console.WriteLine("Student 2");
        break;
      case 3:
        Console.WriteLine("Student 3");
        break;
      default:
        Console.WriteLine("No student exists with the given Id.");
        break;
    }
  }
}
```

Conditional Looping: C# provides 4 different loops that allow us to execute a block of code repeatedly until a certain condition is met and those are:

1. for loop 2. while loop 3. do..while loop 4. foreach loop

Every loop requires 3 things in common:

1. **Initialization:** This set's the start point for a loop.
2. **Condition:** This decides when the loop has to end.
3. **Iteration:** This takes the loop to the next level either in forward or backward direction.

Syntax of "for loop":

```
for (initializer;condition;iteration) {
  -<statements>;
}
```

Example:

```
for(int i = 1;i <= 100;i++)
{
  Console.WriteLine(i);
}
```

Syntax of "while loop":

```
while (<condition>) {
  -<statements>;
}
```

Example:

```
int i = 1;
while(i <= 100) {
  Console.WriteLine(i);
  i++;
}
```

Syntax of "do..while loop":

```
do {
  -<statements>;
}
while (<condition>);
```

Example:

```
int i = 1;
do {
  Console.WriteLine(i);
  i++;
}
while (i <= 100);
```

Note: the minimum no. of executions in case of a "for loop" and "while loop" are "0" because in both these cases the loop starts its execution only when the given condition is satisfied whereas the minimum no. of executions in case of a "do...while loop" is "1" because in this case after executing the loop for first time, then it will check for the condition to continue the loop's execution.

Syntax of "foreach loop":

```
foreach(type var_name in array_name|collection_name)
{
  -<statements>;
}
```

Note: foreach loop is specially designed for accessing values from an array or collection.

Jump Statements: these are statements which will transfer the control from 1 line of execution to another line. C# has no. of statements that allows jumping to another line in a program, those are:

1. goto 2. break 3. continue 4. return

goto: it allows us to jump directly to another specified line in the program, indicated by a label which is an identifier followed by a colon.

```
goto xxx;
  Console.WriteLine(“Hello World”);
xxx:
  Console.WriteLine(“Goto Called.”);
```

break: it is used to exit from a case in a switch statement and also used to exit from any conditional loop statement which will switch the control to the statement immediately after end of the loop.

```
for (int i = 1;i <= 100;i++) {
  Console.WriteLine(i);
  if (i == 50)
    break;
}
Console.WriteLine(“End of the loop.”);
```

continue: it is used only in a loop which will jump the control to iteration part of the loop without executing any other statement that is present next to it.

```
for (int i = 1;i <= 100;i++)
{
  if (i == 7 || i == 77)
    continue;
  Console.WriteLine(i);
}
```

return: this is used to terminate the execution of a method in which it is used and jumps out of that method, while jumping out it can also carry a value out of that method which was only optional.

```
using System;
class Table
{
  static void Main()
  {
    Console.Write("Enter a number: ");
    bool Status = uint.TryParse(Console.ReadLine(), out uint x);
    if(Status == true)
    {
      if(x == 0)
        return;
      Console.Clear();
      for(uint i=1;i<=10;i++)
      {
        Console.WriteLine("{0} * {1} = {2}", x, i, x * i);
      }
    }
    else
      Console.WriteLine("Please enter un-signed integer value as input.");
  } //End of the method
} //End of the class
```

Array: it is a set of similar type values that are stored in a sequential order either in the form of a row or rows & columns. In C# language also we access the values of an array by using the index only which will start from "0" and ends at the "no. of items - 1". In C# arrays can be declared either as fixed length or dynamic, where a fixed length array can store a pre-defined no. of items whereas the size of a dynamic array increases as we add new items to it. _

1-Dimensional Array's: these arrays will store data in the form of a row and are declared as following:

Syntax: <type>[] <array_name> = new <type>[length|size]

Example:

```
int[] arr = new int[5];           //Declaration and Initialization with default values
        or
int[] arr;                        //Declaration
arr = new int[5];                 //Initialization with default values
        or
int[] arr = { <list of values> }; //Declaration and Initialization with given set of values
```

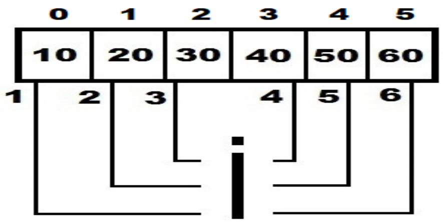
```
using System;
class SDArray1 {
 static void Main() {
  Console.Clear();
  int x = 0;
  int[] arr = new int[6];

  //Accessing values of a SD Array by using for loop
  for(int i=0;i<6;i++)
   Console.Write(arr[i] + "  ");
  Console.WriteLine();

  //Assigning values to a SD Array by using for loop
  for(int i=0;i<6;i++)
  {
   x += 10;
   arr[i] = x;
  }

  //Accessing values of a SD Array by using foreach loop
  foreach(int i in arr)
   Console.Write(i + " ");
  Console.WriteLine();
 }
}
```

foreach loop: this loop is specially designed for accessing values from an array or a collection. When we use foreach loop for accessing values, the loop starts providing access to values of the array or collection by assigning the values to loop variable in a sequential order as following:



Differences between for loop and foreach loop in accessing values of an array or collection:

1. In case of a “for loop”, the loop variable refers to index of the array whereas in case of a “foreach loop”, the loop variable refers to values of the array.
2. By using a “for loop” we can either access or assign values to an array whereas by using a “foreach loop” we can only access the values from an array.
3. In case of a “for loop”, the data type of loop variable is always int only irrespective of the type of values in the array, whereas in case of a “foreach loop”, the data type of loop variable will be same as the type of values in the array.

```
int[] iarr = { 10, 20, 30, 40, 50 };
double[] darr = { 12.34, 34.56, 56.78, 78.90, 90.12 };
string[] sarr = { "Red", "Blue", "Green", "Yellow", "Magenta" };
```

```
for (int i=0;i<iarr.Length;i++)          foreach(int i in iarr)
for (int i=0;i<darr.Length;i++)          foreach(double d in darr)
for (int i=0;i<sarr.Length;i++)          foreach(string s in sarr)
```

Array Class: this is a pre-defined class under the “System” namespace which provides with a set of members in it to perform actions on an array, those are:

Sort(Array arr)	**=> void**	**//Method**
Reverse(Array arr)	**=> void**	**//Method**
Copy(Array source, Array target, int n)	**=> void**	**//Method**
GetLength(int dimension)	**=> int**	**//Method**
Length	**=> int**	**//Property (Field)**

```
using System;
class SDArray2 {
  static void Main() {
    Console.Clear();
    int[] arr = { 54, 79, 59, 8, 42, 22, 93, 3, 73, 38, 67, 48, 18, 61, 32, 86, 15, 27, 81, 96 };

    for(int i=0;i<arr.Length;i++)
      Console.Write(arr[i] + " ");
    Console.WriteLine();

    Array.Sort(arr);
    foreach(int i in arr)
      Console.Write(i + " ");
    Console.WriteLine();
    Array.Reverse(arr);
    foreach(int i in arr)
      Console.Write(i + " ");
    Console.WriteLine();

    int[] brr = new int[10];
    Array.Copy(arr, brr, 7);
    foreach(int i in brr)
      Console.Write(i + " ");
    Console.WriteLine();
  }
}
```

2-Dimensional Array's: these arrays will store data in the form of rows & columns, and are declared as following:

Syntax: <type>[,] <array_name> = new <type>[rows, columns]

Example:

```
        int[,] arr = new int[4,5];          //Declaration and Initialization with default values
                or
        int[,] arr;                         //Declaration
        arr = new int[4,5];                 //Initialization with default values
                or
        int[,] arr = { <list of values> };  //Declaration and Initialization with given set of values
```

```
using System;
class TDArray {
 static void Main() {
  int x = 0; int[,] arr = new int[4, 5];
  //Accessing values of TD Array by using foreach loop
  foreach(int i in arr)
   Console.Write(i + " ");
  Console.WriteLine();

  //Assigning values to TD Array by using nested for loop
  for(int i=0;i<arr.GetLength(0);i++) {
   for(int j=0;j<arr.GetLength(1);j++) {
    x += 5; arr[i,j] = x;
   }
  }

  //Accessing values of TD Array by using nested for loop
  for(int i=0;i<arr.GetLength(0);i++) {
   for(int j=0;j<arr.GetLength(1);j++)
    Console.Write(arr[i,j] + " ");
   Console.WriteLine();
  }
 }
}
```

Assigning values to 2-D Array at the time of its declaration:

```
int[,] arr = {
            { 11, 12, 13, 14, 15 },
            { 21, 22, 23, 24, 25 },
            { 31, 32, 33, 34, 35 },
            { 41, 42, 43, 44, 45 }
          };
```

Jagged Arrays: these are also 2-Dimensional arrays only which will store the data in the form of rows and columns but the difference is in-case of a 2-Dimensional array all the rows will be having equal no. of columns whereas in case of a jagged array the column size varies from row to row. Jagged arrays are also known as "array of arrays" because here each row is considered as a single dimensional array and multiple single dimensional arrays with different sizes are combined together to form a new array.

Syntax: <type>[][] <array_name> = new <type>[rows][]

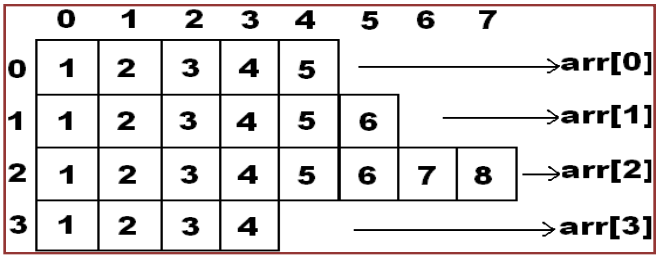
Example:

```
int[][] arr = new int[4][];             //Declaration
        or
int[][] arr = { <list of values> };     //Declaration & Initialization with given set of values
```

Note: in case of a jagged array we can't initialize the array with default values at the time of its declaration i.e. first we need to specify the no. of rows and then pointing to each row we need to specify the no. of columns to that row, as following:

```
int[][] arr = new int[4][];        //Declaration
arr[0] = new int[5];               //Initialization of 1st row
arr[1] = new int[6];               //Initialization of 2nd row
arr[2] = new int[8];               //Initialization of 3rd row
arr[3] = new int[4];               //Initialization of 4th row
```

Internally the memory is allocated for the array as following:



	0	1	2	3	4	5	6	7	
0	1	2	3	4	5				→arr[0]
1	1	2	3	4	5	6			→arr[1]
2	1	2	3	4	5	6	7	8	→arr[2]
3	1	2	3	4					→arr[3]

```
using System;
class JArrayDemo {
  static void Main() {
    Console.Clear();

    int[][] arr = new int[4][];
    arr[0] = new int[5];
    arr[1] = new int[6];
    arr[2] = new int[8];
    arr[3] = new int[4];
    //Accessing values of Jagged Array by using nested foreach loop
    foreach(int[] iarr in arr) {
     foreach(int x in iarr)
      Console.Write(x + " ");
     Console.WriteLine();
    }
    Console.WriteLine("--------------------------------------");

    //Accessing values of Jagged Array by using for loop in foreach loop
    foreach(int[] iarr in arr) {
     for(int i=0;i<iarr.Length;i++)
      Console.Write(iarr[i] + " ");
     Console.WriteLine();
    }
    Console.WriteLine("--------------------------------------");

    //Assigning values to Jagged Array by using for loop in foreach loop
    foreach(int[] iarr in arr) {
     for(int i=0;i<iarr.Length;i++) {
      iarr[i] = i + 1;
     }
    }

    //Accessing values of Jagged Array by using nested for loop
    for(int i=0;i<arr.GetLength(0);i++) {
     for(int j=0;j<arr[i].Length;j++)
      Console.Write(arr[i][j] + " ");
     Console.WriteLine();
    }
    Console.WriteLine("--------------------------------------");

    //Assigning values to Jagged Array by using nested for loop
    for(int i=0;i<arr.GetLength(0);i++) {
     for(int j=0;j<arr[i].Length;j++) {
      arr[i][j] = i + 1;
    }
  }

    //Accessing values of Jagged Array by using foreach loop in for loop
    for(int i=0;i<arr.GetLength(0);i++) {
     foreach(int x in arr[i])
      Console.Write(x + " ");
     Console.WriteLine();
    }
  }
}
```

Assigning values to Jagged Array at the time of its declaration:

```
        int[][] arr = {
                    new int[5] { 11, 12, 13, 14, 15 },
                    new int[6] { 21, 22, 23, 24, 25, 26 },
                    new int[8] { 31, 32, 33, 34, 35, 36, 37, 38 },
                    new int[4] { 41, 42, 43, 44 }
                 };
```

Implicitly typed arrays: Just like we can declare variables by using "var" keyword we can also declare arrays by using the same "var" keyword and here also the type identification is performed based on the values that are assigned to the array.

```
var iarr = new[] { 10, 20, 30, 40, 50 };                            //Implicitly type integer array
var sarr = new[] { "Red", "Blue", "Green", "Yellow", "Magenta" };    //Implicitly typed string array
var darr = new[] { 12.34, 34.56, 56.78, 78.96, 90.12 };             //Implicitly typed double array
var jarr = new[] {
                new[] { 11, 12, 13, 14, 15 },
                new[] { 21, 22, 23, 24, 25, 26 },
                new[] { 31, 32, 33, 34, 35, 36, 37, 38 },
                new[] { 41, 42, 43, 44 },
                new[] { 51, 52, 53, 54, 55, 56, 57 }
              };                                                    //Implicitly typed jagged integer array
```

Command Line Arguments: Arguments which are passed by the user or programmer to the Main method are known as Command-Line Arguments. Main method is the entry point for the execution of a program and this Main method can accepts an array of strings.

```
using System;
class Params {
  static void Main(string[] args) {
    foreach(string str in args)
      Console.WriteLine(str);
  }
}
```

After compilation of the program execute the program at Command Prompt as following:

```
<drive>:\CSharp> Params 100 Hello 34.56 A true
```

Note: We can pass any no. of values as well as any type of values as Command Line Arguments to the program, but each value should be separated with a space and all those values we passed will be captured in the string array (args) of main method. In the above case (100, Hello, 34.56, A, true) are 5 values we have supplied to the Main method of Params class.

Adding a given set of numbers that are passed as Command Line Arguments:

```
using System;
class AddParams {
  static void Main(string[] args) {
    double Sum = 0;
```

```
   foreach(string str in args)
     Sum = Sum + double.Parse(str);
   Console.WriteLine("Sum of given {0} no's is: {1}", args.Length, Sum);
  }
}
```

<u>After compilation of the program execute the program at Command Prompt as following:</u>

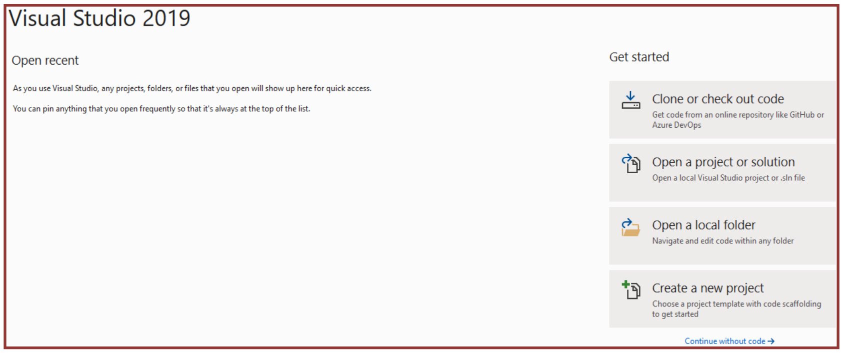
```
<drive>:\CSharp> AddParams 10 20 30
<drive>:\CSharp> AddParams 34.56 28.93 98.45 63.28
<drive>:\CSharp> AddParams 938.387 534 348.378 836 174.392
<drive>:\CSharp> AddParams 18 48.37 75 56.43 97 85.19
```

<u>Working with Visual Studio.Net</u>

Visual Studio is an IDE (Integrated Development Environment) used for developing .Net Applications by using any .Net Language like CSharp, VB etc., as well as we can develop any kind of applications like Console, Windows and Web etc.

<u>Note:</u> the current version of Visual Studio is 16 which come as Visual Studio 2019.

To open Visual Studio, go to Windows Search and search for Visual Studio 2019 and click on it to open, which will launch as following:



Visual Studio 2019

Open recent

As you use Visual Studio, any projects, folders, or files that you open will show up here for quick access.

You can pin anything that you open frequently so that it's always at the top of the list.

Get started

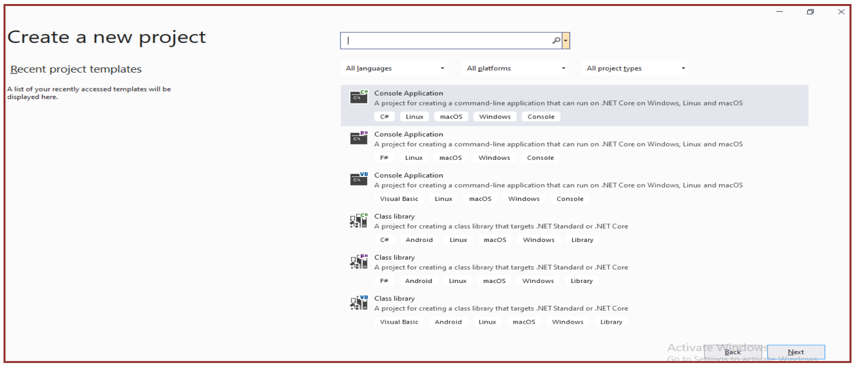
Get code from an online repository like GitHub or Azure DevOps

Open a local Visual Studio project or .sln file

Navigate and edit code within any folder

Choose a project template with code scaffolding to get started

Applications that are developed under VS are known as projects, where each project is a collection of items like Class, Interface, Structure, Enum, Delegate, Html Files, XML Files, and Text Files etc. To create a Project click on "Create a new project" option in the above Page which opens a new window as following:



Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

All languages
All platforms
All project types

Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
F# Linux macOS Windows Console

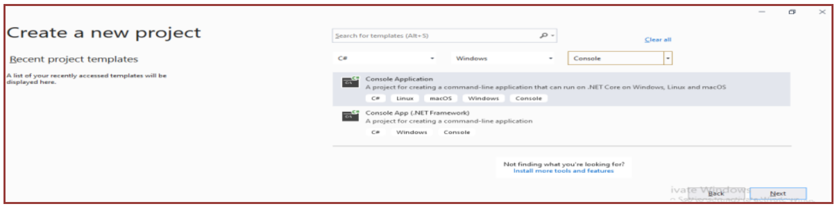
Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS

Class library
A project for creating a class library that targets .NET Standard or .NET Core

Class library
A project for creating a class library that targets .NET Standard or .NET Core

Class library
A project for creating a class library that targets .NET Standard or .NET Core

In the above window under "All languages" DropDownList select "C#", under "All platforms" DropDownList select "Windows" and under "All project_types" DropDownList select "Console" which will display the options as following:



Create a new project

Recent project templates

A list of your recently accessed templates will be displayed here.

Search for templates (Alt+S)

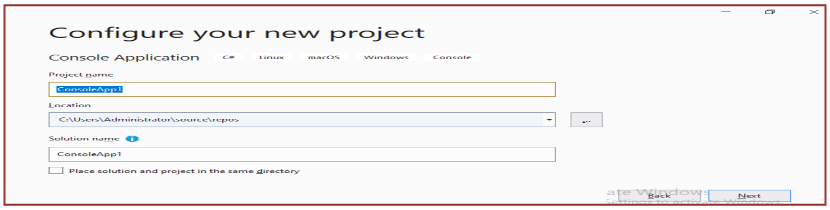
C# | Windows | Console

Console Application
A project for creating a command-line application that can run on .NET Core on Windows, Linux and macOS
C# Linux macOS Windows Console

Console App (.NET Framework)
A project for creating a command-line application
C# Windows Console

Not finding what you're looking for?
Install more tools and features

Now select “Console Application” in the above window and click “Next” button which opens a new window as following:



Configure your new project

Console Application C# Linux macOS Windows Console

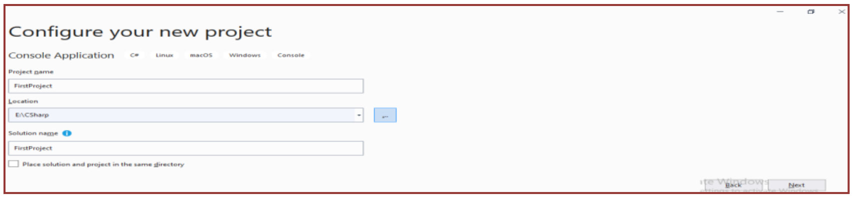
Project name
ConsoleApp1

Location
C:\Users\Administrator\source\repos

Solution name
ConsoleApp1

□ Place solution and project in the same directory

In that above window under ProjectName TextBox enter the name of project as "FirstProject", under Location TextBox enter or select our personal folder location i.e. "<drive>:\CSharp" and click on "Next" button:



Configure your new project

Console Application C# Linux macOS Windows Console

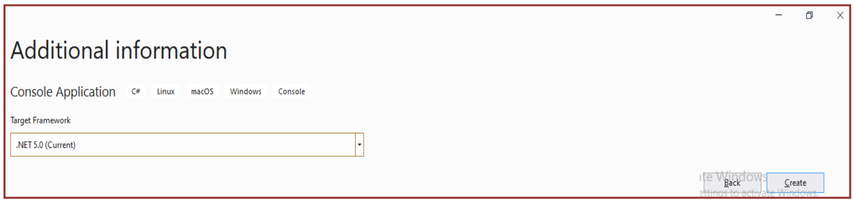
Project name
FirstProject

Location
E:\CSharp

Solution name
FirstProject

□ Place solution and project in the same directory

This will open a new window asking to select the Target Framework choose .NET 5.0 (Current) which is the latest version of .NET and then click on “Create” button:



Additional information

Console Application C# Linux macOS Windows Console

Target Framework
.NET 5.0 (Current)

This action will create a new project with a default class Program under a file “Program.cs” which will be as following:

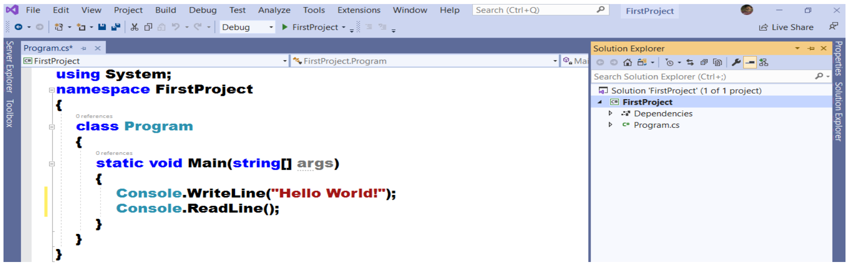
```
using System;
namespace FirstProject
{
 class Program
 {
  static void Main(string[] args)
  {
   Console.WriteLine("Hello World!");
  }
 }
}
```

When types are defined in Visual Studio by default all those types will be under a namespace whose name will be same as project name i.e. “FirstProject” in our case and from now each and every type of the project will be defined with-in this namespace only. We have already learnt earlier, that a namespace is a logical container of types like class, structure, interface, enum and delegate.

To run the class either hit Ctrl + F5 or go to “Debug” menu and select the option “Start Without Debugging” which will save, compile and executes the program by displaying output “Hello World!” on the console window because we have opened a “Console App.” Project. To close that console window it will display a message “Press any key to continue . . .”, so once we hit any key it will close the window and takes us back to the studio. We can also run the class by hitting F5 or clicking on the ▶ FirstProject ▾ button in the ToolBar or go to “Debug” menu and select the option “Start Debugging”, but in this case it save compile and executes the program but we can’t view the output because the console window gets closed immediately and in this case to view the output we need to hold console window and to do that use “Console.ReadLine();” method after “Console.WriteLine("Hello World!");” in Main method of the program.

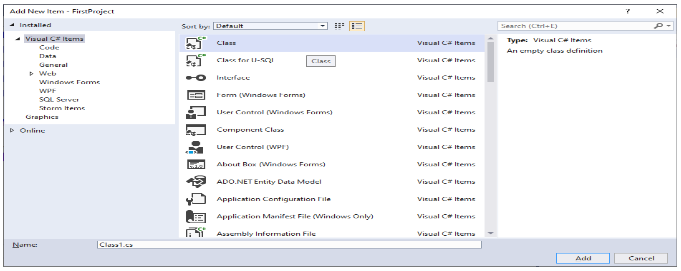
Adding new items in the project:

Under VS we find a window in the RHS known as Solution Explorer used for organizing the complete application, which allows us to view, add and delete items under the projects, if it is not visible in the RHS then go to “View” menu and select “Solution Explorer” which will launch on RHS that looks as following:



```
using System;
namespace FirstProject
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.ReadLine();
        }
    }
}
```

To add new classes under project, open Solution Explorer, right click on the project, select Add => choose “New Item” option, which opens the "Add New Item" window as following:



Class	Visual C# Items
Class for U-SQL	Visual C# Items
Interface	Visual C# Items
Form (Windows Forms)	Visual C# Items
User Control (Windows Forms)	Visual C# Items
Component Class	Visual C# Items
User Control (WPF)	Visual C# Items
About Box (Windows Forms)	Visual C# Items
ADO.NET Entity Data Model	Visual C# Items
Application Configuration File	Visual C# Items
Application Manifest File (Windows Only)	Visual C# Items
Assembly Information File	Visual C# Items

Type: Visual C# Items
An empty class definition

Name: Class1.cs

In this window select "Class" template, specify a name to it in the bottom or leave the existing name and click on Add button, which adds the class under project with the name “Class1.cs”. The new class added, also comes under the Namespace, i.e. “FirstProject”. Now write the below code in the class Class1:

```
static void Main() {
  Console.WriteLine("Second class under the project.");
  Console.ReadLine();
}
```

Now when we run the project we get an error stating that there are multiple entry points in the project because the 2 classes in the project contains a Main method and each Main method is an entry point, To resolve the above problem, open Solution Explorer, right click on the project, select properties, which opens project property window, under it we find an option "Startup Object" which lists all the classes of project that contains a valid Main method in them, choose your class i.e. “FirstProject.Class1” and run. If the “Startup Object” doesn’t list your class, again open Solution Explorer, right click on the project, select the option “Edit Project File” which opens an “XML File” and in that we find “<StartupObject></StartupObject>” tags with the existing Startup class name listed, change that name to your class name: **“<StartupObject>FirstProject.Class1</StartupObject>”** and run.

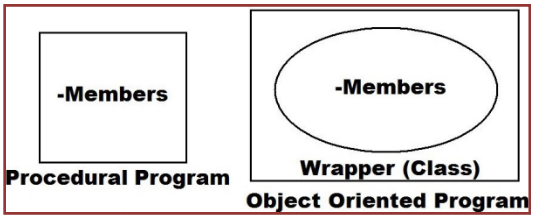
Object Oriented Programming

This is an approach that we use in the industry for developing Application, introduced in late 70's replacing the traditional Procedural Programming Approach. Procedural Programming Approach doesn't provide Security and Re-usability, where as these 2 are the major strength of Object Oriented Programming Approach.

Any language to be called as Object Oriented needs to satisfy 4 important principals that are prescribed under the standards of Object Oriented Programming and they are:

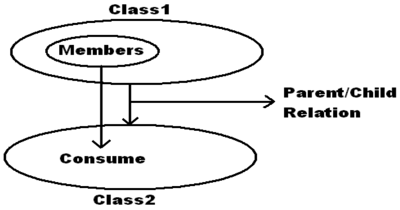
1. **Encapsulation => hiding the data**
2. **Abstraction => hiding the complexity**
3. **Inheritance => re-usability**
4. **Polymorphism => behaving in different ways based on input received**

Encapsulation: this is all about hiding of data or members of a program by wrapping them under a container known as a Class, which provides security for all its contents.



Abstraction: this is all about hiding the complexity of code and then providing with a set of interfaces to consume those functionalities, for example functions/methods in a program are good example for this, because here we are only aware of how to call them but we are never aware of the underlying logic behind that implementation.

Inheritance: this provides re-usability i.e. members that are defined in 1 class can be consumed from other classes by establishing parent/child relation between the classes.



Polymorphism: behaving in different ways based on the input received is known as polymorphism i.e. whenever the input changes then the output or behavior also changes accordingly.

Class: It's a user-defined type which is in-turn a collection of members like:

- **Fields**
- **Methods**
- **Constructors**
- **Finalizers**
- **Properties**
- **Indexers**
- **Events**
- **De-constructors (Introduced in C# 7.0)**

Method: It is a named block of code which performs an action whenever it is called and after completion of that action it may or may not return any result of that action, and they are divided into 2 categories:

1. **Value returning method (Function)**
2. **Non-value returning method (Sub-Routine)**

Syntax to define a method:

```
[<modifiers>] void|type <Name>( [<Parameter List>] )
{
 -Stmt's or Logic                                    [] => Optional
}                                                    <> => Any
```

Modifiers are some special keywords which can be used on a method if required like public, internal, protected, static, virtual, abstract, override, sealed, partial, etc.

void|type is to tell whether our method is value returning or non-value returning i.e. “void” implies that the method is non-value returning, whereas if we want our method to return any value then we need to specify the type of value it has to return by using the “type”.

Example for non-value returning method:

```
public static void Clear()
public static void WriteLine(<type> var)
```

Example for value returning method:

```
public static string ReadLine()
```

Note: the return type of a method need not be any **pre-defined type** like int, float, char, bool, DateTime, Guid, string, object, etc, but it can also be any **user-defined type** also.

<Name> refers to the “ID” of method for identification.

[<Parameter list>]: if required we can pass parameters to our methods for execution, and parameters of a method will make an action dynamic, for example:

```
GetLength(0)     => Returns rows
GetLength(1)     => Returns columns
```

Syntax to pass parameters to a method:

```
[ref|out] [params] <type> <var> [=default value] [,...n]
```

Where should we define methods?

Ans: As per the rule of Encapsulation methods should be defined inside of a class.

How to execute a method that is defined under a class?

Ans: The methods that are defined in a class must be explicitly called for execution, except Main method because Main is implicitly called.

How to call a method that is defined in a class?

Ans: Methods are of 2 types:

1. Non-Static   2. Static

Note: By default every method of a class is non-static only, and if we want to make it as static we need to prefix the "static" modifier before the method as we are doing in case of Main method.

To call a method that is defined under any class we require to create instance of that class provided the methods are non static, whereas if the methods are static we can call them directly by using class name, for example WriteLine and ReadLine are static methods in class Console which we are calling in our code as Console.WriteLine and Console.ReadLine.

How to create instance of a class?
Ans: We create the instance of class as following:

Syntax: <class_name> <instance_name> = new <class_name> ([<List of values>])

Example:

```
Program p = new Program();          //Declaration and Initialization
            or
Program p;                          //Declaration
p = new Program();                  //Initialization
```

Note: with-out using "new" keyword we can't create the instance of a class in Java and .NET Languages.

Where should we create the instance of a class?
Ans: instance of a class can be created either with-in the same class or in other classes also.

If instance is created in the same class it should be created under any static block; generally we create instances in Main method because of 2 reasons:

1. **Entry Point of the program.**
2. **It is a static block.**

If instance is created in another class then it can be created in any block of that class i.e. either static or non-static also.

To try all the above, create a new project in Visual Studio naming it as "OOPSProject" and write the below code under the default class "Program":

```
class Program {
 //Non-value returning method without parameters
 public void Test1() {                          //Static in behavior
  int x = 5;
  for (int i = 1; i <= 10; i++)
    Console.WriteLine("{0} * {1} = {2}", x, i, x * i);
 }

 //Non-value returning method with parameters
 public void Test2(uint x, uint ub) {           //Dynamic in behavior
  for (uint i = 1; i <= ub; i++)
    Console.WriteLine("{0} * {1} = {2}", x, i, x * i);
 }
 //Value returning method without parameters
 public string Test3() {                          //Static in behavior
   string str = "Hello world";
   str = str.ToUpper();
   return str;
 }

 //Value returning method with parameters
 public string Test4(string str) {                //Dynamic in behavior
   str = str.ToUpper();
   return str;
 }

 static void Main(string[] args) {
   //Creating the instance of class
   Program p = new Program();

   //Calling non-value returning methods
   p.Test1();
   Console.WriteLine();
   p.Test2(8, 12);
   Console.WriteLine();

   //Calling value returning methods
   string s1 = p.Test3();
   Console.WriteLine(s1);
   string s2 = p.Test4("india");
   Console.WriteLine(s2);
   Console.ReadLine();
 }
}
```

Consuming a class from other classes: It is possible to consume a class and its members from other classes in 2 different ways:

1. **Inheritance**
2. **Creating an instance**

To test the second, add a new class in the project naming it as “TestProgram.cs” and write the below code under class:

```
class TestProgram {
 public void CallMethods() {
   Program p = new Program();
   p.Test1(); p.Test2(9, 15);
   Console.WriteLine(p.Test3());
   Console.WriteLine(p.Test4("america"));
 }
 static void Main() {
  //Creating an un-named instance and calling the method
  new TestProgram().CallMethods();
  Console.ReadLine();
 }
}
```

Note: Un-named instances are created and used when we want to call any single member of a class or when we want to use that instance only for 1 time.

Code files in a project: When we want to add a new class under any project we first open the "Add New Item" window and in that we choose "Class Item Template", which when added will add a file with a class template in it, same as that we also find "Code File Item Template" which when added will add a blank file and we need to write everything manually in it, just like we write code using Notepad.

Defining multiple classes in a file: It's possible to define "n" no. of classes under a single ".cs" file, but "Main" method can be defined under 1 class only. Even if it is not mandatory it is advised to use the "Class Name" under which we defined "Main" method as the "File Name".

User-defined return types to a Methods: The return type of a method need not be any pre-defined type, but can also be any user-defined type also i.e. a type which is defined representing some complex data.

To test all the above, add a "Code File" under the project naming it as "UserDefinedTypes.cs" and write the below code in it:

```
using System;
namespace OOPSProject {
 class Emp {
  public int Eno;
  public string Name, Job;
  public double Salary;
  public bool Status;
 }
 class UserDefinedTypes {
  public Emp GetEmpDetails(int Eno) {
   Emp e = new Emp();
   e.Eno = Eno;
   e.Name = "Ajay";
   e.Job = "Salesman";
   e.Salary = 6000.00;
   e.Status = true;
   return e;
  }
  static void Main() {
   UserDefinedTypes udt = new UserDefinedTypes();
   Emp obj = udt.GetEmpDetails(1008);
   Console.WriteLine(obj.Eno + "  " + obj.Name + "  " + obj.Job + "  " + obj.Salary + "  " + obj.Status);
   Console.ReadLine();
  }
 }
}
```

In the above case "Emp" is a new type (User-Defined and Complex) and that type is used as a return type for our method "GetEmpDetails".

Parameters of a Method:

We define parameters to methods for making actions dynamic i.e. as discussed earlier every method is an action and to make those actions dynamic we define parameters to methods.

Syntax for defining parameters to a method:

[ref|out] [params] <type> <parameter_name> [=default value] [,...n]

Parameters of a method are classified as:

1. Input Parameters
2. Output Parameters
3. InOut Parameters

- Input parameters will bring values into the method for execution.
- Output parameters will carry results out of the method after execution.
- InOut Parameters are a combination of above 2 i.e. these parameters will 1st bring a value into the method for execution and after execution, the same parameter will carry results out of the method.

By default every parameter is an input parameter whereas if we want to declare any parameter as output we need to prefix "out" keyword and to declare a parameter as InOut we need to prefix "ref" keyword before the parameter, as following:

public void Test(int a, out int b, ref int c)

To test Output Parameters add a new class in the Project naming it as "OutputParameters.cs" and write the below code in it:

```
class OutputParameters {
 public void Math1(int a, int b, out int c, out int d) {   //Before C# 7.0
  c = a + b;
  d = a * b;
 }
 public (int, int) Math2(int a, int b) {                   //From C# 7.0
  int c = a + b;
  int d = a * b;
  return (c, d);
 }
 static void Main() {
  OutputParameters obj = new OutputParameters();

  int Sum1, Product1;
  obj.Math1(100, 25, out Sum1, out Product1);
  Console.WriteLine("Sum of given no's is: " + Sum1);
  Console.WriteLine("Product of given no's is: " + Product1);
  Console.WriteLine("----------------------------------------------------------------------------------------------------------");
  obj.Math1(165, 15, out int Sum2, out int Product2);            //From C# 7.0
  Console.WriteLine("Sum of given no's is: " + Sum2);
  Console.WriteLine("Product of given no's is: " + Product2);
  Console.WriteLine("----------------------------------------------------------------------------------------------------------");

  var (Sum3, Product3) = obj.Math2(252, 12);
  Console.WriteLine("Sum of given no's is: " + Sum3);
  Console.WriteLine("Product of given no's is: " + Product3);
  Console.WriteLine("----------------------------------------------------------------------------------------------------------");

  (int Sum4, int Product4) = obj.Math2(315, 9);
  Console.WriteLine("Sum of given no's is: " + Sum4);
  Console.WriteLine("Product of given no's is: " + Product4);
  Console.WriteLine("----------------------------------------------------------------------------------------------------------");
  Console.ReadLine();
 }
}
```

To test InOut parameters add a new class in the Project naming it as “InOutParameters.cs” and write the below code in it:

```
class InOutParameters {
 public void Factorial(ref uint fact) {
  if (fact == 0 || fact == 1) {
   fact = 1;
  }
  else {
   uint result = 1;
   for(uint i = 2;i <= fact;i++) {
    result = result * i;
   }
   fact = result;
  }
 }
 static void Main() {
  InOutParameters p = new InOutParameters();
  uint num = 8;
  Console.Write($"Factorial of {num} is: ");
  p.Factorial(ref num);
  Console.WriteLine(num);
  Console.ReadLine();
 }
}
```

<u>Params KeyWord:</u> By prefixing this keyword before an array parameter of any method we get a chance to call that method without explicitly creating an array and pass to the method, but we can directly pass a set of values in a “Comma-Seperated” list.

public void AddParams(params double[] args)

For example WriteLine method of Console class is defined as below:

public static void WriteLine(string format, params object[] args)

So we are able to call that method in our earlier program as following:

Console.WriteLine("{0} * {1} = {2}", x, i, x * i);

Note: while using the "params" keyword we have 2 restrictions:

1. We can use it only on 1 parameter of the method.
2. It can be used only on the last parameter of that method.

Default values to parameters: While defining methods we can assign default values to parameters of that method, so that those parameters will become "optional" and while calling that method it is not mandatory to pass values to those parameters. If the method is called without passing a value to those parameters then default value of that parameter will be used, for example:

public void AddNums(int x, int y = 50, int z = 25)

Note: In the above case x is a mandatory parameter whereas y and z are optional parameters and while defining methods with mandatory and optional parameters, mandatory parameters should be in the 1st place of parameter list, followed by optional parameters in the last.

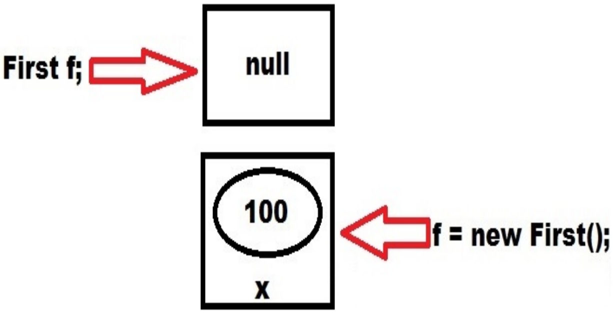
To test "params" keyword and "default valued parameters" add a new class in the project naming it as "MethodParameters.cs" and write the below code in it:

```
class MethodParameters {
  public void AddParams(params double[] args) {    //Method using params keyword on an array parameter
    double Sum = 0;
    foreach (double d in args) {
      Sum = Sum + d;
    }
    Console.WriteLine($"Sum of given {args.Length} no's is: {Sum}");
  }
  //Method with default valued parameters
  public void AddNums(int x, int y = 50, int z = 25) {
    Console.WriteLine($"Sum of the 3 no's is: { x + y + z}");
  }
  static void Main() {
    MethodParameters obj = new MethodParameters();
    //Calling a method which uses array as parameter with params keyword
    obj.AddParams();
    obj.AddParams(56.19);
    obj.AddParams(23.45, 78);
    obj.AddParams(45, 67.82, 92);
    obj.AddParams(32.91, 87, 56.23, 17);
    Console.WriteLine("----------------------------------------------------------------------------------------------------");

    //Calling a method which has default valued parameters
    obj.AddNums(100);
    obj.AddNums(100, 100);
    obj.AddNums(100, z: 100);
    obj.AddNums(100, 100, 100);
    Console.ReadLine();
  }
}
```

Understanding the difference between variable, instance and reference of a class: to understand about a variable, instance and reference of a class add a new class in our Project naming it as “First.cs” and write below code in it:

```
class First {
  public int x = 100;
  static void Main() {
    First f; //f is a variable of class
    f = new First(); //f is a instance of class
    Console.WriteLine(f.x);
    Console.ReadLine();
  }
}
```



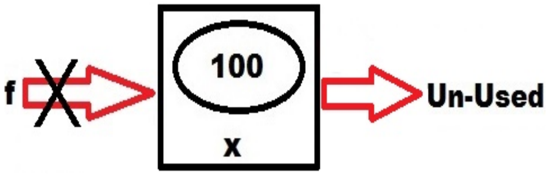
Every member of a class, if it is non-static can be accessed from the Main Method only by using instance of that class. So in the above case to print the value of “x”, we created instance of class First under Main method.

A variable of class is a copy of class which is not initialized so it doesn't have any memory allocation and can't be used for calling or accessing the members.

An instance of class is a copy of class which is initialized by using “new” keyword and for an instance memory is allocated, so by using this instance we can access or call members of that class.

De-referencing an Instance: it is possible to dereference the instance of any class by assigning “null” to the instance and once “null” is assigned to instance we can't use that instance for calling members of class and if we try to do so, we get a runtime error. To test this, re-write the code under “Main Method” of class “First” as below:

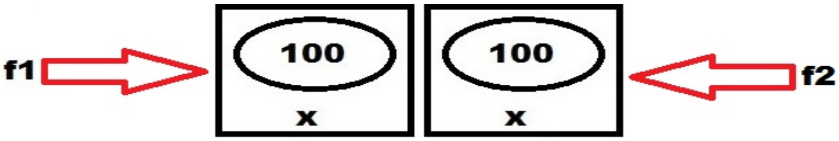
```
First f = new First();
Console.WriteLine(f.x); //Valid
f = null;
Console.WriteLine(f.x); //Invalid (Causes Runtime error)
Console.ReadLine();
```



Note: once null is assigned to a instance, internally the memory which is allocated for that instance is not de-allocated immediately, but only gets marked as un-used and all those un-used objects memory will be de-allocated by “Garbage Collector” whenever it comes into action.

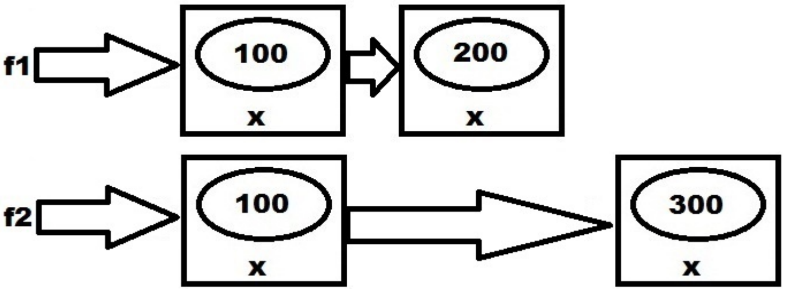
Creating multiple instances to a class: it is possible to create multiple instances to a class and each instance we create for the class will be having a separate memory allocation for its members as following:

```
First f1 = new First();
First f2 = new First();
```



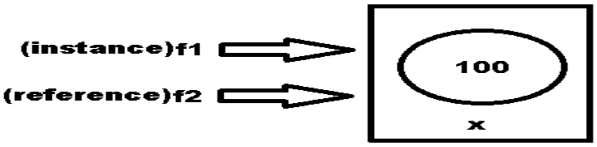
Instances are unique i.e. any modifications that we perform on the members of 1 instance will not reflect to the members of other instances of the class, and to test this rewrite the code under "Main method" of class First as below:

```
First f1 = new First();
First f2 = new First();
Console.WriteLine(f1.x + "  " + f2.x);
f1.x = 200;
Console.WriteLine(f1.x + "  " + f2.x);
f2.x = 300;
Console.WriteLine(f1.x + "  " + f2.x);
Console.ReadLine();
```



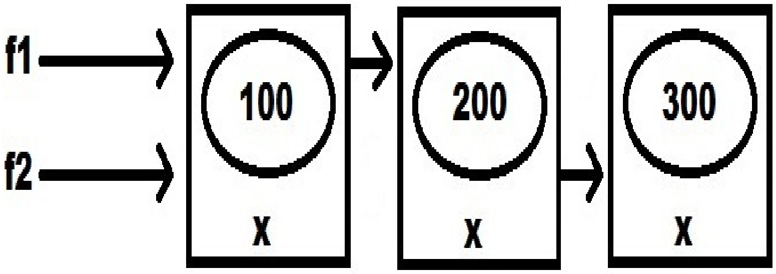
Reference of a class: we can initialize the variable of a class by using any existing instance of that class and we call it as a reference of the class. References of class will not have any memory allocation like instances, i.e. they will be consuming the memory of instance using which they are initialized, so a reference is just a pointer to an instance, as following:

```
First f1 = new First();
First f2 = f1;   //f2 is a reference of class First
```



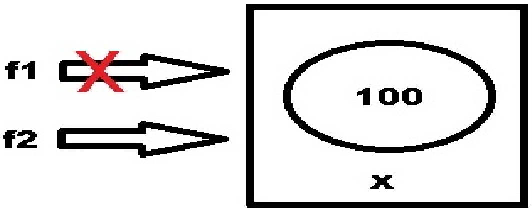
Because an instance and reference are accessing the same memory, changes that are performed on the members by using the instance will reflect when those members are accessed by using reference and vice versa. To test this, rewrite code under "Main method" of class First as below:

```
First f1 = new First();
First f2 = f1;
Console.WriteLine(f1.x + "  " + f2.x);
f1.x = 200;
Console.WriteLine(f1.x + "  " + f2.x);
f2.x = 300;
Console.WriteLine(f1.x + "  " + f2.x);
Console.ReadLine();
```



Note: when an instance and references are accessing the same memory and if "null" is assigned to any 1 of them, then the 1 to whom null is assigned can't access the memory any more, but still the others can access it as is for calling the members. To test this re-write code under "Main method" of class First as below:

```
First f1 = new First();
First f2 = f1;
f1 = null;
Console.WriteLine(f2.x); //Valid
Console.WriteLine(f1.x); //Invalid (Causes runtime error)
Console.ReadLine();
```



Variable of Class: this is a copy of class which is not initialized, so by using this we can't call any members of that class.

Instance of Class: this is a copy of class which is initialized by using the "new" keyword and by using this we can call members of that class.

Reference of Class: this is a copy of class which is initialized by using any existing instance of that class and this works same as an instance. By using the reference also we can call members of that class.

What happens internally when we create the instance of a class?

Ans: When we create the instance of any class internally following actions will take place:

1. Reads the classes to identify their members.
2. Invokes the constructors of all those classes.
3. Allocates the memory that is required for execution.

Constructor

This is a special method present under a class responsible for initializing the data members (fields) of that class. This method is invoked automatically when we create the instance of class. The name of constructor method is exactly the same name of the class and more over it's a non-value returning method. Every class requires a constructor in it, if we want to create the instance of that class or else we can't create the instance of any class.

Note: While defining a class it's the responsibility of developers to define a **constructor explicitly** under his class, and if they fails to do so, on-behalf of the developers an **implicit constructor** gets defined in those classes; so till now we are creating instances of the classes we defined, by using those implicit constructors only.

For example if we define a class as following:

```
class Test {
 int i = 10; string s; bool b;            //Fields
}
```

After compilation of the above class it will be as following with an implicit constructor:

```
class Test {
 int i = 10; string s; bool b;            //Fields
 public Test() {                          //Implicit Constructor
  i = 10; s = null; b = false;
 }
}
```

- Implicit constructors are public.
- While declaring a field if we assign any value to it, then constructor will initialize the field with that value only or else it will initialize the field with default value of that type.
- We can also define our own constructors in our classes and if we do that implicit constructor will not be defined again.

Syntax to define a Constructor Explicitly:

```
[<modifiers>] <Class Name>( [<Parameter List>] )
{
 -Stmt's to execute
}
```

To test defining a constructor explicitly, add a new class in the project naming it as "ConDemo.cs" and write the below code in it:

```
class ConDemo {
 public ConDemo() {
  Console.WriteLine("Constructor is called.");
 }
 public void Demo() {
  Console.WriteLine("Method is called.");
 }
 static void Main() {
  ConDemo cd1 = new ConDemo();
  ConDemo cd2 = new ConDemo();
  ConDemo cd3 = cd2;
  cd1.Demo(); cd2.Demo(); cd3.Demo();
  Console.ReadLine();
 }
}
```

Constructor of a class must be **explicitly called** for execution and we do that while creating the instance of class as following:

Syntax: <Class_Name> <Instance_Name> = new <Constructor_Name>([<List of Values>])

Example: ConDemo obj = new **ConDemo();**

Calling the constructor

If constructor is called then only memory allocation is performed, so instances of a class will have memory allocation because they call the constructor explicitly whereas reference of class will not have memory allocation because they do not call the constructor.

Constructors are defined implicitly or explicitly?
Ans: Either Or.

Constructors must be called explicitly or called implicitly?
Ans: Must be called explicitly.

Constructors are of 2 types:

1. Default or Parameter-less
2. Parameterized

Constructors can also be parameterized i.e. just like a method can be defined with parameters, constructor can also be defined with parameters. If constructor is defined with parameters we call it as "Parameterized Constructor" where as a constructor without any parameters is called as "Default/Parameter-less Constructor".

Default constructors can be defined either explicitly or will be defined implicitly provided there is no explicit constructor defined under that class, whereas implicit constructors will never be parameterized i.e. if a constructor is parameterized then it is very sure that it is an explicit constructor.

Note: if Constructors of a class are parameterized then values to those parameters should be sent while creating instance of that class.

To test Parameterized Constructors, add a new class in our project naming it as “ParamConDemo.cs” and write the below code in it:

```
class ParamConDemo
{
 public ParamConDemo(int i)
 {
  Console.WriteLine("Parameterized constructor is called: " + i);
 }
 static void Main()
 {
  ParamConDemo cd1 = new ParamConDemo(10);
  ParamConDemo cd2 = new ParamConDemo(20);
  ParamConDemo cd2 = new ParamConDemo(20);
  Console.ReadLine();
 }
}
```

Why to define a constructor explicitly in our class when there are implicit constructors?
Ans: We define constructors explicitly in our class for various reasons like:

1. Implicit constructors are parameter-less which will initialize fields of a class either with a default value of that type or a fixed given value even if we create multiple instances of class, whereas if constructors are defined explicitly (parameterized), then we get a chance of passing new values to the fields every time the instance of class is created. To test this, add a new class under our project naming it as “Second.cs” and write the below code in it:

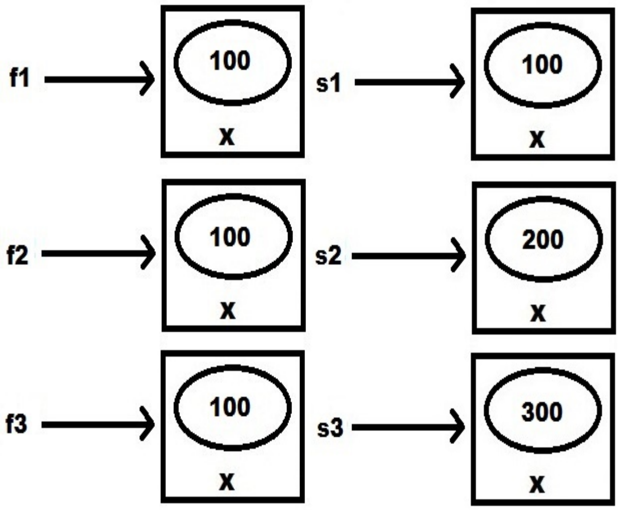
```
class Second
{
 public int x;
 public Second(int x)
 {
  this.x = x;
 }
}
```

Note: “this” is a keyword which refers to the class and by using this we can access non-static members of a class from other non-static blocks when there is a naming conflict.

Earlier we have defined a class "First" with a public field “x” in it, and we have initialized it with a static value “100”, and in the above class also we have a public field “x” which was initialized thru a Constructor, so in the 1st case even if we create multiple instances of class First; under every instance the value of “x” will be “100” only whereas in case of class Second for each instance of class we create we can pass a new value for initialization because initialization is performed thru the constructor. To test that add a new class in our project naming it as “TestClasses.cs” and write the below code in it:

```
class TestClasses {
 static void Main() {
  First f1 = new First();
  First f2 = new First();
  First f3 = new First();
  Console.WriteLine(f1.x + "  " + f2.x + "  " + f3.x);

  Second s1 = new Second(100);
  Second s2 = new Second(200);
  Second s3 = new Second(300);
  Console.WriteLine(s1.x + "  "+ s2.x + "  " + s3.x);
  Console.ReadLine();
 }
}
```



2. Every class requires some values for execution and the values that are required for a class to execute should be passed to the class with the help of a constructor.
3. Just like parameters of a method will make a method dynamic, same as that parameters of constructor will make the whole class dynamic.

Static Modifier

It is a keyword using which we can declare a class and its members as static i.e. if static keyword is pre-fixed before a class or its members then they will become static or else by default every class and its member is non-static only.

Members of a class are divided into 2 categories, like:

1. Non-Static or Instance Members
2. Static Members

Members that require instance of a class for initialization and execution are known as non-static or instance members, whereas members that doesn't require instance of the class for initialization and execution are known as static members.

Non-Static Fields Vs Static Fields:

- If a field is explicitly declared by using static modifier it is a static field, whereas rest of every other field is non-static only.

```
class Test
{
 int x = 100;              //Non-Static
 static int y = 200;       //Static
 static void Main()
 {
  int z = 300;             //Static
 }
}
```

Note: variables declared under static blocks are also static.

- Static fields of a class are initialized immediately once the execution of that class starts whereas non-static fields are initialized only after creating the instance of that class as well as each and every time a new instance is created.

- In the life cycle of a class a static field gets initialized 1 & only 1 time whereas a non-static field gets initialized for “0” times if no instances are created & “n” times if “n” instances are created.

- The initialization of non-static fields is associated with a constructor call, so the best place to initialize non-static fields is a constructor.

Note: static fields can also be initialized thru constructor but still we never do that because, it's a single copy thru out the life cycle of class and every new instance will override the old values.

Constant Fields: If a field is explicitly declared by using “const” keyword we call it as a constant field and those constant field can't be modified once after their declaration, so it is must to initialize them at the time of declaration only because they do not have a default value.

E.g.: const float pi = 3.14f;

The behavior of a constant field will be very similar to the behavior of a static field i.e. initialized immediately once the execution of class starts maintaining a single copy thru-out the life cycle of a class and the only difference between static and constant fields is static fields can be modified but not constant fields.

ReadOnly Fields: If a field is explicitly declared by using readonly keyword we call it as a readonly field and like constant fields, readonly fields also can't be modified, but after their initialization i.e. it's not mandatory to initialize readonly fields at the time of declaration because they can also be initialized after their declaration i.e. under a constructor.

E.g.: readonly bool flag;

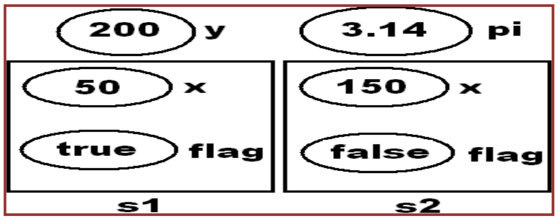
The behavior of readonly fields will be similar to the behavior of non-static fields i.e. they are initialized only after creating the instance of class and maintain a separate copy for each instance that is created.

The only difference between non-static and readonly fields is non-static fields can be modified but not readonly fields.

The difference between constant and readonly fields is constant is a single fixed value for the whole class whereas readonly is a fixed value specific to each instance of the class.

To test all the above add a new class in our project naming it as “Fields.cs” and write the below code in it:

```
class Fields {
  int x;
  static int y = 200;
  const float pi = 3.14f;
  readonly bool flag;
  public Fields(int x, bool flag) {
    this.x = x;
    this.flag = flag;
  }
  static void Main() {
    Console.WriteLine("Static field y is: " + y);
    Console.WriteLine("Constant field pi is: " + pi);
    y = 500; //Can be modified
    //pi = 5.67f; //Can't be modified & error if un-commented
    Console.WriteLine("Modified static field y is: " + y);
    Console.WriteLine("-------------------------------------------------");
    Fields s1 = new Fields(50, true);
    Fields s2 = new Fields(150, false);
    Console.WriteLine("Non-Static Fields: " + (s1.x + "  " + s2.x));
    Console.WriteLine("ReadOnly Fields: " + (s1.flag + "  " + s2.flag));
    s1.x = 100; s2.x = 300; //Can be modified
    //s1.flag = false; s2.flag = true; //Can't be modified & Error if un-commented
    Console.WriteLine("Modified Non-Static Fields: " + (s1.x + "  "           + s2.x));
    Console.ReadLine();
  }
}
```



Note: While accessing fields of a class from other classes use class name for accessing static and constant fields whereas use instance of class for accessing non-static and readonly fields.

- Static field initializes immediately once the execution of class starts maintaining a single copy thru out the life cycle of class and its value is modifiable.
- Constant field also initializes immediately once the execution of class starts maintaining a single copy thru out the life cycle of class and its value is non-modifiable.
- Non-static field initializes only after creating the instance of class, as well as for each instance of the class that is created, maintaining a separate copy for each instance and its value is modifiable.
- Readonly field also initializes only after creating the instance of class, as well as for each instance of the class that is created, maintaining a separate copy for each instance and its value is non-modifiable.

Non-Static Methods Vs Static Methods:

If a method is explicitly declared by using static keyword, then it is a static method whereas rest of every other method is non-static only.

While defining methods, if a method is non-static and if we want to consume any static members of class in it, we can consume them directly where as if the method is static, we can consume non-static members of class in that method only by using class instance.

Rules for consuming members with in a class:

Static Member => Static Block	//Direct Access
Static Member => Non Static Block	//Direct Access
Non Static Member => Non Static Block	//Direct Access
Non Static Member => Static Block	//Can be accessed only by using the class instance

Rules for consuming members out of the class:

Static Members	//Using class name
Non-Static Members	//Using class instance

To test all the above add a new "Code File" in the project naming it as "TestMethods.cs" and write the below code in it:

```
using System;
namespace OOPSProject {
  class Methods {
    int x = 200;
    static int y = 100;
    public void Add() {
      Console.WriteLine(x + y);
    }
    public static void Sub() {
      Methods m = new Methods();
      Console.WriteLine(m.x - y);
    }
  }
  class TestMethods {
    static void Main() {
      Methods obj = new Methods();
      obj.Add();                   //Add is non-static so calling it with instance
      Methods.Sub();               //Sub is static so calling it with class name
      Console.ReadLine();
    }
  }
}
```

Non-Static Constructor Vs Static Constructor:

- A Constructor if explicitly declared by using static modifier is a static Constructor whereas rest of the other are non-static only and till now every Constructor we defined is non-static only.
- Static Constructors are implicitly called whereas non-static Constructors must be explicitly called.
- As we are aware that Constructors are responsible for initializing fields in a class; Non-Static Constructor will initialize Non-Static and Readonly Fields, whereas Static Constructor will initialize Static and Constant fields.
- Static Constructor executes immediately once the execution of class starts and more over it is the first block of code to execute in a class, whereas Non-Static Constructor gets executed only after creating the instance of class as well as each and every time a new instance is created i.e. Static Constructor executes 1 and only 1 time in the life cycle of a class whereas Non-Static Constructor get executed for "0" times if no instances are created and "n" times if "n" instances are created.
- Static Constructor can't be parameterized because they are implicitly called and more over it's the first block of code to execute in a class, so we don't have any chance of sending values to its parameter's where as parameterized Non-Static Constructors can be defined.

Note: We have already learnt earlier that, every class will contain an implicit constructor if not defined explicitly and those implicit constructors are defined based on the following criteria:

1. Non-static constructor will be defined in every class except in a static class.
2. Static constructor will be defined only if the class contains any static fields.

```
class Test                    //Case 1
```

```
{
}
```

*After compilation there will be a non-static constructor in class.

```
class Test {                //Case 2
  int i = 10;
}
```

*After compilation there will be a non-static constructor in class.

```
class Test {                //Case 3
  static int i = 100;
}
```

*After compilation there will be both static and non-static constructors also.

```
static class Test           //Case 4
{
}
```

*After compilation there will not be any constructor in class.

```
static class Test {         //Case 5
  static int i = 100;
}
```

*After compilation there will be a static constructor in class.

To test all the above add a new class in the project naming it as “Constructors.cs” and write the below code in it:

```
class Constructors {
  static Constructors() {
    Console.WriteLine("Static constructor is called.");
  }
  Constructors() {
    Console.WriteLine("Non-static constructor is called.");
  }
  static void Main() {
    Console.WriteLine("Main method is called.");
    Constructors c1 = new Constructors();
    Constructors c2 = new Constructors();
    Constructors c3 = new Constructors();
    Console.ReadLine();
  }
}
```

Static Class: These are introduced in C# 2.0. If a class is explicitly declared by using static modifier we call it as a static class and this class can contain only static members in it. We can't create the instance of static class and more over it is not required also.

```
static class Class1
{
  //Define only static members.
}
```

Note: Console is a static class in our Libraries so each and every member of Console class is static member only and to check that, right click on Console class in Visual Studio and choose the option "Go to definition" which will open "Metadata" of that class.

Entity

Any living or non-living object that is associated with a set of attributes is known as an entity and application development is all about dealing and managing these entities only. To develop an application we follow the below process:

Step 1: Identify each and every entity that is associated with the application.

- **School Application:** Student, Teacher
- **Business Application:** Customer, Employee, Product, Supplier

Step 2: Identify each and every attribute of that entity.

- **Student:** Id, Name, Address, Phone, Class, Section, Fees, Marks, Grade
- **Teacher:** Id, Name, Address, Phone, Qualification, Subject, Salary, Designation
- **Customer:** Id, Name, Address, Phone, Balance, Account Type, EmailId, PanCard, Aadhar
- **Employee:** Id, Name, Address, Phone, Job, Salary, Department, EmailId, PanCard

Step 3: Design a DB based on the following guidelines:

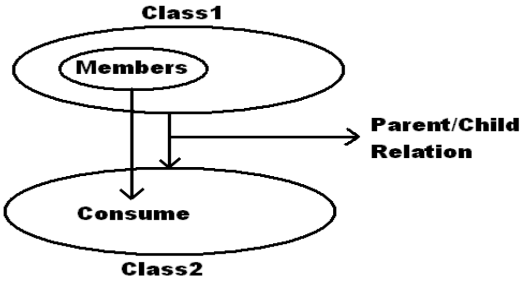
1. Create a table representing each entity.
2. Every column of the table should represent each attribute of the entity.
3. Each record under table should be a unique representation for the entity.

Step 4: Design an application by using any Programming Language of your choice which should act as an UI (User Interface) between the End User and Database in managing the data present under DB, by adopting following guidelines:

1. Define a class where each class should represent an entity.
2. Define properties where each property should be a representation for each attribute.
3. Each instance of the class we create will be a unique representation for each entity.

Inheritance

It is a process of consuming members that are defined in one class from other classes by establishing parent/child relation-ship between the classes, so that child class can consume members of its parent class as if they are owner of those members.



Note: Child class even if it can consume members of its parent class as an owner, still it can't access any private members of their parent.

Syntax: [<modifiers>] class <CC Name> : <PC Name>

Example:

```
class Class1 {
  -Define Members
}
class Class2 : Class1 {
  -Consume members of parent i.e. Class1 from here
}
```

To test inheritance, add a new class under the project naming it as “Class1.cs” and write the below code in it:

```
class Class1 {
  public Class1() {
    Console.WriteLine("Class1 constructor is called.");
  }
  public void Test1() {
    Console.WriteLine("Method 1");
  }
  public void Test2() {
    Console.WriteLine("Method 2");
  }
}
```

Now add another class in the project naming it as “Class2.cs” and write the below code in it:

```
class Class2 : Class1 {
  public Class2() {
    Console.WriteLine("Class2 constructor is called.");
  }
  public void Test3() {
    Console.WriteLine("Method 3");
  }
  public void Test4() {
    Console.WriteLine("Method 4");
  }
  static void Main() {
    Class2 c = new Class2();
    c.Test1(); c.Test2(); c.Test3(); c.Test4();
    Console.ReadLine();
 }
}
```

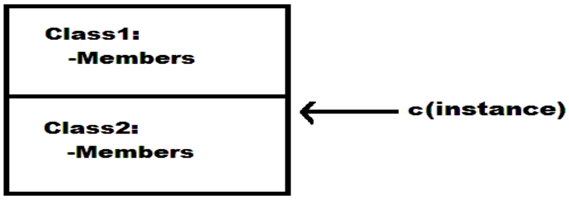
Rules and regulations that has to be followed while working with Inheritance:

Rule 1: In inheritance parent class Constructor must be accessible to child class or else inheritance will not be possible. The reason why parent’s Constructor should be accessible to child is, because when ever child class instance is created, control first jumps to child class Constructor and child class Constructor will in turn call its parent class Constructor for execution and to test this, place a break point at child class’s Main method and debug the code by hitting F11.

The reason why child Constructor calls its parent class Constructor is, because if child class wants to consume members of its parent class, they must be initialized first and then only child classes can consume them and we are already aware that members of a class are initialized by its own Constructor.

Note: Constructors are never inherited i.e. Constructors are specific to any class which can initialize members of that particular class only but not of parent or child classes.

When we create the instance of any class, it will first read all its parent classes to gather the information of members that are present under those classes, so in our previous case when the instance of Class2 is created it gathers information of Class1 also as following:



Rule 2: In inheritance child class can access members of their parent class whereas parent classes can never access members of their child class which are **purely defined** under the child class. To test this re-write the code under Main method of child class i.e. Class2 as following:

```
Class1 p = new Class1();
p.Test1(); p.Test2();        //Valid
//p.Test3(); p.Test4();      //Invalid and in-accessible
Console.ReadLine();
```

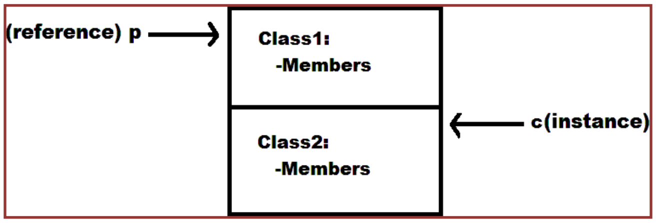
Rule 3: Earlier we have learnt that variable of a class can be initialized by using instance of same class to make it as a reference, for example:

```
Class2 c1 = new Class2();
Class2 c2 = c1;
```

Same as the above we can also initialize variables of parent class by using its child classes instance as following:

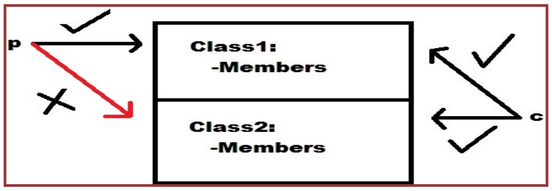
```
Class2 c = new Class2();
Class1 p = c;
```

In this case both parent class reference and child class instance will be accessing the same memory, but owner of that memory is child class instance.



In the above case even if parent class reference is initialized by the child class instance and consuming the memory of child class instance, now also it is not possible to access the member's which are **purely defined** under the child class and to test that rewrite the code under Main method of child class i.e. Class2 as following:

```
Class2 c = new Class2();
Class1 p = c;
p.Test1(); p.Test2();        //Valid
//p.Test3(); p.Test4();      //Invalid and in-accessible now also
Console.ReadLine();
```



Note: We can never initialize child class variables by using parent class instance either implicitly or explicitly also.

```
Class1 p = new Class1();    //Creating parent class instance
Class2 c = p;               //Invalid (Implicit conversion and compile time error)
Class2 c = (Class2)p;       //Invalid (Explicit conversion and runtime error)
```

We can initialize child class variables by using a parent class reference which is initialized by using the same child class instance by performing an explicit conversion.

Creating parent's reference by using child class instance:

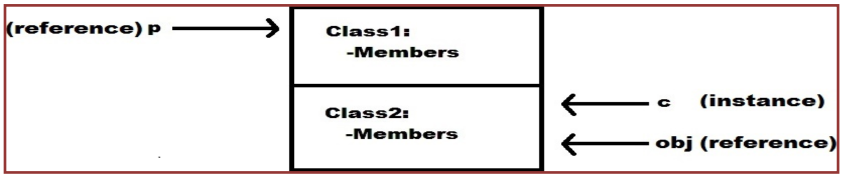
```
Class2 c = new Class2();
Class1 p = c;
```

Initializing child's variable by using the above parent's reference:

```
Class2 obj = (Class2)p;      //Valid (Explicit)
              or
Class2 obj = p as Class2;    //Valid (Explicit)
```

Child Instance	=> Parent Reference		//Valid
Child Instance	=> Parent Reference	=> Child Reference	//Valid
Parent Instance	=> Child Reference		//Invalid

Note: in the above case the new reference “obj” also start's accessing the same memory allocated for the instance “c” and with the new reference we call the members of both “Class1” and “Class2” also.

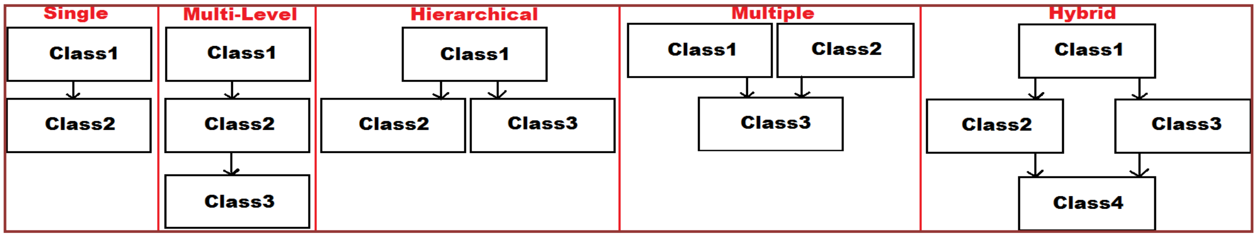


Rule 4: Each and every class that is pre-defined or user-defined has a default parent class i.e. Object class of System namespace. Object is the ultimate parent of all classes in .NET class hierarchy providing low level services to child classes. So every class by default contains 4 methods that are inherited from the “Object” Class and those are “Equals”, “GetHashCode”, “GetType” and “ToString”, and these 4 methods can be called or consumed from any class. To test this re-write code under Main method of child class (Class2) as following:

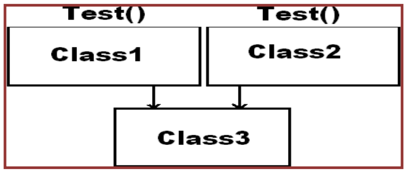
```
Object obj = new Object();
Console.WriteLine(obj.GetType() + "\n");
Class1 p = new Class1();
Console.WriteLine(p.GetType()  + "\n");
Class2 c = new Class2();
Console.WriteLine(c.GetType());
Console.ReadLine();
```

Types of Inheritance: This talk about no. of child classes a parent have or the no. of parent classes a child have. According to the standards of Object Oriented Programming we have 5 types of inheritances and they are:

i. Single
ii. Multi-Level
iii. Hierarchical
iv. Multiple
v. Hybrid



Rule 5: Both Java and .Net Language's doesn't provide the support for Multiple and Hybrid inheritances thru classes, and what they support is Single, Multi-Level and Hierarchical inheritances only because multiple inheritance suffers from ambiguity problem, for example:



Note: C++ Language supports all 5 types of inheritances because it is the 1st Object Oriented Programming Language that came into existence and at the time of its introduction, this problem was not anticipated.

Rule 6: In the first rule of inheritance we have discussed that whenever the instance of child class is created it will implicitly call its parent class constructor for execution, but this implicit calling will take place only if parent classes Constructor is "default or parameter less", whereas if at all the parent classes Constructor is parameterized then child class constructor can't implicitly call parent class Constructor for execution because it requires parameter values for execution. To resolve the above problem developer needs to explicitly call parent classes Constructor from child class constructor using "base" keyword and pass all the required parameter values.

To test the above re-write constructor of parent class i.e. Class1 as following:

```
public Class1(int i) {
  Console.WriteLine("Class1 constructor is called: " + i);
}
```

Now when we run child class i.e. Class2, we get an error stating that there is no value sent to formal parameter "i" of Class1 (Parent Class) and to resolve this problem re-write constructor of Class2 as following:

```
public Class2(int x) : base(x)  {
  Console.WriteLine("Class2 constructor is called.");
}
```

In the above case child classes constructor is also parameterized so while creating the instance of child class we need to explicitly pass all the required values to its constructor and those values are first loaded into the constructor and from there those values are passed to parent classes constructor thru the base keyword, and to test this go to Main method of Class2, and re-write the code in it as following:

```
Class2 c = new Class2(50); Console.ReadLine();
```

How do we use inheritance in application development?

Ans: Inheritance is a process which comes into picture from the initial stages of an application development. As discussed earlier, if we want to develop an Application we need to follow the below process:

Step 1: Identification of the entities.

E.g.: School Application: Student, TeachingStaff, NonTeachingStaff

Step 2: Identification of attributes for each entity.

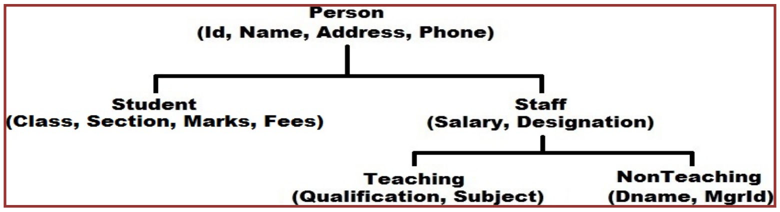
Student	Teaching Staff	Non Teaching Staff
Id	Id	Id
Name	Name	Name
Phone	Phone	Phone
Address	Address	Address
Class	Designation	Designation
Section	Salary	Salary
Marks	Qualification	Dname
Fees	Subject	MgrId

Step 3: Designing the Database.

Step 4: Developing an application that works like an UI.

While developing the UI, to bring re-usability into the applications we use inheritance and to do that follow the below guidelines:

1. Identify all the common attributes between entities and put them in a hierarchical order as below:



2. Now define classes based on the above hierarchy:

```
public class Person {
  public int Id;       public string Name, Phone, Address;
}
public class Student : Person {
  int Class;           char Section;              float Marks, Fees;
}
public class Staff : Person {
  public double Salary;       public string Designation;
}
public class Teaching : Staff  {
  string Subject, Qualification;
}
public class NonTeaching : Staff {
  int MgrId;           string Dname;
}
```

Polymorphism

Behaving in different ways depending upon the input received is known as Polymorphism i.e. when ever input changes then automatically the output or behaviour also changes accordingly. This can be implemented in our language in 3 different ways:

1. Overloading
2. Overriding
3. Hiding/Shadowing

Overloading: This is again of different types like Method Overloading, Operator Overloading, Constructor Overloading, Indexer Overloading and Deconstructor Overloading.

Method Overloading: It is an approach of defining multiple methods in a class with the same name by changing their parameters. Changing parameters means we can change any of the following:

1. Change the no. of parameters passed to method.
2. Change the type of parameters passed to method.
3. Change the order of parameters passed to method.

- public void Show()
- public void Show(int i)
- public void Show(string s)
- public void Show(int i, string s)
- public void Show(string s, int i)

Note: in overloading a return type change with out parameter change is not taken into consideration, for example:

public string Show() => Invalid

To test method overloading, add a new class in the project naming it as "OverloadMethods.cs" and write the following code in it:

```
class OverloadMethods {
 public void Show() {
  Console.WriteLine(1);
 }
 public void Show(int i) {
  Console.WriteLine(2);
 }
 public void Show(string s) {
  Console.WriteLine(3);
 }
 public void Show(int i, string s) {
  Console.WriteLine(4);
 }
 public void Show(string s, int i) {
  Console.WriteLine(5);
 }
 static void Main() {
   OverloadMethods obj = new OverloadMethods();
   obj.Show();      obj.Show(10);      obj.Show("Hello");      obj.Show(10, "Hello");      obj.Show("Hello", 10);
   Console.ReadLine();
 }
}
```

What is Method Overloading?

Ans: It's an approach of defining a method with multiple behaviours and those behaviours will vary based on the number, type and order of parameters. For example IndexOf is an overloaded method under String class which returns the index position of a characater or string based on input values of that method, for example:

```
string str = "Hello World";
str.IndexOf('o'); => 4 => Returns the first occurance of character
str.IndexOf('o', 5); => 7 => Returns the next occurance of character
```

* WriteLine method of Console class is also overloaded for printing any type of value that is passed as input to the method, as following:

- WriteLine()
- WriteLine(int value)
- WriteLine(bool value)
- WriteLine(double value)
- WriteLine(string value)
- WriteLine(string format, params object[] values)
- +12 more overloads

Inheritance based overloading: It's an approach of overloading parent classes' methods under the child class, and to do this child class doesn't require taking any permission from the parent class, for example:

```
Class1
public void Test()

Class2 : Class1
public void Test(int i)
```

Method Overriding: it's an approach of re-implementing parent classes' methods under child class exactly with the same name and signature (parameters).

Difference between Method Overloading and Method Overriding:

Method Overloading	Method Overriding
It's all about defining multiple methods with the same name by changing their parameters.	It's all about defining multiple methods with the same name and same parameters.
This can be performed with-in a class or between parent-child classes also.	This can be performed only between parent-child classes but can't be perfomed with-in a class.
To overload parent's method under child, child doesn't require any permission from parent.	To override parent's method under child, parent should first grant the permission to child.
This is all about defining multiple behaviours to a method.	This is all about changing existing behaviour of a parent's method under child.

How to override a parent classes method under child class?

Ans: To override any parent classes method under child class, first that method should be declared "overridable" by using "virtual" modifier in parent class as following: ***Class1 => public virtual void Show() //Overridable***

Every virtual method of parent class can be overriden by child class, if required by using "override" modifier as following: ***Class2 : Class1 => public override void Show() //Overriding***

Note: overriding virtual methods of parent class under child class is not mandatory for child class.

In overriding, parent class defines a method in it as virtual and gives it to the child class for consumption, so that it's giving a permission to the child class either to consume the method “as is” or override the method as per it's requirement, if at all the original behaviour of that method is not satisfactory to the child class.

To test inheritance based method overloading and method overriding, add a new class in the project naming it as LoadParent.cs and write the following code in it:

```
class LoadParent {
  public void Test() {
    Console.WriteLine("Parent Class Test Method Is Called.");
  }
  public virtual void Show() {
    Console.WriteLine("Parent Class Show Method Is Called.");
  }
  public void Display() {
    Console.WriteLine("Parent Class Display Method Is Called.");
  }
}
```

Now add another class in the project naming it as LoadChild.cs and write the following code in it:

```
class LoadChild : LoadParent {
  //Overloading parent's Test method in child
  public void Test(int i) {
    Console.WriteLine("Child Class Test Method Is Called.");
  }
  static void Main() {
    LoadChild c = new LoadChild();
    c.Test();         //Executes parent class Test method
    c.Test(10);       //Executes child class Test method
    c.Show();         //Executes parent class Show method
    c.Display();      //Executes parent class Display method
    Console.ReadLine();
  }
}
```

Inheritance-Based Overloading: In the above classes Test method of parent class has been overloaded in child class and then by using child class instance we are able to call both parent and child classes methods also, from the child class.

Method Overriding: In the above classes Show method of parent class is declared virtual which gives a chance for child classes to override that method but the child class did not override the method, so a call to that method by using child classes instance will invoke the parent classes Show method only and this proves us overriding is only optional and to confirm that run the child class LoadChild and watch the output of Show method.

In this case if child class overrides the parent classes virtual method then a call to that method by using child class instance will execute or invoke its own method but not of the parent classes, and to test that add a new method in class LoadChild as following:

```
//Overriding parent's Show method in child class
public override void Show() {
  Console.WriteLine("Child Class Show Method Is Called.");
}
```

Now if we run the child class i.e. LoadChild and watch the output of Show method we will notice child classes Show method getting executed in place of the parent classes Show method and this is what we call as changing the behaviour.

Can we override any parent classes' methods under child classes with out declaring them as virtual?
Ans: No.

Can we re-implement any parent classes' methods under the child classes with out declaring them as virtual?
Ans: Yes.

We can re-implement a parent class method under the child class by using 2 different approaches:

- Overriding
- Hiding/Shadowing

Method Hiding/Shadowing: This is also an approach of re-implementing parent classes methods under child class exactly with the same name and signature just like overriding but the difference between the 2 is; in overriding child class can re-implement only virtual methods of parent class where as in-case of hiding/shadowing child class can re-implement any method of the parent class i.e. even if the method is not declared as virtual also re-implementation can be performed.

```
Class1            => public void Display()

Class2 : Class1   => public [new] void Display()      //Hiding/Shadowing
```

In the above case using "new" keyword while re-implementing the method in child class is only optional and if we don't use it, compiler gives a warning message at the time of compilation, telling that there is already a method with the same name in parent class and your new method in child class will hide that old method, so by using "new" keyword we are informing compiler that we are intentionally defining a new method with the same name and signature under our child class.

Before testing hiding/shadowing first run the child class i.e. LoadChild and watch the output of Display method and here we notice that parent classes Display method getting executed, now add a new method in the child class LoadChild as following:

```
//Hiding/Shadowing parent's Display method in child
public new void Display() {
  Console.WriteLine("Child Class Display Method Is Called.");
}
```

Now run the child class LoadChild again and watch the difference in output i.e. in this case child classes Display() method is called in place of parent class Display() method.

In the above 2 classes we have performed the following:

LoadParent

```
public void Test()
public virtual void Show()
public void Display()
```

LoadChild : LoadParent

```
public void Test(int i)              => Overloading
public override void Show()          => Overriding
public new void Display()            => Hiding/Shadowing
```

In case of Overriding and Hiding, after re-implementing the parent classes methods under child class, instance of child class starts calling it's own methods but not of parent class, whereas if required there is still a chance of calling those parent class methods from child class in 2 different ways:

1. By creating the parent classes instance under the child class we can call parent class methods from child class and to test that re-write code under Main method of child class i.e. LoadChild as following:

    ```
    LoadParent p = new LoadParent();
    p.Show();          //Executes parent class Show method
    p.Display();       //Executes parent class Display method

    LoadChild c = new LoadChild();
    c.Show();          //Executes child class Show method
    c.Display();       //Executes child class Display method
    Console.ReadLine();
    ```

2. By using base keyword also we can call parent class methods from child class, but keywords like "this" and "base" can't be used in static blocks.

To test this first add 2 new methods under the child class i.e. LoadChild as following:

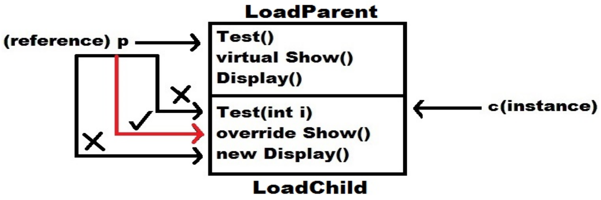
```
public void PShow() {
  base.Show();
}
public void PDisplay() {
  base.Display();
}
```

In the above case the 2 new methods we defined in child class acts as an interface in calling parent classes methods from the child class, so now by using child class instance only we can call both parent and child classes methods also. To test this re-write code under Main method of child class i.e. LoadChild as following:

```
LoadChild c = new LoadChild();
c.PShow();        //Executes parent class Show method
c.PDisplay();     //Executes parent class Display method
c.Show();         //Executes child class Show method
c.Display();      //Executes child class Display method
Console.ReadLine();
```

Note: Earlier in the 3rd rule of inheritance we have learnt that parent class reference even if created by using the child class instance can't access any members of the child class which are **purely defined** under child class but we have an exemption for that rule, that is, parent's reference can call or access **overriden members** of the child class because overriden members are not considered as pure child class members because they have been re-implemented with permission from the parent class. To test that re-write code under Main method of child class i.e. LoadChild as following:

```
LoadChild c = new LoadChild();
LoadParent p = c;
p.Show(); //Executes child class Show method
p.Display(); //Executes parent class Display method only
Console.ReadLine();
```



In the above case Display is considered as pure child class member only because it's re-implemented by child class with out taking any permission from parent, so parent will never recognize it.

Polymorphism is divided into 2 types:

1. Static or Compile-time Polymorphism
2. Dynamic or Run-time Polymorphism

In static or compile-time polymorphism, the decision which polymorphic method has to be executed for a method call is performed at compile time. Method overloading is an example for this and here compiler identifies which overloaded method it has to execute for a particular method call at the time of program compilation by checking the type and number of parameters that are passed to the method and if no method matchs the method call it will give an error.

In dynamic or run-time polymorphism, the decision which polymorphic method has to be executed for a method call is made at runtime rather than compile time. Run-time polymorphism is achieved by method overriding because method overriding allows us to have methods in the parent and child classes with the same name and the same parameters. By runtime polymorphism, we can point to any child class by using the reference of the parent class which is initialized by child class object, so the determination of the method to be executed is based on the instance being referred to by reference.

Static Polymorphism	**Dynamic Polymorphism**
1. Occurs at compile-time.	1. Occurs at runtime.
2. Achieved through static binding.	2. Achieved through dynamic binding.
3. Method overloading should exist.	3. Method overriding should exist.
4. Inheritance is not involved.	4. Inheritance is involved.
5. Happens in the same class.	5. Happens between parent-child classes.
6. Reference creation thru instance is not required.	6. Requires parent class reference creation thru child class instance.

Operator Overloading

It's an approach of defining mutiple behaviours to an operator, which varies based on the operands between which we use the operator. For example: "+" is an addition operator when used between numeric operands and it is a concatenation operator when used between string operands.

Number + Number => Addition

String + String => Concatenation

The behaviour for an operator is pre-defined i.e. developers of the language have already implemented, logic that has to be executed when an operator is used between 2 operands under the libraries of a language with the help of a special method known as "Operator Method".

Syntax of an Operator Method:

```
[<modifiers>] static <type> operator <opt>(<operand types>) {
 -Logic
}
```

- Operator methods must be static only.
- <type> refers to the return type of method i.e. when the operator is used between 2 types what should be the result type.
- operator is name of the method, which should be in lower case and can't be changed.
- <opt> refers to the operator for which we want to write behaviour like "+" or "-" or "==", etc.
- <operand types> refers to type of operands between which we want to use the operator.

Under Libraries, operator methods have been defined as following:

```
public static int operator +(int a, int b)
public static string operator +(string a, string b)
public static string operator +(string a, int b)
public static bool operator >(int a, int b)
public static float operator +(int a, float b)
public static decimal operator +(double a, decimal b)
public static bool operator ==(string a, string b)
public static bool operator !=(string a, string b)
```

Note: same as the above we can also define operator methods for using an operator between new types of operands.

To test "Operator Overloading", "Method Overriding" and "Method Hiding/Shadowing" add a new class naming it as "Matrix.cs" under the project and write the following code:

```
class Matrix {
 int a, b, c, d;  //Declaring attributes for a 2 * 2 Matrix
 public Matrix(int a, int b, int c, int d) {
  //Initializing attributes of the Matrix
  this.a = a; this.b = b; this.c = c; this.d = d;
 }
 //Overriding the ToString() method inherited from Object class to return values of the Matrix in 2 * 2 format
 public override string ToString() {
  return a + "  " + b + "\n" + c + "  " + d + "\n";
 }
 //Implementing the + operator so that it can be used between 2 Matrix operands
 public static Matrix operator +(Matrix obj1, Matrix obj2) {
  Matrix obj = new Matrix(obj1.a + obj2.a, obj1.b + obj2.b, obj1.c + obj2.c, obj1.d + obj2.d);
  return obj;
 }
 //Implementing the - operator so that it can be used between 2 Matrix operands
 public static Matrix operator -(Matrix obj1, Matrix obj2) {
  Matrix obj = new Matrix(obj1.a - obj2.a, obj1.b - obj2.b, obj1.c - obj2.c, obj1.d - obj2.d);
  return obj;
 }
 //Re-Implementing the == operator using Hiding/Shadowing so that it can be used between 2 Matrix's to perform
   values == comparision b'coz original implementation for the operator in Object class is reference == comparision.
 public static bool operator ==(Matrix obj1, Matrix obj2) {
  if (obj1.a == obj2.a && obj1.b == obj2.b && obj1.c == obj2.c && obj1.d == obj2.d) {
   return true;
  }
  else {
   return false;
  }
 }
 //Re-Implementing the != operator using Hiding/Shadowing so that it can be used between 2 Matrix's to perform
   values != comparision b'coz original implementation for the operator in Object class is reference != comparision.
 public static bool operator !=(Matrix obj1, Matrix obj2) {
  if (obj1.a != obj2.a || obj1.b != obj2.b || obj1.c != obj2.c || obj1.d != obj2.d) {
   return true;
  }
  else {
   return false;
  }
 }
}
```

ToString is a method defined in parent class "Object" and by default that method returns "Name" of the type to which an instance belongs when we call it on any type's instance.

ToString() method is declared as virtual under the class "Object" so any child class can override it as per their requirements as we performed it in our "Matrix" class to change the behaviour of that method, so the new method will return values that are associated with "Matrix" but not the type name.

The "==" and "!=" operators are also implemented in the parent class "Object", but their original behaviour is to perform a reference equal and reference not-equal comparision between type instances but not values equal and values non-equal comparision. We can also change the behaviour of those operator methods by using the concept of hiding (but not overriding because they are not declared as virtual) as we have done in our Matrix class, so that now the 2 operators will perform values equal and values not-equal comparision in place of reference equal and reference not-equal comparision.

To consume all the above add a new class TestMatrix.cs and write the following code:

```
class TestMatrix {
 static void Main() {
  //Creating 4 instances of Matrix class with different values
  Matrix m1 = new Matrix(20, 19, 18, 17);          Matrix m2 = new Matrix(15, 14, 13, 12);
  Matrix m3 = new Matrix(10, 9, 8, 7);             Matrix m4 = new Matrix(5, 4, 3, 2);

  //Performing Matrix Arithmatic
  Matrix m5 = m1 + m2 + m3 + m4;                   Matrix m6 = m1 - m2 - m3 - m4;

  //Printing values of each Matrix:
  Console.WriteLine(m1);         Console.WriteLine(m2);         Console.WriteLine(m3);
  Console.WriteLine(m4);         Console.WriteLine(m5);         Console.WriteLine(m6);

  //Performing Matrix equal comparision
  if (m1 == m2) { Console.WriteLine("Yes, m1 is equal to m2."); }
  else { Console.WriteLine("No, m1 is not equal to m2."); }

  //Performing Matrix not equal comparision
  if (m1 != m2) { Console.WriteLine("Yes, m1 is not equal to m2."); }
  else { Console.WriteLine("No, m1 is equal to m2."); }
  Console.ReadLine();
 }
}
```

In the above case when we call WriteLine method by passing Matrix class instance as a parameter to it, will internally invoke the overloaded WriteLine Method which takes "Object" as a parameter and that method will internally call ToString() method on that instance, and because we have overwritten the ToString method in our Matrix class, a call to it in WriteLine method will invoke Matrix classes ToString method which will return the values that are associated with Matrix instance and prints them in a 2 * 2 Matrix format.

Constructor Overloading

Just like methods in a class can be overloaded, constructors in a class also can be overloaded and it is called as “Constructor Overloading”. It’s an approach of defining multiple constructors under a class and if constructors of a class are overloaded then instance of that class can be created by using any available constructor i.e. it is not mandatory to call any particular constructor for instance creation. To test this, add a new code file under the project naming it as “TestOverloadCons.cs” and write the below code in it:

```
using System;
namespace OOPSProject {
 class OverloadCons {
  int i; bool b;
  public OverloadCons() {
   //Initializes both i & b with default values
  }
  public OverloadCons(int i) {
   //Initializes i with given value and b with default value
   this.i = i;
  }
  public OverloadCons(bool b) {
   //Initializes i with default value and b with given value
   this.b = b;
  }
  public OverloadCons(int i, bool b) {
   //Initializes both i & b with given values
   this.i = i; this.b = b;
  }
  public void Display() {
   Console.WriteLine("Value of i is: {0} and value of b is: {1}", i, b);
  }
 }
 class TestOverloadCons {
  static void Main() {
   OverloadCons c1 = new OverloadCons(); c1.Display();
   OverloadCons c2 = new OverloadCons(10); c2.Display();
   OverloadCons c3 = new OverloadCons(true); c3.Display();
   OverloadCons c4 = new OverloadCons(10, true); c4.Display();
   Console.ReadLine();
  }
 }
}
```

By overloading constructors in a class we get a chance to initialize fields of that class in 3 different ways:

1. With a default or “parameter-less” constructor defined in class we can initialize all fields of that class with default values.
2. With a parameterized constructor defined in class we can initialize all fields of that class with given values.
3. With a parameterized constructor defined in class we can initialize some fields of that class with default values and some fields with given values.

Note: If a class contains multiple attributes in it and if we want to initialize them in a “mix & match” combination then we overload constructors, and the no. of constructors to be defined will be 2 power “n” where “n” is the no. of attributes. In our above class we have 2 attributes so we have defined 4 constructors.

Copy Constructor

It is a constructor using which we can create a new instance of the class with the help of an existing instance of the same class, which copies the attribute values from the existing instance into the new instance and the main purpose of this constructor is to initialize a new instance with the values from an existing instance. The “Formal Parameter Type” of a copy constructor will be the same “Class Type” in which it is defined.

To test Copy Constructors add a new class under the project naming it as “CopyConDemo.cs” and write the following code:

```
class CopyConDemo {
 int Custid;       string Name;       double Balance;
 public CopyConDemo(int Custid) {
   this.Custid = Custid;        this.Name = "Scott";     this.Balance = 5000.00;
 }
 public CopyConDemo(CopyConDemo cd) //Copy Constructor {
   this.Custid = cd.Custid;     this.Name = cd.Name;     this.Balance = cd.Balance;
 }
 public void Display() {
   Console.WriteLine("Custid: {0}; Name: {1}; Balance: {2}", Custid, Name, Balance);
 }
 static void Main() {
   CopyConDemo cd1 = new CopyConDemo(101);
   CopyConDemo cd2 = new CopyConDemo(cd1);       //Calling Copy Constructor
   cd1.Display(); cd2.Display(); Console.WriteLine();
   cd1.Balance = 10000.00; cd1.Display(); cd2.Display(); Console.WriteLine();
   cd2.Balance = 20000.00; cd1.Display(); cd2.Display(); Console.ReadLine();
 }
}
```

In the above case “cd2” is a new instance of the class which is created by copying the values from “cd1” and here any changes that are performed on members of “cd1” will not reflect to members of “cd2” and vice versa because they have their own individual memory which is not accessbile to others.

Types of Constructors: constructors are divided into 5 Categories like:

1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor
4. Static Constructor
5. Private Constructor

Default Constructor: a constructor defined with out any parameters is known as a default constructor, which will initialize fields of a class with default values. If a class is not defined with any explicit constructor then the class will contain an implicit default constructor.

Parameterized Constructor: if a constructor is defined with atleast 1 parameter then we call it as parameterized constructor and these constructors must be explicitly defined but never implicitly defined. Parameterized constructors are used for initializing fields of a class with given set of values which we can pass while creating the instance of that class.

Copy Constructor: it’s a constructor which takes the same type as its “Parameter” and initializes the fields of class by copying values from an existing instance of the same class. A Copy constructors will not create a reference to the class i.e. it will create a new instance for the class by allocating memory for all the members of that class and very importantly any changes made on the source will not reflect to the new instance and vice-versa.

Static Constructor: if a constructor is defined explicitly by using static modifier we call it as a static constructor and this constructor is the first block of code which executes under the class and they are responsible for initializing static fields and more over these constructors can't be overloaded because they can't be parameterized. This constructor is called implicitly before the first instance is created or any static members are referenced.

Private Constructor: If a constructor is explicitly declared by used private modifier we call it as a private constructor. If a class contains only private constructors and no public constructors, other classes cannot create instances of that class. The use of private constructor is to serve singleton classes where a singleton class is one which limits the number of instances created to one.

Sealed Class: if a class is explicitly declared by using sealed modifier we call it as a sealed class and these classes can't be inherited by other classes, for example:

```
sealed class Class1 {
  -Members
}
```

In the above case Class1 is a sealed class so it can't be inherited by any other class, for example:

class Class2 : Class1 ***=> Invalid***

Note: even if a sealed class can't be inherited it is still possible to consume the members of a sealed class by creating its instance, for example String is a sealed class in our libraries, so we can't inherit from that class but we can still consume it in all our classes by creating the instance of String class.

Sealed Method: If a parent class method can't be overriden under a child class, then we call that method as Sealed Method. By default every method of a class is sealed because we can never override any method of parent class under the child class unless the method is declared as virtual. If a method is declared as virtual under a class then any child class of it in a linear hierarchy can override that method, for example:

```
Class1
public virtual void Show()

Class2 : Class1
public override void Show()                //Valid

Class3 : Class2
public override void Show()                //Valid
```

Note: in the above case even if Class2 is not overriding the method also Class3 can override the method.

When a child class is overriding any of its parent classes' virtual methods, it can seal those methods by using sealed modifier on them, so that furthur overriding of those methods can't be performed by its child classes, for example:

```
Class1
public virtual void Show()

Class2 : Class1
public sealed override void Show()         //Valid

Class3 : Class2
public override void Show()                //Invalid
```

Note: in the above case Class2 has sealed the method while overriding the method so Class3 can't override the method.

Abstract Class and Abstract Method

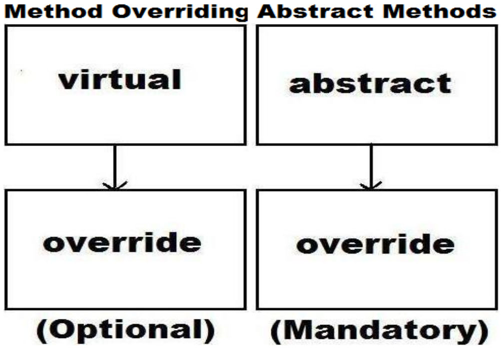
Abstract Method: a method without any body is known as abstract method i.e. an abstract method contains only declaration without any implementation. To declare a method as abstract it is must to use "abstract" modifier on that method explicitly.

Abstract Class: a class under which we declare abstract members is known as abstract class and must be declared by using "abstract" modifier.

```
abstract class Math {
  public abstract void Add(int x, int y);
}
```

Note: each and every abstract member of an abstract class must be implemented by the child class of abstract class without fail (mandatory).

The concept of abstract method's is near similar to method overriding i.e. in case of overriding, if at all a parent class contains any methods declared as virtual then child classes can re-implement those methods by using override modifier whereas in case of abstract methods if at all a parent class contains any methods declared as abstract then every child class must implement all those methods by using the same override modifier only.



An abstract class can contain both abstract and non-abstract (concrete) members also, and if at all any child class of the abstract class wants to consume any non-abstract members of its parent, must first implement all the abstract members of its parent.

Abstract Class:

- Non-Abstract/Concrete Members
- Abstract Members

Child Class of Abstract Class:

- Implement each and every abstract member of parent class
- Now only we can consume concrete members of parent class

Note: we can't create the instance of an abstract class, so abstract classes are never useful to themselves, i.e an abstract class is always a parent providing services to child classes.

To test an abstract class and abstract methods add a new class under the project naming it as "AbsParent.cs" and write the following code in it:

```
abstract class AbsParent {
  public void Add(int x, int y) {
    Console.WriteLine(x + y);
  }
  public void Sub(int x, int y) {
    Console.WriteLine(x - y);
  }
  public abstract void Mul(int x, int y);
  public abstract void Div(int x, int y);
}
```

Now add another class “AbsChild.cs” to implement the above abstract classes - abstract methods and write the following code in it:

```
class AbsChild : AbsParent {
  public override void Div(int x, int y) {
    Console.WriteLine(x / y);
  }
  public override void Mul(int x, int y) {
    Console.WriteLine(x * y);
  }
  static void Main() {
    AbsChild c = new AbsChild();
    c.Add(100, 23); c.Sub(78, 19); c.Mul(15, 18); c.Div(580, 24);
    Console.ReadLine();
  }
}
```

Note: even if the instance of an abstract class can’t be created it is still possible to create the reference of an abstract class by using it’s child classes instance, and with that reference we can call each and every concrete method of abstract class as well as its abstract methods which are implemented by child class, and to test this re-write code under Main method of class “AbsChild” as following:

```
AbsChild c = new AbsChild();
AbsParent p = c;
p.Add(100, 23); p.Sub(78, 19); p.Mul(15, 18); p.Div(580, 24);
Console.ReadLine();
```

What is the need of Abstract Classes and Abstract Methods?

Ans: The concept of Abstract Classes and Abstract Methods is an extension to inheritance i.e. in inheritance we have already learnt that, we can eliminate redundancy between entities by identifying all the common attributes between the entities we wanted to implement, by defining them under a parent class.

For example if we are designing a Mathematical Application then we follow the below process of implementation:

Step1: Identifying the Entities of Mathematical Application.

- *Cone, Circle, Triangle, Rectangle*

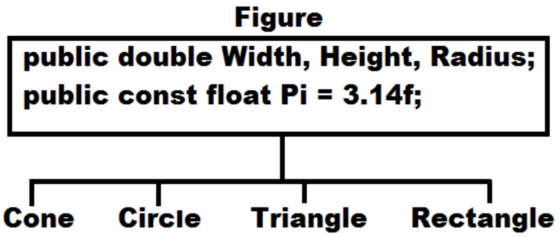
Step 2: Identifying the Attributes of each Entity.

- Cone: Height, Radius, Pi
- Circle: Radius, Pi
- Triangle: Base (Width), Height
- Rectangle: Length (Height), Breadth (Width)

Step 3: Designing the database by following the rules we learnt in entity implementations.

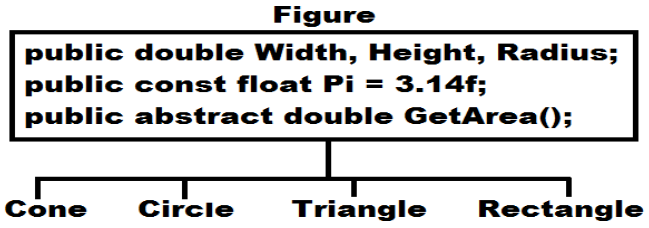
Step 4: Develop an application and define classes representing each and every entity.

Note: while defining classes representing entities, as learnt in inheritance first we need to define a parent class with all the common attributes as following:



```
public double Width, Height, Radius;
public const float Pi = 3.14f;
```

In the above case, “Figure” is a Parent class containing all the common attributes between the 4 entities. Now we want a method that returns Area of each and every figure, and even if the method is common for all the classes, still we can't define it in the parent class Figure, because the formula to calculate area varies from figure to figure. So to resolve the problem, without defining the method in parent class we need to declare it in the parent class Figure as abstract and restrict each and every child class to implement logic for that method as per their requirement as following:



```
public double Width, Height, Radius;
public const float Pi = 3.14f;
public abstract double GetArea();
```

In the above case because GetArea() method is declared as abstract in the parent class, so it is mandatory for all the child classes to implement that method under them, but logic can be varying from each other whereas signature of the method can't change and now all the child classes have to do the following:

1. Define a constructor to initialize the attributes that are required for that entity.
2. Implement GetArea() method and write logic for calculating the Area of that corresponding figure.

To test the above add a new “Code File” under project naming it as “TestFigures.cs” and write the following code:

```
using System;
namespace OOPSProject {
 public abstract class Figure {
  public double Width, Height, Radius;
  public const float Pi = 3.14f;
  public abstract double GetArea();
 }
 public class Cone : Figure {
  public Cone(double Radius, double Height) {
    this.Radius = Radius; this.Height = Height;
  }
  public override double GetArea() {
    return Pi * Radius * (Radius + Math.Sqrt((Height * Height) + (Radius * Radius)));
  }
 }
 public class Circle : Figure {
  public Circle(double Radius) {
    this.Radius = Radius;
  }
  public override double GetArea() {
    return Pi * Radius * Radius;
  }
 }
 public class Triangle : Figure {
  public Triangle(double Base, double Height) {
    this.Width = Base; this.Height = Height;
  }
  public override double GetArea() {
    return 0.5 * Width * Height;
  }
 }
 public class Rectangle : Figure {
  public Rectangle(double Length, double Breadth) {
    this.Width = Length; this.Height = Breadth;
  }
  public override double GetArea() {
    return Width * Height;
  }
 }
 class TestFigures {
  static void Main() {
    Cone cone = new Cone(12.54, 67.34);
    Console.WriteLine("Area of Cone: " + cone.GetArea());
    Circle circ = new Circle(45.29);
    Console.WriteLine("Area of Circle: " + circ.GetArea());
    Triangle trin = new Triangle(67.32, 56.98);
    Console.WriteLine("Area of Triangle: " + trin.GetArea());
    Rectangle rect = new Rectangle(29.45, 32.78);
    Console.WriteLine("Area of Rectangle: " + rect.GetArea());
    Console.ReadLine();
  }
 }
}
```

Interface

Interface is also a user-defined type like a class but can contain only “Abstract Members” in it and all those abstract members should be implemented by a child class of the interface.

Non-Abstract Class: Contains only non-abstract/concrete members
Abstract Class: Contains both non-abstract/concrete and abstract members
Interface: Contains only abstract members

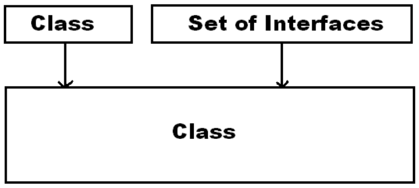
Just like a class can have another class as its parent, it can also have an interface as its parent but the main difference is if a class is a parent we call it as inheriting whereas if an interface is a parent we call it as implementing.

Inheritance is divided into 2 categories:

1. Implementation Inheritance
2. Interface Inheritance

If a class is inheriting from another class we call it as Implementation Inheritance whereas if a class is implementing an interface we call it as Interface Inheritance. Implementation Inheritance provides re-usability because by inheriting from a class we can consume members of a parent in child class whereas Interface Inheritance doesn't provide any re-usability because in this case we need to implement abstract members of a parent in child class, but not consume.

Note: we have already discussed in the 5th rule of inheritance that Java and .NET Languages doesn’t support multiple inheritance thru class, because of ambiguity problem i.e. a class can have 1 and only 1 immediate parent class to it; but both in Java and .NET languages multiple inheritance is supported thru the interfaces i.e. a class can have more than 1 interface as its immediate parent.



Syntax to define a interface:

```
[<modifiers>] interface <Name> {
 -Abstract member declarations.
}
```

- We can't declare any fields under an interface.
- Defualt scope for members of an interface is public whereas it is private in case of a class.
- Every member of an interface is by default abstract, so we again don't require using abstract modifier on it.
- Just like a class can inherit from another class, an interface can also inherit from another interface, but not from a class.

Adding an Interface under Project: Just like we have “Class Item Template” in “Add New Item” window to define a class we are also provided with an “Interface Item Template” to define an Interface. To test working with interfaces add 2 interfaces under the project naming them as IMath1.cs, IMath2.cs and write the following code:

```
interface IMath1  {                          interface IMath2 {
  void Add(int a, int b);                      void Mul(int a, int b);
  void Sub(int a, int b);
 }
```

```
  void Div(int a, int b);
 }
```

To implement methods of both the above interfaces add a class under project naming it as ClsMath.cs and write the following code:

```
class ClsMath : Program, IMath1, IMath2 {
 public void Add(int a, int b) {
  Console.WriteLine(a + b);
 }
 public void Sub(int a, int b) {
  Console.WriteLine(a - b);
 }
 public void Mul(int a, int b) {
  Console.WriteLine(a * b);
 }
 public void Div(int a, int b) {
  Console.WriteLine(a / b);
 }
 static void Main() {
  ClsMath obj = new ClsMath();
  obj.Add(100, 34); obj.Sub(576, 287); obj.Mul(12, 38); obj.Div(456, 2);
  Console.ReadLine();
 }
}
```

Points to Ponder:

1. The implementation class can inherit from another class and also implement "n" no. of interfaces, but class name must be first in the list followed by interface names.

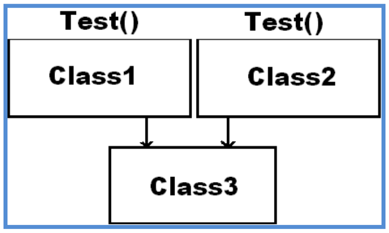
   E.g.: *class ClsMath : Program, IMath1, IMath2*
2. While declaring abstract members in an interface we don't require using "abstract" modifier on them and in the same way while implementing those abstract members we don't require to use "override" modifier also.

Just like we can't create instance of an abstract class, we can't create instance of an interface also; but here also we can create a reference of interface by using it's child class instance and with that reference we can call all the members of parent interface which are implemented in child class and to test this re-write code under Main method of class "ClsMath" as following:

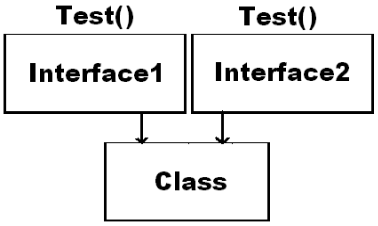
```
ClsMath obj = new ClsMath();
IMath1 i1 = obj; IMath2 i2 = obj;
i1.Add(100, 34); i1.Sub(576, 287);
i2.Mul(12, 38); i2.Div(456, 2);
Console.ReadLine();
```

Multiple Inheritance with Interfaces:

Earlier in the 5th rule of inheritance we have discussed that Java and .Net Languages doesn't support multiple inheritances thru classes because of ambiguity problem.



Whereas in Java and .Net Languages multiple inheritance is supported thru interfaces i.e. a class can have any no. of interfaces as its parent, but still we don't come across any ambiguity problem because child class of an interface is not consuming parent's members but implements them.



If we come across any situation as above we can implement the interface methods under class by using 2 different approaches:

1. Implement the method of both interfaces only for 1 time under the class and both interfaces will assume the implemented method is of its only and in this case we can call the method directly by using class instance.
2. We can also implement the method of both interfaces separately for each interface under the class by pre-fixing interface name before method name and we call this as **explicit implementation**, but in this case we need to call the method by using reference of interface that is created with the help of a child class instance.

To test the above add 2 new interfaces under the project naming them as Interface1.cs, Interface2.cs and write the following code:

```
interface Interface1 {
  void Test();
  void Show();
}
```

```
interface Interface2 {
  void Test();
  void Show();
}
```

Now add a new class under the project naming it as ImplClass.cs for implementing both the above interfaces and write the following code:

```
class ImplClass : Interface1, Interface2 {
  public void Test() {          //Implementing Test method using 1st approach
    Console.WriteLine("Method declared under 2 interfaces.");
  }
  void Interface1.Show() {   //Implementing Show method using 2nd approach
    Console.WriteLine("Method declared under Interface1.");
 }
  void Interface2.Show() {   //Implementing Show method using 2nd approach
    Console.WriteLine("Method declared under Interface2.");
 }
 static void Main() {
  ImplClass obj = new ImplClass();
  obj.Test();
  Interface1 i1 = obj; Interface2 i2 = obj;
  i1.Show(); i2.Show();
  Console.ReadLine();
 }
}
```

Structure

Structure is also a user-defined type like a class, which can contain members what a class can contain like constructor, static constructor, constants, fields, methods, properties, indexers, operators and events.

Differences between Class and Structure

Class	Structure
This is a reference type.	This is a value type.
Memory is allocated for its instances on Managed Heap so we get the advantage of Automatic Memory Management thru Garbage Collector.	Memory is allocated for its instances on Stack, so Automatic Memory Management is not available but faster in access.
Recommended for representing entities with larger volumes of data.	Recommended for representing entities with smaller volumes of data.
All pre-defined reference types in our Libraries like string (System.String) and object (System.Object) are defined as classes.	All pre-defined value types in our Libraries like int (System.Int32), float (System.Single), bool (System.Boolean), char (System.Char) and Guid (System.Guid) are defined as structures.
"new" keyword is mandatory for creating the instance and in this process we need to call any constructor that is available in the class.	"new" keyword is optional for creating the instance and if "new" is not used it will call default constructor which is defined implicitly, whereas it is still possible to use "new" and call other constructors also.
Contains an implicit default constructor if no constructor is defined explicitly.	Contains an implicit default constructor every time.
We can declare fields and those fields can be initialized at the time of declaration.	We can declare fields but those fields can't be initialized at the time of declaration.
Fields can also be initialized thru a constructor as well as referring thru instance also we can initialize them.	Fields can only be initialized thru a constructor as well as referring thru instance also we can initialize them.
Constructor is mandatory for creating the instance which can either be default or parameterized also.	Default constructor is mandatory for creating the instance without using new keyword and apart from that we can also define parameterized constructors.
Developers can define any constructor like default or parameterized also, or else implicit default constructor gets defined.	Developers can define parameterized constructors only because there is always an implicit default constructor, and this default constructor is mandatory if at all we want to create instance without using "new" keyword.
If defined with “0” constructors after compilation there will be “1” constructor and if defined with “n”	If defined with “0” constructors after compilation here also there will be “1” constructor whereas if defined

constructors after compilation there will be “n” constructors only.	with “n” constructors after compilation there will be “n + 1” constructors.
Supports both, implementation as well as interface inheritances also i.e. a class can inherit from another class as well as implement an interface also.	Supports only interface inheritance but not implementation inheritance i.e. a structure can implement an interface but can’t inherit from another structure.

Syntax to define a structure:

```
[<modifiers>] struct <Name> {
 -Define Members
}
```

Adding a Structure under Project: we are not provided with any Structure Item template in the add new item window, like we have class and interface item templates, so we need to use code file item template to define a structure under the project. Add a Code File under project, naming it as “MyStruct.cs” and write the following in it:

```
using System;
namespace OOPSProject {
  struct MyStruct {
    int x;
    public MyStruct(int x) {
      this.x = x;
    }
    public void Display() {
      Console.WriteLine("Method in a structure: " + x);
    }
    static void Main() {
      MyStruct m1 = new MyStruct(); m1.Display();
      MyStruct m2; m2.x = 10; m2.Display();
      MyStruct m3 = new MyStruct(20); m3.Display();
      Console.ReadLine();
    }
  }
}
```

Consuming a Structure: we can consume a Structure and its members from another structure or a class also; but only by creating its instance because structure doesn't support inheritance. To test this, add a new class under the project naming it as “TestStruct.cs” and write the following code:

```
struct TestStruct {
  static void Main() {
    MyStruct obj = new MyStruct(50);
    obj.Display();
    Console.ReadLine();
 }
}
```

Extension Methods

It's a new feature added in C# 3.0 which allows adding new methods to existing types (class/structure) without creating a new derived type or re-compiling or modifying the original type. Extension methods are a special kind of static methods, but they are called or consumed as if they are non-static methods. There is no difference between calling an extension method and methods that are directly defined in the type.

Rules for defining Extension Methods:

1. Extension methods should be defined in a static class only.
2. The first parameter of an extension method is the type name to which the method has to be bound with, pre-fixed by “this” keyword and we call this parameter as binding parameter which will not be taken into consideration while calling the method.
3. We can also define any additional parameters (formal parameters) to an extension method from 2nd place of parameter list and all those parameters will be taken into consideration while calling the method.

Note: If an extension method is defined with “n” no. of parameters, while calling the method there will be “n - 1” parameters only, because it excludes the binding parameter.

4. We can't bind an extension method with multiple types because it can have only 1 binding parameter.
5. Extension methods can't access private members in the original type, but can access other members.
6. If we create an extension method which has the same signature method inside the original type we are extending, then the extension method will not be called.

To test Extension Methods add a new class naming it as ExtensionMethods.cs and write the following code:

```
static class ExtensionMethods {
 public static void Test5(this Program p, int i) {  //Adding extension method to user-defined class Program
  Console.WriteLine("Extension Method into Program class: " + i);
 }
 public static void Show(this MyStruct m) {  //Adding extension method to user-defined structure MyStruct
  Console.WriteLine("Extension method into MyStruct structure.");
 }
 public static ulong Factorial(this UInt32 i) { //Adding extension method to pre-defined structure UInt32
  if (i == 0 || i == 1) { return 1; }
  if (i == 2) { return 2; }
  else { return i * Factorial(i - 1); }
 }
 public static string ToPascal(this String OldStr) {  //Adding extension method to pre-defined class (Sealed Class)
  if (OldStr.Trim().Length > 0) {
   string NewStr = null;
   OldStr = OldStr.ToLower();
   string[] sarr = OldStr.Split(' ');
   foreach (string str in sarr) {
    char[] carr = str.ToCharArray(); carr[0] = char.ToUpper(carr[0]);
    if (NewStr == null) { NewStr = new String(carr); }
    else { NewStr += " " + new String(carr); }
  }
  return NewStr;
 }
 return OldStr;
}
```

Now add a new class TestExtensionMethods.cs and write the following:

```
class TestExtensionMethods {
  static void Main() {
    Program p = new Program(); p.Test5(100);
    MyStruct obj = new MyStruct(); obj.Show();
    uint fact = 15; ulong result = fact.Factorial(); Console.WriteLine($"Factorial of {fact} is: {result}");
    string str = "hElLo hOw aRe yoU"; str = str.ToPascal(); Console.WriteLine(str); Console.ReadLine();
  }
}
```

Working with Multiple Projects and Solution

While developing an application sometimes code will be written under more than 1 project also, where collection of all those projects is known as a Solution. Whenever we open a new project by default Visual Studio will create one Solution and under it the project gets added, where a solution is a collection of projects and project is a collection of items or files and each item is a collection of types (Class, Structure, Interface, Enum & Delegate).

A Solution also requires a name, which can be specified by us while creating a new project or else it will take name of the first project that is created under solution, if not specified. In our case solution name is "OOPSProject" because our project name is "OOPSProject". A solution can have projects of different .NET languages as well as can be of different project templates also like Windows App's, Console App's, Class Library etc. but a project cannot contain items of different .NET languages i.e. they must be specific to one language only.

To add a new project under our OOPSProject solution, right click on solution node in Solution Explorer and select add "New Project" which opens the new project window, under it select language as Visual C#, template as Console Application, name the project as "SecondProject" and click Ok which adds the new project under the "OOPSProject" solution.

By default the new project also comes with a class "Program" but under "SecondProject" namespace, now write the following code in its main method and execute:

```
Console.WriteLine("Second Project under the solution.");
Console.ReadLine();
```

To run the above class, first we need to set a property i.e. "StartUp Project", because there are multiple project's under the solution and Visual Studio by default runs first project of the Solution only i.e. "OOPSProject" under the solution. To set the "StartUp Project" property and run classes under "SecondProject" open Solution Explorer, right click on "SecondProject", select "Set as StartUp Project" and then run the project.

Note: if the new project is added with new classes we need to again set "StartUp Object" property under Second Project's project file, because each project has its own property window.

Saving Solution and Projects:

The application what we have created right now is saved physically on hard disk in the same hierarchy as seen under Solution Explorer i.e. first a folder is created representing the solution and under that a separate folder is created representing each project and under that items corresponding to that project gets saved as following:

```
<drive>:\<our_folder>\OOPSProject\OOPSProject => Project1
<drive>:\<our_folder>\OOPSProject\SecondProject => Project2
```

Note: a solution will be having a file called solution file, which gets saved with **.sln** extension and a project also has a file called project file, where a C# Project file gets saved with **.csproj** extension which can contain C# items only.

Compilation of Projects: whenever a project is compiled it generates an output file known as **"Assembly"** that contains "CIL Code" of all the "Types" that are defined in the project.

What is an Assembly?

- It's an output file that is generated after compilation of a project which contains CIL Code in it.
- Assembly file contains the CIL Code of each and every type that is defined under the project.
- An Assembly is a unit of deployment, because when we need to install an Application on client machines what we install is these Assemblies only and all the .NET Libraries are installed on our machines in the form of Assemblies when we install Visual Stuido.
- In .NET Framework the assembly file of a project will be present under the project folder's "bin\debug" folder. In .NET Core, assembly file of a project will be present under "bin\debug\netcoreapp<Version>" folder and here version represents the Core runtime version. From .NET 5, assembly file of a project will be present under "bin\debug\net<Version>" folder and here also version represents the runtime version.
- The name of an assembly file is exactly the same name of the project and can't be changed.
- In .NET Framework the extension of an assembly file can either be a ".exe" or ".dll" which is based on the type of project we open, for example if the project is an "Application Project" then it will generate ".exe" assembly whereas if it is a "Lirary Project" then it will generate ".dll" assembly. From .NET Core every project will generate ".dll" assembly and apart from that "Application Project's" will generate an additional ".exe" assembly also i.e. "Library Projects" will be generating ".dll" only now also where as "Application Project's" will generate both ".exe" and ".dll" also.

.NET Framework:

- Application Projects => Generates only ".exe".
- Library Projects => Generates only ".dll".

.NET Core & .NET 5:

- Application Projects => Generates both ".exe" and ".dll" also.
- Library Projects => Generates only ".dll"

Note: Generally ".dll" assemblies can't run but a ".dll" assembly that are generated by Application Projects can run or execute on Linux and MAC Machines also by using the tool: ".NET Core CLI (Command Line Interface)" as following:

```
dotnet <Assembly_Name>.dll
```

What is a ".exe" assembly?

Ans: In Windows OS ".exe" assemblies are known as in-process components i.e. these assemblies are physically loaded into the memory for execution and run on Windows O.S.

What is a ".dll" assembly?

Ans: In Windows O.S. ".dll" assembly are known as out-process components i.e. these assemblies sit out of the memory providing support to the 1 who is running in the memory. In .NET Framework, ".dll" assemblies can never run on their own i.e. they can only be consumed from other projects, whereas in .NET Core and .NET 5 the ".dll" assemblies that are generated by "Application Projects" can run on Windows, Linux and Mac Machine with the help of .NET Core CLI.

The assembly files of our 2 projects i.e. OOPSProject and SecondProject will be at the following location:

<drive>:\<folder>\OOPSProject\OOPSProject\bin\Debug\net5.0\OOPSProject.dll|.exe => Assembly1
<drive>:\<folder>\OOPSProject\SecondProject\bin\Debug\net5.0\SecondProject.dll|.exe => Assembly2

ildasm: Intermediate Language Dis-Assembler. We use it to dis-assemble an Assembly file and view the contents of it. To check it out, open VS Developer Command Prompt, go to the location where the assembly files of the project are present and use it as following: **ildasm <name of the .dll assembly file>**

Note: in .NET Framework we can dis-assemble both ".exe" and ".dll" assemblies also where as from .NET Core we can dis-assemble only ".dll" assemblies.

E.g.: Open VS Developer Command Prompt, go to the below location and try the following:

```
<drive>:\<our_folder>\OOPSProject\OOPSProject\bin\Debug\net5.0> ildasm OOPSProject.dll
<drive>:\<our_folder>\OOPSProject\SecondProject\bin\Debug\net5.0> ildasm SecondProject.dll
```

Q. Can we consume classes of a project from other classes of same project?

Ans: Yes, we can consume them directly because all those classes were under the same project and will be considered as a family.

Q. Can we consume the classes of a project from other projects?

Ans: Yes we can consume them, but not directly, as they are under different projects. To consume them first we need to add reference of the assembly in which the class is present to the project who wants to consume it.

Q. How to add the reference of an assembly to a project?

Ans: To add reference of an assembly to a project open solution explorer, right click on the project to whom reference has to be added, select "Add => Project Reference" option, which opens a window "Reference Manager" and in that window select "Browse" option in LHS, then click on "Browse" button below, select the assembly we want to consume from its physical location and click ok. Now we can consume types of that assembly by prefixing with their namespace or importing the namespace.

Note: In .NET Framework we can add reference to ".exe" or ".dll" assemblies also and consume them in other projects, whereas from .NET Core onwards we can't add reference to ".exe" assemblies i.e. we can add reference only to ".dll" assemblies.

To test this go to "OOPSProject" Solution, right click on the "SecondProject" we have newly added, select add reference and add the reference of "OOPSProject.dll" assembly from its physical location (**<drive>:\<our_folder>\OOPSProject\OOPSProject\bin\Debug\net5.0>**). Now add a new class under the "SecondProject" naming it as "Class1.cs" and write the below code in it:

```
using OOPSProject;
class Class1 {
  static void Main() {
    Rectangle r = new Rectangle(12.34, 56.78); Console.WriteLine(r.GetArea());
    Circle c = new Circle(34.56); Console.WriteLine(c.GetArea());
    Cone cc = new Cone(12.67, 34.89); Console.WriteLine(cc.GetArea());
    Triangle t = new Triangle(45.23, 98.42); Console.WriteLine(t.GetArea());
    Console.ReadLine();
  }
}
```

Assemblies and Namespaces:

An assembly is an output file which gets generated after compilation of a project and it is physical. The name of an assembly file will be same as project name and can't be changed at all.

Project: Source Code **Assembly:** Compiled Code (IL Code)

A namespace is a logic container of types which are used for grouping types. By default every project has a namespace and its name is same as the project name, but we can change namespace names as per our requirements and more over a project can contain multiple namespaces in it also.

For Example: TestProject (Console Applicaion) when compiled generates an assembly with the names as TestProject.exe and TestProject.dll, under it, namespaces can be as following:

```
namespace NSP1 {
 Class1
 Class2
}
namespace NSP2 {
 Class3
 Class4
}
```

Whenever we want to consume a type which are defined under 1 project from other projects we need to follow the below process:

Step 1: add reference of the assembly corresponding to the project we want to consume.
Step 2: import the namespace under which the class is present.
Step 3: now either create the instance of the class or inherit from the class and consume it.

Access Specifier's

These are a special kind of modifiers using which we can define the scope of a type and its members i.e. who can access them and who cannot. C# supports 5 access specifiers in it, those are:

1. Private 2. Internal 3. Protected 4. Protected Internal 5. Public 6. Private Protected (C# 7.3)

Note: members that are defined in a type with any scope or specifier are always accessible with in the type, restrictions comes into picture only when we try to access them outside of the type.

Private: members declared as private under a class or structure can be accessed only with-in the type in which they are defined and more over their default scope is private only. Interfaces can't contain any private members and their default scope is public. Types can't be declared as private, so this applies only to members.

Protected: members declared as protected under a class can be accessed only from their child class i.e. non-child classes can't consume them. Types can't be declared as protected, so this applies only to members.

Internal: members and types that are declared as internal can be consumed only with in the project, both from a child or non-child. The default scope for a type in C# is internal only.
Protected Internal: members declared as protected internal will have dual scope i.e. within the project they behave as internal providing access to whole project and out-side the project they will change to protected and provide access to their child classes. Types can't be declared as protected internal, so this applies only to members.

<u>Public:</u> a type or member of a type if declared as public is global in scope which can be accessed from anywhere.

<u>Private Protected (Introducted in C# 7.3 Version):</u> members declared as private protected in a class are accessible only from the child classes that are defined in the same project. Types can't be declared as private protected, so this applies only to members.

To test access specifiers open a new CSharp Project of type "Console App.", name the project as "AccessDemo1" and re-name the solution as "MySolution". By default the project comes with a class Program; write the following code in it making it as public.

```
public class Program {                    //Case 1 (Accessing members of a class from same class)
 private void Test1_Private() {
  Console.WriteLine("Private Method");
 }
 internal void Test2_Internal() {
  Console.WriteLine("Internal Method");
 }
 protected void Test3_Protected() {
  Console.WriteLine("Protected Method");
 }
 protected internal void Test4_ProtecedInternal() {
  Console.WriteLine("Protected Internal Method");
 }
 public void Test5_Public() {
  Console.WriteLine("Public Method");
 }
 private protected void Test6_PrivateProtected() {
  Console.WriteLine("Private Protected Method");
 }
 static void Main(string[] args) {
  Program p = new Program();
  p.Test1_Private();              p.Test2_Internal();          p.Test3_Proteced();
  p.Test4_ProtecedInternal();     p.Test5_Public();            p.Test6_PrivateProtected();
 }
}
```

Now add a new class Two.cs under the project and write the following:

```
class Two : Program {       //Case 2 (Accessing members of a class from a child class in the same project)
 static void Main() {
  Two obj = new Two();
  obj.Test2_Internal();     obj.Test3_Protected();           obj.Test4_ProtectedInternal();
  obj.Test5_Public();       obj.Test6_PrivateProtected();    Console.ReadLine();
 }
}
```

Now add another new class Three.cs in the project and write the following:

```
class Three {               //Case 3 (Accessing members of a class from a non-child class in the same project)
 static void Main() {
  Program p = new Program();
  p.Test2_Internal();      p.Test4_ProtectedInternal();       p.Test5_Public();          Console.ReadLine();
 }
}
```

Now Add a new "Console App" project under "MySolution", name it as "AccessDemo2", rename the default file Program.cs as Four.cs so that class name also changes to Four, add a reference to AccessDemo1 assembly from its physical location to the new project and write the following code in the class Four:

```
class Four : AccessDemo1.Program {  //Case 4 (Accessing members of a class from a child class in another project)
 static void Main() {
  Four obj = new Four();
  obj.Test3_Protected();   obj.Test4_ProtectedInternal();      obj.Test5_Public();         Console.ReadLine();
 }
}
```

Now add a new class under AccessDemo2 project, naming it as Five.cs and write the following:

```
using AccessDemo1
class Five {                   //Case 5(Accessing members of a class from a non-child class in another project)
 static void Main() {
  Program p = new Program();
  p.Test5_Public(); Console.ReadLine();
 }
}
```

Cases	Private	Internal	Protected	Private Protected	Protected Internal	Public
Case 1: Same Class - Same Project	Yes	Yes	Yes	Yes	Yes	Yes
Case 2: Child Class - Same Project	No	Yes	Yes	Yes	Yes	Yes
Case 3: Non-Child Class - Same Project	No	Yes	No	No	Yes	Yes
Case 4: Child Class - Another Project	No	No	Yes	No	Yes	Yes
Case 5: Non-Child Class - Another Project	No	No	No	No	No	Yes

Language Interoperability

As discussed earlier the code written in 1 .NET Language can be consumed from any other .NET Languages and we call this as Language Interoperability.

- VB is not a case-sensitive language.
- VB does not have any curly braces; the end of a block is represented with a matching End Stmt.
- VB does not have any semi colon terminators each statement must be in a new line.
- The extension of source files will be ".vb".

To test this, add a new "Console App" project under "MySolution" choosing the language as Visual Basic and name the project as "AccessDemo3". By default the project comes with a file "Module1.vb", so open solution explorer, delete that file under project and add a new class naming it as "TestCS.vb". Now add the reference of "AccessDemo1" assembly to the current project, choosing it from its physical location and write the following code under the Class TestCS:

```
Imports AccessDemo1
Public Class TestCS : Inherits Program
```

```
 Shared Sub Main()
   Dim obj As New TestCS()            'Creating a instance of a class in VB.Net Language
   obj.Test3_Protected()
   obj.Test4_ProtectedInternal()
   obj.Test5_Public()
   Console.ReadLine()
 End Sub
End Class
```

Note: to run the class set Startup Project and execute.

Consuming VB.Net Code in CSharp: Now to test consuming VB.Net code in CSharp, add a new project under “MySolution”, choosing the language as VB, project type as "Class Library" and name the project as “AccessDemo4”. A Class Library is a collection of types that can be consumed but not executed. After compilation the extension of project’s assembly will be “.dll”.

In VB.Net language methods are divided into 2 categories like:

1. Functions (Value returning methods)   2. Sub-Routines (Non-value returning methods)

By default the project comes with a class Class1 within the file Class1.vb, write the below code in it:

```
Public Class Class1
 Public Function SayHello(name As String) As String
   Return "Hello " & name
 End Function
 Public Sub AddNums(x As Integer, y As Integer)
   Console.WriteLine(x + y)
 End Sub
 Public Sub Math(a As Integer, b As Integer,  ByRef c As Integer, ByRef d As Integer)
   c = a + b
   d = a * b
 End Sub
End Class
```

Now to compile the project open solution explorer, right click on “AccessDemo4” project and select “Build” option which compiles and generates an assembly “AccessDemo4.dll”.

Now add a new class under “AccessDemo2” project with the name “TestVB.cs”, add reference of “AccessDemo4.dll” assembly from its physical location and write the following code under the class TestVB:

```
using AccessDemo4;
class TestVB {
 static void Main() {
  Class1 obj = new Class1(); obj.AddNums(100, 50);
  int x = 0, y = 0; obj.Math(100, 25, ref x, ref y); Console.WriteLine(x + "  " + y);
  string str =obj.SayHello("Raju"); Console.WriteLine(str); Console.ReadLine();
  }
}
```

Note: to run the class set both Startup Project and Startup Object properties also.

Q. How to restrict a class not to be accessible for any other class to consume?

Ans: This can be done by declaring all the class constructors as private.

Q. How to restrict a class not to be inherited for any other class?
Ans: This can be done by declaring class as sealed.

Q. How to restrict a class not to be accessible for any other class to consume by creating its instance?
Ans: This can be done by declaring all the class constructors as protected.

Finalizer

Finalizers are used to destruct objects (instances) of classes. A finalizer is also a special method just like a constructor, whereas constructors are called when instance of a class is created and finalizers are called when instance of a class is destroyed. Both of them will have the same name i.e. the name of class in which they are defined, but to differentiate between each other we prefix finalizer with a tilde (~) operator. For Example:

```
class Test {
 Test() {
  //Constructor
 }
 ~Test() {
  //Finalizer
 }
```

Remarks:

- Finalizers cannot be defined in structs. They are only used with classes.
- A finalizer does not take modifiers or have parameters.
- A class can only have one finalizer and cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.

The programmer has no control over when the finalizer is called because this is determined by the garbage collector; garbage collector calls the finalizer in any of the following cases:

1. Called in the end of a programs execution and destroys all instances that are associated with the program.
2. In the middle of a programs execution also the garbage collector checks for instances that are no longer being used by the application. If it considers an instance is eligible for destruction, it calls the finalizer and reclaims the memory used to store the instance.
3. It is possible to force garbage collector by calling GC.Collect() method to check for un-used instances and reclaim the memory used to store those instances.

Note: we can force the garbage collector to do clean up by calling the GC.Collect method, but in most cases this should be avoided because it may create performance issues, i.e. when the garbage collector comes into picture for reclaiming memory of un-used instances, it will suspend the execution of programs.

To test this open a new Console App. Project in .NET Framework naming it as “FinalizersProject”, rename the default class “Program.cs” as “DestDemo1.cs” using Solution Explorer and write the below code over there:

```
class DestDemo1 {
 public DestDemo1() {
  Console.WriteLine("Instance1 is created.");
 }
 ~DestDemo1() {
  Console.WriteLine("Instance1 is destroyed.");
 }
 static void Main()
 {
  DestDemo1 dd1 = new DestDemo1();
  DestDemo1 dd2 = new DestDemo1();
  DestDemo1 dd3 = new DestDemo1();
  //dd1 = null; dd3 = null; GC.Collect(); (Write all the 3 statements in the same line with comments).
  Console.ReadLine();
 }
}
```

Execute the above program by using Ctrl + F5 and watch the output of program, first it will call Constructor for 3 times because 3 instances are created and then waits at ReadLine() statement to execute; now press enter key to finish the execution of ReadLine(), immediately finalizer gets called for 3 times because it is the end of programs execution, so all 3 instances associated with the program are destroyed. This proves that finalizer is called in the end of a programs execution.

Now un-comment the commented code in Main method of above program and re-execute the program again by using Ctrl + F5 to watch the difference in output, in this case 2 instances are destroyed before execution of ReadLine() because we have marked them as un-used by assigning “null” and called Garbage Collector explicitly and the third instance is destroyed in end of programs execution.

Finalizers and Inheritance:

As we are aware that whenever a child class instance is created, child class constructor will call its parents class constructor implicitly, same as this when a child class instance is destroyed it will also call its parent classes finalizer, but the difference is constructor are called in "Top to Bottom" hierarchy and finalizer are called in "Bottom to Top" hierarchy. To test this add a new class DestDemo2.cs and write the following code:

```
class DestDemo2 : DestDemo1 {
 public DestDemo2() {
  Console.WriteLine("Instance2 is created.");
 }
 ~DestDemo2() {
  Console.WriteLine("Instance2 is destroyed.");
 }
 static void Main() {
  DestDemo2 obj = new DestDemo2();
  Console.ReadLine();
 }
}
```

Conclusion about Finalizers: In general, C# does not require as much memory management; it is needed when you develop with a language that does not target a runtime with garbage collection, for example CPP Language. This is because the .NET Framework garbage collector implicitly manages the allocation and release of memory for your instances. However, when your application encapsulates un-managed resources such as files, databases and network connections, you should use finalizers to free those resources.

Properties

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public fields, but they are actually special methods called accessors. Suppose a class is associated with any value and if we want to expose that value outside of the class, access to that value can be given in 4 different ways:

I. By storing the value under a public field, access can be given to that value outside of the class, for Example:

```
public class Circle {
  public double Radius = 12.34;
}
```

Now by creating the instance of above class we can get or set a value to the field as following:

```
class TestCircle {
  static void Main() {
   Circle c = new Circle();
   double Radius = c.Radius;                    //Getting the old value of Radius
   c.Radius = 56.78;                            //Setting a new value for Radius
  }
}
```

Note: in this approach it will provide Read/Write access to the value i.e. anyone can get the old value of the field as well as anyone can set with a new value for the field.

II. By storing the value under a private field also we can provide access to the value outside of the class by defining a property on that field. The advantage in this approach is it can provide access to the value in 3 different ways:
   I. Only get access (Read Only Property)
   II. Only set access (Write Only Property)
   III. Both get and set access (Read/Write Property)

Syntax to define a property:

```
[<modifiers>] <type> Name {
  [ get { -Stmts } ]            //Get Accessor
  [ set { -Stmts } ]            //Set Accessor
}
```

- A property is one or two code blocks, representing a get accessor and/or a set accessor.
- The code block for the get accessor is executed when the property is read and the body of the get accessor resembles that of a method. It must return a value of the property type. The get accessor resembles a value returning method without any parameters.
- The code block for the set accessor is executed when the property is assigned with a new value. The set accessor resembles a non value returning method with parameter, i.e. it uses an implicit parameter called "value", whose type is the same type of the property.
- A property without a set accessor is considered as read-only. A property without a get accessor is considered as write-only. A property that has both accessors is considered as read-write.

Remarks:

- Properties can be marked as public, private, protected, internal, or protected internal. These access modifiers define how users of the class can access the property. The get and set accessors for the same property may have different access modifiers. For example, the get may be public to allow read-only access from outside the class, and the set may be private or protected.
- A property may be declared as a static property by using the static keyword. This makes the property available to callers at any time, even if no instance of the class exists.
- A property may be marked as a virtual property by using the virtual keyword, which enables derived classes to override the property behavior by using the override keyword. A property overriding a virtual property can also be sealed, specifying that for derived classes it is no longer virtual.
- A property can be declared as abstract by using the abstract keyword, which means that there is no implementation in the class, and derived classes must write their own implementation.

To test properties first add a new code file Cities.cs and write the following code:

```
namespace OOPSProject {
 public enum Cities { Bengaluru, Chennai, Delhi, Hyderabad, Kolkata, Mumbai }
}
```

Now add a new class Customer.cs and write the following code:

```
public class Customer {
 int _Custid; bool _Status; string _Name, _State; double _Balance; Cities _City;
 public Customer(int Custid) {
  _Custid = Custid; _Status = false; _Name = "John"; _Balance = 5000.00; _City = 0; _State = "Karnataka";
  Country = "India";
 }
 public int Custid {                      //Read Only Property
  get { return _Custid; }
 }
 public bool Status {                     //Read/Write Property
  get { return _Status; } set { _Status = value; }
 }
 public string Name {                     //Read/Write Property with condition in set
  get { return _Name; }
  set { if(_Status) { _Name = value; } }
 }
 public double Balance {                  //Read/Write Property with condition in get & set
  get { if(_Status) { return _Balance; } return 0; }
  set { if (_Status) { if(value >= 500) { _Balance = value; } } }
 }
 public Cities City {                     //Read/Write Enumerated Property
  get { return _City; }
  set { if (_Status) { _City = value; } }
 }
 public string State {                    //Read/Write property with different scope to each accessor (2.0)
  get { return _State; }
  protected set { if (_Status) { _State = value; } }
 }
 public string Country {                  //Auto-Implemented or Automatic Property (3.0)
```

```
    get; private set;
  }
  public string Continent { get; } = "Asia";       //Auto-Property Initializer (6.0)
}
```

Note: The contextual keyword value is used in the set accessor in ordinary property declarations. It is similar to an input parameter of a method. The word value references the value that client code is attempting to assign to the property.

Enumerated Property:

It is a property that provides with a set of constants to choose from, for example BackgroundColor property of the Console class that provides with a list of constant colors to choose from, under an enum ConsoleColor. *E.g.: Console.BackgroundColor = ConsoleColor.Blue;*

An enum is a distinct type that consists of a set of named constants called the enumerator list. Usually it is best to define an enum directly within a namespace so that all classes in the namespace can access it with equal convenience. However, an enum can also be nested within a class or structure.

Syntax to define an enum:

[<modifiers>] enum <Name> { <list of constant values> }

E.g.: public enum Days { Monday, Tuesday, Wednesday, Thursday, Friday }

By default, the first value is represented with an index 0, and the value of each successive enumerator is increased by 1. For example, in the above enumeration, Monday is 0, Tuesday is 1, Wednesday is 2, and so forth.

To define an Enumerated Property adopt the following process:

Step 1: define an enum with the list of constants we want to provide for the property to choose.

E.g.: public enum Days { Monday, Tuesday, Wednesday, Thursday, Friday };

Step 2: declare a field of type enum on which we want to define a property.

E.g.: Days _Day = 0; or Days _Day = Days.Monday; or Days _Day = (Days)0;

Step 3: now define a property on the enum field for providing access to its values.

```
public Days Day {
  get { return _Day; }
  set { _Day = value; }
}
```

Auto-Implemented properties:

In C# 3.0 and later, auto-implemented properties make property-declaration more concise when **no additional logic is required in the property accessors**. E.g.: Country property in our Customer class, but up to CSharp 5.0 it is important to remember that auto-implemented properties must contain both get and set blocks either with the same access modifier or different also whereas from CSharp 6.0 it’s not mandatory because of a new feature called “Auto Property Initializer”, which allows to initialize a property at declaration time. In our Customer class the Country property we have defined can be implemented as following also:

E.g.: public string Country { get; } = "India";

To consume the properties we have defined above add a new class TestCustomer.cs and write the following:

```
class TestCustomer {
  static void Main() {
    Customer obj = new Customer(101);
    Console.WriteLine("Custid: " + obj.Custid);
    //obj.Custid = 102; //Can't be assigned to as property is read only

    Console.WriteLine("Old Name: " + obj.Name);
    obj.Name = "John Smith"; Console.WriteLine("New Name: " + obj.Name);
    if(obj.Status) { Console.WriteLine("Status: Active"); }
    else {Console.WriteLine("Status: In-Active"); }
    Console.WriteLine("Current Balance: " + obj.Balance); //Prints Balance as 0, because status is in-active

    obj.Status = true; //Activating the status again
    if(obj.Status) { Console.WriteLine("Status: Active"); }
    else {Console.WriteLine("Status: In-Active"); }
    Console.WriteLine("Current Balance: " + obj.Balance); //Prints actual Balance, as status is active again

    obj.Balance -= 4600; //Transaction fails
    Console.WriteLine("Balance when transaction failed: " + obj.Balance); //Prints old balance value only
    obj.Balance -= 4500; //Transaction succeeds
    Console.WriteLine("Balance when transaction succeeded " + obj.Balance); //Prints new balance value

    Console.WriteLine("Current City: " + obj.City);
    obj.City = Cities.Hyderabad; Console.WriteLine("Modified City: " + obj.City);

    Console.WriteLine("Current State: " + obj.State);
    //obj.State = "AP"; //Can't be assigned with a value, as current class is not a child class of Customer

    Console.WriteLine("Country: " + obj.Country); //Auto-Implemented Property
    Console.WriteLine("Continent: " + obj.Continent); //Auto-Property Initializer
    Console.ReadLine();
  }
}
```

Object Initializers (Introduced in C# 3.0)

Object initializers let you assign values to any accessible **properties** of an instance at creation time without having to explicitly invoke a parameterized constructor. You can use object initializers to initialize type objects in a declarative manner without explicitly invoking a constructor for the type. Object Initializers will use the default constructor for initializing **properties**. To test these add a new CodeFile naming it as “TestStudent.cs” and write the following code in it:

```
using System;
namespace OOPSProject {
  public class Student {
    int? _Id, _Class; string _Name; float? _Marks, _Fees;
    public int? Id { get { return _Id; } set { _Id = value; } }
    public int? Class { get { return _Class; } set { _Class = value; } }
    public string Name { get { return _Name; } set { _Name = value; } }
    public float? Marks { get { return _Marks; } set { _Marks = value; } }
    public float? Fees { get { return _Fees; } set { _Fees = value; } }
  public override string ToString() {
   return "Id: " + _Id + "\nName: " + _Name + "\nClass: " + _Class + "\nMarks: " + _Marks + "\nFees: " + _Fees;
  }
 }
 class TestStudent {
  static void Main() {
   Student s1 = new Student { Id = 101, Name = "Raju", Class = 10, Marks = 550.00f, Fees = 5000.00f };
   Student s2 = new Student { Id = 102, Name = "Venkat", Class = 10 };
   Student s3 = new Student { Id = 103, Name = "Suresh", Marks = 575.00f };
   Console.WriteLine(s1); Console.WriteLine(s2); Console.WriteLine(s3); Console.ReadLine();
  }
 }
}
```

Indexers

Indexers allow instances of a class or struct to be indexed just like arrays. Indexers resemble properties except that their accessors take parameters. Indexers are syntactic conveniences that enable you to create a class or struct that client applications can access just as an array. Defining an indexer allows you to create classes that act like "virtual arrays." Instances of that class or structure can be accessed using the [] array access operator. Defining an indexer in C# is similar to defining operator [] in C++, but is considerably more flexible. For classes or structure that encapsulate array or collection - like functionality, defining an indexer allows the users of that class or structure to use the array syntax to access the class or structure. An indexer doesn't have a specific name like a property it is defined by using “this” keyword.

Syntax to define Indexer:

```
[<modifiers>] <type> this[<Parameter List>] {
 [ get { -Stmts } ]  //Get Accessor
 [ set { -Stmts } ]  //Set Accessor
}
```

Indexers Overview:

- “this” keyword is used to define the indexers.
- The out and ref keyword are not allowed on parameters.
- A get accessor returns a value. A set accessor assigns a value.
- The value keyword is only used to define the value being assigned by the set indexer.
- Indexers do not have to be indexed by integer value; it is up to you how to define the look-up mechanism.
- Indexers can be overloaded.
- Indexers can't be defined as static.
- Indexers can have more than one formal parameter, for example, accessing a two-dimensional array.

To test indexers add a CodeFile under the project naming it as TestEmployee.cs and write the below code in it:

```
using System;
namespace OOPSProject {
 public class Employee {
  int _Eno; bool _Status; double _Salary; string _Name, _Job;
  public Employee(int Eno) {
   _Eno = Eno; _Status = true; _Name = "Scott"; _Job = "Manager"; _Salary = 25000.00;
  }
  public object this[int Index] {
    get {
      if (Index == 1) { return _Eno; }
      else if (Index == 2) { return _Name; }
      else if (Index == 3) { return _Job; }
      else if (Index == 4) { return _Salary; }
      else if (Index == 5) { return _Status; }
      else { return null; }
    }
    set {
      if (Index == 2) { _Name = (string)value; }
      else if (Index == 3) { _Job = (string)value; }
      else if (Index == 4) { _Salary = (double)value; }
      else if (Index == 5) { _Status = (bool)value; }
    }
  }
  public object this[char Index] {
    get {
      if (Char.ToUpper(Index) == 'A') { return _Eno; }
      else if (Char.ToUpper(Index) == 'B') { return _Name; }
      else if (Char.ToUpper(Index) == 'C') { return _Job; }
      else if (Char.ToUpper(Index) == 'D') { return _Salary; }
      else if (Char.ToUpper(Index) == 'E') { return _Status; }
      else { return null; }
    }
    set {
      if (Char.ToLower(Index) == 'b') { _Name = (string)value; }
      else if (Char.ToLower(Index) == 'c') { _Job = (string)value; }
      else if (Char.ToLower(Index) == 'd') { _Salary = (double)value; }
      else if (Char.ToLower(Index) == 'e') { _Status = (bool)value; }
    }
  }
}
class TestEmployee {
  static void Main() {
    Employee Emp = new Employee(1001);
    Console.WriteLine("Employee No: " + Emp[1]);     Console.WriteLine("Employee Name: " + Emp[2]);
    Console.WriteLine("Employee Job: " + Emp[3]);    Console.WriteLine("Employee Salary: " + Emp[4]);
    Console.WriteLine("Employee Status: " + Emp[5] + "\n");

    Emp[3] = "Sr. Manager"; Emp['D'] = 30000.00;
    Console.WriteLine("Employee No: " + Emp['A']);   Console.WriteLine("Employee Name: " + Emp['b']);
    Console.WriteLine("Employee Job: " + Emp['C']);  Console.WriteLine("Employee Salary: " + Emp['d']);
    Console.WriteLine("Employee Status: " + Emp['E']);
    Console.ReadLine();
  }
 }
}
```

Deconstructor

These are newly introduced in C# 7.0 which can also be used to provide access to the values or expose the values associated with a class to the outside environment, apart from public fields, properties and indexers. Deconstructor is a special method with the name “Deconstruct” that is defined under the class to expose (Read Only) the attributes of a class and this will be defined with a code that is reverse to a constructor.

To understand deconstructors, add a code file in our project naming it as “TestTeacher.cs” and write the below code in it:

```
using System;
namespace OOPSProject {
 public class Teacher {
  int Id;
  string Name, Subject, Designation;
  double Salary;
  public Teacher(int Id, string Name, string Subject, string Designation, Double Salary) {
   this.Id = Id;
   this.Name = Name;
   this.Subject = Subject;
   this.Designation = Designation;
   this.Salary = Salary;
  }
  public void Deconstruct(out int Id, out string Name, out string Subject, out string Designation, out double Salary)
  {
   Id = this.Id;
   Name = this.Name;
   Subject = this.Subject;
   Designation = this.Designation;
   Salary = this.Salary;
  }
 }

 class TestTeacher {
  static void Main() {
   Teacher obj = new Teacher(1005, "Suresh", "English", "Lecturer", 25000.00);
   (int Id, string Name, string Subject, string Designation, double Salary) = obj;
   Console.WriteLine("Teacher Id: " +Id);
   Console.WriteLine("Teacher Name: " + Name);
   Console.WriteLine("Teacher Subject: " + Subject);
   Console.WriteLine("Teacher Designation: " + Designation);
   Console.WriteLine("Teacher Salary: " + Salary);
   Console.ReadLine();
```

```
    }
  }
}
```

In the above case “Deconstruct” (name cannot be changed) is a special method which will expose the attributes of teacher class. We can capture the values exposed by “Deconstructors” by using tuples, through the instance of class we have created.

E.g: (int Id, string Name, string Subject, string Designation, double Salary) = obj;

The above statement can be implemented as following also:

E.g: (var Id, var Name, var Subject, var Designation, var Salary) = obj;

The above statement can be implemented as following also:

E.g: var (Id, Name, Subject, Designation, Salary) = obj;

Note: Deconstructors will provide read-only access to the attributes of a class.

We can also overload Deconstructors to access particular values from the list of attributes and to test that add the following deconstructor in the Teacher class.

```
public void Deconstruct(out int Id, out string Name, out double Salary) {
  Id = this.Id; Name = this.Name; Salary = this.Salary;
}
```

Now we can capture only those 3 values and to test that write the following code in the “Main()” method of “TestTeacher” class just above the “ReadLine()” method.

```
var (Id1, Name1, Salary1) = obj;
Console.WriteLine("Teacher Id: " + Id1);
Console.WriteLine("Teacher Name: " + Name1);
Console.WriteLine("Teacher Salary: " + Salary1);
```

Without OverLoading the Deconstructors also we can access required attribute values by just putting "_" at the place whose values we don't want to access, and to test this Add the following code in the “Main()” method of “TestTeacher” class just above the “ReadLine()” method.

```
var (Id2, Name2, Subject2, _, _) = obj;
Console.WriteLine("Teacher Id: " + Id2);
Console.WriteLine("Teacher Name: " + Name2);
Console.WriteLine("Teacher Subject: " + Subject2);

var (Id3, _, _, Designation3, Salary3) = obj;
Console.WriteLine("Teacher Id: " + Id3);
Console.WriteLine("Teacher Designation: " + Designation3);
Console.WriteLine("Teacher Salary: " + Salary3);

var (Id4, _, Subject4, _, Salary4) = obj;
Console.WriteLine("Teacher Id: " + Id4);
Console.WriteLine("Teacher Subject: " + Subject4);
Console.WriteLine("Teacher Salary: " + Salary4);
```

Exceptions and Exception Handling

In the development of an application we will be coming across 2 different types of errors, like:

- Compile time errors.
- Runtime errors.

Errors which occur in a program due to syntactical mistakes at the time of program compilation are known as compile time errors and these are not considered to be dangerous.

Errors which occur in a program while the execution of a program is taking place are known as runtime errors, which can occur due to various reasons like wrong implementation of logic, wrong input supplied to the program, missing of required resources etc. Runtime errors are dangerous because when they occur under the program, the program terminates abnormally at the same line where the error got occurred without executing the next lines of code. To test this, add a new class naming it as ExceptionDemo.cs and write the following code:

```
class ExceptionDemo {
 static void Main() {
  Console.Write("Enter 1st number: " ); int x = int.Parse(Console.ReadLine());
  Console.Write("Enter 2nd number: " ); int y = int.Parse(Console.ReadLine());
  int z = x / y; Console.WriteLine("The result of division is: " + z); Console.WriteLine("End of the Program.");
 }
}
```

Execute the above program by using Ctrl + F5, and here there are chances of getting few runtime errors under the program, to check them enter the value for y as ‘0’ or enter character input for x or y values, and in both cases when an error got occurred program gets terminated abnormal on the same line where error got occurred.

Exception: In C#, errors in the program at run time are caused through the program by using a mechanism called Exceptions. Exceptions are classes derived from class Exception of System namespace. Exceptions can be thrown by the .NET Framework CLR (Common Language Runtime) when basic operations fail or by code in a program. Throwing an exception involves creating an instance of an Exception-derived class, and then throwing that instance by using the throw keyword. There are so many Exception classes under the Framework Class Library where each class is defined representing a different type of error that occurs under the program, for example: FormatException, NullReferenceException, IndexOutOfRangeException, ArithmeticException etc.

Exceptions are basically 2 types like SystemExceptions and ApplicationExceptions. System Exceptions are pre-defined exceptions that are fatal errors which occur on some pre-defined error conditions like DivideByZero, FormatException, NullReferenceException etc. ApplicationExceptions are non-fatal errors i.e. these are error that are caused by the programs explicitly. Whatever the exception it is every class is a sub class of class Exception only and the hierarchy of these exception classes will be as following:

- ➢ Exception
    - ➢ SystemException
        - ➢ FormatException
        - ➢ NullReferenceException
        - ➢ IndexOutOfRangeException
        - ➢ ArithmeticException
            - ➢ DivideByZeroException
            - ➢ OverflowException
    - ➢ ApplicationException

Exception Handling: It is a process of stopping the abnormal termination of a program whenever a runtime error occurs under the program; if exceptions are handled under the program we will be having the following benefits:

1. As abnormal termination is stopped, statements that are not related with the error can be still executed.
2. We can also take any corrective actions which can resolve the problems that may occur due to the errors.
3. We can display user friendly error messages to end users in place of pre-defined error messages.

How to handle an Exception: to handle an exception we need to enclose the code of the program under some special blocks known as try and catch blocks which should be used as following:

```
try {
  -Statement’s where there is a chance of getting runtime errors.
  -Statement’s which should not execute when the error occurs.
}
catch(<Exception Class Name> [<Variable>]) {
  -Statement’s which should execute only when the error occurs.
}
[---<multiple catch blocks if required>---]
```

To test handling exceptions add a new class TryCatchDemo.cs and write the following code:

```
class TryCatchDemo {
 static void Main() {
  try {
   Console.Write("Enter 1st number: " ); int x = int.Parse(Console.ReadLine();
   Console.Write("Enter 2nd number: " ); int y = int.Parse(Console.ReadLine());
   int z = x / y; Console.WriteLine(“The result of division is: “ + z);
  }
  catch(DivideByZeroException) { Console.WriteLine("Divisor value should not be zero."); }
  catch(FormatException) { Console.WriteLine("Input values must be numeric."); }
  catch(Exception ex) { Console.WriteLine(ex.Message); }
  Console.WriteLine("End of the program.");
 }
}
```

If we enclose the code under try and catch blocks the execution of program will take place as following:

- If all the statements under try block are successfully executed (i.e. no error in the program), from the last statement of try the control directly jumps to the first statement which is present after all the catch blocks.
- If any statement under try causes an error from that line, without executing any other lines of code in try, control directly jumps to catch blocks searching for a catch block to handle the error:
  - If a catch block is available that can handle the exception then exceptions are caught by that catch block, executes the code inside of that catch block and from there again jumps to the first statement which is present after all the catch blocks.
  - If a catch block is not available to handle that exception which got occurred, abnormal termination takes place again on that line.

Note: Message is a property under the Exception class which gets the error message associated with the exception that got occurred under the program, this property was defined as virtual under the class Exception and overridden under all the child classes of class Exception as per their requirement, that is the reason why when we call “ex.Message” under the last catch block, even if “ex” is the reference of parent class, it will get the error message that is associated with the child exception class but not of itself because we have already learnt in overriding that “parent’s reference which is created by using child classes instance will call child classes overridden members” i.e. nothing but dynamic polymorphism.

Finally Block: this is another block of code that can be paired with try along with catch or without catch also and the specialty of this block is code written under this block gets executed at **any cost** i.e. when an exception got occurred under the program or an exception did not occur under the program. All statements under try gets executed only when there is no exception under the program and statements under catch block will be executed only when there is exception under the program whereas code under finally block gets executed in both the cases. To test finally block add a new class FinallyDemo.cs and write the following code:

```
class FinallyDemo {
  static void Main() {
    try {
      Console.Write("Enter 1st number: " ); int x = int.Parse(Console.ReadLine());
      Console.Write("Enter 2nd number: " ); int y = int.Parse(Console.ReadLine());
      if (y == 1) { return; } int z = x / y; Console.WriteLine(“The result of division is: “ + z);
    }
    catch(Exception ex) { Console.WriteLine(ex.Message); }
    finally { Console.WriteLine("Finally block executed."); }
    Console.WriteLine("End of the program.");
  } //End of the method
}
```

Execute the above program for 2 times, first time by giving input which doesn’t cause any error and second time by giving the input which causes an error and check the output where in both the cases finally block is executed.

In both the cases not only finally block along with it “End of the program.” statement also gets executed, now test the program for the third time by giving the divisor value i.e. value to y as 1, so that the if condition in the try block gets satisfied and return statement gets executed. As we are aware that return is a jump statement which jumps out of the method in execution, but in this case it will jump out only after executing the finally block of the method because once the control enters into try we cannot stop the execution of finally block.

Note: try, catch and finally blocks can be used in 3 different combinations like:

I. **try and catch:** in this case exceptions that occur in the program are caught by the catch block so abnormal termination will not take place.
II. **try, catch and finally:** in this case behavior will be same as above but along with it finally block keeps executing in any situation.
III. **try and finally:** in this case exceptions that occur in the program are not caught because there is no catch block so abnormal termination will take place but still the code under finally block gets executed.

Application Exceptions: these are non-fatal application errors i.e. these are errors that are caused by the programs explicitly. Application exceptions are generally raised by programmers under their programs basing on their own error conditions, for example in a division program we don’t want the divisor value to be an odd number. If a programmer wants to raise an exception explicitly under his program he needs to do 2 things under the program.

1. Create the instance of any exception class.
2. Throw that instance by using throw statement. E.g.: throw <instance of exception class>

While creating an Exception class instance to throw explicitly we are provided with different options in choosing which exception class instance has to be created to throw, like:

1. We can create instance of a pre-defined class i.e. ApplicationException by passing the error message that has to be displayed when the error got occurred as a parameter to the class constructor and then throw that instance.

   E.g. ApplicationException ex = new ApplicationException (“<error message>”);
      throw ex;

   or

      throw new ApplicationException (“<error message>”);
2. We can also define our own exception class, create instance of that class and throw it when required. If we want to define a new exception class we need to follow the below process:
   - I. Define a new class inheriting from any pre-defined Exception class (but ApplicationException is preferred choice as we are dealing with application exceptions) so that the new class also is an exception.
   - II. Override the Message property inherited from parent by providing the required error message.

To test this first add a new class under the project naming it DivideByOddNoException.cs and write the following:

```
public class DivideByOddNoException : ApplicationException {
  public override string Message {
    get { return "Attempted to divide by odd number."; }
  }
}
```

Now add a new class ThrowDemo.cs and write the following code:

```
class ThrowDemo {
  static void Main() {
    Console.Write("Enter 1st number: " ); int x = int.Parse(Console.ReadLine());
    Console.Write("Enter 2nd number: " ); int y = int.Parse(Console.ReadLine());
    if(y % 2 > 0) {
      throw new ApplicationException(“Divisor value should not be an odd number.”);
      //throw new DivideByOddNoException();
    }
    int z = x / y; Console.WriteLine(“The result of division is: “ + z);
    Console.WriteLine("End of the program.");
  }
}
```

Test the above program for the first time by giving the divisor value as an odd number and now ApplicationException will raise and displays the error message “Divisor value should not be an odd number.”. Now comment the first throw statement and uncomment the second throw statement so that when the divisor value is an odd number DivideByOddNoException will raise and displays the error message “Attempted to divide by odd number.”.

Delegates

Delegate is a type which holds the method(s) reference in an object. It is also referred to as a type safe function pointer. Delegates are roughly similar to function pointers in C++; however, delegates are type-safe and secure. A delegate instance can encapsulate a static or a non-static method also and call that method for execution. Effective use of a delegate improves the performance of applications.

Methods can be called in 2 different ways in C#, those are:

1. Using instance of a class if it is non-static and name of the class if it is static.
2. Using a delegate (either static or non-static).

To call a method by using delegate we need to adopt the following process:

1. Define a delegate.
2. Instantiate the delegate.
3. Call the delegate by passing required parameter values.

-

<u>Defining a Delegate:</u>

[<modifiers>] delegate void|<type> Name(<Parameter List>)

<u>Note:</u> while defining a delegate you should follow the same signature of the method i.e. parameters of delegate should be same as the parameters of method and return types of delegate should be same as the return types of method we want to call by using the delegate.

```
public void AddNums(int x, int y) {
  Console.WriteLine(x + y);
}
public delegate void AddDel(int x, int y);

public static string SayHello(string name) {
  return "Hello " + name;
}
public delegate string SayDel(string name);
```

<u>Instantiate the delegate:</u> In this process we create the instance of delegate and bind the method with delegate.

```
AddDel ad = new AddDel(AddNums);        or        AddDel ad = AddNums;
SayDel sd = new SayDel(SayHello);       or        SayDel sd = SayHello;
```

<u>Calling the delegate:</u> Call the delegate by passing required parameter values, so that the method which is bound with delegate gets executed.

```
ad(100, 50);                                      string str = sd("Raju");
```

<u>Where to define a delegate:</u> Delegates can be defined either with in a class or with in a namespace also just like we define other types.

Add a code file under the project naming it as Delegates.cs and write the following code:

```
namespace OOPSProject {
  public delegate void MathDelegate(int x, int y);
  public delegate string WishDelegate(string name);
  public delegate void CalculatorDelegate(int a, int b, int c);
}
```

Add a new class DelDemo1.cs under the project and write the following code:

```
class DelDemo1 {
  public void AddNums(int x, int y, int z) {
    Console.WriteLine(x + y + z);
  }
  public static string SayHello(string name) {
    return "Hello " + name + ", good morning.";
  }
  static void Main() {
    DelDemo1 obj = new DelDemo1();
    CalculatorDelegate cd = obj.AddNums; (or) CalculatorDelegate cd = new CalculatorDelegate(obj.AddNums);
    cd(100, 50, 25); cd(123, 456, 789); cd(396, 224, 156);

    WishDelegate wd = DelDemo1.SayHello; (or) WishDelegate wd = WishDelegate(DelDemo1.SayHello);
    Console.WriteLine(wd("Raju")); Console.WriteLine(wd("Naresh")); Console.WriteLine(wd("Praveen"));
    Console.ReadLine();
  }
}
```

<u>Multicast Delegate:</u> It is a delegate which holds the reference of more than one method. Multicast delegates must contain only methods that return void. If we want to call multiple methods using a single delegate all the methods should have the same Parameter types. To test this, add a new class DelDemo2.cs under the project and write the following code:

```
class DelDemo2 {
  public void Add(int x, int y) {
    Console.WriteLine("Add: " + (x + y));
  }
  public void Sub(int x, int y) {
    Console.WriteLine("Sub: " + (x - y));
  }
  public void Mul(int x, int y) {
    Console.WriteLine("Mul: " + (x * y));
  }
  public void Div(int x, int y) {
    Console.WriteLine("Div: " + (x / y));
  }
  static void Main() {
    DelDemo2 obj = new DelDemo2();
    MathDelegate md = obj.Add; md += obj.Sub; md += obj.Mul; md += obj.Div;
    md(100, 50); Console.WriteLine(); md(575, 25); Console.WriteLine(); md -= obj.Mul; md(678, 28);
    Console.ReadLine();
  }
}
```

Anonymous Methods (Introduced in C# 2.0):

In versions of C# before 2.0, the only way to instantiate a delegate was to use named methods. C# 2.0 introduced anonymous methods which provide a technique to pass a code block as a delegate parameter. Anonymous methods are basically methods without a name. An anonymous method is inline unnamed method in the code. It is created using the delegate keyword and doesn't require modifiers, name and return type. Hence we can say an anonymous method has only body without name, return type and optional parameters. An anonymous method behaves like a regular method and allows us to write inline code in place of explicitly named methods. To test this, add a new class DelDemo3.cs and write the following code:

```
class DelDemo3 {
 static void Main() {
  WishDelegate wd = delegate(string name) {
   return "Hello Mr./Ms./Mrs. " + name + " have a nice day.";
  };
  CalculatorDelegate cd = delegate(int a, int b, int c) {
   Console.WriteLine(a * b * c);
  };
  Console.WriteLine(wd("Raju")); Console.WriteLine(wd("Naresh")); Console.WriteLine(wd("Praveen"));
  cd(10, 20, 30); cd(40, 50, 60); cd(70, 80, 90);
  Console.ReadLine();
 }
}
```

Lambda Expression (Introduced in CSharp 3.0):

While Anonymous Methods were a new feature in 2.0, Lambda Expressions are simply an improvement to syntax when using Anonymous method. Lambda Operator "=>" was introduced so that there is no longer a need to use the delegate keyword, or provide the type of the parameter. The type can usually be inferred by compiler from usage based on the delegate. To test this, add a new class DelDemo4.cs and write the following code:

```
class DelDemo4 {
 static void Main() {
  WishDelegate wd = name => {
   return "Hello Mr./Ms./Mrs. " + name + " have a nice day.";
  };
  Console.WriteLine(wd("Raju")); Console.WriteLine(wd("Naresh")); Console.WriteLine(wd("Praveen"));

  CalculatorDelegate cd = (a, b, c) => {
   Console.WriteLine(a * b * c);
  };
  cd(10, 20, 30); cd(40, 50, 60); cd(70, 80, 90);
  Console.ReadLine();
 }
}
```

Expression Bodied Members (Introduced in CSharp 6.0 & 7.0):

Expression body definitions let you provide a member's implementation in a very concise, readable form. You can use an expression body definition whenever the logic for any supported member consists of a single expression. An expression body definition has the following general syntax:

member => expression;

To test this, add a new class DelDemo5.cs and write the following code:

```
class DelDemo5 {
  static void Main() {
    WishDelegate sd = name => "Hello Mr./Ms./Mrs. " + name + " have a nice day.";
    CalculatorDelegate cd = (a, b, c) => Console.WriteLine(a * b * c);
    Console.WriteLine(wd("Raju")); Console.WriteLine(wd("Naresh")); Console.WriteLine(wd("Praveen"));
    cd(10, 20, 30); cd(40, 50, 60); cd(70, 80, 90); Console.ReadLine(); Console.ReadLine();
  }
}
```

Why would we need to write a method without a name is convenience i.e. it's a shorthand that allows you to write a method in the same place you are going to use it. Especially useful in places where a method is being used only once and the method definition are short. It saves you the effort of declaring and writing a separate method to the containing class. Benefits are like Reduced typing, i.e. no need to specify the name of the method, its return type, and its access modifier as well as when reading the code you don't need to look elsewhere for the method definition. Anonymous methods should be short; a complex definition makes calling code difficult to read.

Support for expression body definitions was introduced for methods and read-only properties in C# 6.0 and was expanded in C# 7.0. Expression body definitions can be used with type members listed in following table:

Member	**Supported as of...**
Method	C# 6.0
Read-only property	C# 6.0
Property	C# 7.0
Constructor	C# 7.0
Finalizer	C# 7.0
Indexer	C# 7.0

For example, below is a class defined with out using expression bodied members:

```
public class Circle {
  double _Radius;              const float _Pi = 3.14f;
  public Circle(double Radius) {
    _Radius = Radius;
  }
  ~Circle() {
    Console.WriteLine("Instance is destroyed.");
  }
  public float Pi {
    get { return _Pi; }
  }
  public double Radius {
    get { return _Radius; }    set { _Radius = value; }
  }
  public double GetArea() {
    return _Pi * _Radius * _Radius;
  }
  public double GetPerimeter() {
    return 2 * _Pi * _Radius;
  }
}
```

Above class can be defined as following by using expression bodied members:

```
public class Circle {
 double _Radius;
 const float _Pi = 3.14f;
 public Circle(double Radius) => _Radius = Radius;                          //C# 7.0
 ~Circle() => Console.WriteLine("Instance is destroyed.");                  //C# 7.0
 public float Pi => _Pi;                                                    //C# 6.0
 public double Radius                                                       //C# 7.0
 {
   get => _Radius;
   set => _Radius = value;
 }
 public double GetArea() => _Pi * _Radius * _Radius;                        //C# 6.0
 public double GetPerimeter() => 2 * _Pi * _Radius;                         //C# 6.0
}
```

Anonymous Types (Introduced in C# 3.0):

Anonymous type, as the name suggests, is a type that doesn't have any name. C# allows you to create an instance with the new keyword without defining a class. The implicitly typed variable - “var” or “dynamic” is used to hold the reference of anonymous types.

```
var EmpType = new { Id = 1001, Name = "Raju", Job = "Manager", Salary = 25000.00, Status = true };
```

In the above example, “EmpType” is an instance of the anonymous type which is created by using the new keyword and object initializer syntax. It includes 5 properties of different data types. An anonymous type is a temporary type that is inferred based on the data that you include in an object initializer. Properties of anonymous types will be read-only properties so you cannot change their values.

Notice that the compiler applies the appropriate type to each property based on the value assigned. For example Id is of integer type, Name and Job are of string type, Salary is of double type and Status is of boolean type. Internally, the compiler automatically generates the new type for anonymous types. You can check that by calling GetType() method on an anonymous type instance which will return the following value:

```
<>f__AnonymousType0`5[System.Int32,System.String,System.String,System.Double,System.Boolean]
```

Remember that Anonymous Types are derived from the Object class and they are sealed classes, and all the properties are created as read only properties. An anonymous type will always be local to the method where it is defined. Usually, you cannot pass an anonymous type to another method; however, you can pass it to a method that accepts a parameter of dynamic type. Anonymous types can be nested i.e. an anonymous type can have another anonymous type as a property.

Points to Remember:

- Anonymous type can be defined using the new keyword and object initializer syntax.
- The implicitly typed variable - “var” or “dynamic” keyword, is used to hold an anonymous type.
- Anonymous type is a reference type and all the properties are read-only.
- The scope of an anonymous type is local to the method where it is defined.

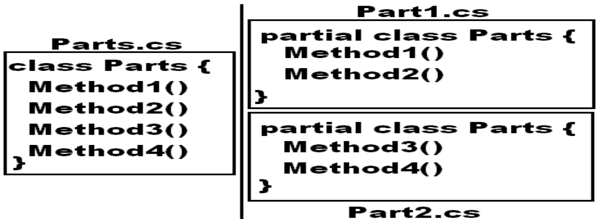
To test anonymous types add a new Code File under the project naming it as “TestAnonymousTypes.cs” and write the following code in it:

```
using System;
namespace OOPSProject {
 class Printer {
  public static void Print(dynamic d) {
   Console.WriteLine("Employee No: " + d.Eno);
   Console.WriteLine("Employee Name: " + d.Name);
   Console.WriteLine("Employee Job: " + d.Job);
   Console.WriteLine("Employee Salary: " + d.Salary);
   Console.WriteLine("Employee Status: " + d.Status);
   Console.WriteLine("Department No: " + d.Dept.Did);
   Console.WriteLine("Department Name: " + d.Dept.Dname);
   Console.WriteLine("Department Location: " + d.Dept.Location);
  }
 }
 class TestAnonymousTypes {
  static void Main() {
    var EmpType = new { Eno = 1005, Name = "Scott", Job = "Manager", Salary = 25000.00, Status = true,
         Dept = new { Did = 30, Dname = "Sales", Location = "Hyderabad" } };
   Printer.Print(EmpType); Console.ReadLine();
  }
 }
}
```

Partial Types/Partial Classes (Introduced in CSharp 2.0):

It is possible to split the definition of a class or struct or an interface over two or more source files. Each source file contains a section of the type definition, and all parts are combined when the application is compiled. There are several situations when splitting a class definition is desirable like:

- When working on large projects, spreading a type over separate files enables multiple programmers to work on it at the same time.
- Visual Studio uses these partial classes for auto generation of source code in the development of Windows Forms Apps, WPF Apps, Web Forms Apps, and Web Services and so on.



Parts.cs

```
class Parts {
  Method1()
  Method2()
  Method3()
  Method4()
}
```

Part1.cs

```
partial class Parts {
  Method1()
  Method2()
}
```

```
partial class Parts {
  Method3()
  Method4()
}
```

Part2.cs

Points to Remember:

- The partial keyword indicates that other parts of the class, struct, or interface can be defined in the namespace.
- All the parts must use the partial keyword.
- All the parts must be available at compile time to form the final type.
- All the parts must have the same accessibility, such as public or internal.
- If any part is declared abstract, then the whole type is considered abstract.

- If any part is declared sealed, then the whole type is considered sealed.
- If any part declares a base type, then the whole type inherits that class.
- Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations.
- Any class, struct, or interface members declared in a partial definition are available to all the other parts.
- The final type is the combination of all the parts at compile time.
- The partial modifier is not available on delegate or enumeration declarations.

To test partial classes add 2 new code files under the project Part1.cs and Part2.cs and write the following code:

```
using System;
namespace OOPSProject {
 partial class Parts {
  public void Method1() {
   Console.WriteLine("Method1 defined in Part1");
  }
  public void Method2() {
   Console.WriteLine("Method2 defined in Part1");
  }
 }
}
using System;
namespace OOPSProject {
 partial class Parts {
  public void Method3() {
   Console.WriteLine("Method3 defined in Part2");
  }
  public void Method4() {
   Console.WriteLine("Method4 defined in Part2");
  }
 }
}
```

Now to test the above partial class, add a new class TestParts.cs under the project and write the following code:

```
class TestParts {
 static void Main() {
  Parts p = new Parts();
  p.Method1(); p.Method2(); p.Method3(); p.Method4(); Console.ReadLine();
 }
}
```

Partial Methods (Introduced in C# 3.0):

A partial class or struct may contain a partial method. One part of the type contains the signature of the method and an optional implementation may be defined in the same part or another part. If the implementation is not supplied, then the method and all calls to the method are removed at compile time.

Partial methods enable the implementer of one part of a type to define a method and the implementer of the other part of the type can decide whether to implement the method or not. If the method is not implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial type can freely use a partial method, even if the implementation is not supplied, no compile-time or run-time errors will result if the method is called but not implemented.

Partial methods are especially useful as a way to customize generated code. They allow for a method name and signature to be reserved, so that generated code can call the method but the developer can decide whether to implement the method or not.

A partial method consists of two parts: the declaration, and the implementation. These may be in separate parts of a partial type, or in the same part. If there is no implementation for the declaration, then the compiler removes both the defining declaration and all calls to the method.

```
partial void TestMethod();          // Declaration in file1.cs
partial void TestMethod() {         // Implementation in file2.cs
  //Method Body
}
```

There are several rules to follow with partial types and partial methods as defined by Microsoft and those rules are listed below:

- Partial methods are indicated by the partial modifier and can be declared within partial classes only.
- Partial methods must be private and must return void.
- Partial methods do not always have an implementation and can also be declared as static.
- Partial methods can have arguments including ref but not out.
- You cannot make a delegate to a partial method.

To test partial methods add 2 new code files Test1.cs and Test2.cs under the project and write the following code:

```
using System;
namespace OOPSProject {
  partial class Test {
    partial void Method1();  //Partial method declaration
    public void Method2() {
      Console.WriteLine("Method 2.");
      Method1(); //Calling Partial Method
    }
  }
}
using System;
namespace OOPSProject {
  partial class Test {
    partial void Method1() { //Partial method implementation
      Console.WriteLine("Method 1.");
    }
    static void Main() {
      Test obj = new Test();
      obj.Method2; Console.ReadLine();
    }
```

```
 }
}
```

Collections

Arrays are simple data structures used to store data items of a specific type. Although commonly used, arrays have limited capabilities. For instance, you must specify an array's size, and if at execution time, you wish to modify it, you must do so manually by creating a new array or by using Array class's Resize method, which creates a new array and copies the existing elements into the new array.

Collections are a set of pre-packaged data structures that offer greater capabilities than traditional arrays. They are reusable, reliable, powerful and efficient and have been carefully designed and tested to ensure quality and performance. Collections are similar to arrays but provide additional functionalities, such as dynamic resizing - they automatically increase their size at execution time to accommodate additional elements, inserting of new elements and removing of existing elements.

Initially .NET introduced so many collection classes under the namespace System.Collections like **Stack, Queue, LinkedList, SortedList, ArrayList, Hashtable etc** and you can work out with these classes in your application where you need the appropriate behaviour.

To work with Collection classes, create a new project of type "Console App" naming it as "CollectionsProject", now under the first class Program.cs write the following code to use the Stack class which works on the principle First In Last Out (FILO) or Last In First Out (LIFO):

```
using System.Collections;
class Program {
 static void Main(string[] args) {
  Stack s = new Stack(); s.Push(10); s.Push("Hello");
  s.Push(DateTime.Now.ToShortDateString()); s.Push(true); s.Push(67.8); s.Push('A');
  foreach (object obj in s) { Console.Write(obj + "  "); } Console.WriteLine();
  Console.WriteLine(s.Pop());
  foreach (object obj in s) { Console.Write(obj + "  "); } Console.WriteLine();
  Console.WriteLine(s.Peek());
  foreach (object obj in s) { Console.Write(obj + "  "); } Console.WriteLine();
  Console.WriteLine(s.Count); s.Clear(); Console.WriteLine(s.Count);
  Console.ReadLine();
 }
}
```

Using Queue class which works on the principle First In First Out (FIFO):

```
using System.Collections;
class Class1 {
 static void Main() {
  Queue q = new Queue(); q.Enqueue(50); q.Enqueue("World"); q.Enqueue(false);
  q.Enqueue('A'); q.Enqueue(DateTime.Now.ToShortDateString()); q.Enqueue(45.73);
  foreach (object obj in q) { Console.Write(obj + "  "); } Console.WriteLine();
  Console.WriteLine(q.Dequeue());
  foreach (object obj in q) { Console.Write(obj + "  "); } Console.ReadLine();
 }
}
```

Auto-Resizing of Collections: The capacity of a collection increases dynamically i.e. when we add new elements to a Collection the size keeps on incrementing automatically. Every collection class has 3 constructors to it and the behaviour of collections will be as following when created instance using different constructor:

i. **Default Constructor**: initializes a new instance of the collection class that is empty and has the default initial capacity as zero which becomes 4 after adding the first element and from then when ever needed the current capacity doubles.

ii. **Collection(int Capacity)**: Initializes a new instance of the collection class that is empty and has the specified initial capacity, here also when requirement comes current capacity doubles.

iii. **Collection(Collection c):** Initializes a new instance of the collection class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied, here also when requirement comes current capacity doubles.

ArrayList: this collection class works same as an array but provides auto resizing, inserting and deleting of items.

```
using System.Collections;
class Class2 {
  static void Main() {
    ArrayList Coll1 = new ArrayList(2); Console.WriteLine("Initial capacity of collection: " + Coll1.Capacity);
    Coll1.Add(10); Console.WriteLine("Capacity after adding 1st item: " + Coll1.Capacity);
    Coll1.Add(true); Coll1.Add(DateTime.Now.ToShortDateString()); Coll1.Add(true);
    Console.WriteLine("Capacity after adding 4th item: " + Coll1.Capacity);
    Coll1.Add(45.7); Console.WriteLine("Capacity after adding 5th item: " + Coll1.Capacity); Coll1.Add("Hello");
    for (int i = 0; i < Coll1.Count; i++) { Console.Write(Coll1[i] + "  "); } Console.WriteLine();
    //Coll1.Remove(true); or //Coll1.RemoveAt(1); or Coll1.RemoveRange(1, 1); //Removes an item from ArrayList
    foreach (object obj in Coll1) { Console.Write(obj + "  "); } Console.WriteLine();
    Coll1.Insert(1, 'Z'); foreach (object obj in Coll1) { Console.Write(obj + "  "); } Console.WriteLine("\n");

    ArrayList Coll2 = new ArrayList(Coll1);
    foreach (object obj in Coll2) { Console.Write(obj + "  "); } Console.WriteLine();
    Console.WriteLine("Initial capacity of new collection: " + Coll2.Capacity);
    Coll2.Add(3.14f); Console.WriteLine("New capacity of new collection: " + Coll2.Capacity);
    Coll2.TrimToSize(); Console.WriteLine("Capacity of new collection after calling Trim: " + Coll2.Capacity);
    Console.ReadLine();
  }
}
```

Hashtable: it is a collection with stores elements in it as “Key/Value Pairs” i.e. Array and ArrayList also has a key to access the values under them which is the index that starts at 0 to number of elements - 1, where as in case of Hashtable these keys can also be defined by us and can be of any data type.

```
using System.Collections;
class Class3 {
  static void Main() {
    Hashtable Emp = new Hashtable();
    Emp.Add("EmpId", 1005); Emp.Add("Name", "John"); Emp.Add("Job", "Manager");
    Emp.Add("HireDate", DateTime.Parse("01/21/2018"));
    Emp.Add("MgrId", 1001); Emp.Add("Salary", 25000.00); Emp.Add("Status", true);
    Emp.Add("Mobile", "98327 12742"); Emp.Add("Email", "john@gmail.com");
    Emp.Add("DeptId", 30); Emp.Add("Dname", "Sales"); Emp.Add("Location", "Mumbai");
    foreach(object key in Emp.Keys) { Console.WriteLine(key + ": " + Emp[key]); } Console.ReadLine();
  }
}
```

Generics: Generics are added in CSharp version 2.0 introducing to the .NET Framework the concept of type parameters, which make it possible to design classes, and methods that defer the specification of one or more types until the class or method is declared and instantiated by client code. For example, by using a generic type parameter “T” you can write a single class that other client code can use without incurring the cost or risk of runtime casts or boxing operations, in simple words Generics allow you to define a class with placeholders for the type of its fields, methods, parameters, etc. Generics replace these placeholders with some specific type at consumption time. To understand these add a class naming it as GenericMethods.cs and write the following code:

```
class GenericMethods {
  public bool AreEqual<T>(T a, T b) {
    if(a.Equals(b)) { return true; }
    else { return false; }
  }
  static void Main() {
    GenericMethods obj = new GenericMethods();
    bool b1 = obj.AreEqual<int>(10, 20); bool b2 = obj.AreEqual<string>("Nit", "Nit");
    bool b3 = obj.AreEqual<double>(12.34, 56.678); bool b4 = obj.AreEqual<char>(‘A’, ‘A’);
    Console.WriteLine(b1); Console.WriteLine(b2); Console.WriteLine(b3); Console.WriteLine(b4); Console.ReadLine();
  }
}
```

Just like we are passing Type parameter to methods it is possible to pass them to a class also, to test this add a code file naming it as TestGenericClass.cs and write the following:

```
using System;
namespace CollectionsProject {
  class Math<T> {
    public void Add(T x, T y) {
      dynamic a = x; dynamic b = y; Console.WriteLine(a + b);
    }
    public void Sub(T x, T y) {
      dynamic a = x; dynamic b = y; Console.WriteLine(a - b);
    }
    public void Mul(T x, T y) {
      dynamic a = x; dynamic b = y; Console.WriteLine(a * b);
    }
    public void Div(T x, T y) {
      dynamic a = x; dynamic b = y; Console.WriteLine(a / b);
    }
  }
  class TestGenericClass {
    static void Main() {
      Math<int> pi = new Math<int>();
      pi.Add(100, 200); pi.Sub(234, 123); pi.Mul(12, 46); pi.Div(900, 45);
      Math<double> pd = new Math<double>();
      pd.Add(145.35, 12.5); pd.Sub(45.6, 23.4); pd.Mul(15.67, 3.4); pd.Div(168.2, 14.5); Console.ReadLine();
    }
  }
}
```

Generic Collections: these are also introduced in C# 2.0 which are extension to collections we have been discussing above, in case of collection classes the elements being added in them are of type object, so we can store any type of values in them which requires boxing and un-boxing, where as in case of generic collections we can store specified type of values which provides type safety. Microsoft has re-implemented all the existing collection classes under a new namespace System.Collections.Generic but the main difference is while creating object of generic collection classes we need to explicitly specify the type of values we want to store under them. In this namespace we have been provided with many classes like classes in System.Collections namespace as following:

Stack<T>, Queue<T>, LinkedList<T>, SortedList<T>, List<T>, Dictionary<TKey, TValue>

Note: <T> referes to the type of values we want to store under them. For example:

```
Stack<int> si = new Stack<int>();              //Stores integer values only
Stack<float> sf = new Stack<float>();          //Stores float values only
Stack<string> ss = new Stack<string>();        //Stores string values only
```

List: this class is same as ArrayList we have discussed under collections above.

```
class Class4 {
  static void Main() {
    List<int> Coll = new List<int>();
    Coll.Add(10); Coll.Add(20); Coll.Add(30); Coll.Add(40); Coll.Add(50); Coll.Add(60); Coll.Add(70); Coll.Add(80);
    for (int i = 0; i < Coll.Count; i++) { Console.Write(Coll[i] + "  "); } Console.WriteLine();
    Coll.Insert(4, 45); foreach (int i in Coll) { Console.Write(i + "  "); } Console.WriteLine();
    Coll.RemoveAt(6); foreach (int i in Coll) { Console.Write(i + "  "); } Console.WriteLine();
    Console.ReadLine();
  }
}
```

Dictionary: this class is same as Hashtable we have discussed under collections but here while creating the object we need to specify the type for keys as well as for values also, as following:

Dictionary<TKey, TValue>

```
class Class5 {
  static void Main() {
    Dictionary<string, object> Emp = new Dictionary<string, object>();
    Emp.Add("EmpId", 1005); Emp.Add("Name", "John"); Emp.Add("Job", "Manager");
    Emp.Add("HireDate", DateTime.Parse("01/21/2018"));
    Emp.Add("MgrId", 1001); Emp.Add("Salary", 25000.00); Emp.Add("Status", true);
    Emp.Add("Mobile", "98327 12742"); Emp.Add("Email", "john@gmail.com");
    Emp.Add("DeptId", 30); Emp.Add("Dname", "Sales"); Emp.Add("Location", "Mumbai");
    foreach (string key in Emp.Keys)
      Console.WriteLine(key + ": " + Emp[key]);
   Console.ReadLine();
  }
}
```

Collection Initializers: this is a new feature added in C# 3.0 which allows to initialize a collection directly at the time of declaration like an array, as following:

```
List<int> Coll1 = new List<int>() { 10, 20, 30, 40, 50 };
List<string> Coll2 = new List<string>() { "Red", "Blue", "Green", "White", "Yellow" };
```

Add a new class in the project naming it as Class6.cs and write the below code in it:

```
class Class6 {
 static void Main() {
  //Copying values > 40 from 1 list to another list and arranging them in descending order
  List<int> coll1 = new List<int>() { 13,56,29,98,24,54,79,39,8,42,22,93,6,73,35,67,48,18,61,32,86,15,21,81,2 };
  List<int> coll2 = new List<int>();
  foreach (int i in coll1) { if (i > 40) { coll2.Add(i); } }    //Retrieving values of list > 40
  coll2.Sort();                                                 //Sorting the new list values in ascending order
  coll2.Reverse();                                              //Reversing new list values to arrange in descending order
  Console.WriteLine(String.Join(" ", coll2));
  Console.ReadLine();
 }
}
```

The above program if used an array, code will be as following (Add a new class Class7.cs and write the below):

```
class Class7 {
 static void Main() {
  //Copying values > 40 from 1 array to another array and arranging them in descending order
  int[] arr = { 13, 56, 29, 98, 24, 54, 79, 39, 8, 42, 22, 93, 6, 73, 35, 67, 48, 18, 61, 32, 86, 15, 21, 81, 2 };
  int Count = 0, Index = 0;
  foreach(int i in arr) { if (i > 40) Count += 1; }
  int[] brr = new int[Count];
  for(int i=0;i<arr.Length;i++) {
   if (arr[i] > 40) {
    brr[Index] = arr[i]; Index += 1;
   }
  }
  Array.Sort(brr); Array.Reverse(brr);
  Console.WriteLine(String.Join(" ", brr)); Console.ReadLine();
 }
}
```

In the above programs we are filtering the values of a List and Array which are greater than 40 and then arranging them in descending order; to do this we have written a substantial amount of code which is the traditional process of performing filters on Arrays and Collections.

In C# 3.0 Microsoft has introduced a new language known as "LINQ" much like SQL (which we use universally with Relational Databases to perform queries). LINQ allows you to write query expressions (similar to SQL Queries) that can retrieve information from a wide variety of Data Sources like Objects, Databases and XML.

Introduction to LINQ: LINQ stands for Language Integrated Query. LINQ is a data querying methodology which provides querying capabilities to .NET languages with syntax similar to an SQL Query. LINQ has a great power of querying on any source of data, where the Data Source could be collections of objects (arrays & collections), Database or XML Source and it is divided into 3 parts:

LINQ to Objects:

- used to perform queries against the in-memory data like an array or collection.

LINQ to XML (XLinq):

- used to perform queries against an XML source.

LINQ to Databases: under this we again have 2 options like,

- LINQ to SQL is used to perform queries against a relation database, but only Microsoft SQL Server.
- LINQ to Entities is used to perform queries against any relation database like SQL Server, Oracle, etc.

Advantages of LINQ:

i. LINQ offers an object-based, language-integrated way to Query over data, no matter where that data came from. So through LINQ we can query Database, XML as well as Collections & Arrays.

ii. Compile time syntax checking.

iii. It allows you to Query Collections, Arrays, and classes etc. in the native language of your application like VB or C#.

LINQ to Objects

This is designed to write queries against the in-memory data like an array or collection and filter or sort the information present under them. Syntax of the query we want to use on objects will be as following:

from <alias> in <array_name | collection_name> [<clauses>] select <alias> | new { <Column List> }

- A Linq-Query starts with from and ends with select.
- In clauses we need to use the alias name just like we use column names in case of SQL.
- Clauses in LINQ are where, group by and orderby.
- To use LINQ in your application first we need to import "System.Linq" namespace.

We can write our previous 2 programs where we have filtered the data of a List or Array and arranged in sorting order by using LINQ and to test that a*dd a new class with the name Class8.cs and write the below code:*

```
class Class8 {
 static void Main() {
  List<int> coll1 = new List<int>() { 13,56,29,98,24,54,79,39,8,42,22,93,6,73,35,67,48,18,61,32,86,15,21,81,2 };
  var coll2 = from i in coll1 where i > 40 orderby i descending select i;
  Console.WriteLine(String.Join(" ", coll2));

  int[] arr = { 13, 56, 29, 98, 24, 54, 79, 39, 8, 42, 22, 93, 6, 73, 35, 67, 48, 18, 61, 32, 86, 15, 21, 81, 2 };
  var brr = from i in arr where i > 40 orderby i descending select i;
  Console.WriteLine(String.Join(" ", brr)); Console.ReadLine();
 }
}
```

Note: the values that are returned by a LINQ query can be captured by using implicitly typed local variables, so in the above case "**coll2**" & "**brr**" are implicitly declared collection/array that stores the values retrieved by the query.

In traditional process of filtering data of an array or collection we have repetition statements that filter arrays focusing on the process of getting the results i.e. iterating through the elements and checking whether they satisfy the desired criteria, whereas LINQ specifies, not the steps necessary to get the results, but rather the conditions that selected elements must satisfy and this is known as declarative programming - as opposed to imperative programming (which we've been using so far) in which we specify the actual steps to perform a task. Procedural & Object Oriented Languages are a subset of imperative.

The queries we have used above specifies that the result should consist of all the int's in the List that are greater than 40, but it does not specify how to obtain the result, C# compiler generates all the necessary code automatically, which is one of the great strengths of LINQ.

LINQ Providers: The syntax of LINQ is built into the language, and LINQ can be used in many different contexts because of the libraries known as providers. A LINQ provider is a set of classes that implement LINQ operations and enable programs to interact with Data Sources to perform tasks such as sorting, grouping and filtering elements.

To test writing queries on a Collection add a new Class naming it as Class9.cs and write the below code in it:

```
class Class9 {
 static void Main() {
  string[] colors = { "Red", "Blue", "Green", "Black", "White", "Brown", "Orange", "Purple", "Yellow", "Aqua" };

  //var coll = from s in colors select s;                           //Gets the list of all colors as is
  //var coll = from s in colors orderby s select s;                 //Gets the list of all colors in ascending order
  //var coll = from s in colors orderby s descending select s;      //Gets the list of all colors in descending order
  //var coll = from s in colors where s.Length == 5 select s;       //Gets the list of colors whose length is 5 characters

   //Getting the list of colors whose name starts with character "B":
  //var coll = from s in colors where s[0] == 'B'  select s;
  //var coll = from s in colors where s.StartsWith("B") select s;
  //var coll = from s in colors where s.Substring(0, 1) == "B" select s;
  //var coll = from s in colors where s.IndexOf("B") == 0 select s;

  //Getting the list of colors whose name ends with character "e":
  //var coll = from s in colors where s.EndsWith("e") select s;
  //var coll = from s in colors where s[s.Length - 1] == 'e' select s;
  //var coll = from s in colors where s.Substring(s.Length - 1) == "e" select s;
  //var coll = from s in colors where s.LastIndexOf("e") == s.Length - 1 select s;

  //Getting the list of colors whose name contains character "a" at 3rd place:
  //var coll = from s in colors where s[2] == 'a' select s;
  //var coll = from s in colors where s.IndexOf("a") == 2 select s;
  //var coll = from s in colors where s.Substring(2, 1) == "a" select s;

  //Getting the list of colors whose name contains character "o" in it:
  //var coll = from s in colors where s.Contains('o') || s.Contains('O') select s;
  //var coll = from s in colors where s.IndexOf('o') >= 0 || s.IndexOf('O') >= 0 select s;
  //var coll = from s in colors where s.ToLower().Contains('o') select s;
  //var coll = from s in colors where s.ToUpper().IndexOf('O') >= 0 select s;

  //Getting the list of colors whose name dosn't contains character "o" in it:
```

```
  //var coll = from s in colors where s.IndexOf('o') == -1 && s.IndexOf('O') == -1 select s;
  //var coll = from s in colors where s.Contains('o') == false && s.Contains('O') == false select s;
  //var coll = from s in colors where s.ToUpper().Contains('O') == false select s;
  var coll = from s in colors where s.ToLower().IndexOf('o') == -1 select s;
  Console.WriteLine(String.Join(" ", coll)); Console.ReadLine();
 }
}
```

Note: The type of values being stored in a generic collection can be of user-defined type values also like a class type or structure type that is defined to represent an entity as following:

List<Customer> Customers = new List<Customer>();

In the above code assume Customer is a user-defined class type that represents an entity Customer, so we can store objects of Customer type under the List where each object can internally represent different attributes of Customer like Id, Name, City, Balance, Status etc.

To test this above add a class in the project with the name Customer.cs and write the below code in it:

```
public class Customer {
 public int Id { get; set; }
 public string Name { get; set; }
 public string City { get; set; }
 public double Balance { get; set; }
 public bool Status { get; set; }
 public override string ToString() => Id + " " + Name + " " + City + " " + Balance + " " + Status;
}
```

Add another class in the project with the name Class10.cs and write the below code in it:

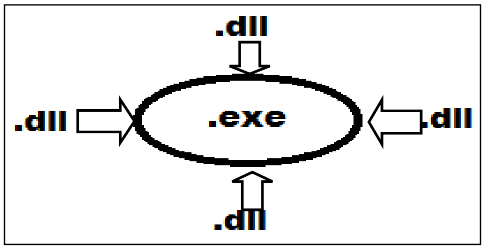
```
class Class10 {
 static void Main() {
  Customer c1 = new Customer { Id = 101, Name = "Scott", City = "Delhi", Balance = 15000.00, Status = true };
  Customer c2 = new Customer { Id = 102, Name = "Dave", City = "Mumbai", Balance = 10000.00, Status = true };
  Customer c3 = new Customer { Id = 103, Name = "Sunitha", City = "Chennai", Balance = 15600.00, Status = false };
  Customer c4 = new Customer { Id = 104, Name = "David", City = "Delhi", Balance = 22000.00, Status = true };
  Customer c5 = new Customer { Id = 105, Name = "John", City = "Kolkata", Balance = 34000.00, Status = true };
  Customer c6 = new Customer { Id = 106, Name = "Jane", City = "Hyderabad", Balance = 19000.00, Status = true };
  Customer c7 = new Customer { Id = 107, Name = "Kavitha", City = "Mumbai", Balance = 16500.00, Status = true };
  Customer c8 = new Customer { Id = 108, Name = "Steve", City = "Bengaluru", Balance = 34600.00, Status = false };
  Customer c9 = new Customer { Id = 109, Name = "Sophia", City = "Chennai", Balance = 6300.00, Status = true };
  Customer c10 = new Customer { Id = 110, Name = "Rehman", City = "Delhi", Balance = 9500.00, Status = true };
  Customer c11 = new Customer { Id = 111, Name = "Raj", City = "Hyderabad", Balance = 9800.00, Status = false };
  Customer c12 = new Customer { Id = 112, Name = "Rupa", City = "Kolkata", Balance = 13200.00, Status = true };
  Customer c13 = new Customer { Id = 113, Name = "Ram", City = "Bengaluru", Balance = 47700.00, Status = true };
  Customer c14 = new Customer { Id = 114, Name = "Joe", City = "Hyderabad", Balance = 26900.00, Status = false };
  Customer c15 = new Customer { Id = 115, Name = "Peter", City = "Delhi", Balance = 17400.00, Status = true };

  List<Customer> Customers = new List<Customer>() {c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15};
  //var Coll = from C in Customers select C;
```

```
    //var Coll = from C in Customers select new { C.Id, C.Name, C.Balance };
    //var Coll = from C in Customers orderby C.Name select C;
    //var Coll = from C in Customers orderby C.Balance descending select C;
    //var Coll = from C in Customers where C.Balance > 15000 select C;
    //var Coll = from C in Customers where C.City == "Hyderabad" select C;
    //var Coll = from C in Customers where C.City == "Delhi" && C.Balance > 10000 select C;
    //var Coll = from C in Customers where C.City == "Mumbai" || C.Balance > 25000 select C;
    //var Coll = from C in Customers group C by C.City into G select new { City = G.Key, Count = G.Count() };
    //var Coll = from C in Customers group C by C.City into G where G.Count() > 2 select new { City = G.Key,
                                            Count = G.Count() };
    //var Coll = from C in Customers group C by C.City into G select new { City = G.Key,
                                   MaxBalance = G.Max(C => C.Balance) };
    //var Coll = from C in Customers group C by C.City into G select new { City = G.Key,
                                   MinBalance = G.Min(C => C.Balance) };
    //var Coll = from C in Customers group C by C.City into G select new { City = G.Key,
                                   TotalBalance = G.Sum(C => C.Balance) };
    var Coll = from C in Customers group C by C.City into G where G.Sum(C => C.Balance) > 30000
                           select new { City = G.Key, TotalBalance = G.Sum(C => C.Balance) };
    foreach (var customer in Coll)
     Console.WriteLine(customer);
    Console.ReadLine();
  }
}
```

Assemblies

An assembly is a unit of deployment, because when we need to install or deploy an Application on client machines what we install is these assemblies only and every application is a blend of .dll's and .exe assemblies combined together to give better efficiency.

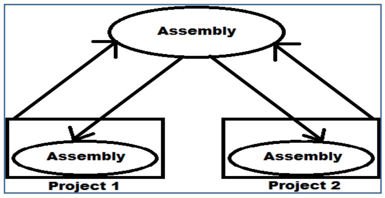


Assemblies are of 2 types:

1. Private Assembly
2. Shared Assembly

Private Assembly:

By default every assembly is private, if reference of these assemblies was added to any project; a copy of the assembly is created and given to that project, so that each project maintains a private copy of that assembly.



Creating an assembly to test it is by default private:

Open a new project of type Class Library and name it as "PAssembly", which will by default come with a class Class1 under the file Class1.cs. Now write the following code under the class:

```
public string SayHello() {
  return "Hello from private assembly.";
}
```

Now compile the project by opening the Solution Explorer, right click on the project and select "Build" which will compile and generate an assembly with the name as PAssembly.dll.

Note: we can find path of assembly in the output window present at bottom of the studio after build.

Testing the assembly we have created:

Create 2 new projects of type Console App., naming them as as "TestPAssembly1" and "TestPAssembly2". Now add the reference of "PAssembly.dll" we have created above to both the new projects, from its physical location and write the following code under Main method of Class1:

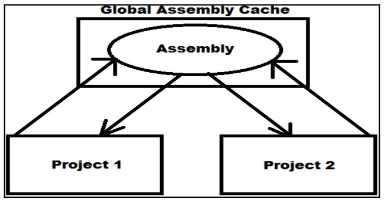
```
PAssembly.Class1 obj = new PAssembly.Class1();
Console.WriteLine(obj.SayHello());
Console.ReadLine();
```

Run both the projects to test them and then go and verify under bin/debug/net5.0 folder of both the new projects where we can find a copy of "PAssembly.dll" as it is a private assembly.

Note: the advantage of a private assembly is faster execution as it was in the local folder of consumer project, whereas the draw back was multiple copies gets created when multiple projects adds the reference to consume it.

Shared Assemblies:

If we intend to use an assembly among several applications, private assemblies are not feasible, in such cases we can install the Assembly into a centralized location known as the "Global Assembly Cache". Each computer where the ".NET Runtime" is installed has this machine-wide code cache. The Global Assembly Cache stores assemblies specifically designated to be shared by several applications on the computer. All .NET Libraries are assemblies and are shared ".dll" assemblies only, so we can find them under GAC. If an assembly is shared multiple copies of the assembly will not be created even if being consumed by multiple projects i.e. only a single copy under GAC serves all the projects.



Note: administrators often protect the Windows directory using an access control list (ACL) to control write and execute access. Because the global assembly cache is installed in the Windows directory, it inherits that directory's ACL. It is recommended that only users with Administrator privileges be allowed to add or delete files from the global assembly cache.

Location of GAC: <OS Drive>:\Windows\Microsoft.NET\assembly\GAC_MSIL

How to make an assembly as Shared?

Ans: to make an assembly as Shared we need to install the assembly into GAC.

How to install an assembly into GAC?

Ans: To manage assemblies in GAC like install, un-install and view we are provided with a tool known as Gacutil.exe (Global Assembly Cache Tool). This tool is automatically installed with VS. To run the tool, use the Visual Studio Command Prompt. These utilities enable you to run the tool easily, without navigating to the installation folder. To use Global Assembly Cache on your computer: On the taskbar, click Start, click All Programs, click Visual Studio, click Visual Studio Tools, and then click Visual Studio Command Prompt and type the following:

```
gacutil -i | -u | -l [<assembly name>]          or          gacutil /i | /u | /l [<assembly name>]
```

What assemblies can be installed into the GAC?

Ans: We can install only Strong Named Assemblies into the GAC.

What is a Strong Named Assembly?

Ans: assemblies deployed in the global assembly cache must have a strong name. When an assembly is added to the global assembly cache, integrity checks are performed on all files that make up the assembly.

Strong Name:

A strong name consists of the assembly's identity - its simple text name, version number, and public key.

1. **Name:** it was the name of an assembly used for identification. Every assembly by default has name.
2. **Version:** software's maintain versions for discriminating changes that has been made from time to time. As an assembly is also a software component it will maintain versions, whenever the assembly is created it has a default version for it i.e. 1.0.0.0, which can be changed when required.
3. **Public Key:** as GAC contains multiple assemblies in it, to identify each assembly it will maintain a key value for the assembly known as public key, which should be generated by us and associate with the assembly to make it Strong Named.

You can ensure that a name is globally unique by signing an assembly with a strong name. In particular, strong names satisfy the following requirements:

- Strong names guarantee name uniqueness by relying on unique key pairs. No one can generate the same assembly name that you can. Strong names protect the version lineage of an assembly.
- A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version the application was built with.
- Strong names provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built.

Generating a Public Key:

To sign an assembly with a strong name, you must have a public key pair. This public cryptographic key pair is used during compilation to create a strong-named assembly. You can create a key pair using the Strong Name tool (Sn.exe) from visual studio command prompt as following:

Syntax: sn -k <file name>

E.g.: sn -k Key.snk

Note: the above statement generates a key value and writes it into the file "Key.snk". Key pair files usually have .snk extension.

Creating a Shared Assembly:

Step 1: generate a public key. Open VS command prompt, go into your personal folder and generate a public key as following:

```
<drive>:\<folder> sn -k Key.snk
```

Step 2: develop a new project and add the key file to it before compilation so that the assembly which is generated will be Strong Named. To do this open a new project of type Class Library, name it as “SAssembly” and write the following code under the class Class1:

```
public string SayHello() {
  return "Hello from shared assembly 1.0.0.0";
}
```

To associate key file we have generated with the project, open project properties window and to do that open Solution Explorer, right click on the Project and choose the option “Properties” and in the window opened select Signing tab on LHS, which displays a CheckBox as “Sign the Assembly” in RHS, select it, now in the ComboBox below select browse, select the “Key.snk” file from its physical location which adds the file under solution explorer, then compile the project using “Build” option that will generate “SAssembly.dll” which is Strong Named.

Step 3: installing assembly into GAC by using the “Global Assembly Cache Tool”.

To install the assembly into GAC open Developer Command Prompt for Vs, go to the location where “SAssembly.dll” was present and write the following:

```
<drive>:\<folder\SAssembly\SAssembly\bin\Debug\net5.0> gacutil -i SAssembly.dll
```

Step 4: testing the Shared Assembly.

Open a new project of type Console App., name it as “TestSAssembly1”, add a reference to “SAssembly.dll” from its physical location and write the following code under the Main method of Class1:

```
SAssembly.Class1 obj = new SAssembly.Class1();
Console.WriteLine(obj.SayHello());
Console.ReadLine();
```

Run the project, test it, and now this project i.e. “TestSAssembly1” can run by using the “SAssembly.dll” which is present in “GAC” i.e. we don’t require a copy of the “SAssembly.dll” to be present under the local folder.

Versioning Assemblies: Every assembly is associated with a set of attributes that describes about general info of an assembly like Title, Company, Description, Version etc. These attributes will be under “.csproj” file of the project. To view the “.csproj” file right click on the project in Solution Explorer and select “Edit Project File” which will open the “.csproj” file of the project. This is an XML file and under this file right now we find code as below:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <SignAssembly>true</SignAssembly>
    <AssemblyOriginatorKeyFile>Key.snk</AssemblyOriginatorKeyFile>
  </PropertyGroup>
</Project>
```

We can specify various details regarding the assembly like Company, Version, etc, etc, under this file by using XML Tags, for example if we want to specify the name of the Company who designed and developed this assembly we can use “<Company>” tag and specify it inside of the “<PropertyGroup>” tag as following:

```
<Company>NIT</Company>
```

Why do we maintain version numbers to an assembly?

Ans: Version no. is maintained for discriminating the changes that has been made from time to time. Version no's are changed for an assembly if there are any modifications or enhancements made in the code. The default version of every assembly is 1.0.0.0 and version no is a combination of 4 values like:

1. Major Version
2. Minor Version
3. Build Number
4. Revision

What are the criteria for changing the version no. of an assembly?

Ans: we change version no. of an assembly basing on the following criteria:

1. Change the Major version value when we add new types under the assembly.
2. Change the Minor version value when we modify any existing types under the assembly.
3. Change the Build Number when we add new members under types.
4. Change the Revision value when we modify any existing members under types.

Where do we change the version no. of an assembly?

Ans: we need to change the version no. of an assembly under the project property file, and to do that open Solution Explorer, right click on the project, select the option “Edit Project File” which opens “SAssembly.csproj” file in XML Format. Now under <PropertyGroup> tag we need to add the below statements in the last:

```
<AssemblyVersion>-Specify the new version no. here</AssemblyVersion>
<FileVersion>-Specify the new version no. here</FileVersion>
```

Testing the process of changing version no of an assembly:

Open the SAssembly project we have developed earlier and add a new method under Class1 as following:

```
public string SayHello2() {
  return "Hello from shared assembly 1.0.1.0";
}
```

Now open project property file and set “Company”, “AssemblyVersion” & “FileVersion” attributes as following:

```
<Company>NIT</Company>
<AssemblyVersion>1.0.1.0</AssemblyVersion>
<FileVersion>1.0.1.0</FileVersion>
```

Now Re-build the project and add the new version of “SAssembly.dll” i.e. “1.0.1.0” also into GAC using the Gacutil Tool.

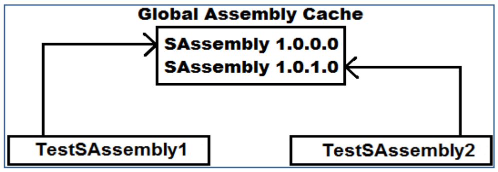
Note: GAC allows placing of multiple versions of an assembly in it and provides different applications using different versions of the assembly to execute correctly using their required version. Now if we open the GAC folder there we will find 2 versions of SAssembly i.e. “1.0.0.0” and “1.0.1.0”.

Assemblies and Side-by-Side Execution:

Side-by-side execution is the ability to store and execute multiple versions of an application or component on the same computer. Support for side-by-side storage and execution of different versions of the same assembly is an integral part of strong naming and is built into the infrastructure of the .NET Runtime. Because the strong-named assembly's version number is part of its identity, the .NET Runtime can store multiple versions of the same assembly in the global assembly cache and load those assemblies at run time. To test this open a new Console App. project naming it as “TestSAssembly2”, add reference of SAssembly “1.0.1.0” version from its physical location and write the below code under Main method of Class1:

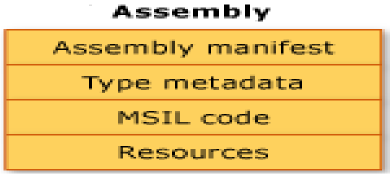
```
SAssembly.Class1 obj = new SAssembly.Class1();
Console.WriteLine(obj.SayHello2());
Console.ReadLine();
```

To check side by side execution of projects run the **“exe files”** of “TestSAssembly1” and “TestSAssembly2” projects at the same time, where each project will use its required version of “SAssembly” and execute, as below:



In general, an assembly is divided into four sections:

- The assembly manifest, which contains assembly metadata.
- Type metadata.
- Microsoft intermediate language (MSIL) Code or CIL Code that implements the types.
- A set of resources.



Assembly Manifest: contains information about the attributes that are associated with an assembly like Assembly Name, Assembly Version, File Version, Company, Strong Name Information, List of files in the assembly etc.

Type Metadata: describes every type and member defined in your code in a language-neutral manner. Metadata stores the following information:

- Description of the assembly.
  - Identity (name, version, culture, public key).
  - Other assemblies that this assembly depends on.
  - Security permissions needed to run.

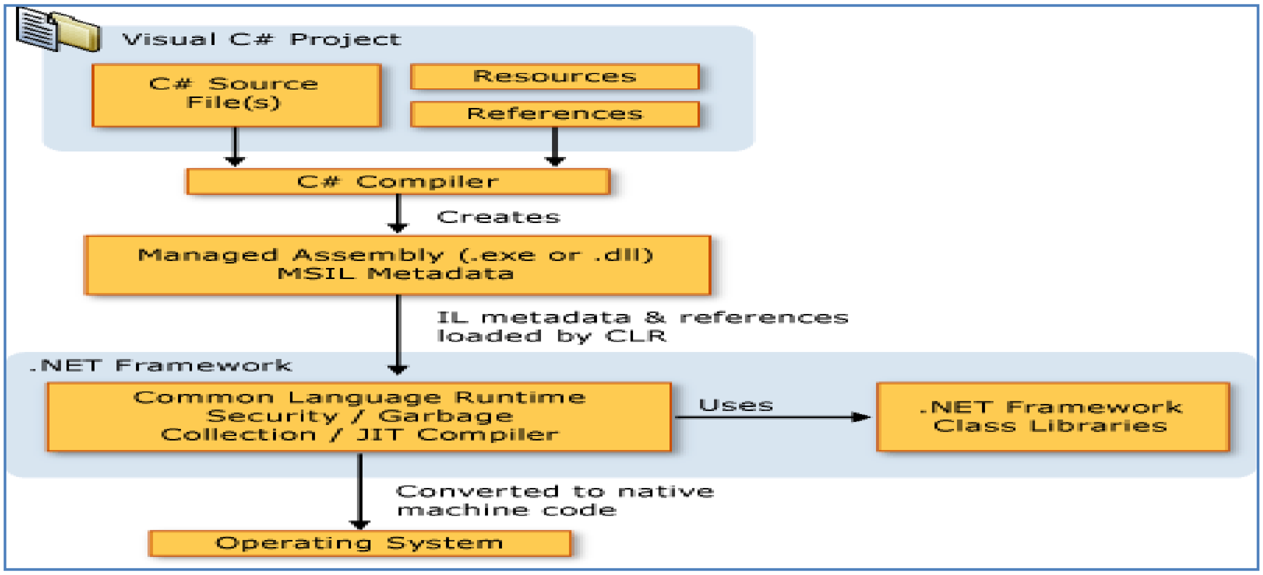
- Description of types.
  - Name, visibility, base class, and interfaces implemented.
  - Members (methods, fields, properties, events, nested types).

Metadata provides the following major benefits:

1. Self-describing files, common language runtime modules and assemblies are self-describing.
2. Language interoperability, metadata provides all the information required about compiled code for you to inherit a class from a file written in a different language.

MSIL Code or CIL Code: during compilation of any .NET programming languages, the source code is translated into CIL code rather than platform or processor-specific code. CIL is a CPU and platform-independent instruction set that can be executed in any environment supporting the Common Language Infrastructure, such as the .NET runtime on Windows, or the cross-platform Mono runtime.

Compilation and Execution Process of a C# Project:


Visual C# Project
C# Source File(s)
Resources
References
C# Compiler
Creates
Managed Assembly (.exe or .dll) MSIL Metadata
IL metadata & references loaded by CLR
.NET Framework
Common Language Runtime Security / Garbage Collection / JIT Compiler
Uses
.NET Framework Class Libraries
Converted to native machine code
Operating System

Task Parallel Library (TPL)

The Task Parallel Library (TPL) is a set of public types “System.Threading” and “System.Threading.Tasks” namespaces. The purpose of TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications. The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available. In addition, the TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details. By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish.

Starting with .NET Framework 4, the TPL is the preferred way to write multithreaded and parallel code. However, not all code is suitable for parallelization. For example, if a loop performs only a small amount of work on each iteration, or it doesn't run for many iterations, then the overhead of parallelization can cause the code to run more slowly. Furthermore, parallelization like any multithreaded code adds complexity to your program execution. Although the TPL simplifies multithreaded scenarios, it is recommend that you have a basic understanding of threading concepts, for example, locks, deadlocks, and race conditions, so that you can use the TPL effectively.

To test the examples given below create a new “Console Application” Project naming it as “TPLProject” and choose the Target Framework as: “.NET 5.0 (Current)” and click on the “Create” button.

First lets write a program without using Multi-Threading or Task Parallelism and to do that write the below code in the default class “Program” which is present under “Program.cs” file by deleting the existing code in the class:

```
using System.Threading;
class Program
{
 static void Print1()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print1 Method; { Thread.CurrentThread.ManagedThreadId }; {i}");
 }
 static void Print2()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print2 Method; {Thread.CurrentThread.ManagedThreadId}; {i}");
 }
 static void Print3()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print3 Method; {Thread.CurrentThread.ManagedThreadId}; {i}");
 }
 static void Main()
 {
  Print1(); Print2(); Print3();
  Console.ReadLine();
 }
}
```

Note: in the above program we have defined 3 methods and called them in a single threaded model so each method is executed 1 after the other and all the methods are executed by the Main Thread of the program and we can see the Id of that Thread which will be printed by the statement "Thread.CurrentThread.ManagedThreadId".

Now lets re-write the above program using Multi-Threading, and to do that add a new class in the project naming it as "Class1.cs" and write the below code in the class:

```
using System.Threading;
class Class1
{
 static void Print1()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print1 Method; {Thread.CurrentThread.ManagedThreadId}; {i}");
 }
 static void Print2()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print2 Method; {Thread.CurrentThread.ManagedThreadId}; {i}");
 }
 static void Print3()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print3 Method; {Thread.CurrentThread.ManagedThreadId}; {i}");
 }
 static void Main()
 {
  Thread t1 = new Thread(Print1);
  Thread t2 = new Thread(Print2);
  Thread t3 = new Thread(Print3);
  t1.Start(); t2.Start(); t3.Start();
  t1.Join(); t2.Join(); t3.Join();
  Console.WriteLine($"Main Method; {Thread.CurrentThread.ManagedThreadId}");
  Console.ReadLine();
 }
}
```

Note: in the above program we have defined 3 methods and called them by using 3 separate threads so each thread will execute 1 method concurrently and in the program we will be having 4 threads along with the Main thread and we can see the Id of those Threads which will be printed by the statement "Thread.CurrentThread.ManagedThreadId".

Now lets re-write the above program using Task Parallelism, and to do that add a new class in the project naming it as "Class2.cs" and write the below code in the class:

```
using System.Threading;
class Class2
{
 static void Print1()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print1 Method; {Thread.CurrentThread.ManagedThreadId}; {i}");
 }
 static void Print2()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print2 Method; {Thread.CurrentThread.ManagedThreadId}; {i}");
 }
 static void Print3()
 {
  for (int i = 1; i <= 100; i++)
   Console.WriteLine($"Print3 Method; {Thread.CurrentThread.ManagedThreadId}; {i}");
 }
 static void Main()
 {
  Task t1 = new Task(Print1);
  Task t2 = new Task(Print2);
  Task t3 = new Task(Print3);
  t1.Start(); t2.Start(); t3.Start();
  t1.Wait(); t2.Wait(); t3.Wait();
  Console.WriteLine($"Main Method; {Thread.CurrentThread.ManagedThreadId}");
  Console.ReadLine();
 }
}
```

In the above case in place of Threads we have used Tasks and these Tasks will internally use Threads to execute the code and the "Wait" method we called here is same as the "Join" method we use in Threads. The process of creating Tasks, calling Start and Wait methods can be simplified and implemented i.e. we can implement the first 5 lines of code in Main method of the above program as following also:

```
Task t1 = Task.Factory.StartNew(Print1);
Task t2 = Task.Factory.StartNew(Print2);
Task t3 = Task.Factory.StartNew(Print3);
Task.WaitAll(t1, t2, t3);
```

In the above code Factory is a Static property of the Task class which will refer to TaskFactory class and the StartNew method of TaskFactory class will create a new Thread, starts it and returns the reference of it.

Note: in the above program we have defined 3 methods and called them by using 3 separate tasks so each task will execute 1 method concurrently. In this program also we will be having 4 threads along with the Main thread and we can see the Id of those Threads which will be printed by the statement “Thread.CurrentThread.ManagedThreadId”.

Calling value returning methods by using Tasks: in the above program the methods that we called by using Tasks are not value returning as well as they do not take any parameters also. Now lets learn how to call the value returning method by using Task and to do that add a new class in the project naming it as “Class3.cs” and write the below code in it:

```
class Class3
{
  static int GetLength()
  {
    string str = "";
    for (int i = 1; i <= 100000; i++)
      str += i;
    return str.Length;
  }
  static void Main()
  {
    Task<int> t = new Task<int>(GetLength);
    t.Start();
                     or
    Task<int> t = Task.Factory.StartNew(GetLength);

    int result = t.Result;
    Console.WriteLine(result);
    t.Wait();
    Console.ReadLine();
  }
}
```

Note: in the above program the GetLength() method of the class concatinates 1 to 100000 and then returns the length of that string, so in this case to capture that value we need to use the Task class which takes the generic parameter <T> and in this case <T> is of type integer and after execution of the method we can capture the result by calling “Result” property of Task class.

Calling value returning method with parameters by using Tasks: in the above program the method that we called by using Task is a value returning method and now lets learn how to call the value returning method which takes parameters also, by using Task and to do that add a new class in the project naming it as “Class4.cs” and write the below code in it:

```
class Class4
{
  static int GetLength(int ub)
  {
    string str = "";
    for (int i = 1; i <= ub; i++)
      str += i;
    return str.Length;
  }
  static void Main()
  {
    Task<int> t = new Task<int>(() => GetLength(100000));
    t.Start();
                            or
    Task<int> t = Task.Factory.StartNew(() => GetLength(100000));
    int result = t.Result;
    Console.WriteLine(result);
    t.Wait();
    Console.ReadLine();
  }
}
```

Note: in the above program GetLength() method of class concatinates 1 to a number that is passed to the method as parameter value and then returns the length of that string, so in this case to pass values to the method we need to take the help of a delegate.

Thread Synchronization: synchronization is a technique that allows only one thread to access the resource for the particular time. No other thread can interrupt until the assigned thread finishes its task. In multithreading program, threads are allowed to access any resource for the required execution time. Threads share resources and executes asynchronously. Accessing shared resources (data) is critical task that sometimes may halt the system. We deal with it by making threads synchronized. It is mainly used in case of transactions like deposit, withdraw etc. We can use C# lock keyword to execute program synchronously. It is used to get lock for the current thread, execute the task and then release the lock. It ensures that other thread does not interrupt the execution until the execution finish. To test this, add a new class in the project naming it as “Class5.cs” and write the below code in it:

```
using System.Threading;
class Class5
{
  public static void Print()
  {
    lock (typeof(Class5))
    {
      Console.Write("[CSharp Is ");
      Thread.Sleep(5000);
      Console.WriteLine("Object Oriented]");
    }
  }
  static void Main()
  {
    Thread t1 = new Thread(Print);
    Thread t2 = new Thread(Print);
    Thread t3 = new Thread(Print);
    t1.Start(); t2.Start(); t3.Start();
    t1.Join(); t2.Join(); t3.Join();
    Console.ReadLine();
  }
}
```

If we want to perform synchronization with Tasks then here also the process is same and to test this, add a new class in the project naming it as “Class6.cs” and write the below code in it:

```
class Class6
{
  public static void Print()
  {
    lock (typeof(Class6))
    {
      Console.Write("[CSharp Is ");
      Task.Delay(5000).Wait();
      Console.WriteLine("Object Oriented]");
    }
  }
  static void Main()
  {
    Task t1 = Task.Factory.StartNew(Print);
    Task t2 = Task.Factory.StartNew(Print);
    Task t3 = Task.Factory.StartNew(Print);
    Task.WaitAll(t1, t2, t3);
    Console.ReadLine();
  }
}
```

Data Parallelism: this refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source like an array or collection. In data parallel operations, the source is partitioned so that multiple threads can operate on different segments concurrently. The Task Parallel Library (TPL) supports data parallelism through “Parallel” class which is present under System.Threading.Tasks namespace. This class provides method-based parallel implementations of for and foreach loops. You write the loop logic for a “Parallel.For” or “Parallel.ForEach” loop much as you would write a sequential loop. You do not have to create threads or queue the work items i.e. TPL handles all the low-level work for you.

Sequential Version:

```
foreach (var item in sourceCollection)
{
   Process(item);
}
```

Parallel Equivalent:

```
Parallel.ForEach(sourceCollection, item => Process(item));
```

Lets now write a program to understand the difference between sequential for loop and parallel for loop and to do that add a new class in the project naming it as “Class7.cs” and write the below code in it:

```
using System.Diagnostics;
class Class7
{
  static void Main()
  {
    Stopwatch sw1 = new Stopwatch();
    sw1.Start();
    string str1 = "";
    for (int i = 1; i < 200000; i++)
    {
      str1 = str1 + i;
    }
    sw1.Stop();
    Console.WriteLine("Time taken to execute the code by using sequential for loop: " +
                                   sw1.ElapsedMilliseconds);

    Stopwatch sw2 = new Stopwatch();
    sw2.Start();
    string str2 = "";
    Parallel.For(1, 200000, i =>
    {
      str2 = str2 + i;
    });
    sw2.Stop();
    Console.WriteLine("Time taken to execute the code by using parallel for loop: " +
                                  sw2.ElapsedMilliseconds);
    Console.ReadLine();
  }
}
```

Lets now write another program to understand the difference between sequential foreach loop and parallel foreach loop and to do that add a new class in the project naming it as “Class8.cs” and write the below code in it:

```
using System.Threading;
using System.Diagnostics;
class Class8
{
  static void Main()
  {
    int[] arr = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
       21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40 };
    Stopwatch sw1 = new Stopwatch();
    sw1.Start();
    foreach (int i in arr)
    {
      Console.WriteLine(Thread.CurrentThread.ManagedThreadId + ": " + i);
    }
    sw1.Stop();
    Console.WriteLine("Time taken to execute the code by using sequential foreach loop: "
                            + sw1.ElapsedMilliseconds);
    Console.WriteLine();
    Stopwatch sw2 = new Stopwatch();
    sw2.Start();
    Parallel.ForEach(arr, i =>
    {
      Console.WriteLine(Thread.CurrentThread.ManagedThreadId + ": " + i);
    });
    sw2.Stop();
    Console.WriteLine("Time taken to execute the code by using parallel foreach loop: "
                            + sw2.ElapsedMilliseconds);
    Console.ReadLine();
  }
}
```

Note: If you observe the above 2 programs in the first code parallel for loop executed much faster than a sequential for loop where as in the second case sequential foreach loop executed faster than parallel foreach loop because when we are doing any bulk task inside the loop then parallel loops are faster whereas if you are just iterating and doing a small task inside a loop then sequential loops are faster.

Chaining Tasks using Continuation Tasks: in asynchronous programming, it's common for one asynchronous operation, on completion, to invoke a second operation. Continuations allow descendant operations to consume the results of the first operation. A continuation task (also known just as a continuation) is an asynchronous task that's invoked by another task, known as the antecedent, when the antecedent finishes. To test this, add a new class in the project naming it as “Class9.cs” and write the below code in it:

```
class Class9
{
  static void Method1(int x, int ub)
  {
    for (int i = 1; i <= ub; i++)
      Console.WriteLine($"{x} * {i} = {x * i}");
  }
  static void Method2(int x, int ub)
  {
    for (int i = ub; i > 0; i--)
      Console.WriteLine($"{x} * {i} = {x * i}");
  }
  static void Main()
  {
    Task t = Task.Factory.StartNew(() => Method1(5, 12)).ContinueWith((antecedent) =>
       Console.WriteLine()).ContinueWith((antecedent) => Method2(5, 12));
    t.Wait();
    Console.ReadLine();
  }
}
```

Asynchronous programming with async and await: async and await in C# are the code markers, which marks code positions from where the control should resume after a task completes. When we are dealing with UI, and on a button click we called a long-running method like reading a large file or something else which will take a long time and in that case the entire application must wait to complete the task. In other words, if a process is blocked in a synchronous application, the whole application gets blocked and stops responding until the whole task completes.

```
class Class10
{
  static async void Test()
  {
    Console.WriteLine("Started reading values from DB.....");
    await Task.Delay(10000);
    Console.WriteLine("Completed reading values from DB.....");
  }
  static void Main()
  {
    Test();
    Console.Write("Please enter your name: ");
    string Name = Console.ReadLine();
    Console.ReadLine();
  }
}
```

Task Cancellation: task classes support cancellation through the use of cancellation tokens. In the Task classes, cancellation involves cooperation between the user delegate, which represents a cancelable operation, and the code that requested the cancellation. A successful cancellation involves the requesting code calling the CancellationTokenSource.

```
using System.Threading;
class Class11
{
  private static void CancellableWork(CancellationToken cancellationToken)
  {
    for (int i = 0; i < 10; i++)
    {
      Thread.Sleep(1000);
      if (cancellationToken.IsCancellationRequested)
      {
        Console.WriteLine($"Cancelled on iteration {i + 1}");
        cancellationToken.ThrowIfCancellationRequested();
      }
      Console.WriteLine($"Iteration {i + 1} completed");
    }
  }
  static async Task Main(string[] args)
  {
    Console.WriteLine("Starting application...");
    CancellationTokenSource source = new CancellationTokenSource();
    CancellationToken token = source.Token;
    Task task = Task.Factory.StartNew(() => CancellableWork(token));
    Console.WriteLine("Press C to cancel");
    char ch = Console.ReadKey().KeyChar;
    if (ch == 'c' || ch == 'C')
    {
      source.Cancel();
      Console.WriteLine("\nTask cancellation requested.");
    }
    try
    {
      await Task.WhenAll(task);
    }
    catch (OperationCanceledException)
    { Console.WriteLine($"\n{nameof(OperationCanceledException)} Thrown.\n"); }
    finally
    { source.Dispose(); }
    Console.WriteLine("Process completed.");
    Console.ReadLine();
  }
}
```

Understanding Async and Await with a GUI Application: Create a new project of type "Windows Forms App", naming it as "TPLProjectWindows" and click on the Create button. By default it opens an item with the name "Form1.cs [Design]". Go to "View Menu" and select the MenuItem "Toolbox" with will launch that in RHS and in that under the tab "Common Windows Forms" we will find "Button" control, drag and drop 2 Buttons on to the Form. Select "button1", hit F4 which launch Property Window in the RHS, identify the "Text" property and set the value as "Get Length", same as this set the "Text" for "button2" as "Say Hello". Now let's implement logic in the class and to do that double click on "button1" which will take you to a file "Form1.cs" and there we find a method "button1_Click" and do the same with "button2" also which will generate a method "button2_Click", now write the below code under the class Form1:

```
private int Length()
{
  string s = "";
  for (int i = 1; i <= 100000; i++)
    s += i;
  return s.Length;
}
```

Write the below code under button1_Click method:

```
int result = Length();
MessageBox.Show(result.ToString());
```

Write the below code under button2_Click method:

```
MessageBox.Show("Hello World");
```

Now run the Form by hitting "F5" and click on "button1" which will be executing the logic we have implemented, but this takes time because it has to iterate for 1 lakh times and the drawback here is until this action is completed we can't execute the code under "button2" and if we try to click also the UI will be non-responsive, so to overcome this problem change the method "button1_Click" as below:

```
private async void button1_Click(object sender, EventArgs e)
{
  Task<int> task = Task.Factory.StartNew(Length);
  int result = await task;
  MessageBox.Show(result.ToString());
}
```

Now run and watch the difference i.e. you click on "button1" which will be executing the logic and meanwhile we can click on "button2" also and the UI will be responsive.

Logical Programs

Write a program to print the given no is a prime number or not?

```
class PrimeNumberTest
{
  static void Main()
  {
    Console.Write("Enter a number to check it's a prime: ");
    uint Number = uint.Parse(Console.ReadLine());
    if(Number == 0 || Number == 1)
    {
      Console.WriteLine("Please enter a number other than 0 & 1");
      return;
    }
    bool IsPrime = true;
    uint HalfNumber = Number / 2;
    for(uint i = 2;i<=HalfNumber;i++)
    {
      if(Number % i == 0)
      {
        IsPrime = false;
        break;
      }
    }
    if(IsPrime == true)
      Console.WriteLine("Given number is a prime.");
    else
      Console.WriteLine("Given number is not a prime.");
    Console.ReadLine();
  }
}
```

Write a program to swap 2 numbers with out using 3 variable?

```
class SwapNumbers1          //Solution 1
{
  static void Main()
  {
    int a = 342, b = 784;
    Console.WriteLine($"Numbers Before Swap: a => {a}; b => {b}");
    a = a * b;     b = a / b;     a = a / b;
    Console.WriteLine($"Numbers After Swap: a => {a}; b => {b}");
    Console.ReadLine();
  }
}
class SwapNumbers2          //Solution 2
{
  static void Main()
  {
    Console.Write("Enter 1st number: ");
    int a = int.Parse(Console.ReadLine());
    Console.Write("Enter 2nd number: ");
    int b = int.Parse(Console.ReadLine());
    Console.WriteLine($"Numbers Before Swap: a => {a}; b => {b}");
    a = a + b;     b = a - b;     a = a - b;
    Console.WriteLine($"Numbers After Swap: a => {a}; b => {b}");
    Console.ReadLine();
  }
}
```

Write a program to print the reverse of a given number?

```
class ReverseNumber
{
  static void Main()
  {
    Console.Write("Enter a number: ");
    int Number = int.Parse(Console.ReadLine());
    int Reminder, Reverse = 0;
    while(Number != 0)
    {
      Reminder = Number % 10;
      Reverse = Reverse * 10 + Reminder;
      Number = Number / 10;
    }
    Console.WriteLine("Reversed Number is: " + Reverse);
    Console.ReadLine();
  }
}
```

Write the program to print the binary value of of a given number?

```
class NumberToBinary
{
  static void Main()
  {
    Console.Write("Enter an number to convert into binary: ");
    int Number = int.Parse(Console.ReadLine());
    int[] arr = new int[16];
    int i;
    for(i = 0;Number > 0;i++)
    {
      arr[i] = Number % 2;
      Number = Number / 2;
    }
    Console.Write("Binary value of the given number is: ");
    for(i = i - 1;i >= 0;i--)
    {
      Console.Write(arr[i]);
    }
    Console.ReadLine();
  }
}
```

Write a program to check whether a given number is a palindrome?

```
class PalindromeNumber
{
  static void Main()
  {
    Console.Write("Enter a Number: ");
    int Number = int.Parse(Console.ReadLine());
    int OldNumber = Number;
    int Reminder, Reverse = 0;
    while(Number != 0)
    {
      Reminder = Number % 10;
      Reverse = (Reverse * 10) + Reminder;
      Number = Number / 10;
    }
    if (OldNumber == Reverse)
      Console.WriteLine("Given number is a palindrome");
    else
      Console.WriteLine("Given number is not a palindrome");
    Console.ReadLine();
  }
}
```

Write a program to print the fibanocci series upto a given upper bound?

```
class FibanocciSeries
{
  static void Main()
  {
    Console.Write("Enter the number of elements for Fibanocci Series: ");
    int Number = int.Parse(Console.ReadLine());
    int Num1 = 0, Num2 = 1, Num3;
    Console.Write(Num1 + "  " + Num2 + "  ");
    for(int i = 2;i < Number;i++)
    {
      Num3 = Num1 + Num2;
      Console.Write(Num3 + "  ");
      Num1 = Num2;
      Num2 = Num3;
    }
    Console.ReadLine();
  }
}
```

Write a program to print the factorial of a given number?

```
class Factorial
{
  static void Main()
  {
    Console.Write("Enter a number to find it's factorial: ");
    uint Number = uint.Parse(Console.ReadLine());
    uint Result = 1;
    for(uint i=1;i<=Number;i++)
    {
      Result = Result * i;
    }
    Console.WriteLine("Factorial of given number is: " + Result);
    Console.ReadLine();
  }
}
```

Write a program to find whether the give number is an Armstrong number or not?

```
class ArmstrongNumber
{
  static void Main()
  {
    Console.Write("Enter a number to find it is Armstrong: ");
    int Number = int.Parse(Console.ReadLine());
    int Original = Number;
    int Reminder, Sum = 0;
    while(Number > 0)
    {
      Reminder = Number % 10;
      Sum = Sum + (Reminder * Reminder * Reminder);
      Number = Number / 10;
    }
    if (Original == Sum)
      Console.Write($"{Original} is an armstrong number");
    else
      Console.Write($"{Original} is not an armstrong number");
    Console.ReadLine();
  }
}
```

Write a program to find the sum of digits of a given number?

```
class SumOfDigits1
{
  static void Main()
  {
    Console.Write("Enter a number to find sum of it's digits: ");
    int Number = int.Parse(Console.ReadLine());
    int Reminder, Sum = 0;
    while(Number > 0)
    {
      Reminder = Number % 10;
      Sum = Sum + Reminder;
      Number = Number / 10;
    }
    Console.WriteLine("Sum of the digits of given no is: " + Sum);
    Console.ReadLine();
  }
}
```

Write a program to find the sum of digits of a given number until single digit?

```
class SumOfDigits2
{
  static void Main()
  {
    Console.Write("Enter a number to find sum of it's digits: ");
    int Number = int.Parse(Console.ReadLine());
    int Reminder, Sum = 0;
    do
    {
      if(Sum != 0)
      {
        Number = Sum;
    Sum = 0;
   }
   while (Number > 0)
   {
    Reminder = Number % 10;
    Sum = Sum + Reminder;
    Number = Number / 10;
   }
  }
  while (Sum > 9);
   Console.WriteLine("Sum of the digits of given no is: " + Sum);
  Console.ReadLine();
 }
}
```

Write a program to print the given number in words?

```
class NumberToString
{
 static void Main()
 {
  Console.Write("Enter a number: ");
  int Number = int.Parse(Console.ReadLine());
  int Reminder, Reverse = 0;
  while(Number > 0)
  {
   Reminder = Number % 10;
   Reverse = Reverse * 10 + Reminder;
   Number = Number / 10;
  }
  while(Reverse > 0)
  {
   Reminder = Reverse % 10;
   switch (Reminder)
   {
    case 1:
     Console.Write("one ");
     break;
    case 2:
     Console.Write("two ");
     break;
    case 3:
     Console.Write("three ");
     break;
    case 4:
     Console.Write("four ");
     break;
    case 5:
     Console.Write("five ");
     break;
    case 6:
     Console.Write("six ");
     break;
    case 7:
     Console.Write("seven ");
     break;
    case 8:
     Console.Write("eight ");
     break;
    case 9:
     Console.Write("nine ");
     break;
    case 0:
     Console.Write("zero ");
     break;
   }
   Reverse = Reverse / 10;
  }
  Console.ReadLine();
 }
}
```

Write a program to find the given year is a leap year or not?

```
class LeapYear
{
 static void Main()
 {
  Console.Write("Enter the year in 4 digits: ");
  int Year = int.Parse(Console.ReadLine());
  if ((Year % 4 == 0 && Year % 100 != 0) || (Year % 400 == 0))
   Console.WriteLine($"{Year} is a leap year.");
  else
   Console.WriteLine($"{Year} is not a leap year.");
  Console.ReadLine();
 }
}
```

Write a program to print the larger number in an array?

```
class LargerNumberInArray
{
 static void Main()
 {
  Console.Write("Specify the no of items to compare: ");
  int UB = int.Parse(Console.ReadLine());
  Console.Clear();
  int[] arr = new int[UB];
  for(int i=0;i<UB;i++)
  {
   Console.Write($"Enter Item{i + 1}: ");
   arr[i] = int.Parse(Console.ReadLine());
  }
  int LargeNumber = arr[0];
  for(int i=1;i<UB;i++)
  {
   if(arr[i] > LargeNumber)
   {
    LargeNumber = arr[i];
   }
  }
  Console.WriteLine("Larger number in the array is: " + LargeNumber);
  Console.ReadLine();
 }
}
```

Write a program to print the given string in reverse?

```
class StringReverse
{
 static void Main()
```

```
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    string reverse = "";
    foreach(char ch in input)
      reverse = ch + reverse;
    Console.WriteLine($"Reverse of given string '{input}' is: '{reverse}'");
    Console.ReadLine();
  }
}
```

Write a program to print the no. of words in a given string?

```
class WordCount
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    int Count = 0, CharCount = 0;
    bool Flag = true, EndSpace = false;
    bool StartSpace = false;
    foreach (char ch in input)
    {
      CharCount += 1;
      if (CharCount == 1 && ch == 32)
        StartSpace = true;
      if (ch == 32 && Flag == false)
        continue;
      else
      {
        Flag = true;
        EndSpace = false;
      }
      if (Count == 0)
        Count = 1;
      if (ch == 32)
      {
        Count += 1;
        Flag = false;
        EndSpace = true;
      }
    }
    if (StartSpace == true)
      Count -= 1;
    if (EndSpace == true)
      Count -= 1;
    Console.WriteLine("No of words in the given string are: " + Count);
    Console.ReadLine();
  }
}
```

Write a program to print the length of a given string?

```
class StringLength
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    int Length = 0;
    foreach (char ch in input)
      Length += 1;
    Console.WriteLine("Length of given string is: " + Length);
    Console.ReadLine();
  }
}
```

Write a program to print the no. of characters in a given string?

```
class CharCount
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    int Length = 0;
    foreach (char ch in input)
    {
      if(ch != 32)
        Length += 1;
    }
    Console.WriteLine("No. of char's in given string are: " + Length);
    Console.ReadLine();
  }
}
```

Write a program to print the words in reverse order of a given string?

```
class ReverseWords
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    string word = "", reverseWords = "";
    foreach(char ch in input)
    {
      if (ch != 32)
        word = word + ch;
      else
      {
        reverseWords = " " + word + reverseWords;
        word = "";
      }
    }
    if (word != "")
      reverseWords = word + reverseWords;
    Console.WriteLine(reverseWords);
    Console.ReadLine();
  }
}
```

Write a program to convert the given string in to lower case?

```
class StringToLower
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    string output = "";
    foreach(char ch in input)
    {
      if (ch >= 65 && ch <= 90)
        output += (char)(ch + 32);
      else
        output += ch;
    }
    Console.WriteLine(output);
    Console.ReadLine();
  }
}
```

Write a program to convert the given string in to upper case?

```
class StringToUpper
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    string output = "";
    foreach (char ch in input)
    {
      if (ch >= 97 && ch <= 122)
        output += (char)(ch - 32);
      else
        output += ch;
    }
    Console.WriteLine(output);
    Console.ReadLine();
  }
}
```

Write a program to convert the given string in to pascal case?

```
class StringToPascal
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    string lower = "";
    foreach (char ch in input)
    {
      if (ch >= 65 && ch <= 90)
        lower += (char)(ch + 32);
      else
        lower += ch;
    }
    string pascal = "";
    bool firstChar = true, flag = false;
    foreach(char ch in lower)
    {
      if (firstChar == true)
      {
        if (ch >= 97 && ch <= 122)
          pascal += (char)(ch - 32);
        firstChar = false;
        continue;
      }
      if (flag == true)
      {
        if (ch >= 97 && ch <= 122)
          pascal += (char)(ch - 32);
        flag = false;
      }
      else
        pascal += ch;
      if (ch == 32)
        flag = true;
    }
    Console.WriteLine(pascal);
    Console.ReadLine();
  }
}
```

Write a program to find out the unique characters in a given string?

```
class UniqueChars
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    bool Exists = false;
    int Count1 = 0, Count2 = 0;
    foreach(char ch1 in input)
    {
      Count1 += 1;
      foreach(char ch2 in input)
      {
        Count2 += 1;
        if(Count1 != Count2)
        {
          if (ch1 != ch2 && ch1 != 32)
            Exists = false;
          else
          {
            Exists = true;
            break;
          }
        }
      }
      if (Exists == false)
        Console.Write(ch1);
      Count2 = 0; Exists = false;
    }
    Console.ReadLine();
  }
}
```

Write a program to find out the duplicate characters in a given string?

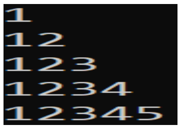
```
class DuplicateChars
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    string input = Console.ReadLine();
    int Length = 0;
    foreach (char ch in input)
      Length += 1;
    char[] arr = new char[Length];
    int Index = 0;
    foreach(char ch in input)
    {
      arr[Index] = ch;
      Index += 1;
    }
    int Count1 = 0, Count2 = 0;
    foreach(char ch1 in arr)
    {
      Count1 += 1;
      foreach(char ch2 in arr)
      {
        Count2 += 1;
        if(Count1 != Count2)
        {
          if(ch1 == ch2 && ch1 != 32)
          {
            Console.WriteLine(ch1);
            arr[Count1 - 1] = ' ';
            arr[Count2 - 1] = ' ';
            break;
          }
        }
      }
      Count2 = 0;
    }
    Console.ReadLine();
  }
}
```

Write a program to print the roman number of a given number?

```
class NumberToRoman
{
  static void Main()
  {
    Console.Write("Enter a string: ");
    int num = int.Parse(Console.ReadLine());
    string roman = ToRoman(num);
    Console.WriteLine(roman);
    Console.ReadLine();
  }
  public static string ToRoman(int num)
  {
    if (num < 0 || num > 3999)
      return "Enter a number between 1 and 3999";
    else if (num >= 1000)
      return "M" + ToRoman(num - 1000);
    else if (num >= 900)
      return "CM" + ToRoman(num - 900);
    else if (num >= 500)
      return "D" + ToRoman(num - 500);
    else if (num >= 400)
      return "CD" + ToRoman(num - 400);
    else if (num >= 100)
      return "C" + ToRoman(num - 100);
    else if (num >= 90)
      return "XC" + ToRoman(num - 90);
    else if (num >= 50)
```

```
      return "L" + ToRoman(num - 50);
    else if (num >= 40)
      return "XL" + ToRoman(num - 40);
    else if (num >= 10)
      return "X" + ToRoman(num - 10);
    else if (num >= 9)
      return "IX" + ToRoman(num - 9);
    else if (num >= 5)
      return "V" + ToRoman(num - 5);
    else if (num >= 4)
      return "IV" + ToRoman(num - 4);
    else if (num >= 1)
      return "I" + ToRoman(num - 1);
    else
      return "";
  }
}
```

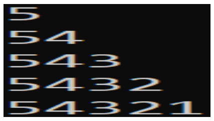
Write a program to print the below output:



```
1
12
123
1234
12345
```

```
class Pattern1
{
  static void Main()
  {
    Console.Write("Enter a number: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    for(int i=1;i<=num;i++)
    {
      for(int j=1;j<=i;j++)
        Console.Write(j);
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

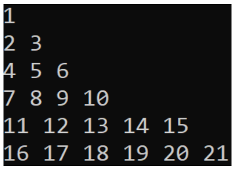
Write a program to print the below output:



```
5
54
543
5432
54321
```

```
class Pattern2
{
  static void Main()
  {
    Console.Write("Enter a number: ");
    int Number = int.Parse(Console.ReadLine());
    Console.Clear();
    for (int i = Number; i >= 1; i--)
    {
      for (int j = Number; j >= i; j--)
        Console.Write(j);
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

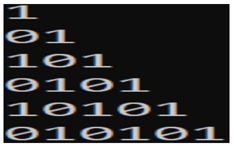
Write a program to print the below output:



```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
```

```
class Pattern3
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int Rows = int.Parse(Console.ReadLine());
    Console.Clear();
    int x = 1;
    for(int i=1;i<=Rows;i++)
    {
      for (int j = 1; j <= i; j++)
        Console.Write($"{x++} ");
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

Write a program to print the below output:

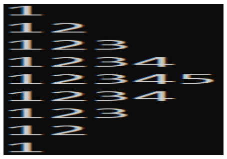


```
1
01
101
0101
10101
010101
```

```
class Pattern4
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int Rows = int.Parse(Console.ReadLine());
    Console.Clear();
    int x = 0, y = 0;
    for (int i = 1; i <= Rows; i++)
    {
      if(i % 2 == 0)
      {
        x = 1;
        y = 0;
      }
      else
      {
        x = 0;
        y = 1;
      }
      for (int j = 1; j <= i; j++)
      {
        if (j % 2 == 0)
```

```
          Console.Write(x);
        else
          Console.Write(y);
      }
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

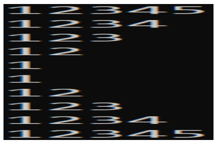
Write a program to print the below output:



```
1
12
123
1234
12345
1234
123
12
1
```

```
class Pattern5
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    for (int i = 1; i < num; i++)
    {
      for (int j = 1; j <= i; j++)
        Console.Write(j);
      Console.WriteLine();
    }
    for(int i=num;i>=0;i--)
    {
      for (int j = 1; j <= i; j++)
        Console.Write(j);
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

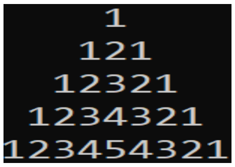
Write a program to print the below output:



```
12345
1234
123
12
1
1
12
123
1234
12345
```

```
class Pattern6
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    for (int i = num; i >= 0; i--)
    {
      for (int j = 1;j<= i; j++)
        Console.Write(j);
      if(i > 0)
        Console.WriteLine();
    }
    for (int i = 1; i <= num; i++)
    {
      for (int j = 1; j <= i; j++)
        Console.Write(j);
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

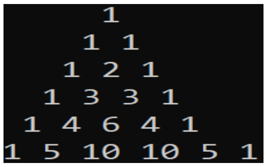
Write a program to print the below output:



```
    1
   121
  12321
 1234321
123454321
```

```
class Pattern7
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    for(int i=1;i<= num;i++)
    {
      for (int space = 1; space <= (num - i); space++)
        Console.Write(" ");
      for (int j = 1; j <= i; j++)
        Console.Write(j);
      for (int k = (i - 1); k >= 1; k--)
        Console.Write(k);
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

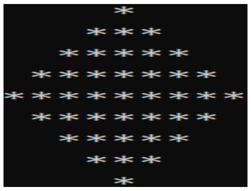
Write a program to print the below output:



```
     1
    1 1
   1 2 1
  1 3 3 1
 1 4 6 4 1
1 5 10 10 5 1
```

```
class Pattern8
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    int result;
    for(int i=0;i<=num;i++)
    {
      result = 1;
      for(int j=i;j <= num - 1;j++)
        Console.Write(" ");
      for(int k=0;k<=i;k++)
      {
        Console.Write(result + " ");
        result = (result * (i - k) / (k + 1));
      }
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

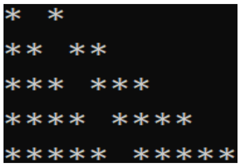
Write a program to print the below output:



```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

```
class Pattern9
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    int count = num - 1;
    for(int i=1;i<num+1;i++)
    {
      for (int j = 1; j <= count; j++)
        Console.Write(" ");
      count--;
      for (int k = 1; k <= 2 * i - 1; k++)
        Console.Write("*");
      Console.WriteLine();
    }
    count = 1;
    for(int i=1;i<=num-1;i++)
    {
      for (int j = 1; j <= count; j++)
        Console.Write(" ");
      count++;
      for (int k = 1; k <= 2 * (num - i) - 1; k++)
        Console.Write("*");
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

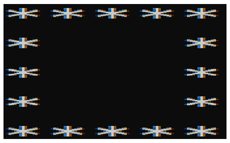
Write a program to print the below output:



```
* *
** **
*** ***
**** ****
***** *****
```

```
class Pattern10
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    for(int i=0;i<num;i++)
    {
      for (int j = 0; j <= i; j++)
        Console.Write("*");
      Console.Write(" ");
      for (int j = 0; j <= i; j++)
        Console.Write("*");
      Console.WriteLine();
    }
    Console.ReadLine();
  }
}
```

Write a program to print the below output:



```
* * * * *
*       *
*       *
*       *
* * * * *
```

```
class Pattern11
{
  static void Main()
  {
    Console.Write("Enter number of rows: ");
    int num = int.Parse(Console.ReadLine());
    Console.Clear();
    for(int i=0;i < num;i++)
    {
      if(i == 0 || i == num - 1)
      {
        for (int j = 0; j < num; j++)
          Console.Write('*');
        Console.WriteLine();
      }
      else
      {
        for(int j=0;j<num;j++)
        {
          if (j == 0 || j == num - 1)
            Console.Write('*');
          else
            Console.Write(' ');
        }
        Console.WriteLine();
      }
    }
    Console.ReadLine();
  }
}
```

Bubble Sort: how does Bubble Sort work is starting at index zero, we take an item and the item next in the array and compare them. If they are in the right order, then we do nothing, if they are in the wrong order (e.g. the item lower in the array is actually a higher value than the next element), then we swap these items. Then we continue through each item in the array doing the same thing (Swapping with the next element if it’s higher).

```
class BubbleSort
```

```
{
  static void Main(string[] args)
  {
    int[] arr = { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14, 28 };
    bool itemMoved = false;
    do
    {
      itemMoved = false;
      for (int i = 0; i < arr.Length - 1; i++)
      {
        if (arr[i] > arr[i + 1])
        {
          int lowerValue = arr[i + 1];
          arr[i + 1] = arr[i];
          arr[i] = lowerValue;
          itemMoved = true;
        }
      }
    } while (itemMoved);
    foreach (int i in arr)
      Console.Write(i + "  ");
    Console.ReadLine();
  }
}
```

Now since we are only comparing each item with its neighbor, each item may only move a single place when it actually needs to move several places. So how does Bubble Dort solve this? Well it just runs the entire process all over again. Notice how we have the variable called “itemMoved”. We simply set this to true if we did swap an item and start the scan all over again. Because we are moving things one at a time, not directly to the right position, and having to multiple passes to get things right, Bubble Sort is seen as extremely inefficient.

Selection Sort: It's remarkably a simple algorithm to explain and the way Selection Sort works is an outer loop visits each item in the array to find out whether it is the minimum of all the elements after it. If it is not the minimum, it is going to be swapped with whatever item in the rest of the array is the minimum. For example, if you have an array of 10 elements, this means that “i” goes from 0 to 9. When we are looking at position 0, we check to find the position of the minimum element in positions 1 .. 9. If the minimum is not already at position “i”, we swap the minimum into place. Then we consider “i = 1” and look at positions 2 .. 9. And so on.

```
class SelectionSort
{
  static void Main()
  {
    int[] arr = { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14, 28 };
    for(int i=0;i<arr.Length;i++)
    {
      int min = i;
      for(int j=i+1;j<arr.Length;j++)
      {
        if(arr[min] > arr[j])
        {
          min = j;
        }
      }
      if(min != i)
      {
        int lowerValue = arr[min];
        arr[min] = arr[i]; arr[i] = lowerValue;
      }
    }
    foreach (int i in arr)
      Console.Write(i + "  ");
    Console.ReadLine();
  }
}
```

Insertion Sort: In the Insertion Sort algorithm, we build a sorted list from the bottom of the array. We repeatedly insert the next element into the sorted part of the array by sliding it down to its proper position. This will require as many exchanges as Bubble Sort, since only one inversion is removed per exchange.

```
class InsertionSort
{
  static void Main()
  {
    int[] arr = { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14, 28 };
    for(int i=0;i<arr.Length;i++)
    {
      int item = arr[i];
      int currentIndex = i;
      while(currentIndex > 0 && arr[currentIndex - 1] > item)
      {
        arr[currentIndex] = arr[currentIndex - 1];
        currentIndex--;
      }
      arr[currentIndex] = item;
    }
    foreach (int i in arr)
      Console.Write(i + "  ");
    Console.ReadLine();
  }
}
```

Shell Sort: Donald Shell published the first version of this sort, hence this is known as Shell sort. This sorting is a generalization of insertion sort that allows the exchange of items that are far apart. It starts by comparing elements that are far apart and gradually reduces the gap between elements being compared. The running time of Shell sort varies depending on the gap sequence it uses to sort the elements.

```
class ShellSort
{
  static void Main()
  {
    int[] arr = { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14, 28 };
    int n = arr.Length;
    int gap = n / 2;
    int temp;
    while (gap > 0)
    {
      for(int i=0;i + gap < n;i++)
      {
        int j = i + gap;
        temp = arr[j];
        while(j - gap >= 0 && temp < arr[j - gap])
        {
          arr[j] = arr[j - gap];
          j = j - gap;
        }
        arr[j] = temp;
      }
      gap = gap / 2;
    }
    foreach (int i in arr)
      Console.Write(i + "  ");
    Console.ReadLine();
  }
}
```

<u>Quick Sort:</u> Like Merge Sort, Quick Sort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of Quick Sort that pick pivot in different ways.

1. Always pick first element as pivot (implemented below).
2. Always pick last element as pivot.
3. Pick a random element as pivot.
4. Pick median as pivot.

The main idea for finding pivot is - the pivot or pivot element is the element of an array, which is selected first to do certain calculations. The key process in Quick Sort is partition. Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x.

```
class QuickSort
{
  static int[] arr;
  public static void Sort(int left, int right)
  {
    int pivot, leftEnd, rightEnd;
    leftEnd = left;
    rightEnd = right;
    pivot = arr[left];
    while (left < right)
    {
      while ((arr[right] >= pivot) && (left < right))
      {
        right--;
      }
      if (left != right)
      {
        arr[left] = arr[right]; left++;
      }
      while ((arr[left] <= pivot) && (left < right))
      {
        left++;
      }
      if (left != right)
      {
        arr[right] = arr[left];
        right--;
      }
    }
    arr[left] = pivot;
    pivot = left;
    left = leftEnd;
    right = rightEnd;
    if(left < pivot)
    {
      Sort(left, pivot - 1);
    }
    if(right > pivot)
    {
      Sort(pivot + 1, right);
    }
  }
  static void Main()
  {
    arr = new int[] { 54, 79, 58, 7, 42, 23, 91, 3, 74, 38, 67, 46, 18, 61, 32, 86, 14 };
    Sort(0, arr.Length - 1);
    foreach (int i in arr)
      Console.Write(i + "  ");
    Console.ReadLine();
  }
}
```