CURRICULUM

AND

WINDOWS FORMS

Index

Introduction to .NET Framework

Microsoft .NET Framework is a programming infrastructure created by Microsoft for building, deploying, and running applications and services that use .NET technologies, such as desktop applications, Web applications and Web services.

.NET is a developer platform made up of tools, programming languages, and libraries for building many different types of applications.

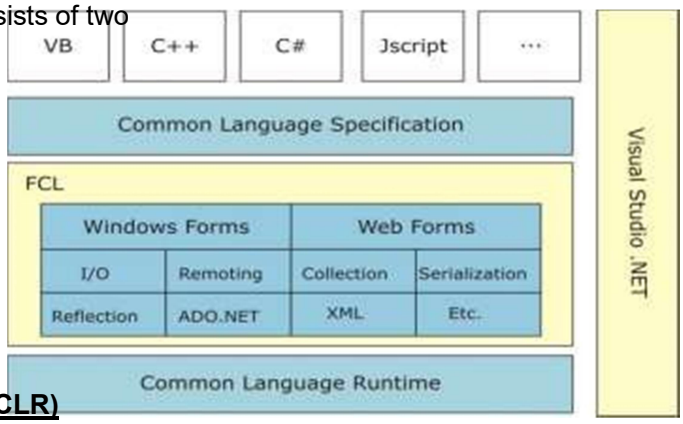
There are various implementations of .NET. Each implementation allows .NET code to execute in different places—Linux, macOS, Windows, iOS, Android, and many more.

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

.NET Standard is a formal specification of the APIs that are common across .NET implementations. This allows the same code and libraries to run on different implementations.

Architecture of .NET Framework

.NET Framework is a managed execution environment for Windows that provides a variety of services to its running apps. It consists of two major components: the **COMMON LANGUAGE RUNTIME** (**CLR**), which is the execution engine that handles running apps, and the **.NET Framework Class Library**, which provides a library of tested, reusable code that developers can call from their own apps. The services that .NET Framework provides to running apps include the following:



1. The **Common Language Runtime (CLR)** is the execution engine that handles running applications. It provides services like thread management, garbage collection, type- safety, exception handling, and more.
2. The **Framework 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.

.NET applications are written in the C#, F#, or Visual Basic programming language. Code is compiled into a language-agnostic Common Intermediate Language (CIL). ***Compiled code is stored in assemblies—files with a .dll or .exe file extension.***

When an app runs, the CLR takes the assembly 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.

The services that .NET Framework provides to running apps include the following:

- **Memory management**: In many programming languages, programmers are responsible for allocating and releasing memory and for handling object lifetimes. In .NET Framework apps, the CLR provides these services on behalf of the app.

- A **Common Type System**: In traditional programming languages, basic types are defined by the compiler, which complicates cross-language interoperability. In .NET Framework, basic types are defined by the .NET Framework type system and are common to all languages that target .NET Framework.

- An **extensive class library**: Instead of having to write vast amounts of code to handle common low-level programming operations, programmers use a readily accessible library of types and their members from the .NET Framework Class Library.

- **Development frameworks and technologies**: .NET Framework includes libraries for specific areas of app development, such as ASP.NET for web apps, ADO.NET for data access, Windows Communication Foundation for service-oriented apps, and Windows Presentation Foundation for Windows desktop apps.

- **Language interoperability**: Language compilers that target .NET Framework emit an intermediate code named Common Intermediate Language (CIL), which, in turn, is compiled at runtime by the common language runtime. With this feature, routines written in one language are accessible to other languages, and programmers focus on creating apps in their preferred languages.

- **Version compatibility**: With rare exceptions, apps that are developed by using a particular version of .NET Framework run without modification on a later version.

- **Side-by-side execution**: .NET Framework helps resolve version conflicts by allowing multiple versions of the common language runtime to exist on the same computer. This means that multiple versions of apps can coexist and that an app can run on the version of .NET Framework with which it was built. Side-by-side execution applies to the .NET Framework version groups 1.0/1.1, 2.0/3.0/3.5, and 4/4.5.x/4.6.x/4.7.x/4.8.

- **Multitargeting**: By targeting .NET Standard, developers create class libraries that work on multiple .NET Framework platforms supported by that version of the standard. For example,

libraries that target .NET Standard 2.0 can be used by apps that target .NET Framework 4.6.1, .NET Core 2.0, and UWP 10.0.16299.

Versions of .NET Framework

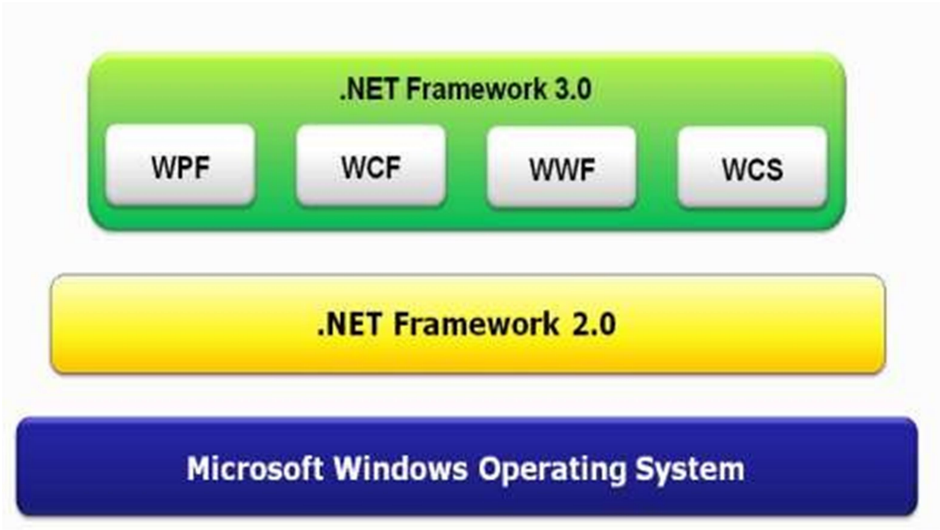
The first version of the .Net framework was released in the year 2002. The version was called .Net framework 1.0. The .Net framework has come a long way since then, and the current version is 4.7.1.
Below is the table of .Net framework versions, which have been released with their release dates. Every version has relevant changes to the framework.
For example, in framework 3.5 and onwards a key framework called the Entity framework was released. This framework is used to change the approach in which the applications are developed while working with databases.

Year	.NET Version	CLR Version	Visual Studio	Released Features and Enhancements
2002	1.0	1.0	Visual Studio .NET 2002	Initial release – Managed code, CLR, CTS, FCL, Use of DLLs as class libraries, OOP support for Web development.
2003	1.1	1.1	Visual Studio .NET 2003	Added ASP.NET Control support for Mobile device development, Added support for ADO.NET classes for Oracle database and ODBC database connectivity. Support for IP6 and fixed issue to Code Access Security for ASP.NET.
2005	2.0	2.0	Visual Studio 2005	New CLR 2.0, Enhancement of ASP.NET & ADO.NET, Generics Types, Partial Types, Anonymous methods, Nullable Types, Iterators, Covariance and Contravariance, and 64bit support we added.
2006	3.0	2.0	Visual Studio 2005	Windows Presentation Foundation (WPF), Windows Communication Foundation (WCF), Windows Workflow Foundation (WWF) introduced.
2007	3.5	2.0	Visual Studio 2008	Build-in-Support for AJAX, Language Integrated Query (LINQ), Expression trees, HashSet collections, WCF and WF integration, Peer-to-Peer networking, Add-ins for extensibility and DateTimeOffset were added.

Year	.NET Version	CLR Version	Visual Studio	Released Features and Enhancements
2010	4.0	4.0	Visual Studio 2010	New CLR 4.0, Task Parallel Library (TPL), Managed Extensibility Framework (MEF), Dynamic Language Runtime (DLR) were a major addition of this release.
2012	4.5	4.0	Visual Studio 2012	Async support, Support for Windows Store, enhancements of WPF, WCF, WF, MEF, and ASP.NET, and base framework classes, such as support for arrays larger than 2GB on 64 -bit platform.
2013	4.5.1	4.0	Visual Studio 2013	Performance and debugging improvements, support for automatic binding redirection, and expanded support for Windows Store application.
2014	4.5.2	4.0	Visual Studio 2013	ASP.NET APIs enhancements, System DPI support for Windows Forms controls, Event Tracing for Windows (ETW) and New Workflow features and Debugging improvements were added in this release.
2015	4.6	4.0	Visual Studio 2015	ASP.NET enhancements, ADO.NET always an encrypted feature for SQL Server 2016, new 64 -bit JIT compiler, Assembly Loader improvements, enhancements to Garbage Collector, etc were added.
2017	4.7	4.0	Visual Studio 2017	High DPI support for Windows Forms controls, Touch support for WPF in Windows 10, Enhanced cryptography support, performance, and reliability improvements.
2019	4.8	4.0	Visual Studio 2019	JIT improvements, Updated ZLib, FIPS improvements, Malware scanning for Assemblies and Accessibility Enhancements

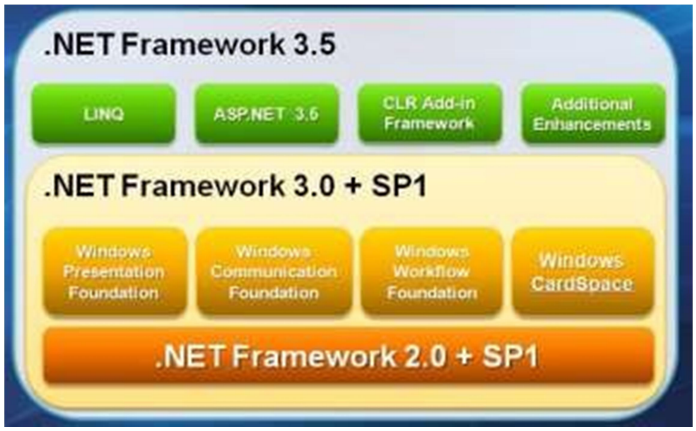
This list presented the NET Framework version history at the time of writing, .NET Framework version is 4.8 and is available for development, it can be downloaded at ***https://dotnet.microsoft.com/download/dotnet-framework*****,** Visual Studio Community Edition can be downloaded at ***https://visualstudio.microsoft.com/downloads/*** here and MSDN documentation at ***https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet-framework***

.NET Framework 3.0 consists of 4 major new components:

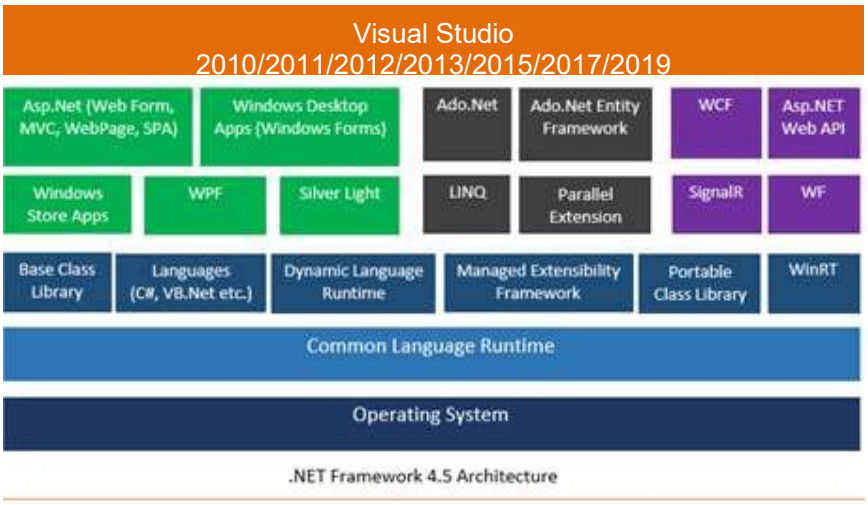


- **Windows Presentation Foundation (WPF), formerly code-named Avalon; a new user interface subsystem and API based on XML and vector graphics, which uses 3D computer graphics hardware and Direct3D technologies.**
- **Windows Communication Foundation (WCF), formerly code-named Indigo; a service- oriented messaging system which allows programs to interoperate locally or remotely similar to web services.**
- **Windows Workflow Foundation (WF) allows for building of task automation and integrated transactions using workflows**

.NET FRAMEWORK 3.5 Provides the first additions to the base class libraries to the .NET Framework since version 2.0



.NET Framework 4 architecture:



About Common Language Runtime (CLR)

CLR is the basic and Virtual Machine component of the .NET Framework.

- It is the run-time environment in the .NET Framework that runs the codes and helps in making the development process easier by providing the various services.
- Basically, it is responsible for managing the execution of .NET programs regardless of any .NET programming language. Internally, CLR implements the ***VES(Virtual Execution System) which is defined in the Microsoft's implementation of the CLI(Common Language Infrastructure).***
- The **code** that runs under the Common Language Runtime is termed as the **Managed Code**. In other words, you can say that CLR provides a managed execution environment for the .NET programs by improving the security, including the cross language integration and a rich set of class libraries etc.

The different version of CLR is discussed in the .NET framework version table earlier.

Role of CLR in code execution:

1. Suppose you have written a C# program and save it in a file which is known as the Source Code.
2. Language specific compiler compiles the source code into the MSIL(Microsoft Intermediate Language) which is also know as the CIL(Common Intermediate Language) or IL(Intermediate Language) along with its metadata. Metadata includes the all the types, actual implementation of each function of the program. MSIL is machine independent code.
3. Now CLR comes into existence. CLR provides the services and runtime environment to the MSIL code. Internally CLR includes the JIT(Just-In-Time) compiler which converts the MSIL code to machine code which further executed by CPU. CLR also uses the .NET Framework class libraries. Metadata provides information about the programming language, environment, version, and class libraries to the CLR by which CLR handles the MSIL code. As

CLR is common so it allows an instance of a class that written in a different language to call a method of the class which written in another language.

As the word specify, Common means CLR provides a common runtime or execution environment as there are more than 60 .NET programming languages.

Main components of CLR:

1. Common Language Specification (CLS)
2. Common Type System (CTS)
3. Garbage Collection (GC)
4. Just In – Time Compiler (JIT)

Common Language Specification (CLS):
It is responsible for converting the different .NET programming language syntactical rules and regulations into CLR understandable format. Basically, it provides the Language Interoperability. Language Interoperability means to provide the execution support to other programming languages also in .NET framework.

Language Interoperability can be achieved in two ways :

- **Managed Code:** The MSIL code which is managed by the CLR is known as the Managed Code. For managed code CLR provides three .NET facilities:
  - CAS(Code Access Security)
  - Exception Handling
  - Automatic Memory Management.
- **Unmanaged Code:** Before .NET development the programming language like .COM Components & Win32 API do not generate the MSIL code. So these are not managed by CLR rather managed by Operating System.

Common Type System (CTS):
Every programming language has its own data type system, so CTS is responsible for understanding all the data type systems of .NET programming languages and converting them into CLR understandable format which will be a common format.
There are 2 Types of CTS that every .NET programming language have :

- **Value Types**: Value Types will store the value directly into the memory location. These types work with stack mechanism only. CLR allows memory for these at Compile Time.
- **Reference Types:** Reference Types will contain a memory address of value because the reference types won't store the variable value directly in memory. These types work with Heap mechanism. CLR allots memory for these at Runtime.

Garbage Collector:
It is used to provide the Automatic Memory Management feature. If there was no garbage collector, programmers would have to write the memory management codes which will be a kind of overhead on programmers.

JIT (Just In Time Compiler):
It is responsible for converting the CIL(Common Intermediate Language) into machine code or native code using the Common Language Runtime environment.

Benefits of CLR:

1. It improves the performance by providing a rich interact between programs at run time.
2. Enhance portability by removing the need of recompiling a program on any operating system that supports it.
3. Security also increases as it analyzes the MSIL instructions whether they are safe or unsafe. Also, the use of delegates in place of function pointers enhances the type safety and security.
4. Support automatic memory management with the help of Garbage Collector.
5. Provides cross-language integration because CTS inside CLR provides a common standard that activates the different languages to extend and share each other's libraries.
6. Provides support to use the components that developed in other .NET programming languages.
7. Provide language, platform, and architecture independence.
8. It allows easy creation of scalable and multithreaded applications, as the developer has no need to think about the memory management and security issues.

Languages from Microsoft compatible with .NET Framework

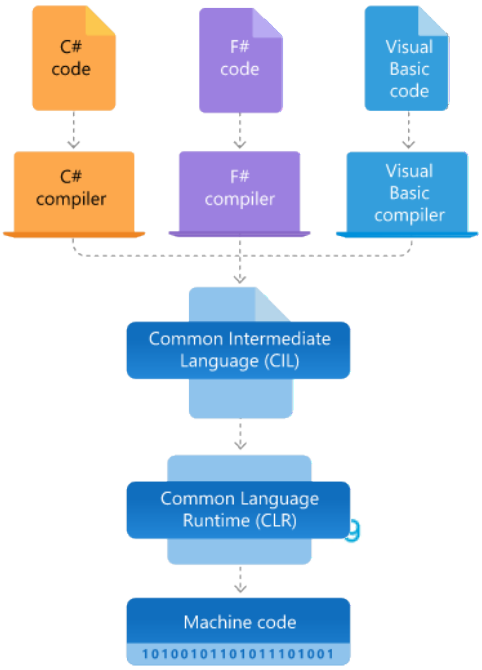
You can write .NET apps in **C#, F#, or Visual Basic**.

- **C#** is a simple, modern, object-oriented, and type-safe programming language. Its roots in the C family of languages makes C# immediately familiar to C, C++, Java, and JavaScript programmers.
- **F#** is a cross-platform, open-source, functional programming language for .NET. It also includes object-oriented and imperative programming.
- **Visual Basic** is an approachable language with a simple syntax for building type-safe, object- oriented apps.

.NET CODE EXECUTION

.Net Framework is a multilanguage execution environment, the runtime supports a wide variety of data types and language features. In order to obtain the full benefits provided by the CLR, you should use respective language (VB.Net, C# etc.) compilers that target the runtime.

(discussed in previous topic about CLR)



Compiling the code to MSIL

Unlike the execution style of compiling source code into machine level code, .Net language compilers translates the source code into **Microsoft Intermediate Language.**

This ensures language interoperability because no matter which language has been used to develop the application, it always gets translated to **Microsoft Intermediate Language**. During the compile time the compiler produces metadata, that contains description of the program like dependencies, versions etc.

Compiling MSIL to native code

Before the program execution, **Just In Time compiler (JIT)** compiles the **MSIL** into native code and stores it in a memory buffer. During JIT compilation, the code is also checked for type safety. Type safety ensures that objects are always accessed in a compatible way. The compiled native code is in memory and is not persisted. So every time we run our application this whole thing has to happen again.

Execution of Code

After translating the IL into native code, it is sent to .Net runtime manager. The .Net runtime manager executes the code. During execution, managed code receives services such as garbage collection, security, interoperability with unmanaged code, cross-language debugging support, and enhanced deployment and versioning support.

.NET Assemblies

- **Microsoft .Net Assembly** is a logical unit of code that contains code which the Common Language Runtime (CLR) executes. It is the smallest unit of deployment of a .net application and it can be a **.dll or an exe** . Assemblies are the building blocks of .NET Framework applications.
- Assembly is really a collection of types and resource information that are built to work together and form a logical unit of functionality. It include both executable application files that you can run directly from Windows without the need for any other programs (.exe files), and libraries (.dll files) for use by other applications.

A .NET assembly can consist of following elements:

a. Assembly Manifest - The Metadata that describes the assembly and its contents
b. Type Metadata - Defines all types, their properties and methods.
c. MSIL - Microsoft intermediate language
d. A set of Resources - All other resources like icons, images etc.

- During the compile time Metadata is created with Microsoft Intermediate Language (MSIL) and stored in a file called **Assembly Manifest** .
- Both Metadata and Microsoft Intermediate Language (MSIL) together wrapped in a Portable Executable (PE) file. Assembly Manifest contains information about itself. This information is called **Assembly Manifest**, it contains information about the members, types, references and all the other data that the runtime needs for execution.
- Every Assembly you create contains one or more program files and a Manifest. There are two types program files : **Process Assemblies (EXE) and Library Assemblies (DLL)**. Each Assembly can have only one entry point (that is, DllMain, WinMain, or Main).

We can create two types of Assembly:

1. **Private Assembly**: A private Assembly is used only by a single application, and usually it is stored in that application's install directory.
2. **Shared Assembly**: It is one that can be referenced by more than one application. If multiple applications need to access an Assembly, we should add the Assembly to the Global Assembly Cache (GAC).
3. **Satellite Assembly** contains only static objects like images and other non-executable files required by the application. It contains .resx files required to translate the UI elements in multiple languages.

.Net Assembly Manifest

An Assembly Manifest is a file that containing Metadata about .NET Assemblies. Assembly Manifest contains a collection of data that describes how the elements in the assembly relate to each other. It describes the relationship and dependencies of the components in the Assembly, versioning information, scope information and the security permissions required by the Assembly.

The Assembly Manifest can be stored in Portable Executable (PE) file with Microsoft Intermediate Language (MSIL) code. You can add or change some information in the Assembly Manifest by using assembly attributes in your code. The Assembly Manifest can be stored in either a PE file (an .exe or
.dll) with Microsoft Intermediate Language (MSIL) code or in a standalone PE file that contains only assembly manifest information. Using **ILDasm**, you can view the manifest information for any managed DLL.

Global Assembly Cache (GAC)

Each computer on which the Common Language Runtime is installed has a machine-wide code cache called the **'Global Assembly Cache'**. GAC is a folder in Windows directory to store the .NET assemblies that are specifically designated to be shared by all applications executed on a system.
Assemblies can be shared among multiple applications on the machine by registering them in global Assembly cache (GAC).

The GAC is automatically installed with the .NET runtime. The global assembly cache is located in 'Windows/WinNT' directory and inherits the directory's access control list that administrators have used to protect the folder.

The approach of having a specially controlled central repository addresses the shared library concept and helps to avoid pitfalls of other solutions that lead to drawbacks like DLL hell.

The Global Assembly Cache Tool (Gacutil.exe), that allows you to view and manipulate the contents of the Global Assembly Cache.

Where is GAC (Global Assembly Cache) located?

GAC is located in C:\Windows\Microsoft.NET\assembly\GAC_MSIL and it is a shared repository of libraries. On a windows machine, if you wish to have a look at what all assemblies are available, go to your windows drive (e.g. C:\>) and underneath your Windows directory, you will find the assembly directory. Further inside the assembly directory you will find the sub-directories named as **GAC, GAC_32, GAC_64 and GAC_MSIL.**

- **GAC_32:** for 32-bit assemblies (defines word size)
- **GAC_64**: for 64-bit assemblies (defines word size)
- **GAC_MSIL**: for assemblies that can be run in either 32-bit or 64-bit mode which is JIT compiled to the required word size as needed.

At the Visual Studio command prompt, type the following command to install in GAC followed by fully qualified path of .NET Assembly:

gacutil -I or gacutil /I

Managed Code vs. Unmanaged Code

- **Managed Code:**
  - Targeted to CLR.
  - Memory managed by Garbage Collector
  - MSIL code.
  - .NET Complaint
- **Unmanaged Code:**
  - Not targeted to CLR
  - Memory not managed by Garbage Collector
  - Operating System Complaint

Microsoft Visual Studio

Microsoft Visual Studio is an integrated development environment (IDE) from Microsoft. It is used to develop computer programs, as well as websites, web apps, web services and mobile apps. Visual Studio uses Microsoft software development platforms such as Windows API, Windows Forms,

Windows Presentation Foundation, Windows Store and Microsoft Silverlight. It can produce both native code and managed code.

- The Design goals are:
  - Maximize Developer Productivity
  - Simplify Server based Development
  - Deliver Powerful Design Tools
- RAD (Rapid Application Development Tool) for the next generation internet
  - Forms Designer
- For creating ASP.NET web form pages and ASP.NET MVC VIEWS
  - Server Explorer
- For accessing web services
- Enhanced RAD support for creating your own components.

Visual Studio includes a code editor supporting **IntelliSense** (the code completion component) as well as code refactoring. The integrated debugger works both as a source-level debugger and a machine-level debugger. Other built-in tools include a code profiler, designer for building GUI applications, web designer, class designer, and database schema designer. It accepts plug-ins that expand the functionality at almost every level—including adding support for source control systems (like Subversion and Git) and adding new toolsets like editors and visual designers for domain- specific languages or toolsets for other aspects of the software development lifecycle (like the Azure DevOps client: Team Explorer).

Visual Studio supports 36 different programming languages and allows the code editor and debugger to support (to varying degrees) nearly any programming language, provided a language-specific service exists. Built-in languages include C, C++, C++/CLI, Visual Basic .NET, C#, F#, JavaScript, TypeScript, XML, XSLT, HTML, and CSS. Support for other languages such as Python, Ruby, Node.js, and M among others is available via plug-ins. Java (and J#) were supported in the past.

The most basic edition of Visual Studio, the Community edition, is available free of charge. The slogan for Visual Studio Community edition is "Free, fully-featured IDE for students, open-source and individual developers".

Visual Studio supports to create the following applications:

- Console Applications
  - Text-Based
  - GUI Based – known as Windows Applications (created using Winforms)
- Web Applications (created using ASP.NET Web Forms and ASP.NET MVC)
- Windows Services
  - Applications that run in the background of the operating system – no GUI.
- Web Services
  - Reusable components on the web – used for Application integration in a SOA.
- Class Libraries
  - Reusable components across .NET applications.
- Windows Presentation Foundation(WPF)
- Windows Communication Foundation(WCF)
- Windows WorkFlow Foundation (WF)

C# Programming Language

C# is a simple & powerful object-oriented programming language developed by Microsoft. C# can be used to create various types of applications, such as web, windows, console applications, or other types of applications using Visual studio.

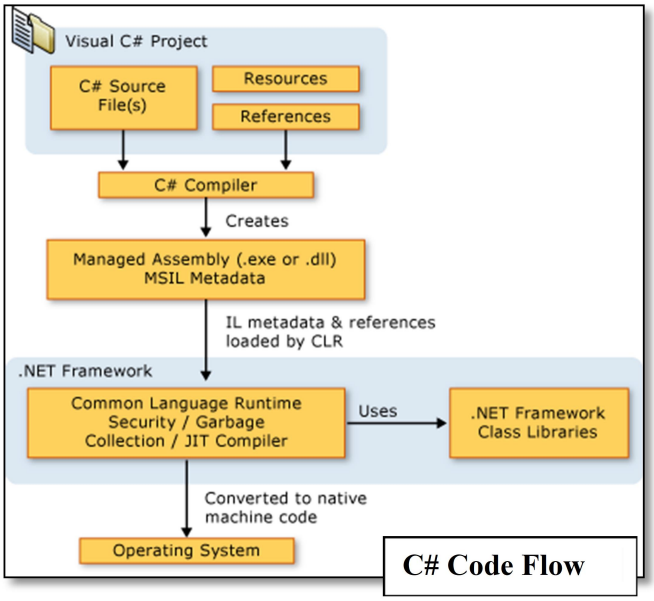
Features:

- ✓ C++ Heritage
  - Namespaces, pointers (in unsafe code), unsigned types, etc.
- ✓ Interoperability
  - C# can talk to COM DLLs and any of the .NET Framework languages.

- ✓ Increased productivity
  - Short learning curve.
- ✓ C# is a type safe object-oriented programming language.
- ✓ C# is case sensitive.

Need of C#:

- Microsoft crafted a new programming language, C# (pronounced “see sharp”), specifically for .NET platform.
- C# is a member of the C family of programming languages (C++, Java etc.) and therefore share a similar syntax.
- C# syntax is highly expressive, yet it is also simple and easy to learn. The curly-brace syntax of C# will be instantly recognizable to anyone familiar with C, C++ or Java. Developers who know any of these languages are typically able to begin to work productively in C# within a very short time.



C# Code Flow

Strength of C#:

- Automatic memory management through garbage collection.
- The ability to build generic types and generic members.
- Support for anonymous methods, which allow you to supply an inline function anywhere a delegate type is required.
- Encapsulated method signatures called delegates, which enable type-safe event notifications.
- Properties, which serve as accessors for private member variables.
- Attributes, which provide declarative metadata about types at run time.

Structure of a typical C# program:

```
using System;
. . .
// A "Hello World!" program in C#
class HelloWorld
{
      static void Main()
      {
```

- Entry point is function.
- **Main()**
- The **using** keyword fers to resources in the
- re Framework class .NET ry.
- libra

```
        Console.WriteLine ("Hello, World");
      }
}
```

Namespaces in .NET:

- Namespace is a collection of types.
- Logical entities.
- Used for grouping related types together.
- Allows compartmentalization and avoids type naming conflicts.
- Types are organized into namespaces.
- Root is System namespace.
- Namespaces can be nested.

C# Version History:

C# was first introduced with .NET Framework 1.0 in the year 2002 and evolved much since then. The following table lists important features introduced in each version of C#:

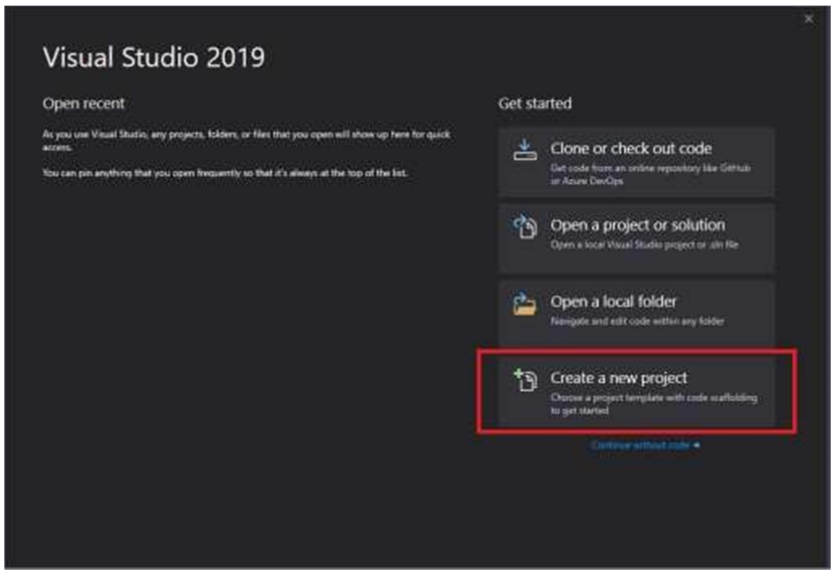
Version	.NET Framework	Visual Studio	Important Features
C# 1.0	.NET Framework 1.0/1.1	Visual Studio .NET 2002	• Basic features
C# 2.0	.NET Framework 2.0	Visual Studio 2005	• Generics • Partial types • Anonymous methods • Iterators • Nullable types • Private setters (properties) • Method group conversions (delegates) • Covariance and Contra-variance • Static classes
C# 3.0	.NET Framework 3.0\3.5	Visual Studio 2008	• Implicitly typed local variables • Object and collection initializers • Auto-Implemented properties • Anonymous types • Extension methods • Query expressions • Lambda expressions • Expression trees • Partial Methods
C# 4.0	.NET Framework 4.0	Visual Studio 2010	• Dynamic binding (late binding) • Named and optional arguments • Generic co- and contravariance • Embedded interop types
C# 5.0	.NET Framework 4.5	Visual Studio 2012/2013	• Async features • Caller information
C# 6.0	.NET Framework 4.6	Visual Studio 2013/2015	• Expression Bodied Methods • Auto-property initializer • nameof Expression • Primary constructor • Await in catch block • Exception Filter • String Interpolation

Version	.NET Framework	Visual Studio	Important Features
C# 8.0	.NET Core 3.0	Visual Studio 2019	• Readonly members • Default interface methods • Using declarations • Static local functions • Disposable ref structs • Nullable reference types

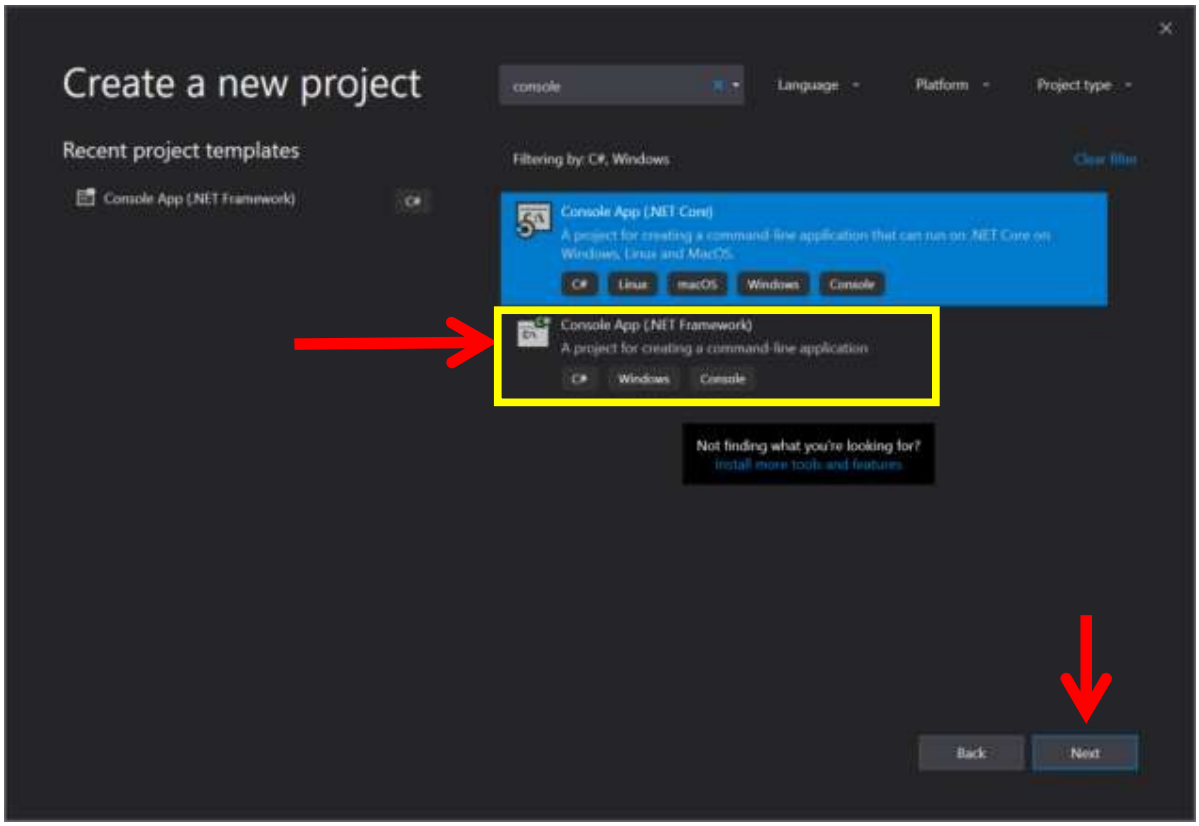
Writing First C# Program:

To start, we'll create a C# application project. The project type comes with all the template files you'll need, before you've even added anything!

1. Open Visual Studio 2019.
2. On the start window, choose **Create a new project**.

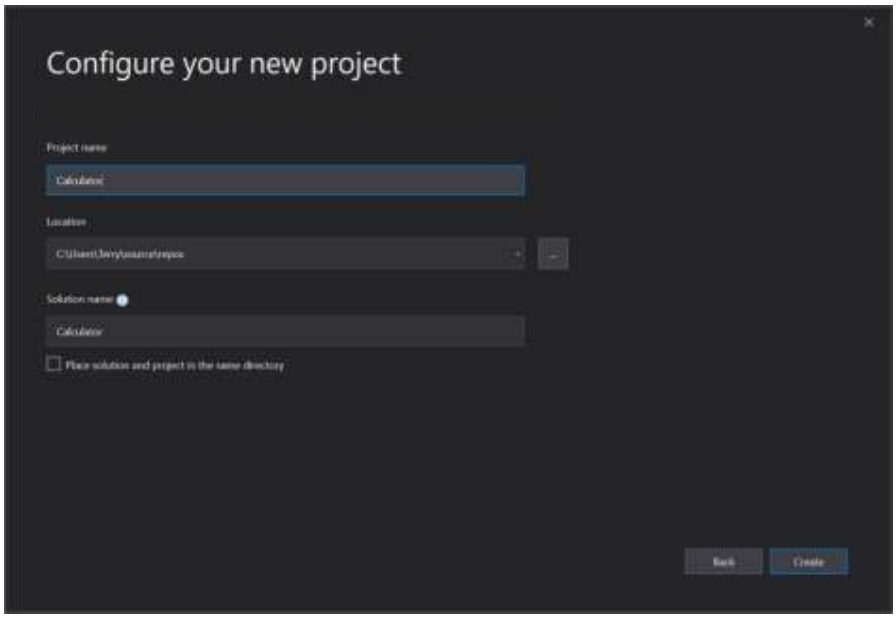


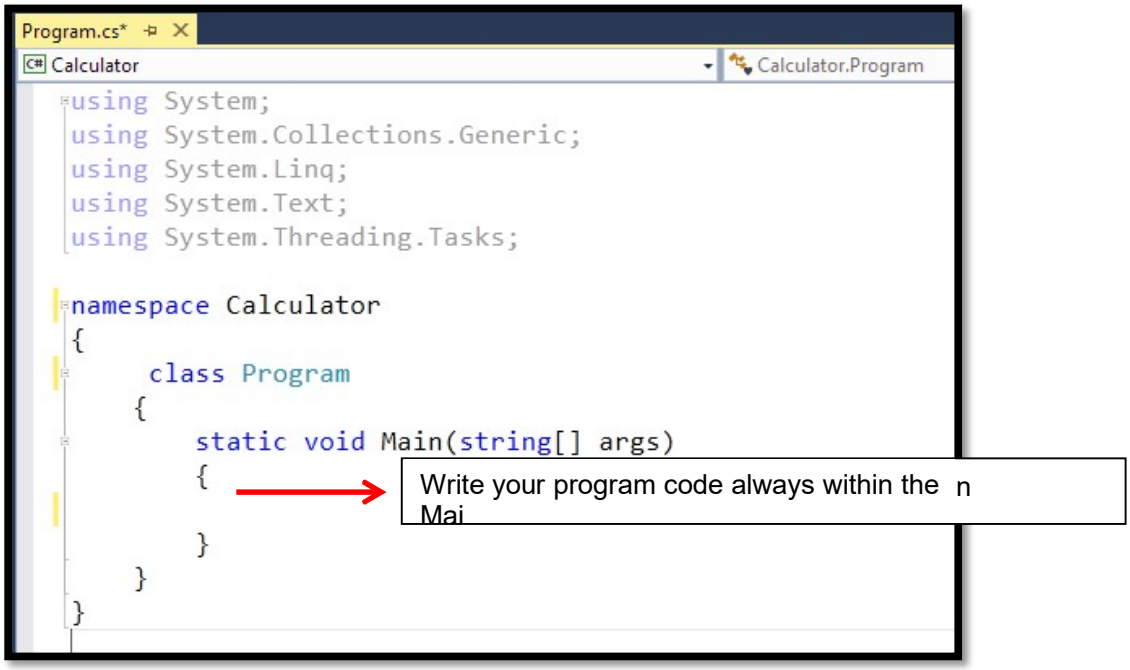
3. On the **Create a new project** window, enter or type *console* in the search box. Next, choose **C#** from the Language list, and then choose **Windows** from the Platform list. After you apply the language and platform filters, choose the **Console App (.NET Framework)** template, and then choose **Next**.



4. In the **Configure your new project** window, type or enter *Calculator* in the **Project name** box and provide the Location where to save the file. Then, choose **Create**.

   Visual Studio opens your new project, which includes default Program class and Main ().





Create the app

First, we'll explore some basic integer math in C#. Then, we'll add code to create a basic calculator. After that, we'll debug the app to find and fix errors. And finally, we'll refine the code to make it more efficient.

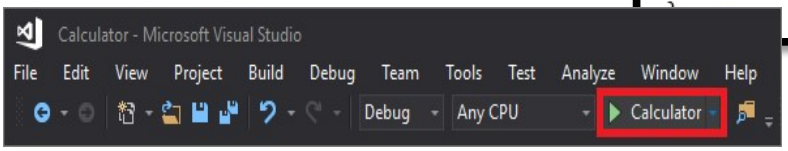
Let's start with some basic integer math in C#.

1. In the code editor, within the Main() write the following code:

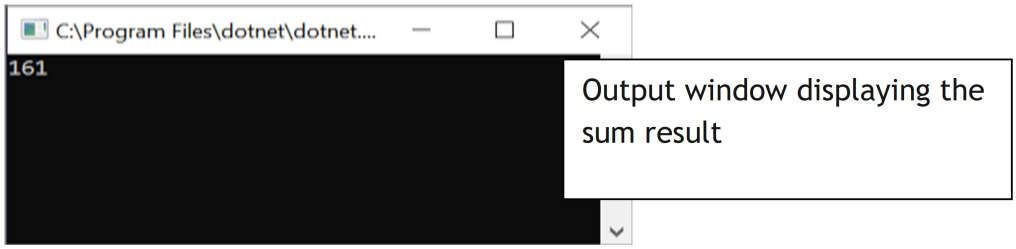
   Notice that when you do so, the IntelliSense feature in Visual Studio offers you the option to autocomplete the entry.

2. Choose the green **Start** button next to **Calculator** to build and run your program, or press **F5**.

```
class Program
{
    static void Main(string[] args)
    {
        int n1 = 42;
        int n2 = 119;

        int s = n1 + n2;

        Console.WriteLine(s);
        Console.ReadLine();

    }
```



A console window opens that reveals the sum of 42 + 119, which is **161**.

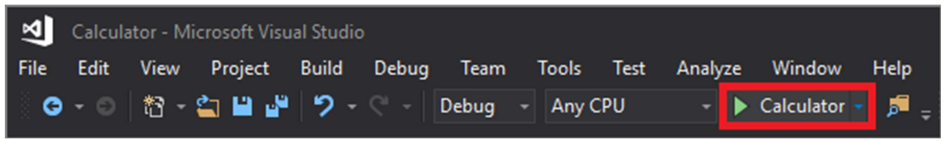


Similarly, instead of providing the value to n1 and n2 we would accept input from the user and display their sum.

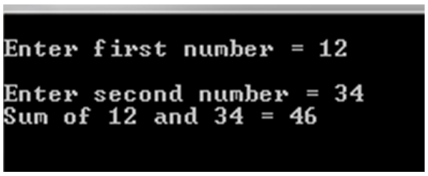
Replace the above code with the following and execute as you did above.

```
int n1, n2, sum;

Console.Write("\nEnter first number = ");
n1 =Convert.ToInt32(Console.ReadLine());
Console.Write("\nEnter second number = ");
n2 = Convert.ToInt32(Console.ReadLine());

sum = n1 + n2;

// Console.WriteLine("Result = " + sum);
//Console.WriteLine("Sum of "
//                  + n1 + " and "
//                  + n2 + " = " + sum);

//formatted output
Console.WriteLine("Sum of {0} and {1} = {2}", n1,n2, sum);
Console.ReadLine();//to hold the screen
```

You can unccmment these statements tcprint the output in different ways



In the output provide the values for first and second number and press enter to see the sum as follows:



Variables and Constants

A variable is a named storage location used to represent data that can be changed while the program is running.

Example :

```
int x=10;
x=20; //here, value of x has changed
```

Constants are values which are known at compile time and do not change. A field is declared as a constant using the **const** keyword.

Example:

```
const int x=10;
x=20; // this is invalid. It will throw compile error
```

Operators in C#

C# supports the following operators:

- Unary operators
- Arithmetic operators
- String operators
- Relational operators
- Conditional operators
- Logical operators
- Assignment operators

Unary Operators

- Unary operators use only one operand.

++	Increment by 1, can be prefix or postfix
--	Decrement by 1, can be prefix or postfix
+	Positive sign
-	Negative sign

Sample code:

```
int num=10;
Console.WriteLine("incrementing/decrementing...");
System.Console.WriteLine(++num);
System.Console.WriteLine(--num);
System.Console.WriteLine(num++);
System.Console.WriteLine(num--);

System.Console.WriteLine("setting signs...");
System.Console.WriteLine(+num);
System.Console.WriteLine(-num);
```

Sample output

```
incrementing/decrementing...
11
10
10
11
setting signs...
10
-10
```

Arithmetic Operators

- Arithmetic operators are used for basic mathematical operations.

Sample code:

```
int num1=15, num2=10;

Console.WriteLine(num1 + num2);
Console.WriteLine(num1 - num2);
Console.WriteLine(num1 * num2);
Console.WriteLine(num1 / num2);
Console.WriteLine(num1 % num2);
```

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulo, remainder

Sample output:

```
calculating...
25
5
150
1
5
```

String Operators

- The string operator (+) is used to concatenate operands.
- If one operand is string, the other operands are converted to string.

Sample code:

```
string fname = "Henry" ;
string lname = "Ford";
string mi = "D";
string fullName = lname + ", " + fname + " " + mi + ".";
string nickName = "Henry";
int age = 21;

Console.WriteLine("My full name is: " + fullName);
Console.WriteLine("You can call me " + nickName + "!");
Console.WriteLine("I'm " + age + " years old.");
```

Sample output:

```
My full name is: Ford, Henry D.
You can call me Henry!
I'm 21 years old.
```

Relational Operators

- Relational operators are used to compare values.
- `boolean` values cannot be compared with non-`boolean` values.
- Only object references are checked for equality, and not their states.
- Objects cannot be compared with `null`.
- `null` is not the same as "".

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equals
!=	Not equals

Relational / Logical Operators

- Logical operators are used to compare `boolean` expressions.
- ! inverts a boolean value.
- `&` , `|` evaluate both operands.
- `&&`, `||` evaluate operands conditionally.
- `&&`, `||` are used to form compound conditions by combining two or more relations.

!	logical NOT
&	bitwise logical AND
|	bitwise logical OR
^	bitwise logical NOT
&&	logical AND
||	logical OR

Truth Table:

Op1	Op2	!Op1	Op1 & Op2	Op1 | Op2	Op1 ^ Op2	Op1 && Op2	Op1 || Op2
false	false	true	false	false	false	false	false
false	true	true	false	true	true	false	true
true	false	false	false	true	true	false	true
true	true	false	true	true	false	true	true

Assignment Operators

- Assignment operators are used to set the value of a variable.

=	Assign
+=	Add and assign
-=	Subtract and assign
*=	Multiply and assign
/=	Divide and assign
%=	Modulo and assign
&=	AND and assign
|=	OR and assign
^=	XOR and assign

Sample code:

```
int a = 12;
a += 4;
Console.WriteLine(" a = " + a);

a -= 3;
Console.WriteLine(" a = " + a);

a *= 2;
Console.WriteLine(" a = " + a);

a /= 5;
Console.WriteLine(" a = " + a);
```

Sample output:

```
a = 16
a = 13
a = 26
a = 5
```

C# Keywords:

- They are an essential part of language definition as they implement specific features of the language.
- They are reserved, and cannot be used as identifiers.

abstract	as	base	bool	break	byte
case	catch	char	checked	class	const
continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false
finally	fixed	float	for	foreach	goto
if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null
object	operator	out	override	params	private
protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	struct
switch	this	throw	true	try	typeof
uint	ulong	unchecked	unsafe	ushort	using
virtual	volatile	void	while	string	

The following keywords are contextual. They can be used as an identifier without an @symbol.

from	get	global	descending	dynamic	equals
join	set	On	orderby	in	into
select	let	Value	group	partial	remove
add	ascending	by	var	where	yield

C# - if, else if, else Statements

C# provides many decision-making statements that help the flow of the C# program based on certain logical conditions.

C# if Statement

The if statement contains a boolean condition followed by a single or multi-line code block to be executed. At runtime, if a boolean condition evaluates to true, then the code block will be executed, otherwise not.

Syntax:

```
if(condition){
    // code block to be executed when if condition evaluates to true
}
```

Example

```
int i = 10, j =
20; if (i < j) {
   Console.WriteLine("{0} is less than {1}", i, j);
}
if (i > j) {
   Console.WriteLine("{0} is greater than {1}", i,j);
}
```

Output:

```
10 is less than 20
```

else if Statement

Multiple else if statements can be used after an if statement. It will only be executed when **the if condition** evaluates to false. So, either if or one of the else if statements can be executed, but not both.

Syntax:

```
if(condition1) { // code block to be executed when if condition1 evaluates to
true } else if(condition2) { // code block to be executed when
                    // condition1 evaluates to false and condition2 evaluates to
true } else if(condition3){     // code block to be executed when condition1
evaluates to false
                     // condition2 evaluates to false
                     // ... condition3 evaluates to true ...}
```

Example:

```
 int n;
Console.Write("Enter a number = ");
 n =
Convert.ToInt32(Console.ReadLine());
if(n%2==0)  {
   Console.WriteLine(n + " is an even number");
}
else   {
   Console.WriteLine(n + " is an odd number");
}
```

switch..case statements

The switch statement can be used instead of if else statement when you want to test a variable against one or more constant values.

Syntax:

```
switch(match expression/variable)
{
  case constant-value:
          statement(s) to be
          executed; break;
  default:
         The default label will be executed if no cases
        executed; break;
}
```

Example:

```
Console.Write("Enter a number = ");
 int n =
 Convert.ToInt32(Console.ReadLine());
 switch (n % 2)
     {
          case 0:
               Console.WriteLine(n + " is an even
               number"); break;
          default:
               Console.WriteLine(n + " is an odd
               number"); break;
     }
```

C# - Ternary Operator ?:

C# includes a decision-making **operator ?:** which is called the conditional operator or ternary operator. It is the short form of the if..else conditions.

```
Syntax: condition ? statement 1 : statement 2
```

The ternary operator starts with a boolean condition. If this condition evaluates to true then it will execute the first statement after ?, otherwise the second statement after : will be executed.

Example:

```
Console.Write("Enter a number = ");
 int n =
Convert.ToInt32(Console.ReadLine());
string result = (n%2==0)? " Even " : " Odd
"
```

Loops

There may be a situation, when you need to execute a block of code several number of times. In general, the statements are executed sequentially.
Looping in a programming language is a way to execute a statement or a set of statements multiple times depending on the result of the condition to be evaluated to execute statements. The result condition should be true to execute statements within loops.
C# supports 4 types of loops:

1. while
2. do..while
3. for
4. for..each

These loops are categorized as follows:

1. **Entry Controlled Loops**: The loops in which condition to be tested is present in beginning of loop body are known as **Entry Controlled Loops**. ***while*** loop and ***for*** loop are entry controlled loops.
    a. **while loop:** The test condition is given in the beginning of the loop and all statements are executed till the given Boolean condition satisfies when the condition becomes false, the control will be out from the while loop. **Syntax:**

```
while(condition) {
   Statements to be executed if the loop condition is true; }
```

Example:

```
public static void Main()
        {
                int x = 1;
                // Exit when x becomes greater than 4
                while (x <= 4)
                {
                        Console.WriteLine("Hello from Nirmaan");
                        // Increment the value of x for next iteration
                        x++;
                }
        }
```

Output: (message will be printed 4 times as shown below)

```
Hello          from
Nirmaan      Hello
from       Nirmaan
Hello          from
```

b. **for loop**: It has similar functionality as while loop but with different syntax. **for *loops are preferred when the number of times loop statements are to be executed is known beforehand***. The loop variable initialization, condition to be tested, and increment/decrement of the loop variable is done in one line in for loop thereby providing a shorter, easy to debug structure of looping.

Syntax:

```
for (loop variable initialization ; testing condition; increment / decrement)
{
   // statements to be executed
}
```

Here, in the syntax :

1. **Initialization of loop variable:** The expression / variable controlling the loop is initialized here. It is the starting point of for loop. An already declared variable can be used or a variable can be declared, local to loop only.
2. **Testing Condition:** The testing condition to execute statements of loop. It is used for testing the exit condition for a loop. It must return a boolean value **true or false**. When the condition became false the control will be out from the loop and for loop ends.
3. **Increment / Decrement:** The loop variable is incremented/decremented according to the requirement and the control then shifts to the testing condition again.

Note: *Initialization part is evaluated only once when the for loop starts.*

Example:

```
public static void Main()
  {
     // for loop begins when x=1
     // and runs till x <=4
     for (int x = 1; x <= 4; x++)
      {
        Console.WriteLine("{0}", i);
      }
  }
```

Output: (it will print numbers from 1 to 4 as follows)

```
1
2
3
4
```

2. **Exit Controlled Loops:** The loops in which the testing condition is present at the end of loop body are termed as **Exit Controlled Loops**. **do-while** is an exit controlled loop.

Note*:* *In Exit Controlled Loops, loop body will be evaluated for at-least one time as the testing condition is present at the end of loop body.*

Syntax:

```
do{
   Statements to execute in the loop;
  }while (condition);
```

Example:

```
public static void Main()
  {
    int x =
    21; do
    {
      // The line will be printed even if the condition is false
      intially Console.WriteLine("Value of x = " + x);
      x++;
    } while (x < 20); // exits the loop as the condition is false now
  }
```

Output: (it will print only once the value of x even if the condition specified is false initially)

```
Value of x = 21
```

3. **Read-Only Collection Elements Loop**: This loop is used to iterate and ***only read*** elements of a collection like an Array, Non-Generic Collections and Generic Collections.
   a. **for..each loop:** The foreach loop is used to iterate over the elements of the collection. The collection may be an array or a list. It executes for each element present in the array.
      i. It is necessary to enclose the statements of for..each loop in curly braces {}.
      ii. Instead of declaring and initializing a loop counter variable, you declare a variable that is the same type as the base type of the array, followed by a colon, which is then followed by the array name.
      iii. In the loop body, you can use the loop variable you created rather than using an indexed array element.

   Syntax:

```
foreach(data_type var_name in collection_variable){
   // statements to be executed
}
```

   Example:

```
static public void Main()
  {
       // Declare and initialize an array
    int[] numbers = new int[] { 1, 2, 3, 4, 5, 6,
    7 }; Console.WriteLine(" Array Elements
    are:");
      // foreach loop begins and it will run till the last element of the array
   //here, n is a local variable of the for..each loop
   //it will hold a single integer value
  //from the array numbers in the loop for each
    iteration foreach(int n in numbers)
   {
       Console.WriteLine(n); //display the current value in n
    }
  }
```

Output:

```
Array Elements
are:
1
2
3
4
5
6
```

Infinite Loops:
The loops in which the test condition does not evaluate false ever tend to execute statements forever until an external force is used to end it and thus they are known as infinite loops.

Example:

```
public static void Main()
  {
     int i=1;
 // statement will be printed infinite times as the value of i is not incremented in the
     loop while(i<=10)
      {
         Console.WriteLine("This is printed infinite times");
      }
  }
```

Output: (statement will be printed till you do not forcefully terminate the program)

```
This is printed infinite times This is printed infinite times This is printed infinite times
```

Nested Loops: When loops are present inside the other loops, it is known as nested loops.

Output:

```
static void Main(string[] args)
{
    for (int i = 1; i <= 5; i++)
    {
        for (int j = 1; j <= i; j++) //  prints the '*'
        {
            Console.Write(" * ");
        }
        Console.WriteLine();
    }

    Console.ReadLine();
}
```

continue statement: is used to skip over the execution part of loop on a certain condition and move the flow to next iteration.

Example:

```
//value of i is printed only 2 times because of continue
  statement for(int i = 1; i < = 3; i++) {
        if(i == 2) {
             continue; //value of i = 2 will not be printed because the iteration was skipped
             when i = 2
         }
    Console.WriteLine("Value of i = " + i);
  }
```

Output:

```
Value of i =
1 Value of i
```

break statement: The break statement in C# has following two usage −

1. When the break statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
2. It can be used to terminate a case in the switch statement.

If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Example(using break in loops) :

```
static void Main()
{
      int a = 10;
      while (a <          // while loop
      20)
          Console.WriteLine("value of a: {0}", a);
          a++;
          if (a > 15) {
               break; // terminate the loop using break
               statement
           }
      }
 }
```

Output: (loop terminates when value of **a** is greater than 15 i.e. when **a = 16**)

```
value of a:
10 value of
a: 11 value
of a: 12
value of a:
13 value of
```

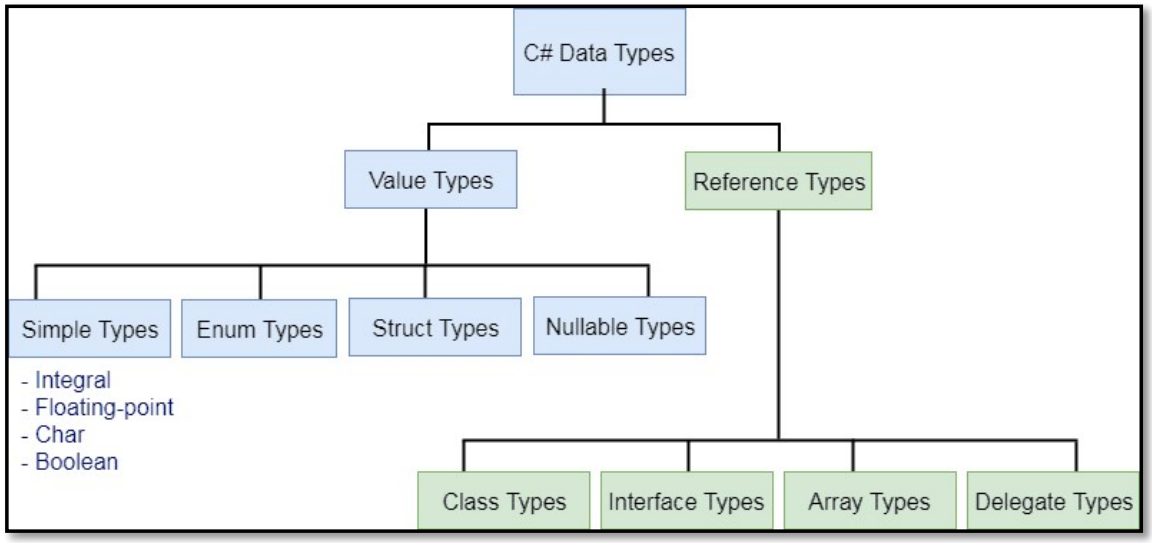
Datatypes in C#

C# is a strongly-typed language. It means we must declare the type of a variable that indicates the kind of values it is going to store, such as integer, float, decimal, text, etc.
Example:

```
string str = "Hello
World!!"; int x = 100;
float f = 10.2f;
char ch = 'A';
bool result=
true;
```

About C# Types:

- A C# program is a collection of types.
  - Classes, structs, enums, interfaces, delegates, etc.
- C# provides a set of predefined (primitive) types.
  - Eg: int, byte, char, bool, double
- Custom types can be created.
- All data and code is defined within a type.
  - No global variables and no global functions
- Types can be instantiated and used by calling methods, get and set properties, etc.
- Can convert from one type to another.
- Types are organized into namespaces and assemblies.
- Types are arranged in a hierarchy.
- There are three categories of types:
  - Value Types
  - Reference Types
  - Pointers



Predefined Data Types in C#

C# includes some predefined value types and reference types. The following table lists predefined data types:

Type	Represents	Range	Default Value
bool	Boolean value	True or False	False
byte	8-bit unsigned integer	0 to 255	0
char	16-bit Unicode character	U +0000 to U +ffff	'\0'
decimal	128-bit precise decimal values with 28-29 significant digits	$(-7.9 \times 10^{28}$ to $7.9 \times 10^{28}) / 10^{0}$ to 28	0.0M
double	64-bit double-precision floating point type	$(+/-)5.0 \times 10^{-324}$ to $(+/-)1.7 \times 10^{308}$	0.0D
float	32-bit single-precision floating point type	-3.4×10^{38} to $+ 3.4 \times 10^{38}$	0.0F
int	32-bit signed integer type	-2,147,483,648 to 2,147,483,647	0
long	64-bit signed integer type	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	0L
sbyte	8-bit signed integer type	-128 to 127	0
short	16-bit signed integer type	-32,768 to 32,767	0
uint	32-bit unsigned integer type	0 to 4,294,967,295	0
ulong	64-bit unsigned integer type	0 to 18,446,744,073,709,551,615	0
ushort	16-bit unsigned integer type	0 to 65,535	0

The predefined data types are alias to their .NET type (CLR class) name.
The following table lists alias for predefined data types and related .NET class name.

Note: struct means it is Value datatype and Class means it is a Reference datatype

Aliases	.NET Type	Type
byte	System.Byte	struct
sbyte	System.SByte	struct
int	System.Int32	struct
uint	System.UInt32	struct
short	System.Int16	struct
ushort	System.UInt16	struct
long	System.Int64	struct
ulong	System.UInt64	struct
float	System.Single	struct
double	System.Double	struct
char	System.Char	struct
bool	System.Boolean	struct
object	System.Object	class
string	System.String	class
decimal	System.Decimal	struct
DateTime	System.DateTime	struct

It means that whether you define a variable of int or Int32, both are the same.

Conversions
The values of certain data types are automatically converted to different data types in C#. This is called an **implicit conversion.**
Example:

```
int i = 345;
double f =
i;
```

In the above example, the value of an integer **variable i** is assigned to the variable of **double type f** because this conversion operation is predefined in

C#. The following is an implicit data type

conversion table.

Sbyte	short, int, long, float, double, decimal
Byte	short, ushort, int, uint, long, ulong, float, double, decimal
Short	int, long, float, double, or decimal
Ushort	int, uint, long, ulong, float, double, or decimal
Int	long, float, double, or decimal.
Uint	long, ulong, float, double, or decimal
Long	float, double, or decimal
Ulong	float, double, or decimal
Char	ushort, int, uint, long, ulong, float, double, or decimal
Float	Double

Conversions from int, uint, long, or ulong to float and from long or ulong to double may cause a loss of precision. No data type implicitly converted to the char type.

However, not all data types are implicitly converted to other data types. For example, int type cannot be converted to uint implicitly. It must be specified **explicitly**.

Implicit conversions

- No information loss
- Occur automatically

Explicit conversions

- Require a cast
- Use the cast operator
- May not succeed
- Information (precision) might be lost

```
int x = 123456;
long y = x;     // implicit
short z = (short)x; //
explicit

double d = 1.2345678901234;
float f = (float)d; // explicit
long l = (long)d; // explicit
```

Value types

- Contains the value directly.
- Amount of memory required for a value type is fixed at compile time and will not change at runtime. This fixed size allows value types to be stored in the area of memory known as the stack.

Reference types

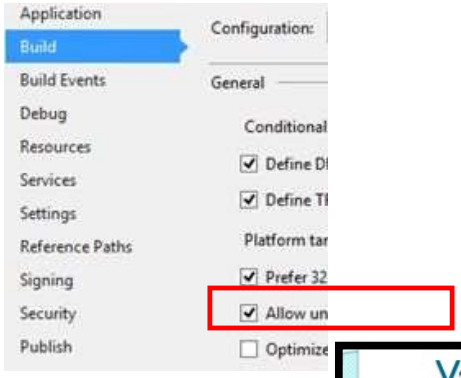
- Stores the reference (memory address) where the data is located.
- The runtime reads the memory location out of the variable and then jumps to the location in the memory that contains the data. The memory area of the data a reference type points to is known as the managed heap.

Pointer Type

- Pointer type variables store the memory address of another type. Pointers in C# have the same capabilities as the pointers in C or C++.
- C# allows using pointer variables in a function of code block when it is marked by the **unsafe** modifier. The **unsafe** code or the **unmanaged** code is a code block that uses a **pointer** variable.
- To maintain type safety and security, **C# does not support pointer** arithmetic, by default. However, using the **unsafe** keyword, you can define an unsafe context in which pointers can be used.
- **Example**:

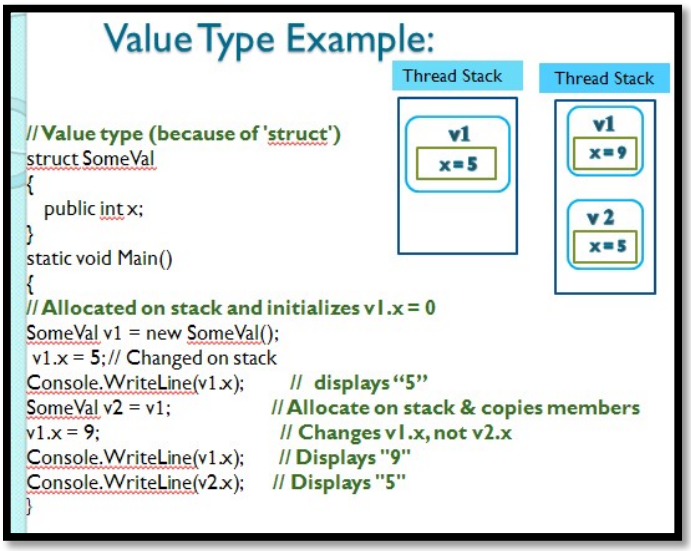
```
unsafe static void Main()
{
    char* ptr =
   value; while (*ptr
   != '\0')
     {

Console.WriteLine(*ptr);
        ++ptr;
     }
```

- **How to run a program in unsafe mode**
  - First go to the View tab.
  - Select the Solution Explorer option.
  - Expand the Solution Explorer a double-click on the Property option.
  - Now select the option of ”Allow unsafe code” and mark it Check.

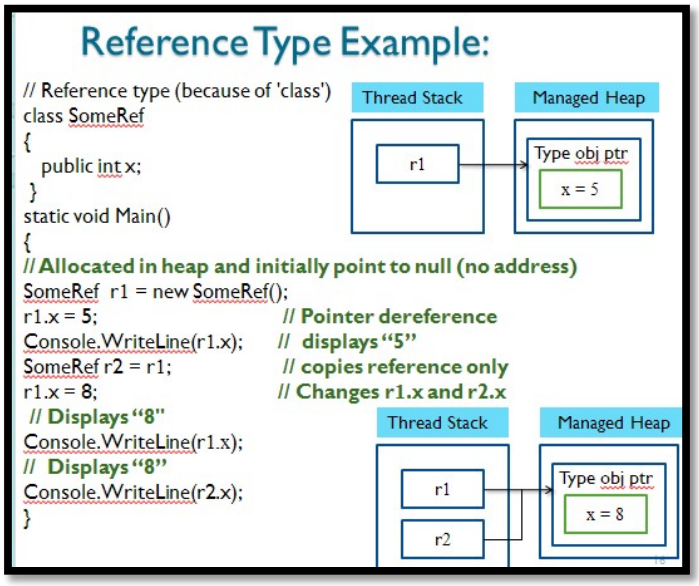


More on Value Types:
Value type variables can be assigned a value directly. They are derived from the class System.ValueType.

The value types directly contain data. Some examples are int, char, and float, which stores numbers, alphabets, and floating point numbers, respectively. When you declare an int type, the system allocates memory to store the value in the stack. In the table specifying all the datatypes in C#, except **string** and **object** all are Value datatypes.



More on Reference Types:
The reference types do not contain the actual data stored in a variable, but they contain a reference to the variables.
In other words, they refer to a memory location. Using multiple variables, the reference types can refer to a memory location. If the data in the memory location is changed by one of the variables, the other variable automatically reflects this change in value. Example of built-in reference types are: **object, dynamic, and string**.



Object Type
The Object Type is the ultimate **base class** for all data types in **C# Common Type System (CTS)**. Object is an alias for **System.Object** class. The object types can be assigned values of any other types, value types, reference types, predefined or user-defined types. However, before assigning values, it needs type conversion.

When a value type is converted to object type, it is called **boxing** and on the other hand, when an object type is converted to a value type, it is called **unboxing**.

- Polymorphic behavior for reference types
  - How does an int (value type) get converted into an object (reference type)?
- Solution: Boxing!
  - Only value types get boxed.
  - Reference types do not get boxed.
- Opposite operation of boxing is **UnBoxing**
  - Copies the value out of the box
  - Copies from reference type to value type
- UnBoxing requires an explicit conversion
  - May not succeed (like all explicit conversions)
  - Essentially a “down cast”

Benefits of Boxing
- Enables polymorphism across all types
- Collection classes work with all types
- Eliminates need for wrapper classes

Disadvantages of Boxing:
- Performance cost
- The Solution : GENERICS

Example:

```
object obj;
obj = 100;          //this is boxing
int x = (int)obj;   //this is
unboxing
```

Dynamic Type:
You can store any type of value in the dynamic data type variable. Type checking for these types of variables takes place at run-time.
Syntax for declaring a dynamic type is −

```
dynamic <variable_name> = value;
```

Example:

```
dynamic d = 20;
```

Dynamic types are similar to object types except that type checking for object type variables takes place at compile time, whereas that for the dynamic type variables takes place at run time.

String Type

The string type allows you to assign any string values to a variable. The string type is an alias for the **System.String** class. It is derived from object type. The value for a string type can be assigned using string literals in two forms: quoted and @quoted.

For example,

```
string str=”Welcome to C# programming";
```

A @quoted string literal (to treat special characters in the string as regular characters) looks as follows :

```
string filepath = @”c:\testfile.txt”;
```

Here, string contains '\' character which is used in C# as Escape character. Hence, if '@' is not prefixed to the string, the compiler throws syntax error. So to treat '\' as regular character use @.

The user-defined reference types are: class, interface, or delegate. These would be discussed in later topics.

Type Casting Operators:

- **is :** Another way to cast in the C# language is to use the is operator. The **is** operator checks whether an object is compatible with a given type, and the result of the evaluation is a Boolean: true or false. The **is** operator will never throw an exception.

  Example:

```
Object o = new Object();
Boolean b1 = (o is Object);     // b1 is true.
Boolean b2 = (o is Employee);
                                // b2 is
false.
```

If the object reference is null, the is operator always returns false because there is no

- **as :**
  - It is used in an expression and used to perform conversions between compatible types.
  - The ***as*** operator works just as casting does except that the *as* operator will never throw an exception.
  - Instead, if the object can't be cast, the result is null.

  Example:

```
Object obj = new Object(); // Creates a new Object
 Employee emp = obj as Employee; // Casts obj to an Employee
// The cast above fails: no exception is thrown, but emp is set to null.
```

Enumerations(enum):

Enumeration (or enum) is a value data type in C#. It is mainly used to assign the names or string values to integral constants that make a program easy to read and maintain.
The main objective of enum is to define our own data types(Enumerated Data Types).

```
enum WeekDays {Mon,Tue,Wed,Thu,Fri,Sat}
WeekDays day = WeekDays.Mon;
Console.WriteLine("{0}", day);          //Displays
Mon Console.WriteLine("{0}",(int)day);
                                        //Displays 0
if(day == WeekDays.Mon){
Console.WriteLine("Hello Monday");
}
```

Enumeration is declared using enum keyword directly inside a namespace, class, or structure.

Points to remember:

- Each member starts from 0 by default and is incremented by 1 for each next member.
- ToString() returns name of the current enumeration's value.
- Cast the enum variable to the underlying storage type to obtain the value.

Initialization of enum: As discussed above, that the default value of first enum member is set to 0 and it increases by 1 for the further data members of enum. However, the user can also change these default value as follows :

```
enum WeekDays {Mon=11,Tue,Wed,Thu,Fri,Sat}
```

here, Mon is assigned 11, so it will implicitly assign the remaining by adding 1.
Which means : **Tue** will be **12** , **Wed** will be **13** and so on. Or else, we can assign custom integers to each as follows:

```
enum WeekDays {Mon=11, Tue=22, Wed=33, Thu=44, Fri=55, Sat=66 }
```

Structures:

- Structure is a **value type** and a collection of variables of different data types under a single unit. It is almost similar to a class because both are user-defined data types and both hold a bunch of different data types.
- Are used for defining a simple composite data type.
- C# provides the ability to use pre-defined data types.
- However, sometimes the user might be in need to define its own data types which are also known as **User-Defined Data Types**.
- Although structures come under the value type, the user can modify it according to requirements and that's why it is also termed as the user-defined data type.
- Members of the structures are by default private.
- Instances of a structure are called as variables.

Defining Structure: In C#, structure is defined using **struct** keyword. Using **struct** keyword one can define the structure consisting of different data types in it. A structure can also contain constructors, constants, fields, methods, properties, indexers and events etc.

Example: Create Book variable of Book structure. Accept and Display the book details.

```
Book b = new Book();
Console.Write("Enter book id =
");
b.bookId =
Convert.ToInt32(Console.ReadLine());
Console.Write("Enter book name = ");
b.bookName = Console.ReadLine();
Console.Write("Enter book price = ");
b.bookPrice = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Book Details:");
Console.WriteLine("Id = " + b.bookId);
Console.WriteLine("Name = " +
b.bookName); Console.WriteLine("Price =
"+b.bookPrice);
```

```
struct Book
  {
    public int bookId;
    public string
    bookName; public int
    bookPrice;
    public BookType bookType;
  }
```

C# Arrays

Like other programming languages, array in C# is a group of similar types of elements that have contiguous memory location. In C#, array is an object of base type **System.Array**. In C#, array index starts from 0. We can store only fixed set of elements in C# array.

The following figure shows how array stores values sequentially :

Explanation :
The index is starting from 0, which stores value. We can also store a fixed number of values in an array. Array index is to be increased by 1 in sequence whenever it's not reach the array size.

Declaring Arrays:
To declare an array in C#, you can use the following syntax −

```
datatype[] arrayName;
```

where,

- datatype is used to specify the type of elements in the array.
- [ ] specifies the rank of the array. The rank specifies the size of the array.
- arrayName specifies the name of the array.

Example :

```
int[] x;                 // can store int values
string[] s;              // can store string values
double[] d;              // can store double values
Student[] stud1;   // can store instances of Student class which is user defined class
```

Note : Only Declaration of an array doesn't allocate memory to the array. For that array must be initialized.

Array Initialization

An array is a reference type so the **new** keyword used to create an instance of the array.
We can assign initialize individual array elements, with the help of the index.

Syntax :

```
datatype[] arrayname = new datatype[size];
```

Example:

```
int[] numbers = new int[10];
```

Here, numbers is an array which can hold maximum 10 integers

Assigning Values to an Array

You can assign values to individual array elements, by using the index number, like −

```
int[] numbers = new int[10];
numbers[0] = 8;
numbers[1]=11;
```

You can assign values to the array at the time of declaration without the **new** keyword, as shown −

```
int[] numbers = {9,4,5,7};
```

You can also create and initialize an array, as shown −

```
int [] marks = new int[5] { 99, 98, 92, 97, 95};
```

You may also omit the size of the array, as shown −

```
int [] marks = new int[] { 99, 98, 92, 97, 95};
```

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array.

Example:

```
static void Main(string[] args)
   {
      int [] n = new int[5]; // n is an array of 5
      integers int i;
      for ( i = 0; i < 5; i++ ) {
         n[ i ] = i + 100;        // initialize elements of array n
      }
```

```
    foreach (int j in n) // iterate each array element's value
    {
      Console.WriteLine(j);
    }
    Console.ReadLine();
  }
```

Output:

```
100
101
102
103
104
```

Single Dimensional Array:

In this array contains only one row for storing the values. All values of this array are stored contiguously starting from 0 to the array size.

Example:

Declaring a single-dimensional array of 5 integers :

```
int[] numbers = new int[5];
```

*The above array contains the elements from numbers[0] to numbers[4]. Here, the **new** operator has to create the array and also initialize its element by their default values. Above example, all elements are initialized by zero, Because it is the int type.*

Multidimensional Arrays

The multi-dimensional array contains more than one row to store the values. It is also known as a **Rectangular Array** in C# because it's each row length is same. It can be a **2D-array or 3D-array** or more. To storing and accessing the values of the array, one required the nested loop.

The multi-dimensional array declaration, initialization and accessing is as follows :

Example :

```
// creates a two-dimensional array of four rows and two
columns. int[, ] n1 = new int[4, 2];

//creates an array of three dimensions, 4, 2, and
3 int[,, ] n2 = new int[4, 2, 3];
```

Example:

```
public static void Main(string[] args)
  {
    int[,] arr=new int[3,3];//declaration of 2D array
    arr[0,1]=10;//initialization
    arr[1,2]=20;
    arr[2,0]=30;
```

```
    Console.WriteLine("Elements stored in the array
      : "); for(int i=0;i<3;i++)
      {
        for(int j=0;j<3;j++){
           Console.Write(arr[i,j]+"
           ");
      }
      Console.WriteLine();//new line at each row
     }
  }
```

Output:

```
0 10 0
0 0 20
30 0 0
```

C# Multidimensional Array Example:

Declaration and initialization at same time

There are 3 ways to initialize multidimensional array in C# while declaration.

```
int[,] arr = new int[3,3]= { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

We can omit the array size.

```
int[,] arr = new int[,]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

We can omit the new operator also.

```
int[,] arr = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

C# Jagged Arrays

In C#, jagged array is also known as "array of arrays" because its elements are arrays.
The element size of jagged array can be different.

Example :

Declare jagged array that has two elements.

```
int[][] arr = new int[2][];
```

C# Jagged Array Example

Here, an example code of jagged array in C# which declares, initializes and traverse jagged arrays.

```
      //creating a jagged array which holds 3 integer arrays each of
      different size int[][] numArray = new int[3][];
      //instantiate each array
    numArray[0] = new int[3]; //allocating an array of size 3 at zeroth index
    numArray[1] = new int[4]; //allocating an array of size 4 at first index
    numArray[2] = new int[2]; //allocating an array of size 2 at second index
      for(int i=0;i<3;i++) {
     //accepting the values and storing in each
       array Console.WriteLine("For Array
       {0}:",(i+1));
```

Output

```
For Array 1:
Enter Number 1 = 4
Enter Number 2 = 5
Enter Number 3 = 6
For Array 2:
Enter Number 1 = 6
Enter Number 2 = 7
Enter Number 3 = 8
Enter Number 4 = 2
For Array 3:
Enter Number 1 = 9
Enter Number 2 = 2
==========================
Array 1 :
4
5
6
Array 2 :
6
7
8
2
Array 3 :
9
2
```

```
        for(int j=0;j<numArray[i].Length; j++)
        {
          Console.Write("Enter Number {0} = ", (j + 1));
          numArray[i][j] =
          Convert.ToInt32(Console.ReadLine());
        }
      }
      int c = 1;
      foreach (int[] array in numArray) {
        Console.WriteLine ("Array {0} : ",
        c); foreach(int n in array) {
         Console.WriteLine(n);  //display elements of the array
         }
         c++
         ;
```

Working with Methods

Methods are generally the block of codes or statements in a program that gives the user the ability to reuse the same code which ultimately saves the excessive use of memory, acts as a time saver and more importantly, it provides a better readability of code. So basically, a method is a collection of statements that perform some specific task and return the result to the caller. A method can also perform some specific task without returning anything.

Syntax of Creating Method:
The following is a general form of a Method declaration.

```
modifiers type method-name (formal-parameter-list)
{
 method_body
}
```

Method declaration consists of five components:

- Methods Modifiers (modifiers).
- Name of the Method (method-name).
- Type of value the Method returns (type).
- List of parameters (formal-parameter-list).
- Body of the Method (method_body).

Types of Method Parameters
C# employs four kinds of parameters that are passed to methods:

- Value Type parameters: (default)
  - Used for passing parameters into methods by *value.*
- Reference Type parameters:
  - Used to pass parameters into methods by *reference.*

- Output parameters:
  - Used to *pass results back* from a method.
- Optional Parameters:
  - Used to pass *optional* parameter.
- Named Argument:
  - Used to pass *argument by position*.
- Parameter arrays:
  - Used in a method definition to enable it *to receive variable number of arguments* when called.

Call By Value (default way to pass parameter to the method)
A value-type variable contains its data directly:

- By passing a value-type variable to a method which passes a copy of the variable to the method.
- Changing the parameter value inside the method does not change the original data stored in the variable.

Example:

```
static void Show(int val) // User defined method
    {
       val++;        // Manipulating value
      Console.WriteLine("Value inside the show function =
      "+val);
      // No return statement
    }
  static void Main(string[] args)
    {
      int val = 50;
       Console.WriteLine("Value before calling the function =
      "+val); Show(val);       // Calling Function by passing
      value Console.WriteLine("Value after calling the function
      = " + val);
```

Output:

```
Value before calling the function 50
Value inside the show function 51
Value after calling the function 50
```

Call By Reference:
C# provides a **ref** keyword to pass argument as reference-type. It passes reference of arguments to the function rather than copy of original value. The changes in passed values are permanent and modify the original variable value.
Example:

```
    static void Swap(ref int x, ref int y) //method to swap 2 given integers
    {
      int t = x;
      x = y;
      y = t;
```

```
        Console.WriteLine("In Swap");
        Console.WriteLine("a = " + x + " b = "
        + y);
    }
    static void Main()
    {
        int a=10, b=45;
        Console.WriteLine("Before Swap");
        Console.WriteLine("a = " + a + " b = "
        + b);
         Swap(ref a, ref b);          //call Swap method by
passing reference a and b
        Console.WriteLine("After Swap");
        Console.WriteLine("a = " + a + " b = " + b);
        Console.WriteLine("b = " + b);
```

Output

```
Before Swap
a = 10
b = 45
In Swap
a = 45
b = 10
After Swap
a = 45
b = 10
```

C# Out Parameter:

C# provides **out** keyword to pass arguments as out-type. It is like reference-type, except that it does not require variable to initialize before passing. We must use **out** keyword to pass argument as out- type. It is useful when we want a function to return multiple values.

Example:

```
static Show(out int val) // Out parameter
    {
        int square =
        5; val =
        square;
        val *= val; // Manipulating value
    }
    static void Main(string[] args)
    {
        int val = 50;
        Console.WriteLine("Value before passing out variable " +
        val); Show(out val); // Passing out argument
        Console.WriteLine("Value after recieving the out variable "
        + val);
       Console.ReadLine();
```

Output:

```
Value before passing out variable 50
Value after receiving the out variable
```

Default or Optional Parameters

As the name suggests **optional parameters** are not compulsory parameters, they are optional. It helps to exclude arguments for some parameters. Or we can say in optional parameters, it is not necessary to pass all the parameters in the method. This concept is introduced in C# 4.0. Here, each and every optional parameter contains a default value which is the part of its definition. If we do not pass any arguments to the optional parameters, then it takes its default value. The optional

parameters are always defined at the end of the parameter list. Or in other words, the last parameter of the method, constructor, etc. is the optional parameter.

Features:

- Methods can declare optional parameters.
- A parameter is optional if it is specified as a default value while declaring.
- Optional parameter can be omitted while calling the method.
- Optional parameter cannot be marked with ref or out keywords.
- Mandatory parameters occur before optional parameters in method declaration and method calling.

Named Parameters:

C# 4.0 introduces named and optional arguments. Named arguments enable you to specify an argument for a particular parameter by associating the argument with the parameter's name rather than with the parameter's position in the parameter list.

Features:

- Identifying an argument by name.
- Named arguments can occur in any order.
- Can mix named and positional parameters.
- Positional parameter must come before named arguments.

Example: (using Optional and Named Parameters)

```
static double ComputeSI(double p, double r=10, double n=3)
{
    double s;
    s = (p * n * r) / 100;
    return s;
}
static void Main()
{
    Console.WriteLine("Simple Interest = " + ComputeSI(10000));
    Console.WriteLine("Simple Interest = " +ComputeSI(50000,15));
    Console.WriteLine("Simple Interest =" + ComputeSI(12000,5,2));
    Console.WriteLine("Simple Interest =" + ComputeSI(n:4,p:15000,r:12));
    Console.WriteLine("Simple Interest =" + ComputeSI(25000,n:6, r: 8));
    Console.WriteLine("Simple Interest =" + ComputeSI(n: 4, p: 15000));//rate value is default
    Console.ReadLine();
}
```

Params

It is useful when the programmer doesn't have any prior knowledge about the number of parameters to be used.

- By using **params** keyword you are allowed to pass any variable number of arguments.

```
static double AddNos(params int[] numbers)
{
    double s = 0;
    foreach (int i in numbers)
    {
        s = s + i;
    }
    return s;
}
static void Main()
{
    Console.WriteLine("sum of 2 ints = " + AddNos(7, 8));
    Console.WriteLine("sum of 5 ints = " + AddNos(8, 5, 7, 3, 2));
    int[] n = new int[] { 4, 5, 6 };
    Console.WriteLine("sum of elements of array = " + AddNos(n));
    Console.ReadLine();
}
```

- Only one **params** keyword is allowed and no additional **Params** will be allowed in method declaration after a **params** keyword.
- The length of **params** will be zero if no arguments will be passed.

OBJECT ORIENTED PROGRAMMING CONCEPTS

The Object Oriented Programming (OOPs) in C# is a design approach where we think in terms of real-world objects rather than functions or methods. Unlike procedural programming language, here in OOPs, programs are organized around objects and data rather than action and logic.

Since C# is an object-oriented language, program is designed using objects and classes in C#. C# provides full support for object-oriented programming including abstraction, encapsulation, inheritance, and polymorphism.

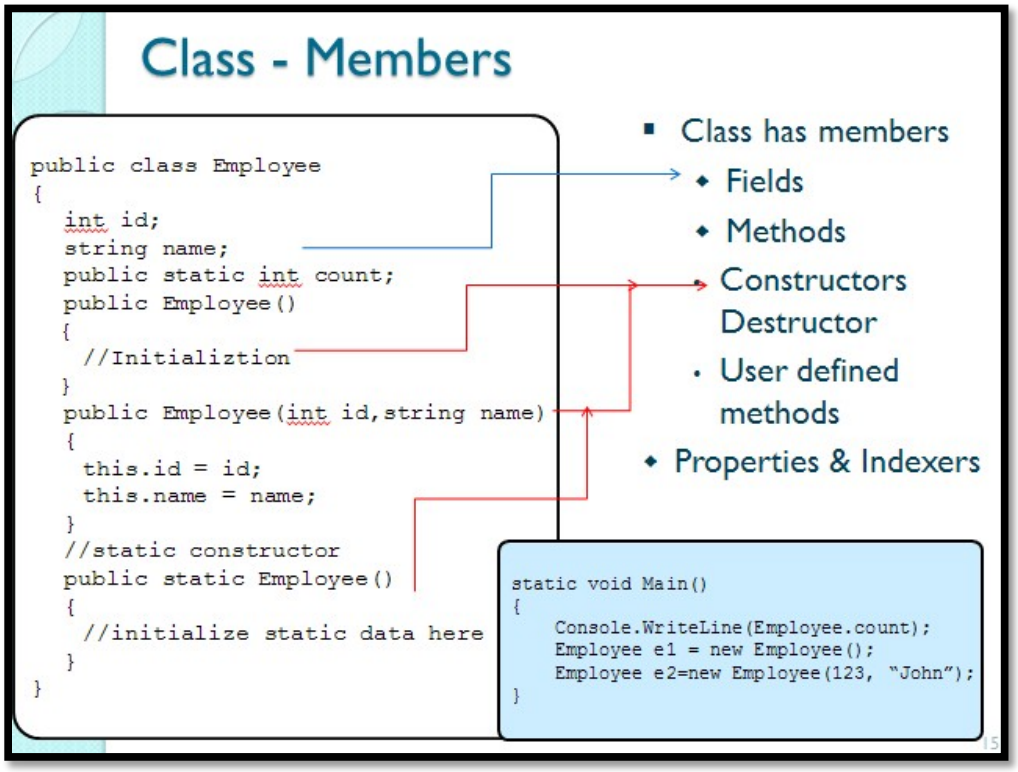
1. **Abstraction** means hiding the unnecessary details from type consumers.
2. **Encapsulation** means that a group of related properties, methods, and other members are treated as a single unit or object.
3. **Inheritance** describes the ability to create new classes based on an existing class.
4. **Polymorphism** means that you can have multiple classes that can be used interchangeably, even though each class implements the same properties or methods in different ways.

Class and **Object** are the basic concepts of **Object-Oriented Programming** which revolve around the real-life entities. A class is a user-defined blueprint or prototype from which objects are created.
Basically, a class combines the fields and methods (member function which defines actions) into a single unit. In C#, classes support polymorphism, inheritance and also provide the concept of derived classes and base classes.

Declaration of class

Generally, a class declaration contains only keyword class, followed by an identifier(name) of the class. But there are some optional attributes that can be used with class declaration according to the application requirement. In general, class declarations can include these components, in order:

- **Modifiers**: A class can be public or internal etc. By default modifier of class is internal.
- **Keyword class**: A **class** keyword is used to declare the type class.
- **Class Identifier**: The variable of type class is provided. The identifier (or name of class) should begin with a initial letter which should be capitalized by convention.
- Base class or Super class: The name of the class's parent (superclass), if any, preceded by the ***: (colon)***. This is optional.
- **Interfaces**: A comma-separated list of interfaces implemented by the class, if any, preceded by the : (colon). A class can implement more than one interface. This is optional.
- **Body**: The class body is surrounded by { } (curly braces).
- **Constructors** in class are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

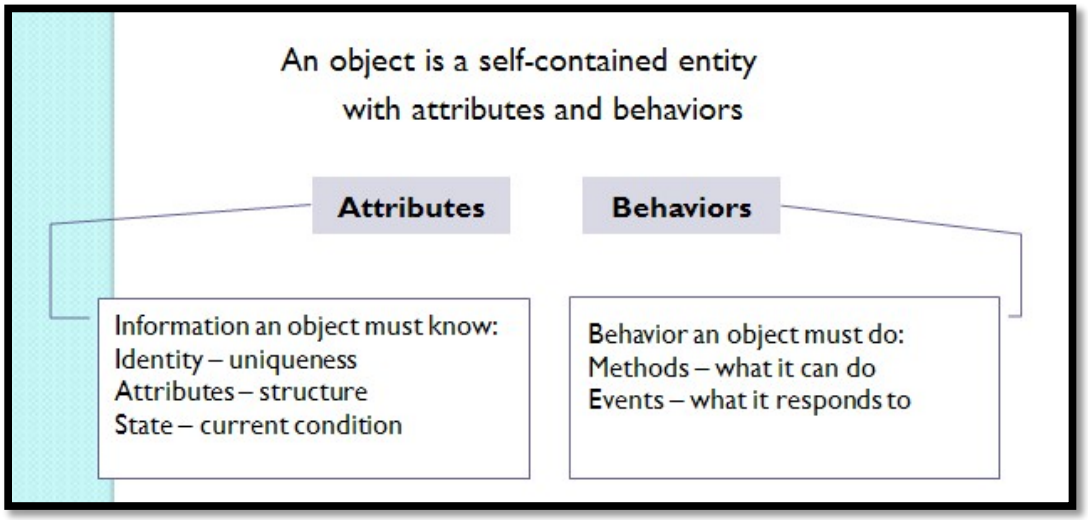


Objects

It is a basic unit of Object-Oriented Programming and represents the real-life entities. A typical C# program creates many objects, which as you know, interact by invoking methods.

An object consists of :

- **State**: It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior**: It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity**: It gives a unique name to an object and enables one object to interact with other objects.



For Example, Consider a Bank Account

Attributes	State	Behavior	Identity	Responsibility
Balance	Balance = 7000	Open Balance Withdraw Report	Account Number = 4141	Keeps a track of a stores money with the facility of deposits and withdraw
Interest Rate	Interest Rate = 6%			
Account number	Account Number= 4141			

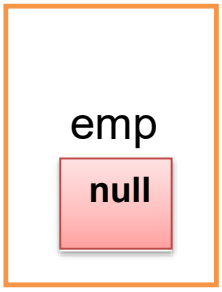
Objects correspond to things found in the real world. For example, a graphics program may have objects such as “circle”, “square”, “menu”. An online shopping system might have objects such as “shopping cart”, “customer”, and “product”.

Declaring Objects (Also called instantiating a class)

When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances. Consider the Employee class created in the picture given above (see topic **Declaraton of Class**). Here, we will declare an object of this Employee class as follows:

```
Employee emp;
```

Here, we declare '**emp'** as an Object of **Employee** class, but did not assign any value to it. Which means it will hold '**null'** value now until an object is instantiated (or memory is allocated) in the heap to hold values for **emp** object.
Note: Simply by declaring an object does not mean the reference is created. So now the memory allocation in the Stack looks like this:



Initializing (Instantiate) an object
The **new** operator instantiates a class by allocating memory for holding values of a new object in the heap and returning a reference of that memory and assign it to the new object. The new operator also invokes the class constructor.

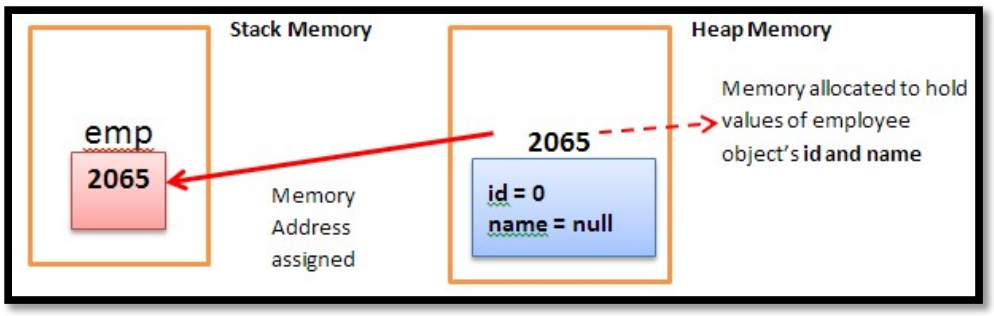
Consider the Employee class created in the picture given above (see topic **Declaraton of Class**). Here, we will declare an object of this Employee class and instantiate as follows:

```
Employee emp = new Employee();
```

Or, can be specified as follows also

```
Employee emp;
emp = new Employee();
```

So, now the memory is allocated in the heap to hold the values of employee object and its reference is assigned to **emp** object in the stack. (Since, Employee is a class and its reference type. Refer the topic Reference type)



Understanding Features of Object Oriented Programming

Abstraction

- Abstraction is "To represent the essential feature without representing the background details."
- Abstraction lets you focus on what the object does instead of how it does it.
- Abstraction provides you a generalized view of your classes or objects by providing relevant information.

- Abstraction is the process of hiding the working style of an object, and showing the information of an object in an understandable manner.

Example: Consider a real-life scenario of withdrawing money from ATM. The user only knows that in ATM machine first enter ATM card, then enter the pin code of ATM card, and then enter the amount which he/she wants to withdraw and at last, he/she gets their money. The user does not know about the inner mechanism of the ATM or the implementation of withdrawing money etc. The user just simply know how to operate the ATM machine, this is called abstraction.

Encapsulation

- Wrapping up a data member and a method together into a single unit (in other words class) is called Encapsulation.
- Encapsulation is like enclosing in a capsule. That is enclosing the related operations and data related to an object into that object.
- Encapsulation is like your bag in which you can keep your pen, book etcetera. It means this is the property of encapsulating members and functions.

```
class
Bag {
book;
pen;
ReadBook();
}
```

- Encapsulation means hiding the internal details of an object, in other words how an object does something.
- Encapsulation prevents clients from seeing it's inside view, where the behaviour of the abstraction is implemented.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as **data-hiding**.
- Encapsulation can be achieved by: Declaring all the variables in the class as private and using C# Properties in the class to set and get the values of variables.

Example: TV operation
It is encapsulated with a cover and we can operate it with a remote and there is no need to open the TV to change the channel. Here everything is private except the remote, so that anyone can access the remote to operate and change the things in the TV.

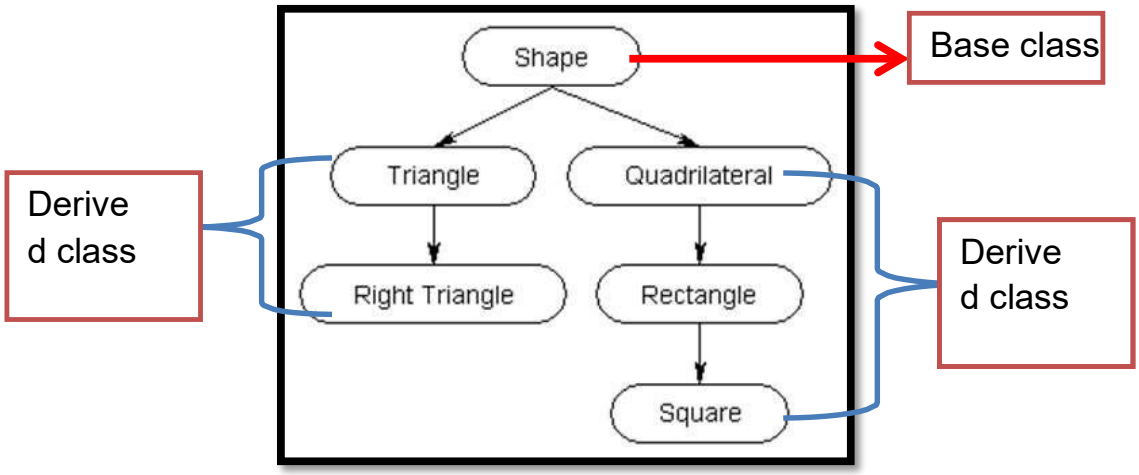
Inheritance

- When a class includes a property of another class it is known as inheritance.
- Inheritance is a process of object reusability.
- Inheritance is an important pillar of OOP(Object Oriented Programming).
- It is the mechanism in C# by which one class is allowed to inherit the features(fields and methods) of another class.

Important terminology used in Inheritance:

- **Base (Super) Class**: The class whose features are inherited is known as **Base** class (or a parent class).
- **Derived (Sub) Class**: The class that inherits the other class is known as **Derived** class (or extended class, or child class). The derived class can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability**: Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example:



Note: Inheritance is discussed in detail in the later topic- click here for reference

Polymorphism

- Polymorphism means one name, many forms.
- One function behaves in different forms.
- In other words, "Many forms of a single object is called Polymorphism."
- Polymorphism can be achieved using **Overloading** and **Overriding**

Example: Your mobile phone, one name but many forms:

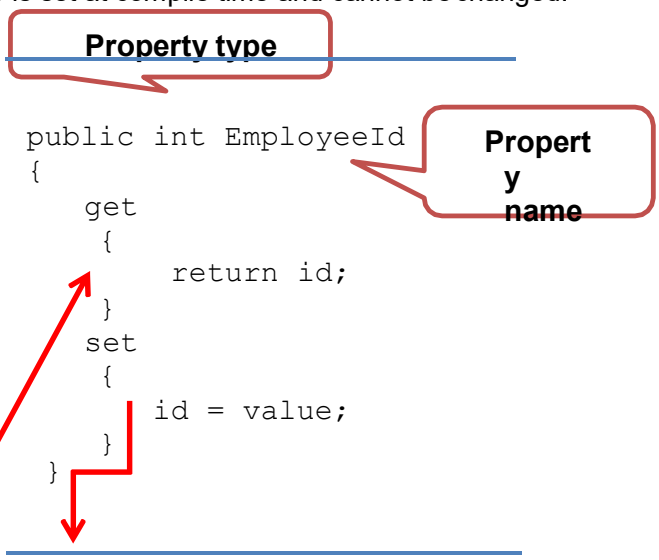
- ✓ **As phone**
- ✓ **As camera**
- ✓ **As mp3 player**
- ✓ **As radio**

Note: Polymorphism is discussed in detail in the later topic

Understanding Class members in detail

Classes and **structs** have members that represent their data and behavior. A class's members include all the members declared in the class, along with all members (except constructors and finalizers) declared in all classes in its inheritance hierarchy. Private members in base classes are inherited but are not accessible from derived classes.

1. **Fields** : Fields are variables declared at class scope. A field may be a built-in numeric type or an instance of another class. For example, a **BankAccount** class may have a field that contains the **AccountHolderName**.
2. **Constants:** Constants are fields whose value is set at compile time and cannot be changed.
3. **Properties:**
   - Properties are methods on a class that are accessed as if they were fields on that class.
   - A property can provide protection for a class field to keep it from being changed without the knowledge of the object.
   - Properties are known as smart fields.
   - Use special methods to assign and retrieve values from the underlying data member.
   - Have accessor and mutator methods:
     - **get** - retrieves data member values.
     - **set** - enables data members to be assigned.



4. **Methods**: Methods define the actions that a class can perform. Methods can take parameters that provide input data, and can return output data through parameters. Methods can also return a value directly, without using a parameter. (discussed in detail in topic Working with Methods)
5. **Events:** Events provide notifications about occurrences, such as button clicks or the successful completion of a method, to other objects. Events are defined and triggered by using delegates. **(will be discussed in later topic)**
6. **Operators:** Overloaded operators are considered type members. When you overload an operator, you define it as a public static method in a type. **(will be discussed in later topic)**
7. **Indexers**: Indexers enable an object to be indexed in a manner similar to arrays. **(will be discussed in later topic)**
8. **Constructors**: Constructors are methods that are called when the object is first created.
   - Implicit method which gets invoked during object creation.
   - Member initialization is feature of constructor
   - More than one constructors could be written
   - Has same name as that of the class
   - Doesn't have return type

Important points to remember about Constructors

- ✓ Constructor of a class must have the same name as the class name in which it resides.
- ✓ A constructor cannot be abstract, final, static and Synchronized.
- ✓ Within a class, you can create only one static constructor.
- ✓ A constructor doesn't have any return type, not even void.
- ✓ A static constructor cannot be a parameterized constructor.
- ✓ A class can have any number of constructors.
- ✓ Access modifiers can be used in constructor declaration to control its access i.e which other class can call the constructor.

Types of Constructor (click here for Examples)

a) **Default Constructor**: A constructor with no parameters is called a default constructor. A default constructor has every instance of the class to be initialized to the same values. The default constructor initializes all numeric fields to zero and all string and object fields to null inside a class.

b) **Parameterized Constructor**: A constructor having at least one parameter is called as parameterized constructor. It can initialize each instance of the class to different values.

c) **Copy Constructor:** This constructor creates an object by copying variables from another object. Its main use is to initialize a new instance to the values of an existing instance.

d) **Private Constructor**: If a constructor is created with private specifier is known as Private Constructor. It is not possible for other classes to derive from this class and also it's not possible to create an instance of this class.

Points To Remember :

- It is the implementation of a singleton class pattern.
- use private constructor when we have only static members.
- Using private constructor, prevents the creation of the instances of that class.

e) **Static Constructor**: Static Constructor has to be invoked only once in the class and it has been invoked during the creation of the first reference to a static member in the class. A static constructor is initialized static fields or data of the class and to be executed only once.

Points To Remember :

- It can't be called directly.
- When it is executing then the user has no control.
- It does not take access modifiers or any parameters.
- It is called automatically to initialize the class before the first instance created.

9. **Finalizers**: Finalizersare used very rarely in C#. They are methods that are called by the runtime execution engine when the object is about to be removed from memory. They are generally used to make sure that any resources which must be released are handled appropriately. Finalizers (which are also called destructors) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector.

Points to Remember:

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

Example:

```
class Car
{
   ~Car() // finalizer for Car class
     {
      // cleanup statements...
     }
}
```

The finalizer implicitly calls Finalize on the base class of the object. Therefore, a call to a finalizer is implicitly translated to the following code:

```
protected override void Finalize()
{
   try
   {
      // Cleanup statements...
   }
   finally
   {
      base.Finalize();
   }
}
```

This design means that the Finalize method is called recursively for all instances in the inheritance chain, from the most-derived to the least-derived.

Points to remember:

- The programmer has no control over when the finalizer is called; the garbage collector decides when to call it. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for finalization, it calls the finalizer (if any) and reclaims the memory used to store the object.
- In .NET Framework applications (but not in .NET Core applications), finalizers are also called when the program exits.
- It's possible to force garbage collection by calling Collect, but most of the time, this call should be avoided because it may create performance issues.

Using finalizers to release resources

In general, C# does not require as much memory management on the part of the developer as languages that don't target a runtime with garbage collection. This is because the .NET garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources, such as windows, files, and network

connections, you should use finalizers to free those resources. When the object is eligible for finalization, the garbage collector runs the Finalize method of the object.

Explicit release of resources
If your application is using an expensive external resource, we also recommend that you provide a way to explicitly release the resource before the garbage collector frees the object. To release the resource, implement a Dispose method from the IDisposable interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the finalizer becomes a safeguard to clean up resources if the call to the Dispose method fails. **(topic discussed in detail later)**

10. Nested Types: A type defined within a class, struct, or interface is called a nested type. For example:

```
class DemoA      //access specifier here is
{
  class DemoB     //access specifier here is
  { }
```

Regardless of whether the outer type is a class, interface, or struct, nested types default to private; they are accessible only from their containing type. In the previous example, the Nested class is inaccessible to external types.
You can also specify an access modifier to define the accessibility of a nested type, as follows:

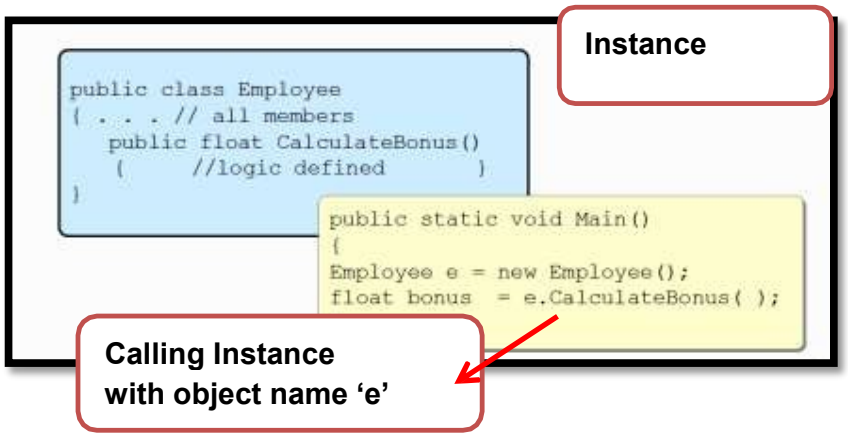
- Nested types of a class can be public, protected, internal, protected internal, private or private protected.
- However, defining a protected, protected internal or private protected nested class inside a sealed class generates compiler warning CS0628, "new protected member declared in sealed class."
- Nested types of a struct can be public, internal, or private.

Static and Non-Static Class Members

Members of a class are either **Static members or Instance (Non-Static)** members. Generally speaking, it is useful to think of static members as belonging to class types and instance(Non-Static) members as belonging to objects (instances of class types).

About Non- Static Members

- Also known as Instance members
- These members are accessible only by object references
- They **do not have static keyword** prefixed to its definition.



About Static Members

static is a modifier in C# which is applicable for the following:

- Classes
- Variables
- Methods
- Constructor

It is also applicable to properties, event, and operators. To create a static member(class, variable, methods, constructor), precede its declaration with the keyword static**. When a member is declared static, it can be accessed with the name of its class directly.**

Static Variable or Fields:

- A static variable is declared with the help of **static** keyword.
- When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level.
- Static variables are accessed with the name of the class, they do not require any object for access.

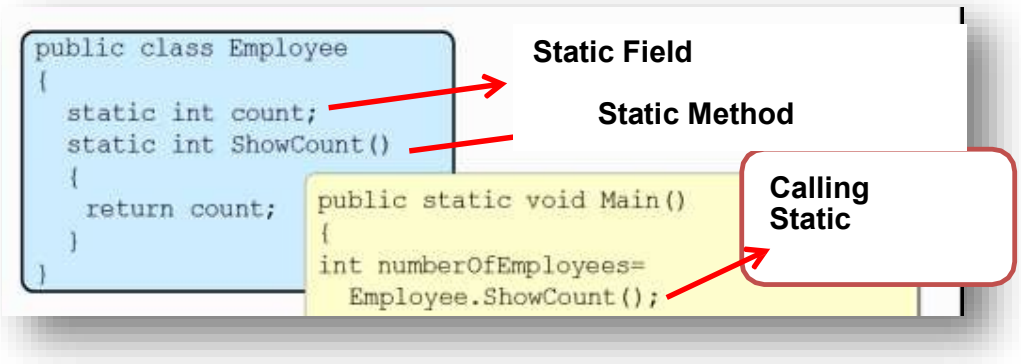
Limitation of using static keyword:

- **static** keyword **cannot** be used by indexers, finalizers, or types other than classes.
- A static member is not referenced through an instance.
- In C#, it is not allowed to use this to reference static methods or property accessors.
- In C#, if static keyword is used with the class, then the static class always contains static members.

Static Method:

A static method is declared with the help of static keyword. Static methods are accessed with the name of the class. A static method can access static and non-static fields, static fields are directly accessed by the static method without class name whereas non-static fields require objects.

Example using Static Field and Static Method:



Constructor Examples (Refer topic Constructor Types)

1. Default Constructor: is an Instance Constructor

```
class User
  {
    string name, location;
    // Default Constructor called every time when an object is created without
    parameters public User()
    {
      name = "Pravin";
      location =
      "Hyderabad";
    }
  }
```

2. Parameterized Constructor: is an Instance Constructor

```
class User
  {
    string name, location;
    // Parameterized Constructor called every time object is created by passing
    parameters public User(string n, string loc)
    {
      name = n;
      location =
      loc;
    }
  }
```

3. Static Constructor: is not an Instance Constructor

```
class User
  {
    string name,
   location; static int
   count;
    // static Constructor called only once when the class is accessed for
    first time. static User() //static constructors are always private
    {
      count++;
    }
  }
```

4. Private Constructor: A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class.

```
class User
  {
    string name, location;
    private User() { Some                    }
    statements;
  }
```

5. Copy Constructor:

```
class User
  {
    public string name, location;
//Copy Constructor
    public User(User
    obj)
    {
      this.name = obj.name;
      this.location =
    } obj.location;
```

```
static void Main()
{
  User u1= new User();
  User u2 = new
  User(u1);
}
```

Indexers

- It is a unique concept of C#, makes the object treated as an array.
- If a class / structure has indexers defined, its object can act as array therefore Indexers allow instances of a class, structure to be indexed in the same way as arrays.
- The Indexers are also known as smart /virtual arrays
- Indexers are similar to properties except that their accessors take parameters.

Example:

```
public class IndexierClass{
private int [] myArray = new int[100];
public int this [int index]
{
 get
  {
   // Check the index limits.
   if (index < 0 || index >= 100)
       return 0;
   else
       return myArray[index];
    }
 set
  {
   if (!(index < 0 || index >= 100))
        myArray[index] = value;
  }
}
}
```

Indexer

```
public static void Main()
{
IndexerClass b = new
      IndexerClass();
b[3] = 256;
}
```

Indexing the object to store 236 in

About Inheritance

- A class that is derived from another class (a base class) automatically contains all the public, protected, and internal members of the base class.
- Inheritance enables you to create a new class that can reuse, extend, and modify the behavior that is defined in the other class.
- The class whose members are inherited is called the base class and the class that inherits those members is called the derived class.
- A derived class can have only one direct base class.
- Inheritance is transitive.
- Conceptually a derived class is a specialization of the base class.

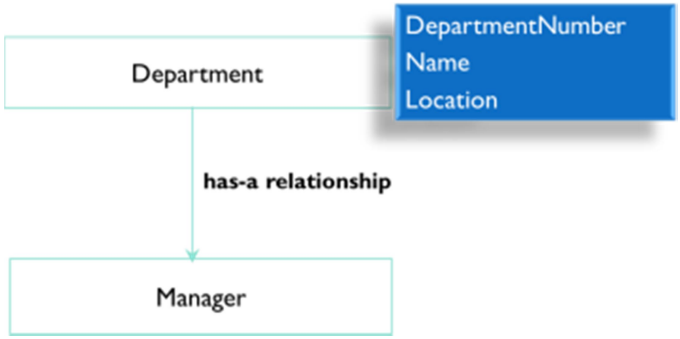
Rules of Inheritance:

- A class can only inherit from one class (known as single inheritance).
- A derived class is guaranteed to do everything the base class can do.
- A derived class inherits members from its base class and can modify or add to its behavior and properties.
- A derived class can define members of the same name in the base class, thus hiding the base class members.
- Inheritance is transitive (i.e., class A inherits from class B, including what B inherited from class C).
- All classes inherit from the highest Object class in the inheritance hierarchy.
- private members and constructors are not inherited by derived classes.

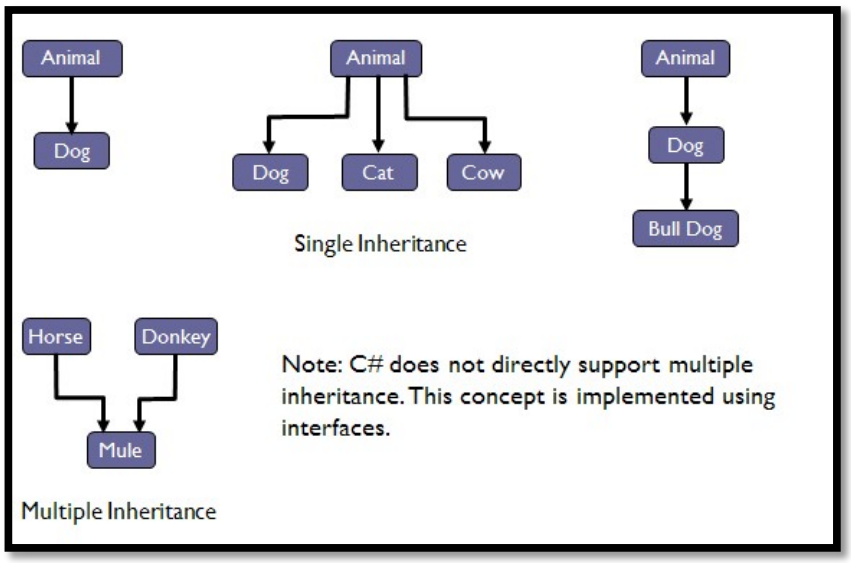
Relationships of Inheritance:

- **"is-a"** relationship in inheritance:
  - Derived class can be used wherever a base class can be used.
  - Is implemented in C# by extending a class.

- **"has-a"** relationship in inheritance(termed as **Containment**):
  - Whole-class relationship between a class and its parts.
  - Implemented in C# by instantiating an object inside a class.



Types of Inheritance



- Inheritance allows you to define a very general class, then later define more specialized classes by adding new detail.
- The general class is called the base or parent class.
- The specialized classes inherit all the properties of the general class. Specialized classes are derived from the base class. They are called derived or child classes.
- After the general class is developed you only have to write the "difference" or "specialization" code for each derived class.
- Single Inheritance can be formed in these ways:
  - only one base class.
  - one base class, many subclasses.
  - Derived from a derived class.
- Multiple Inheritance is formed using several base classes. (C# does not support)

Single Inheritance Example:

```
public class Employee {
    public double salary = 40000;
 }
 public class Manager: Employee
                                    {
    public double bonus = 10000;
 }
 class Demo {
    public static void Main(string[] args) {
        Manager m1 = new Manager();
        Console.WriteLine("Salary: " +
        m1.salary);
        Console.WriteLine("Bonus: " +
        m1.bonus);
```

MultiLevel Inheritance Example: (for this consider the Employee and Manager class created above)

```
public class ProjectManager:
   Employee                              {
   public string projectName =
```

```
class Demo
{
   public static void Main(string[] args)
   {
      ProjectManager p1 = new ProjectManager();
     Console.WriteLine("Project Name: " +
     p1.projectName); Console.WriteLine("Salary: " +
     p1.salary); Console.WriteLine("Bonus: " +
     p1.bonus);
     }
}
```

Abstract Class

- An abstract class cannot be instantiated directly.
- A non-abstract class is derived from an abstract class, the non abstract class must include actual implementations of inherited abstract members, thereby overriding the abstract members.
- An abstract modifier indicates that the thing being modified has missing or incomplete implementation.
- An abstract modifier can be used with classes, methods, properties, indexes, and events.
- **Abstract modifiers are used in class declarations to indicate that a class is intended only to be a base class of other classes.**

Abstract classes have the following rules:

- An Abstract class cannot be instantiated. *i.e.* objects of an abstract class cannot be created
- An Abstract class can contain abstract methods and assessors.
- Abstract modifiers are used in a method or property declaration to indicate that the method or property does not contain implementation.
- They cannot have a sealed access modifier, as this would make inheritance impossible.

Abstract classes are the way to achieve abstraction in C#. Abstraction in C# is the process to hide the internal details and showing functionality only.
Abstraction can be achieved by two ways:

- Abstract class
- Interface

Abstract class and interface both can have abstract methods which are necessary for abstraction.

Abstract Method

A method which is declared abstract and has no body is called abstract method. It can be declared inside the abstract class only. Its implementation must be provided by derived classes.

Example public **abstract** void

Note :

- An abstract method in C# is internally a virtual method so it can be overridden by the derived class.
- You can't use static and virtual modifiers in abstract method declaration

In C#, abstract class is a class which is declared abstract. It can have abstract and non-abstract methods. It cannot be instantiated. Its implementation must be provided by derived classes. Here, derived class is forced to provide the implementation of all the abstract methods.

Let's see an example of abstract class in C# which has one abstract method Draw(). Its implementation is provided by derived classes: Rectangle and Circle. Both classes have different implementation.

```
public abstract class
   Shape { public abstract
   void Draw();
}
public class Rectangle : Shape
   { public override void Draw()
                                    {
      Console.WriteLine("drawing rectangle...");
   }
}
public class Circle : Shape {
   public override void Draw()
   {
      Console.WriteLine("drawing circle...");
```

```
public class Demo{
   public static void Main()
   {
      Shape s;
      s = new Rectangle(); //assign the reference of Rectangle object to Shape
      object s.Draw();  //Invokes Draw method of Rectangle
      s = new Circle(); //assign the reference of Circle object to Shape
      object s.Draw(); //Invokes Draw method of Circle
   }
}
```

Virtual Keyword

It is possible that a derived class defines same method as defined in its base class. This can be achieved by:

- Overriding
- Shadowing

Method overriding is used to achieve runtime polymorphism. It enables you to provide specific implementation of the method which is already provided by its base class.

To perform method overriding in C#, you need to use **virtual** keyword with base class method and override keyword with derived class method.
Example:

```
public class Animal {
   public virtual void
   eat(){
      Console.WriteLine("Eating...");
   }
}
public class Dog: Animal
{
   public override void eat()
   {
      Console.WriteLine("Eating bread...");
   }
}
public class Demo
{
   public static void Main()
   {
      Animal d = new Dog();
      d.eat(); //Invokes eat() of Dog class
   }
}
```

Difference between Abstract and Virtual methods:

Abstract Method	Virtual Method
Always declared in Abstract base class	Can be declared in abstract/ non-abstract base class
Does not have Implementation in base class	Has Implementation in the base class
Derived class must write the implementation by overriding	Derived class may or may not write the implantation by overriding

Interfaces

Interface in C# is a blueprint of a class. It is like abstract class because all the methods which are declared inside the interface are abstract methods. It cannot have method body and cannot be instantiated.

It is used to achieve **multiple inheritance** which can't be achieved by class. It is used to achieve fully abstraction because it cannot have method body.

Its implementation must be provided by class or struct. The class or struct which implements the interface, must provide the implementation of all the methods declared inside the interface.
Note: Interface methods are public and abstract by default. You cannot explicitly use public and abstract keywords for an interface method.

Like a class, Interface can have methods, properties, events, and indexers as its members. But interfaces will contain only the declaration of the members. The implementation of the interface's members will be given by class who implements the interface implicitly or explicitly.

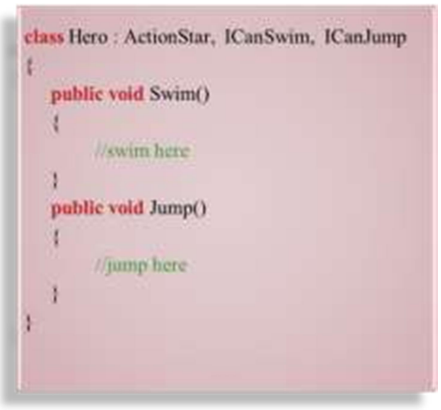
Features:

- It is a Contract
- Interfaces tells the class or struct **"What To Do?"** not **"How to do?**"
- Interfaces can't have private members.
- By default all the members of Interface are public and abstract.
- The interface will always defined with the help of keyword 'interface'.
- Interface cannot contain fields because they represent a particular implementation of data.
- Multiple inheritance is possible with the help of Interfaces but not with classes.

Implementing Interfaces

- Interfaces are implemented using a ":".
- Rules for implementing interface methods are:
- They must have the same method signature and return type.
- They cannot narrow the method accessibility.
- Interface methods are implicitly public.

Example of Multiple Inheritance using Interface



Code Implementing Abstract class, Interface, abstract and virtual methods:

```
abstract class Shape
  {
   protected int side1;
   protected double
   area;
    public abstract void
    ComputeArea(); public virtual
    void DisplayArea()
    {
       Console.WriteLine("Area = " + area);
    }
```

```
interface IDrawable
{
    void Draw();
}
```

Note: Always mention base class first and then Interface when achieving Multiple inheritance using interface

```
class Rectangle : Shape, IDrawable //achieving Multiple Inheritance
  {
     int side2;
     public void AcceptSideValues() {
        Console.Write("Enter length =
        ");
        side1 =
        Convert.ToInt32(Console.ReadLine());
        Console.Write("Enter breadth = ");
        side2 = Convert.ToInt32(Console.ReadLine());
     }
     public override void ComputeArea()
        { area = side1 * side2;
     }
     public new void DisplayArea() {
        Console.WriteLine("Area of Rectangle = " +
        area);
     }
     #region IDrawable Members
     public void Draw()   //implementing IDrawable method
     {
        Console.WriteLine("We can draw rectangle...");
     }
     #endregion
  }
```

Shadowing (using new keyword)

- New Modifier: The new modifier is used to explicitly hide a member inherited from a base class.
- To hide an inherited member, declare it in the derived class using the same name, and modify it with the new modifier.
- Name hiding through inheritance takes one of the following forms:
- A constant, field, property, or type introduced in a class or struct hides all base class members with the same name.
- It is an error to use both new and override on the same member.
- Using the new modifier in a declaration that does not hide an inherited member generates a warning.

Example:

```
class Message
{
   public void DisplayHello()
    {
        Console.WriteLine("Hello from Message class");
    }
}
class WindowMessage: Message
{
   public void DisplayHello()
    {
        Console.WriteLine("Hello from Window Message class");
    }
}
```

```
static void Main()
{
 Message m = new Message();
 m.DisplayHello();

WindowMessage w
          = new
 WindowMessage();
 w.DisplayHello();

}
```

Access Modifiers and Specifiers

Access Modifiers are keywords that define the accessibility of a member, class or datatype in a program. These are mainly used to restrict unwanted data manipulation by external programs or classes. There are 4 access modifiers (public, protected, internal, private) which defines the 6 accessibility levels as follows:

1. public
2. protected
3. internal
4. protected internal
5. private
6. private protected

The Accessibility table of these modifiers is given below:

	PUBLIC	PROTECTED	INTERNAL	PROTECTED INTERNAL	PRIVATE	PRIVATE PROTECTED
Entire program	Yes	No	No	No	No	No
Containing class	Yes	Yes	Yes	Yes	Yes	Yes
Current assembly	Yes	No	Yes	Yes	No	No
Derived types	Yes	Yes	No	Yes	No	No
Derived types within current assembly	Yes	Yes	Yes	Yes	No	Yes

private Accessibility Level:
Access is only granted to the containing class. Any other class inside the current or another assembly is not granted access to these members. Members of a class or struct are private by default.
public Accessibility Level:
Access is granted to the entire program. This means that another method or another assembly which contains the class reference can access these members or types. This access modifier has the most permissive access level in comparison to all other access modifiers. **protected Accessibility Level:**
Access is limited to the class that contains the member and derived types of this class. It means a class which is the subclass of the containing class anywhere in the program can access the protected members.
internal Accessibility Level:
Access is limited to only the current Assembly, that is any class or type declared as internal is accessible anywhere inside the same namespace. It is the default access modifier for types(class, struct, enum, delegates, interface) in C#.
protected internal Accessibility Level:
Access is limited to the current assembly or the derived types of the containing class. It means access is granted to any class which is derived from the containing class within or outside the current Assembly.
private protected Accessibility Level:
Access is granted to the containing class and its derived types present in the current assembly. This modifier is valid in C# version 7.2 and later.

Example showing the syntax of declaring fields of a class with different access modifiers:

```
class Demo //internal by default
{
   int a;   //private by
   default public int b;
   internal int c;
   protected int d;
   protected internal
   p; private internal
   x;
}
```

Understanding Polymorphism

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word.
There are two types of polymorphism in C#:

1. Compile time polymorphism
2. Runtime polymorphism.

Compile time polymorphism is achieved by **method overloading and operator overloading** in C#. It is also known as **static binding or early binding**.
Runtime polymorphism is achieved by **method overriding** which is also known as **dynamic binding or late binding. Runtime** polymorphism was discussed in the topic of Inheritance when we used the keyword override to implement abstract or virtual method in the derived class. (Refer the topic)

Implementing Method Overloading

- Overloading a method means defining multiple versions, using the same name but a different parameter list.
- The purpose of overloading is to define several closely related versions of a method without having to differentiate them by name.
- In a class multiple methods are allowed with unique different signatures.
- Overloaded methods can be implemented in the same class or in a subclass.
- .NET can figure out which method to call during compile based on the parameter types that you pass. This technique is called overloading a method.
- Method overloading is a compile-time mechanism.

Rules of Method Overloading:

There are several rules of method overloading:

- ✓ **Same Name:** Each overloaded version method must use the same method name.
- ✓ **Different Signature**: Each overloaded version must differ from all other overloaded versions in at least following respects:
  - Number of Parameters.
  - Order of Parameters.
  - Data type of the parameters.
- ✓ It can use different access modifiers.

Example: By changing number of arguments but of same datatype

```
public class Calculation{
 public static int AddNos(int a,int b)
 {
    return a + b;
 }
 public static int AddNos (int a, int b, int c)
 {
    return a + b + c;
 }
}
```

```
public class Demo
{
   public static void Main()
   {
      Console.WriteLine(Calculation. AddNos (12, 23));
      Console.WriteLine(Calculation. AddNos (12, 23,
      25));
   }
}
```

C# Member Overloading Example: By changing data type of arguments

```
  static void Display(int a)
     {
        Console.WriteLine("Integer = " + a);
     }
     static void Display(string s)
     {
        Console.WriteLine("String = " + s);
     }
  static void Main()
     {
        Display("abc"); //invokes Display which has string
        argument Display(4);   //invokes Display which has int
        argument Console.ReadLine();
     }
```

Difference between Method Overloading and Method Overriding

Method Overloading	Method Overriding
Methods with same name and different parameters	Methods with same name and same parameters
Access specifiers can be different	Access specifiers has to be same
Can be defined in same or derived class	Can be defined only in derived class. Achieved only in Inheritance.
Compile time polymorphism	Runtime polymorphism

Operator Overloading

The concept of overloading a function can also be applied to operators.

- Operator overloading gives the ability to use the same operator to do various operations.
- It provides additional capabilities to C# operators when they are applied to user-defined data types. It enables to make user-defined implementations of various operations where one or both of the operands are of a user-defined class.

- Only the predefined set of C# operators can be overloaded. To make operations on a user- defined data type is not as simple as the operations on a built-in data type.
- To use operators with user-defined data types, they need to be overloaded according to a programmer's requirement.
- An operator can be overloaded by defining a function to it. The function of the operator is declared by using the **operator** keyword.
  Note: Operator overloading is basically the mechanism of providing a special meaning to an ideal C# operator w.r.t. a user-defined data type such as structures or classes.

The following table describes the overloading ability of the various operators available in C#:

OPERATORS	DESCRIPTION
+, -, !, ~, ++, – –	unary operators take one operand and can be overloaded.
+, -, *, /, %	Binary operators take two operands and can be overloaded.
==, !=, =	Comparison operators can be overloaded.
&&, ||	Conditional logical operators cannot be overloaded directly
+=, -+, *=, /=, %=, =	Assignment operators cannot be overloaded.

In this example, minus operator will be overloaded to find difference between two objects of Time class.
Consider the Time class containing operator overload method

```
class Time {
    public int hrs, mins, secs;
//operator overloaded method to overload – (minus)
operator
    public static Time operator -(Time t1,
      Time t2) { Time temp = new Time();
      int sum1 = (t1.hrs * 3600) + (t1.mins * 60) +
      t1.secs; int sum2 = (t2.hrs * 3600) + (t2.mins *
      60) + t2.secs; int sum; //to hold the difference in
      seconds
      if (sum1 > sum2) { sum = sum1 - sum2; }
      else { sum = sum2 - sum1;
      temp.secs = sum %
      60; sum = sum / 60;
      temp.mins = sum %
      60; temp.hrs = sum /
    } }
```

```
static void Main()
{
    Time starttime = new Time();
    Time endtime = new Time();
    Time DiffTime= endtime -  starttime;
//Translated to Time.operator –
    (endtime,starttime)
}
```

Sealed Class

Sealed classes are used to restrict the users from inheriting the class. A class can be sealed by using the sealed keyword. The keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class. Objects of sealed class can be created.
A method can also be sealed, and in that case, the method cannot be overridden. However, a method can be sealed in the classes in which they have been inherited. If you want to declare a method as sealed, then it has to be declared as virtual in its base class.

Example: The following class definition defines a sealed class in C#:

```
sealed class SimpleMathClass
{
   public int Add(int a, int b)
   {
      return a + b;
   }
}
 class Program {
   static void Main(string[] args)
   {
      // Creating an object of Sealed Class
      SimpleMathClass obj = new
      SimpleMathClass (); int result = obj.Add(6,
      4); Console.WriteLine("Total = " + result);
   }
}
```

this Keyword

this keyword is used to refer to the current instance of the class. It is used to access members from the constructors, instance methods, and instance accessors.
this keyword is also used to track the instance which is invoked to perform some calculation or further processing related to that instance.
The **this** keyword is also used as a modifier of the first parameter of an extension method.

The following are common uses of this:
Example 1: **Using 'this' keyword to refer current class instance members**

```
class Person {
   public string Name;
   public string GetName() {
      return Name;
   }
   public void SetName(string Name)          {
       this.Name = Name;       //"this.Name" refers to class member
   }
}
```

Example 2: To pass an object as a parameter to other methods

```
DisplayDetails(this);
```

Example 3: To declare indexers

```
public int this[int index]
{
   get { return array[index];
   } set { array[index] =
   value; }
}
```

base Keyword

The base keyword is used to access members of the base class from within a derived class:

- Call a method on the base class that has been overridden by another method.
- Specify which base-class constructor should be called when creating instances of the derived class.

A base class access is permitted only in a constructor, an instance method, or an instance property accessor.

Example:

```
public class Person
{
   protected string ssn = "444-55-6666";
   protected string name = "John L.
   Malgraine";

   public virtual void GetInfo()
   {
      Console.WriteLine("Name: {0}", name);
      Console.WriteLine("SSN: {0}", ssn);
   }
}
class Employee : Person
{
   public string id = "ABC567EFG";
   public override void GetInfo()
   {
      // Calling the base class GetInfo
      method: base.GetInfo();
      Console.WriteLine("Employee ID: {0}",
      id);
   }
}
```

Following example shows how to specify the base-class constructor called when creating instances of a derived class.

```
public class BaseClass
{
```

```
  int num;
  public BaseClass(){
    Console.WriteLine("In BaseClass constructor");
  }
  public BaseClass(int i)
  {
    num = i;
    Console.WriteLine("In BaseClass parameterized constructor");
  }
  public int GetNum()
  {
    return num;
  }
}
```

```
public class DerivedClass : BaseClass
{
  // This constructor will call
  BaseClass.BaseClass() public
  DerivedClass() : base()
  {
  }
  // This constructor will call
  BaseClass.BaseClass(int i) public
  DerivedClass(int i) : base(i)
  {
  }
```

```
class Demo{
 static void Main()
  {
    DerivedClass md = new
    DerivedClass(); DerivedClass md1 =
    new DerivedClass(1);
  }
}
```

Output:

```
In BaseClass constructor
In BaseClass parameterized constructor
```

Collections and Generics

Collections standardize the way of which the objects are handled by your program. In other words, it contains a set of classes to contain elements in a generalized manner. With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search, sort etc.

The Collection (Generic and Non-Generic) classes are under the **System.Collections** root namespace

Collections are categorized as:

1. Non-Generic collections
2. Generic collections

Some of the Common Interfaces implemented by the Collection classes

Core Interface	Description
`ICollection`	Defines size, enumerators and synchronization methods for all collections.
`IComparer`	Exposes a method that compares two objects.
`IDictionary`	Represents a collection of key-and-value pairs.
`IDictionaryEnumerator`	Enumerates the elements of a dictionary.
`IEnumerable`	Exposes the enumerator, which supports a simple iteration over a collection.
`IEnumerator`	Supports a simple iteration over a collection.
`IList`	Represents a collection of objects that can be individually accessed by index.

Non-Generic collection

Non-Generic collection in C# is defined in **System.Collections** namespace. It is a general-purpose data structure that works on object references, so it can handle any type of object, but not in a safe- type manner. Non-generic collections are defined by the set of interfaces and classes. Below table contains the frequently used classes of the **System.Collections** namespace:

CLASS NAME	DESCRIPTION
ArrayList	It is a dynamic array means the size of the array is not fixed, it can increase and decrease at runtime.
Hashtable	It represents a collection of key-and-value pairs that are organized based on the hash code of the key.
Queue	It represents a first-in, first out collection of objects. It is used when you need a first-in, first-out access of items.
Stack	It is a linear data structure. It follows LIFO(Last In, First Out) pattern for Input/output.

ArrayList

- Implements the **IList** interface.
- One dimensional array whose size is dynamically increased as required.
- Each element is of **object** datatype

Example:

```
ArrayList countries = new
ArrayList();
countries.Add("India");
countries.Add("Japan");
countries.Add("USA");
countries.Add(88);
Console.WriteLine("Elements in Arraylist
: "); foreach(object obj in countries)
 {
    Console.WriteLine(obj);
 }
```

Stack

- Represents a simple last-in-first-out (LIFO)
- It is dynamic in size
- Each elements is of object type
- Elements are added in the stack using Push method
- Each element is accessed and removed (LIFO) from the Stack using Pop method

Example:

```
Stack countries = new
Stack();
countries.Push("Srilanka");
countries.Push("India");
```

```
Console.WriteLine("Elements in stack : " + countries.Count); //output -> 3
Console.WriteLine("using foreach");
foreach (object obj in countries)
{
   Console.WriteLine(obj);
}
Console.WriteLine("Elements in stack : " + countries.Count); //output -> 3
Console.WriteLine("using Pop()");
while (countries.Count > 0)
{
   Console.WriteLine(countries.Pop());          //elements are read and removed from
   Stack
}
Console.WriteLine("Elements in stack : " + countries.Count); //output -> 0
```

Queue

- Represents a simple first-in-first-out (FIFO)
- It is dynamic in size
- Each elements is of object type
- Elements are added in the Queue using Enqueue method
- Each element is accessed and removed (FIFO) from the Queue using Dequeue method

Example:

```
Queue countries = new
Queue ();
countries.Enqueue("Srilanka")
; countries.Enqueue("India");
countries. Enqueue ("Nepal");

Console.WriteLine("Elements in Queue: " + countries.Count); //output -> 3
Console.WriteLine("using foreach");
foreach (object obj in countries)
{
   Console.WriteLine(obj);
}
Console.WriteLine("Elements in Queue: " + countries.Count); //output -> 3
Console.WriteLine("using Dequeue()");
while (countries.Count > 0)
{
   Console.WriteLine(countries. Dequeue ());//elements are read and removed from
   Stack
}
```

Hashtable

- Represents a collection of key/value pairs that are organized based on the hash code of the key.
- It is dynamic in size
- Each key/value pair is of object type

Example

```
Hashtable currencies = new Hashtable();

        currencies.Add("IN",
        "Rupee");
        currencies.Add("US",
        "Dollar");
        currencies.Add("JP", "Yen");

        Console.WriteLine("Currency of India : " + currencies["IN"]);
        //each element is stored as DictionaryEntry
        object foreach (DictionaryEntry d in
        currencies)
        {
```

```
        Console.WriteLine("All keys : ");
        foreach(object k in
        currencies.Keys)
        {
          Console.WriteLine(k);
        }
        Console.WriteLine("All values : ");
        foreach (object v in
        currencies.Values)
        {
          Console.WriteLine(v);
```

Generic Collections

Generic collection in C# is defined in **System.Collection.Generic** namespace. It provides a generic implementation of standard data structure like linked lists, stacks, queues, and dictionaries. These collections are type-safe because they are generic means only those items that are type-compatible with the type of the collection can be stored in a generic collection, it eliminates accidental type mismatches. Generic collections are defined by the set of interfaces and classes. Below table contains the frequently used classes of the **System.Collections.Generic** namespace:

CLASS NAME	DESCRIPTION
Dictionary<TKey,TValue>	It stores key/value pairs and provides functionality similar to that found in the non-generic Hashtable class.
List<T>	It is a dynamic array that provides functionality similar to that found in the non-generic ArrayList class.
Queue<T>	A first-in, first-out list and provides functionality similar to that found in the non-generic Queue class.
SortedList<TKey,TValue>	It is a sorted list of key/value pairs and provides functionality similar to that found in the non-generic SortedList class.
Stack<T>	It is a first-in, last-out list and provides functionality similar to that found in the non-generic Stack class.
HashSet<T>	It is an unordered collection of the unique elements. It prevent duplicates from being inserted in the collection.
LinkedList<T>	It allows fast inserting and removing of elements. It implements a classic linked list.

Generics

- Generic is a class which allows the user to define classes and methods with the placeholder. Generics were added to version 2.0 of the C# language.
- The basic idea behind using Generic is to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes, and interfaces.
- A primary limitation of collections is the absence of effective type checking. This means that you can put any object in a collection because all classes in the C# programming language extend from the object base class.
- This compromises type safety and contradicts the basic definition of C# as a type-safe language.
- In addition, using collections involves a significant performance overhead in the form of implicit and explicit type casting that is required to add or retrieve objects from a collection.
- To address the type safety issue, the .NET framework provides generics to create classes, structures, interfaces, and methods that have placeholders for the types they use.
- Generics are commonly used to create type-safe collections for both reference and value types.
- The .NET framework provides an extensive set of interfaces and classes in the **System.Collections.Generic** namespace for implementing generic collections.

Features of Generics

- Generics are similar to templates in C++ but are different in implementation and capabilities.

- Generics introduces the concept of **type parameters**, because of which it is possible to create methods and classes that defers the framing of data type until the class or method is declared and is instantiated by client code.
- Generic types perform better than normal system types because they reduce the need for boxing, unboxing, and type casting the variables or objects.
- A type parameter is a placeholder for a particular type specified when creating an instance of the generic type.
- A generic type is declared by specifying a type parameter in an angle bracket after a type name,
  e.g. TypeName<T> where T is a type

Generic class:

- Generic classes encapsulate operations that are not specific to a particular data type.
- The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on.
- Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored
- Generic classes are defined using a type parameter in an angle bracket after the class name.

Example: Define Generic Class

```
class
   BaseDataStore<T> {
   public T Data { get;
```

Here, the BaseDataStore is a generic class. **T is called type parameter**, which can be used as a type of fields, properties, method parameters, return types, and delegates in the BaseDataStore class. For example, Data is generic property because we have used a **type parameter T** as its type instead of the specific data type.

Note: It is not required to use T as a type parameter. You can give any name to a type parameter. Generally, T is used when there is only one type parameter.

You can also define multiple type parameters separated by a comma.

Example: Generic Class with Multiple Type Parameters

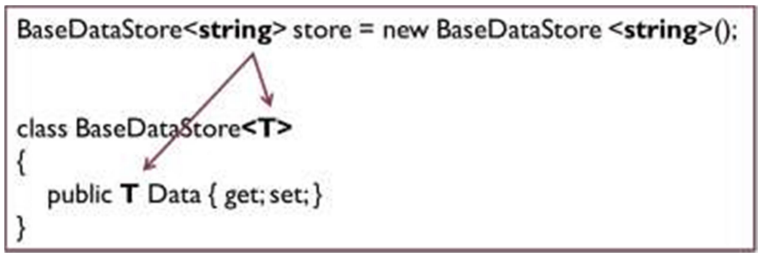
```
class KeyValuePair<TKey, TValue>
{
   public TKey Key { get; set; }
   public TValue Value { get;
   set; }
}
```

Instantiating Generic Class

You can create an instance of generic classes by specifying an actual type in angle brackets. The following creates an instance of the generic class BaseDataStore.

```
BaseDataStore<string> store = new BaseDataStore <string>();
```

Above, we specified the string type in the angle brackets while creating an instance. So, T will be replaced with a string type wherever T is used in the entire class at compile-time. Therefore, the type of Data property would be a string.



You can specify the different data types for different objects, as shown below.

```
BaseDataStore<string> strStore = new BaseDataStore
<string>(); strStore.Data = "Hello World!";
//strStore.Data = 123; // compile-time error

BaseDataStore <int> intStore = new BaseDataStore
<int>(); intStore.Data = 100;
//intStore.Data = "Hello World!"; // compile-time error

KeyValuePair<int, string> kvp1 = new KeyValuePair<int,
string>(); kvp1.Key = 100;
kvp1.Value = "Hundred";

KeyValuePair<string, string> kvp2 = new KeyValuePair<string,
string>(); kvp2.Key = "IT";
kvp2.Value = "Information Technology";
```

Generic Class Characteristics

- ✓ A generic class increases the reusability. The more type parameters mean more reusable it becomes. However, too much generalization makes code difficult to understand and maintain.
- ✓ A generic class can be a base class to other generic or non-generic classes or abstract classes.
- ✓ A generic class can be derived from other generic or non-generic interfaces, classes, or abstract classes.

Generic Methods

A method declared with **the type parameters** for its return type or parameters is called a generic method.

Example of Generic Swap method:

```
static void Swap<T>(ref T a, ref T b)
 {
       T t;
       t    =
       a;   a
       =   b;
```

```
}
  static void Main()
  {
    int x = 20, y = 45;
    string s1 = "abc", s2 = "xyz";
    Console.WriteLine("Before
    swap"); Console.WriteLine("x =
    " + x); Console.WriteLine("y = "
    + y); Console.WriteLine("s1 = "
    + s1); Console.WriteLine("s2 = "
    + s2);

    Swap<int>(ref x, ref y);
    Swap<string>(ref s1, ref
    s2);

    Console.WriteLine("After
    swap"); Console.WriteLine("x
    = " + x); Console.WriteLine("y
    = " + y);
    Console.WriteLine("s1 = " +
    s1); Console.WriteLine("s2 = "
```

Advantages of Generics

- ✓ Generics increase the reusability of the code. You don't need to write code to handle different data types.
- ✓ Generics are type-safe. You get compile-time errors if you try to use a different data type than the one specified in the definition.
- ✓ Generic has a performance advantage because it removes the possibilities of boxing and unboxing.

Generic Collections:

List<T> : Represents a strongly typed ArrayList.

Example

```
static void Main()
  {
    List<int> numberList = new List<int>();
    List<string> nameList = new
    List<string>();

    numberList.Add(5
    );
    numberList.Add(6
    );

    nameList.Add("larry"
    );
    nameList.Add("john"
```

```
        {
           Console.WriteLine(n);
        }
        Console.WriteLine("Names are :
        "); foreach(string s in nameList)
        {
           Console.WriteLine(s);
        }
        Console.ReadLine();
    }
```

Dictionary<K,V>: Represents a generic collection of keys and values.
Example:

```
Dictionary<int,string>emp = new
Dictionary<int,string>(); emp.Add(1, "James");
emp.Add(12, "Jimmy");
emp.Add(3, "James");
emp[12] = "Kim";        //change the name of empcode 12

if(!emp.ContainsKey(4)) //search the key
{
          emp.Add(4,"Tim");                //add a new
          employee Console.WriteLine("New employee
          added");
}
```

```
 //each entry is an object of
KeyValuePair<>
foreach(KeyValuePair<int,string> de in
emp)
{
          Console.WriteLine("{0}  {1}", de.Key, de.Value);
```

Stack<T> : Generic Stack collection similar to Non generic Stack but element of similar type
Example:

```
       Stack<string> countrystack = new
        Stack<string>(); countrystack.Push("Japan");
        countrystack.Push("China");
        Console.WriteLine("elements of Stack");
        while(countrystack.Count>0)
        {
           Console.WriteLine(countrystack.Pop());
        }
```

Queue<T> : Generic Queue collection similar to Non generic Queue but element of similar type
Example:

```
Queue<string> countryqueue = new
 Queue<string>();
 countryqueue.Enqueue("Australia");
 countryqueue.Enqueue("New Zealand");
 Console.WriteLine("elements of Queue");
 while (countryqueue.Count > 0)
 {
    Console.WriteLine(countryqueue.Dequeue());
 }
```

Delegates and Events

- A delegate is an object which refers to a method or you can say it is a reference type variable that can hold a reference to the methods.
- Delegates in C# are similar to the function pointer in C/C++. It provides a way which tells which method is to be called when an event is triggered.
- For example, if you click an Button on a form (Windows Form application), the program would call a specific method. In simple words, it is a type that represents references to methods with a particular parameter list and return type and then calls the method in a program for execution when it is needed.

Important Points About Delegates:

- ✓ Provides a good way to encapsulate the methods.
- ✓ **Delegates are the library class in System namespace.**
- ✓ **All user defined delegates inherit from the System. Delegate type**
- ✓ These are the type-safe pointer of any method.
- ✓ Delegates are mainly used in implementing the call-back methods and events.
- ✓ Delegates can be chained together as two or more methods can be called on a single event.
- ✓ It doesn't care about the class of the object that it references.
- ✓ Delegates can also be used in "anonymous methods" invocation.
- ✓ Anonymous Methods(C# 2.0) and Lambda expressions(C# 3.0) are compiled to delegate types in certain contexts. Sometimes, these features together are known as anonymous functions.
- ✓ Foundation for implementing event handling
- ✓ Types of Delegate:
  - Unicast (Single cast) Delegate
  - Multicast Delegate

Declaration of Delegates

Delegate type can be declared using the delegate keyword. Once a delegate is declared, delegate instance will refer and call those methods whose return type and parameter-list matches with the delegate declaration.

Syntax:

```
[modifier] delegate [return_type] [delegate_name] ([parameter_list]);
```

Here,

modifier: It is the required modifier which defines the access of delegate and it is optional to use.

delegate: It is the keyword which is used to define the delegate.
return_type: It is the type of value returned by the methods which the delegate will be going to call. It can be void. A method must have the same return type as the delegate.
delegate_name: It is the user-defined name or identifier for the delegate.
parameter_list: This contains the parameters which are required by the method when called through the delegate.

Example:

```
// here, MathDelegate is a delegate, whose instances will hold references of only
those methods
 //which has 2 int arguments and returns
```

Note: A delegate will call only a method which agrees with its signature and return type. A method can be a static method associated with a class or can be an instance method associated with an object, it doesn't matter.

Instantiation & Invocation of Delegates

1. After declaring a delegate, a delegate object is created with the help of **new** keyword.
2. Once a delegate is instantiated, a method call made to the delegate is pass by the delegate to that method.
3. The parameters passed to the delegate by the caller are passed to the method, and the return value, if any, from the method, is returned to the caller by the delegate. This is known as invoking the delegate.

Syntax:

```
[delegate_name] [instance_name] = new [delegate_name](calling_method_name);
```

Example using MathDelegate created above (this is example of Single cast delegate

```
// Step 1: Declare the delegate
delegate int MathDelegate(int a, int b);

  class Program
  {
     static int AddNos(int x,int y)
     {
        return (x + y);
     }
     static int SubtractNos(int x, int y)
     {
        return (x - y);
     }
     static void Main(string[] args)
     {
        //step 2: create object of the delegate and assign the method address to it
        MathDelegate mDel = new MathDelegate(AddNos);
```

```
    //another way to do the above step
     // MathDelegate mDel = AddNos;

        //Step 3: invoke the method using delegate
        int r = mDel(5, 4); // invokes the method whose reference it currently holds at

        runtime Console.WriteLine("Result = " + r);

       mDel = SubtractNos;  //now mDel holds reference of
        SubtractNos r = mDel(8, 3);    //this is will invoke only
        SubtractNos

        Console.WriteLine("Result = " +
        r); Console.ReadLine();
    }
}
```

Multicasting of a Delegate

- Multicasting of delegate is an extension of the normal delegate(sometimes termed as Single Cast Delegate).
- It holds references of more than one methods.
- It helps the user to invoke more than one method in a single call.

Properties:

- Delegates are combined and when you call a delegate then a complete list of methods is called.
- All methods are called in First in First Out(FIFO) order.
- '+' or '+=' Operator is used to add the methods to delegates.
- '–' or '-=' Operator is used to remove the methods from the delegates list.

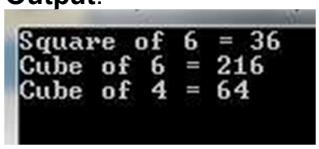
Note: Remember, multicasting of delegate should have a return type of Void otherwise it will throw a runtime exception. Also, the multicasting of delegate will return the value only from the last method added in the multicast. Although, the other methods will be executed successfully.

Example:

```
//declare delegate with name ComputeDelegate, whose instance will hold reference of the
methods
//having 1 int argument and returns
  void delegate void
  ComputeDelegate(int a);

  class MultiCastDemo
  {
     static void ComputeSquare(int x)
     {
        Console.WriteLine("Square of {0} = {1}" x, (x * x));
```

```
static void ComputeCube(int x)
  {
    Console.WriteLine("Cube of {0} = {1}", x, (x * x * x));
  }
  static void Main()
  {
    //it is going to be multicast, as it will hold reference of ComputeSquare and
    ComputeCube ComputeDelegate cDel = ComputeSquare;
    cDel += ComputeCube;          // += is the operator for multicast

    //this will invoke all the methods whose reference cDel holds at
    runtime cDel(6); //prints Square and Cube of 6

    cDel -= ComputeSquare;

    cDel(4); //prints only Cube of

    4

    Console.ReadLine();
  }
```

Output:



Anonymous Method in C#

- An anonymous method is a method which doesn't contain any name which is introduced **in C# 2.0.**
- It is useful when the user wants to create an inline method and also wants to pass parameter in the anonymous method like other methods.
- An Anonymous method is defined using the **delegate** keyword and the user can assign this method to a variable of the delegate type.
- Anonymous methods allow the code associated with a delegate to be written "in-line" where the delegate is used, conveniently tying the code directly to the delegate instance.

Example:

```
 delegate void StringDelegate(string
s); delegate int MathDelegate(int a,
int              b);              class
AnonymousMethodDemo
 {
    static void Main()
    {
      MathDelegate del = delegate (int a, int b)  {  //method body
                                                    return (a * b);  };
```

```
        int r = del(7, 2);
        Console.WriteLine("Result = "
        + r);

        StringDelegate sDel = delegate (string s) {
                                                Console.WriteLine("string entered = " + s);    };

        sDel += delegate (string str) //Anonymous method as Multicast delegate
        {
           Console.WriteLine("string length = " + str.Length);
        };

        sDel("Hello
        world");
        Console.ReadLin
        e();
```

Lambda Expressions in C#

- C# 2.0 introduces anonymous methods, which allow code blocks to be written “in-line” where delegate values are expected.
- A lambda expression is an anonymous function that can contain expressions and statements, and can be used to create delegates or expression tree types.
- All lambda expressions use the lambda **operator =>,** which is read as "goes to". The left side of the lambda operator specifies the input parameters (if any) and the right side holds the expression or statement block.
- The lambda expression **x => x * x** is read "**x goes to x**

times x." **Example:**

```
class LambdaExpressionDemo
 {
     public delegate void Calculate(int a, int
     b); public static void Main()
     {
      Calculate mydel =((a,b) => Console.WriteLine("addition is {0}" , a + b));
        mydel(10, 20);
     }
  }
```

Events

- C# and .NET supports **event driven programming** via **delegates**.
- Delegates and events provide notifications to client applications when some state changes of an object. It is an encapsulation of idea that "Something happened".
- Events and Delegates are tightly coupled concept because event handling requires delegate implementation to dispatch events.

- **An event is an automatic notification sent by an object to signal the occurrence of an action.**
- The action could be caused by user interaction, such as a mouse click, or it could be triggered by some other program logic.
- The object that raises the event is called the event sender(**publisher**).
- The object that captures the event and responds to it is called the event receiver(**subscriber**).
- **An event is built upon a delegate.**
- **The delegate holds the address of the method which will receives (handles) the event when it is raised.**

Following are the key points about Event

✓ Event Handlers in C# return void and take two parameters.
✓ The First parameter of Event - Source of Event means publishing object.
✓ The Second parameter of Event - Object derived from EventArgs.
✓ The publishers determines when an event is raised and the subscriber determines what action is taken in response.
✓ An Event can have so many subscribers.
✓ Events are basically used for the single user action like button click.
✓ If an Event has multiple subscribers then event handlers are invoked synchronously.

Example: In .NET, Windows Forms Application is an Event driven application. Means it performs any task only when an event occurs, like, button is clicked or option is selected from a dropdown. And when the event occurs, the .NET framework predefined delegate **EventHandler will invoke the methods whose references it contains.** (we will discuss this in Windows Forms Application topic)

Exception Handling in the .NET Framework

- An **exception** is any error condition or unexpected behavior that is encountered by a program at runtime.(or also commonly referred as **Runtime Errors**)
- Exceptions can be raised because of a fault in your code or in code that you call (such as a shared library), unavailable operating system resources, unexpected conditions the common language runtime encounters (such as code that cannot be verified), and so on.
- Your application can recover from some of these conditions, but not from others. Although you can recover from most application exceptions, you cannot recover from most runtime exceptions.
- In the .NET Framework, an exception is an object that inherits from the **System. Exception** class.
- An exception is thrown from an area of code where a problem has occurred. The exception is passed up the stack until the application handles it or the program terminates.

Exception Handling is the process to handle the errors generated at runtime, and ensure that the program terminates normally without any causing loss or damage to any system or file resources. The actions to be performed in case of occurrence of an exception is not known to the program. In such a case, we create an exception object and call the exception handler code. **The execution of an exception handler so that the program code does not crash is called exception handling.**

Use the Try/Catch..Finally Block to Handle Exceptions

- ✓ Place the sections of code that might throw exceptions in a try block and place code that handles exceptions in a catch block.
- ✓ The catch block is a series of statements beginning with the keyword catch, followed by an exception type and an action to be taken.
- ✓ Some resource cleanup, such as closing a file, must always be executed even if an exception is thrown. To accomplish this, you can use a finally block. A finally block is always executed, regardless of whether an exception is thrown.

Syntax:

```
try
{
 // statements that may cause an exception
}
catch (Exception obj) //here, obj is object of Exception class to hold information about the error
{
 // code to handle the exception
}
finally //this is an optional block
{
//statements that should be executed whether exception occurred or not
}
```

Following are some of the Common .NET framework defined Exception classes which CLR is aware of and throws when the corresponding situation occurs in the code:

- ➢ **System.ArithmeticException**:
  - o A base class for exceptions that occur during arithmetic operations, such as System.DivideByZeroException and System.OverflowException.
- ➢ **System.ArrayTypeMismatchException**:
  - o Thrown when a store into an array fails because the actual type of the stored element is incompatible with the actual type of the array.
- ➢ **System.DivideByZeroException**:
  - o Thrown when an attempt to divide an integral value by zero occurs.
- ➢ **System.IndexOutOfRangeException**:
  - o Thrown when an attempt to index an array via an index that is less than zero or outside the bounds of the array.
- ➢ **System.InvalidCastException**:
  - o Thrown when an explicit conversion from a base type or interface to a derived type fails at run time.
- ➢ **System.NullReferenceException**:
  - o Thrown when a null reference is used in a way that causes the referenced object to be required.
- ➢ **System.OutOfMemoryException**:
  - o Thrown when an attempt to allocate memory (via new) fails.

- **System.OverflowException**:
  - Thrown when an arithmetic operation in a checked context overflows.
- **System.StackOverflowException**:
  - Thrown when the execution stack is exhausted by having too many pending method calls; typically indicative of very deep or unbounded recursion.
- **System.TypeInitializationException**:
  - Thrown when a static constructor throws an exception, and no catch clauses exists to catch it.

Example: The following program handles any type of Exception which can be thrown by the code.

```
static void Main(string[] args)
    {
        try {
            int[] arr = { 1, 2, 3, 4, 5 }; // Declare an array of max index 4

          // Display values of array
            elements for (int i = 0; i <
            arr.Length; i++)
             {
                 Console.WriteLine(arr[i]);
             }
            Console.WriteLine(arr[7]);       // Try to access invalid index of array
            // An exception is thrown upon executing the above line
          }
         catch (Exception ex) //ex is object of Exception class having information of the
         error
          {
              // The Message property of the object ex of Exception
            // is used to display the default error message of exception
           // that has occurred to the user.

            Console.WriteLine("Error occurred : " + ex.Message);
          }
       //here there is no finally block as it is optional
```

Output:

```
1
2
3
4
5
Error occurred : Index was outside the bounds of the array.
```

Using Multiple try-catch blocks

In the code given below, we attempt to generate an exception in the try block and catch it in one of the multiple catch blocks.
Multiple catch blocks are used when we are not sure about the exception type that may be generated, so we write different blocks to tackle any type of exception that is encountered.

The finally block is the part of the code that has to be executed irrespective of if the exception was generated or not. In the program given below the elements of the array are displayed in the finally block.

Syntax:

```
try
{
 // statements that may cause an exception
}
catch(Specific_Exception_type ex)
{
 // handler code
}
catch(Specific_Exception_type ex)
{
 // handler code
}
.
.
catch(Exception ex)
{
 //code to handle other exceptions which are not handled
//by any of the catch block of Specific_Exception_type mentioned above
}
finally
{
 //default code
}
```

Example: This program has a catch block to handle specifically DivideByZero exception, but if any other exception is thrown by the program it will be handled by catch block with base Exception class

```
static void Main(string[] args)
    {
       try
       {
          int n, d, q;
          Console.Write("enter numerator = ");
          n = Convert.ToInt32(Console.ReadLine());
```

```
            Console.Write("enter denominator =
            "); d =
            Convert.ToInt32(Console.ReadLine())
            ; q = n / d;
               Console.WriteLine("Quotient = " + q);
         }
          catch (DivideByZeroException ex) //Specific Exception class
          {
            Console.WriteLine("Error - Denominator cannot be zero");
          }
          catch(Exception ex) //generalized class
          {
            Console.WriteLine("Error - "+ ex.Message);
          }
          Console.WriteLine("Press enter to
          terminate.."); Console.ReadLine();
       }
```

Note: At a given time only single catch block will execute.

- If denominator provided is zero, it will call the catch block for DivideByZeroException
- if any other exception occurs, it will call the catch block for Exception

Points to remember:

- ✓ An exception is any error condition or unexpected behavior that is encountered by an executing program.
- ✓ Use the Try/Catch..Finally Block to Catch Exceptions.
- ✓ A finally block is always executed, regardless of whether an exception is thrown

C# New features

Following are features added to C# language in different versions:

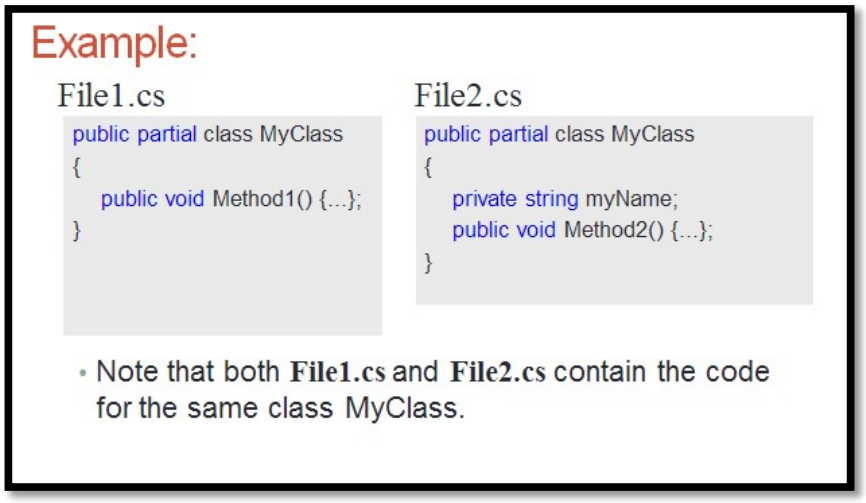
- C# 2.0 features:
  a. Generics (discussed in Collections and Generics topic)
  b. Partial Classes(types)
  c. Anonymous methods (discussed in Delegates topic)
  d. Nullable value types
  e. Iterators
  f. Co-variance and Contra-variance
  g. Getter/setter separate accessibility
  h. Static classes
  i. Delegate inference (discussed in Delegates topic)
- C# 3.0 features:
  a. Auto-implemented properties
  b. Anonymous types
  c. Query expressions
  d. Lambda expressions (discussed in Delegates topic)
  e. Expression trees
  f. Extension methods

g. Implicitly typed local variables
h. Partial methods
i. Object and collection initializers

- C# 4.0 features:
  a. Dynamic binding
  b. Named/optional arguments (discussed in Working with Methods topic)
  c. Generic covariant and contravariant
  d. Embedded interop types

Not all the features are going to be discussed here. Let us understand some commonly used features in C#.

1. **Partial Classes:**
   - C# 2.0 allows you to split the definition and implementation of a class or a struct, interface, and method over two or more Source files.
   - You can put one part of a class in one file and another part of the class in a different file by using the new **partial** keyword.
   - Each source file should contain a section of the class or method definition. When you compile the application, the compiler combines all the source files.



2. Nullable Types:
   - In C#, the compiler does not allow you to assign a null value to a variable. So, C# 2.0 provides a special feature to assign a null value to a variable that is known as the **Nullable** type.
   - The **Nullable** type allows you to assign a null value to a variable of value type.
   - Nullable types can only work with Value Type **not with Reference Type** because it already contains a null value.
   - The Nullable type is an instance of **System.Nullable<T> struct**. Here T is a type which contains non-nullable value types like integer type, floating-point type, a boolean type, etc. For example, in nullable of integer type you can store values from - 2147483648 to 2147483647, or null value.

Syntax:

```
Nullable<data_type> variable_name = null;
```

Or you can also use a shortcut which includes ? operator with the data type.

```
datatype? variable_name = null;
```

Example : showing different ways to declare a Nullable type:

```
int j =                    // this will give compile time
Nullable<int> j = null; // Valid
declaration
```

How to access the value of Nullable type variables?

You cannot directly access the value of the Nullable type. You have to use GetValueOrDefault() method to get an orginal assigned value if it is not null. You will get the default value if it is null. The default value for null will be zero.

Example:

```
static void Main(string[] args)
  {
     int? n1 = null;          // defining Nullable type

     // using the method
     // output will be 0 as default

     // Assigning value of n1 to n2. If n1 is null it will assign zero, or else the
     value. int n2 = n1.GetValueOrDefault(); //here, n2 = 0 because n1 is
     null

     Console.WriteLine("value of n2 = " +

    n2); n1=45;
 // to display value of n1 we have to use property Value because n1 is a
   nullable type Console.WriteLine("value of n1 = " + n1.Value); //here, n1 = 45

    int n3 = n1.GetValueOrDefault(); //here, n3 = 45 because n1 is not null and has value 45.
}
```

3. **getter/setter separate accessibility in C#**

   The get and set portions of a property or indexer are called accessors. By default these accessors have the same visibility or access level of the property or indexer to which they belong. However, it is sometimes useful to restrict access to one of these accessors.

   Typically, this involves restricting the accessibility of the set accessor, while keeping the get accessor publicly accessible.

   Example: private string empname = "Kiran"; public string EmployeeName{

```
   get   {    return empname;   }
  protected set { empname = value;
   }
```

In the above example, a property called **EmployeeName** defines a get and set accessor. The get accessor receives the accessibility level of the property itself, public in this case, while the set accessor is explicitly restricted by applying the protected access modifier to the accessor itself. So only the derived class will be allowed to set the value for empname.

4. **Static classes:**
   - Static class is a collection of only static members.
   - It is implicitly Sealed class, which means it cannot be inherited.
   - An object of static class cannot be created.

Example:

```
static class
    MathCalculations{ static
    int AddNos(int a,int b)
    {
        return (a + b);
     }
  static int SubtractNos(int a,int b)
    {
        return (a - b);
     }
 }
```

5. **AutoImplemented 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.
   - They also enable client code to create objects.
   - When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's get and set accessors.

Example:

```
class Customer
 {
    // Auto-implemented properties for trivial get and set
      public double TotalPurchases { get;
     set; } public string Name { get; set; }
    public int CustomerId { get; set; }
 }
```

6. **Implicitly Typed Local Variables (var keyword):**

- C# 3.0 offers type inference that allows you to define a variable by using the “var” keyword instead of a specific type. This is equivalent to defining a variable of type object.
- When “var” is used, the compiler infers the type from the expression used to initialize the variable. The compiled IL code contains only the inferred type.
- The implicitly typed variable is not designed to replace the normal variable declaration, it is designed to handle some special-case situation like **LINQ(Language-Integrated Query**).

- It is commonly used in the scenarios, when we do not know what type of data the right side would be returning and assign to a variable.
- var does not allow the type of value assigned to be changed after it is assigned to. This means that if we assign an integer value to a var then we cannot assign a string value to it. This is because, on assigning the integer value, it will be treated as an integer type thereafter. So thereafter no other type of value cannot be assigned.

Example:

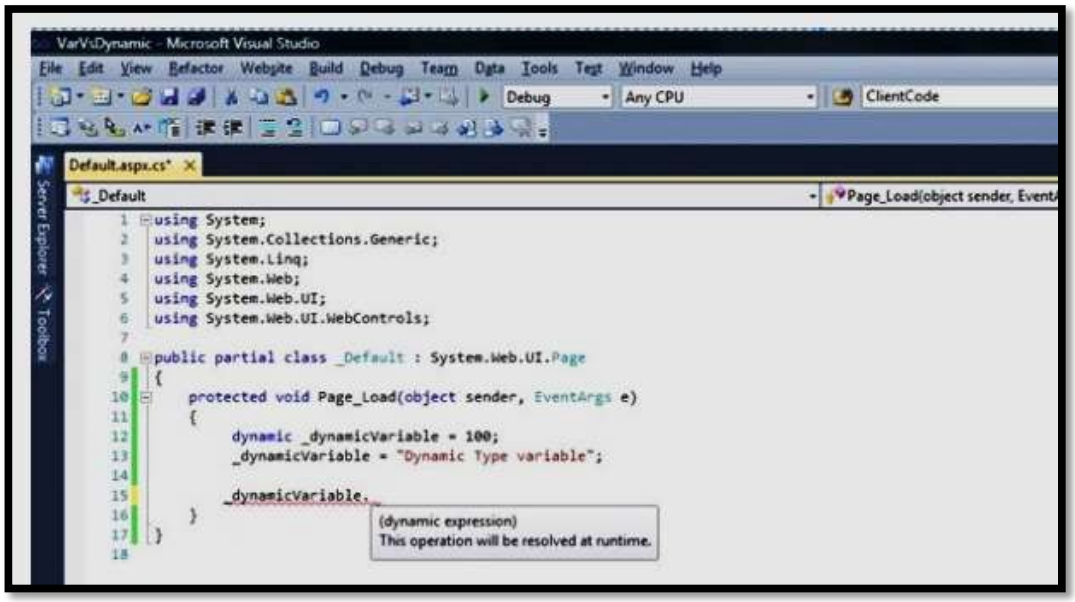
```
// Creating and initializing: implicitly typed variables by Using var
keyword var a = 'f';
var b = "Hello
world"; var c =
30.67d;
var d = false;
var e =
54544;

// Display the datatype
Console.WriteLine("datatype of 'a' is : {0} ",
  a.GetType()); Console.WriteLine("datatype of 'b' is :
  {0} ", b.GetType()); Console.WriteLine("datatype of
  'c' is : {0} ", c.GetType());
```

Output:

```
datatype of 'a' is : System.Char
datatype of 'b' is :
System.String datatype of 'c' is
: System.Double datatype of 'd'
is : System.Boolean datatype
of 'e' is : System.Int32
```

7. **Dynamic type:**
   - In C# 4.0, a new type is introduced that is known as a dynamic type.
   - It is used to avoid the compile-time type checking.
   - The compiler does not check the type of the dynamic type variable at compile time, instead of this, the compiler gets the type at the run time.
   - The dynamic type variable is created using **dynamic** keyword.
   - dynamic are dynamically typed variables. This means, their type is inferred at run-time and not the compile time in contrast to **var** type.
   - dynamic allows the type of value to change after it is assigned to initially.
   - Intellisense help is not available for dynamic type of variables since their type is unknown until run time. So intellisense help is not available. Even if you are informed by the compiler as "This operation will be resolved at run-time".

Example:



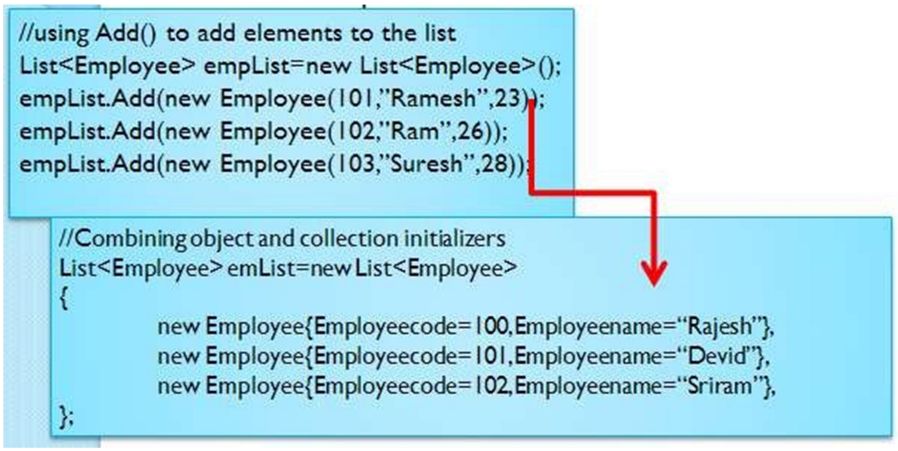
8. **Object Initializers:**
   - It is C# 3.0 feature.
   - Object Initializers let you assign values to any accessible fields or properties of an object at creation time without having to explicitly invoke a constructor.

Example:

```
//With instances and initializing an object
Employee emp=new Employee
{
        EmployeeCode=100,
        EmployeeName="Raj",
        EmployeeAge=25
} ;
```

9. **Collection Initializers:**
   - It is C#3.0 feature.
   - Collection Initializers let you specify one or more element Initializers when you initialize a collection class instead of multiple calls to add method class.

Example: Adding elements to list using Add method or Collection Initializer. Both achieve the same result.



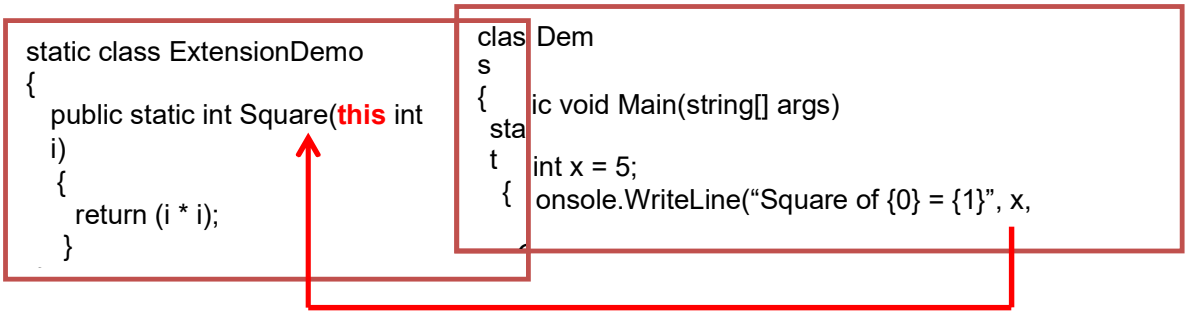
```
//using Add() to add elements to the list
List<Employee> empList=new List<Employee>();
empList.Add(new Employee(101,"Ramesh",23));
empList.Add(new Employee(102,"Ram",26));
empList.Add(new Employee(103,"Suresh",28));
```

```
//Combining object and collection initializers
List<Employee> emList=new List<Employee>
{
        new Employee{Employeecode=100,Employeename="Rajesh"},
        new Employee{Employeecode=101,Employeename="Devid"},
        new Employee{Employeecode=102,Employeename="Sriram"},
};
```

10. Extension Methods :

- It is a C# 3.0 feature.
- Give us the capabilities to asses our own custom methods to datatypes without deriving from the base class.
- Extension methods are special static methods which act like instance methods on extended types.
- Their first parameter specifies which type the method operates on. The parameter is preceded by the **this** modifier.

Example: Here, **Square** is an **Extension method** for **int** datatype. It can be accessed by instances of int datatype.



```
static class ExtensionDemo
{
   public static int Square(this int
   i)
    {
      return (i * i);
    }
```

```
clas Dem
s
{     ic void Main(string[] args)
 sta
 t    int x = 5;
   {  onsole.WriteLine("Square of {0} = {1}", x,
```

11. Anonymous Types:

- An **anonymous type** is a way to encapsulate a set of read-only properties into a single object without having to first explicitly define a type.
- Compiler generates the type name and infers the types for the properties.
- Anonymous types are created by using the new operator with an object initializer.
- Anonymous types are class types that consist of one or more public read-only properties.

Example: Here, an object is created of Anonymous type to store employee information.

```
var emp=new {
        EmployeeNumber=131
        ;
        EmployeeName="Ama
        r"; EmployeeAge=25;
    }
```

File Handling and Serialization in C#

- Generally, a file is used to store the data. The term **File Handling** refers to the various operations like creating the file, reading from the file, writing to the file, appending the file, etc.
- There are two basic operation which is mostly used in file handling is reading and writing of the file. The file becomes stream when we open the file for writing and reading.
- A stream is a sequence of bytes which is used for communication. Two stream can be formed from file one is input stream which is used to read the file and another is output stream is used to write in the file.
- Types of streams:
    - Byte
    - Character
- In C#, **System.IO** namespace contains classes which handle input and output streams and provide information about file and directory structure.
- Following are the classes available under **System.IO:**
    - Directory
    - DirectoryInfo
    - File
    - FileInfo
    - DriveInfo
    - Path
    - FileStream
    - FileSystemInfo

Here we are going to discuss about some of the important classes which are useful for writing in and reading from the text file.

FileStream Class:

- The FileStream is a class used for reading and writing files in C#. It is part of the System.IO namespace.
- To manipulate files using FileStream, you need to create an object of FileStream class.
- This object has four parameters; the **Name of the File, FileMode, FileAccess, and FileShare**.

Parameter	Description	Fields
Name of the file	Name of the file you want to work with along with its extension or the complete path of the file.	Eg: demo.txt, @"C:\demo.txt"
FileMode	It specifies which mode the file has to be opened in.	○ Open – To open an existing file ○ Create – To create a new file if the same file name already exists it will be overwritten ○ OpenOrCreate – To open a file if it exists else create new if it doesn't ○ Create – To specifically create a new file ○ Append – To open an existing file and append more information at the end of the file. If the file doesn't exist a new file will be created ○ Truncate – To open a existing file and truncate its size to Zero bytes
FileAccess	It specifies the access to the file.	○ Read – To read data from a file ○ Write – To write data to a file ○ ReadWrite – To read and write data to a file
FileShare	It specifies the access given to other FileStream objects to this particular file	○ None – To decline the sharing of the file. Any access request will fail until the file is closed. ○ Read – To allow subsequent reading of the file. ○ Write – To allow subsequent writing to the file. ○ ReadWrite – To allow subsequent reading and writing of the file. ○ Delete – To allow subsequent deleting of the file. ○ Inheritable – To allow the file handle inheritable by child processes.

StreamWriter Class

The StreamWriter class implements TextWriter for writing character to stream in a particular format. The class contains the following method which are mostly used.

Following are the methods of the StreamWriter class:

METHOD	DESCRIPTION
Close()	Closes the current StreamWriter object and stream associate with it.
Flush()	Clears all the data from the buffer and write it in the stream associate with it.
Write()	Write data to the stream. It has different overloads for different data types to write in stream.
WriteLine()	It is same as Write() but it adds the newline character at the end of the data.

Example : To write data to a file

```
string fileName = @"c:\msnet\testfile.txt";

 //filestream object to perform input output operations with files
 FileStream fs = new FileStream(fileName, FileMode.OpenOrCreate, FileAccess.Write);

  //StreamWriter class is deriving from TextWriter which is an abstract class which allows
   // to write data to the file as Character
    stream StreamWriter sw = new
    StreamWriter(fs);

   sw.Write(rtb.Text);

 //flush out the data from the buffermemory to the file
    sw.Flush();

    fs.Close();
   Console.WriteLine("File created successfully...");
```

StreamReader Class

The StreamReader class implements TextReader for reading character from the stream in a particular format. The class contains the following method which are mostly used.

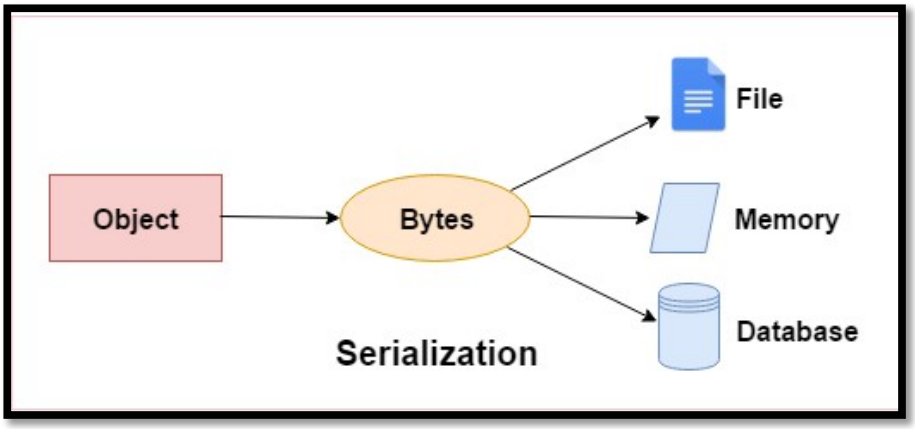
METHOD	DESCRIPTION
Close()	Closes the current StreamReader object and stream associate with it.
Peek()	Returns the next available character but does not consume it.
Read()	Reads the next character in input stream and increment characters position by one in the stream
ReadLine()	Reads a line from the input stream and return the data in form of string
Seek()	It is use to read/write at the specific location from a file

Example : Read the data from a file

```
string fileName = @"c:\msnet\testfile.txt";
 //creating Filestream to read the data of the file
 FileStream fs = new FileStream(fileName, FileMode.Open, FileAccess.Read);

  //StreamReader class inherits from abstract class TextReader it provides reading data from
  the
  //file using Character stream

  StreamReader sr = new StreamReader(fs);
//Reads entire File content and displays it on the
 Console Console.WriteLine(sr.ReadToEnd());
 fs.Close();
```

Serialization

- Serialization is the process of converting the state of an object into a form that can be persisted or transported.
- The complement of serialization is deserialization, which converts a stream into an object.
- Together, these processes allow data to be easily stored and transferred.
- In simple words serialization is a process of storing the object instance to a disk file. Serialization stores state of the object i.e. member variable values to disk. Deserialization is reverse of serialization i.e. it's a process of reading objects from a file



where they have been stored.

Uses for serialization

Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:

- Sending the object to a remote application by using a web service
- Passing an object from one domain to another
- Passing an object through a firewall as a JSON or XML string
- Maintaining security or user-specific information across applications

JSON serialization
The **System.Text.Json** namespace contains classes for **JavaScript Object Notation (JSON)** serialization and deserialization. JSON is an open standard that is commonly used for sharing data across the web.
JSON serialization serializes the public properties of an object into a string, byte array, or stream that conforms to the RFC 8259 JSON specification. To control the way JsonSerializer serializes or deserializes an instance of the class:

- Use a JsonSerializerOptions object
- Apply attributes from the System.Text.Json.Serialization namespace to classes or properties
- Implement custom converters

Binary and XML serialization
The System.Runtime.Serialization namespace contains classes for binary and XML serialization and deserialization.

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-based network streams. In binary serialization, all members, even members that are read- only, are serialized, and performance is enhanced.

XML serialization serializes the public fields and properties of an object, or the parameters and return values of methods, into an XML stream that conforms to a specific XML Schema definition language (XSD) document. XML serialization results in strongly typed classes with public properties and fields that are converted to XML.
System.Xml.Serialization contains classes for serializing and deserializing XML. You apply attributes to classes and class members to control the way the XmlSerializer serializes or deserializes an instance of the class.

Making an object serializable
For binary or XML serialization, you need:

- The object to be serialized
- A stream to contain the serialized object
- A System.Runtime.Serialization.Formatter instance

Apply the SerializableAttribute attribute to a type to indicate that instances of the type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the SerializableAttribute attribute.

To prevent a field from being serialized, apply the NonSerializedAttribute attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked SerializableAttribute, those objects will also be serialized.

Binary Serialization:

- During this process, the public and private fields of the object and the name of the class, including the assembly containing the class, are converted to a stream of bytes.
- The bytes are then written to a data stream.
- When the object is subsequently deserialized, an exact clone of the original object is created.
- All items can be serialized in binary serialization.

Example: to serialize the object of login class name and save loginname of the last logged in user

```
[Serializable] //Attribute to tell CLR object of this class can be serialized and saved
  in a file class Login
  {
     string userName;
     public string
     UserName
     {
        get { return userName;
        } set { userName =
        value; }
     }
     [NonSerialized] //specifies , password should not be serialized and not save in
     the file string password;
     public string Password
     {
        get { return password;
        } set { password =
        value; }
```

Add the following namespace to serialize the login detaisl:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

class Program
{
 static void SerializeLogin(string uname, string pwd)
 {
          Login    currLogin    =    new
           Login();
           currLogin.UserName
           =uname;
           currLogin.Password =pwd;
    //serialize and save the login details in the file
           FileStream fs = new FileStream(@"c:\logindetail.txt", FileMode.OpenOrCreate,
                                                 FileAccess.Writ
           e); BinaryFormatter bf = new BinaryFormatter();
           bf.Serialize(fs, currLogin);
           fs.Close();
     }
```

```
static void DeserializeLogin()
{
 string fileName =
        @"c:\logindetail.txt"; if
        (File.Exists(fileName))
        {
           FileStream fs = new FileStream(fileName, FileMode.Open,
           FileAccess.Read); BinaryFormatter bf = new BinaryFormatter();
           Login lastLogin =
           (Login)bf.Deserialize(fs); if (lastLogin
           != null)
           {
            Console.WriteLine("Last User Logged In was : " + lastLogin.UserName);
           }
           fs.Close();
}
static void Main()
  {
   DeserializeLogin();
    string username, password;
   Console.Write("Enter Username =
   "); username =
   Console.ReadLine();
   Console.Write("Enter Password =
   "); password =
   Console.ReadLine();
   if(username!="" || password!="")
    {
       if(password != "pass123")
       {
          Console.WriteLine("Incorrect password..try again");
       }
      els
       e
       {
         Console.WriteLine("Welcome, {0}",
         username); SerializeLogin(username,
         password);
```

Multithreading

- Multitasking is the simultaneous execution of multiple tasks or processes over a certain time interval. Windows operating system is an example of multitasking because it is capable of running more than one process at a time like running Google Chrome, Notepad, VLC player, etc. at the same time. The operating system uses a term known as a process to execute all these applications at the same time.

- A process is a part of an operating system that is responsible for executing an application. Every program that executes on your system is a process and to run the code inside the application a process uses a term known as a **thread**.
- A **thread** is a lightweight process, or in other words, a thread is a unit which executes the code under the program. So every program has logic and a thread is responsible for executing this logic.
- Every program by default carries one thread to executes the logic of the program and the thread is known as the Main Thread, so every program or application is by default single-threaded model. This single-threaded model has a drawback. The single thread runs all the process present in the program in synchronizing manner, means one after another. So, the second process waits until the first process completes its execution, it consumes more time in processing.

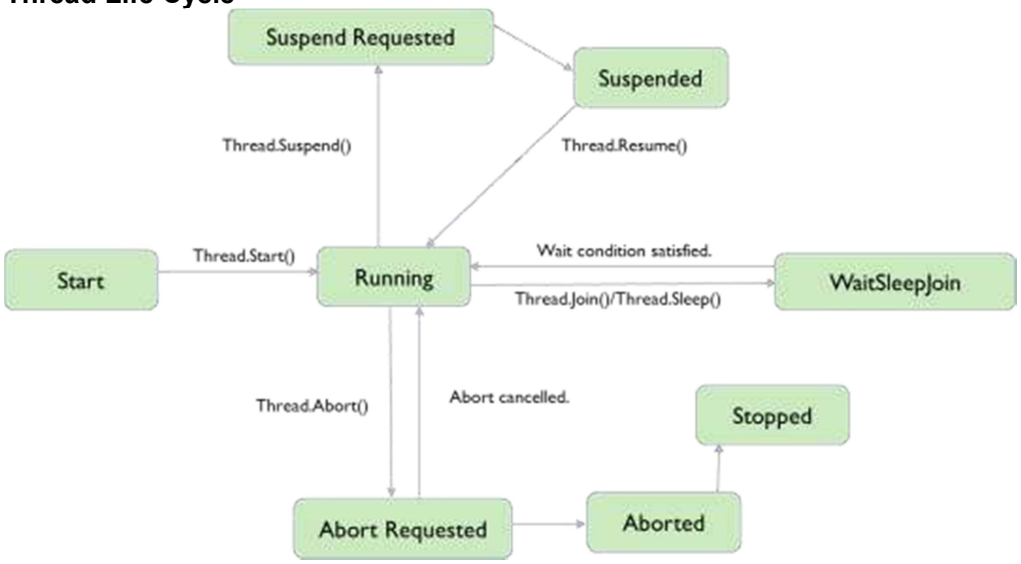
For example, we have a class named as **Program** and this class contains two different methods, i.e method1, method2. Now the main thread is responsible for executing all these methods, so the main thread executes all these methods one by one.

```
using System;
using
 System.Threading;
 class Program
 {
  static void method1()
  {
     for (int x = 1; x <= 5; x++) {
         Console.WriteLine("Method1 is : {0}",
         x);
       }
    }
  public static void method2()
  {
     for (int y = 1; y <= 10; y++) {
         Console.WriteLine("Method2 is : {0}",
         y);
     }
  }
 static public void Main()
   {
    // Calling static
    methods method1();
    method2();
   }
```

Explanation: Here, method2 will start executing only after method1 has finished executing, till then method2 has to wait. So to overcome the drawback of single threaded model multithreading is introduced.

Multi-threading is a process that contains multiple threads within a single process. Here each thread performs different activities.
For example, we have a class and this call contains two different methods, now using multithreading each method is executed by a separate thread. So the major advantage of multithreading is it works simultaneously, which means multiple tasks execute at the same time. And also maximizing the utilization of the CPU because multithreading works on time-sharing concept mean each thread takes its own time for execution and does not affect the execution of another thread, this time interval is given by the operating system.

Thread Life Cycle



Threading Implementation

- Threading in .NET is implemented using the classes under the System.Threading namespace.
- Threading can be implemented in two ways:
  - Using the **ThreadStart** delegate.
  - Using the **ThreadPool** class.
- For long running tasks it is preferable to create your own threads using the ThreadStart delegate.
- ThreadPool can be used for multithreading in two different ways:
  - **Directly**: Using ThreadPool.QueueUserWorkItem.
  - **Indirectly**: Using asynchronous methods like Stream.BeginRead() or the BeginInvoke() method of a delegate. These methods use the ThreadPool class internally.
- By default there are 25 threads in the thread pool. When all the 25 threads are in use then the request for a thread falls into a queue. Whenever any thread gets released, it takes up the item in the queue.
- Since the thread pool is being used by the .NET Framework libraries also, it should be used mainly for tasks running for a short time.

Controlling Threads

We need to maintain explicit control over a thread when creating and maintaining it.

- To switch between the different states of the thread lifecycle, the Thread class offers the following methods:
  - Thread.Join(): This method blocks the calling thread until the thread terminates.
  - Thread.Sleep(): This method blocks the thread for a specified amount of time in milliseconds.
  - Thread.Abort(): When the Abort() method is called, the thread terminates.
  - Thread.Start(): This method starts a thread and the state of the thread changes to running.
- Some Flags / properties which help in tracking a thread's state are:
  - IsAlive: Gets the execution status of the thread.
  - ThreadState: Gets the a value containing the states of the thread.
  - IsBackground: Determines whether a thread is a background thread or not.

Example:

```
using System.Threading;

namespace ThreadingDemo
{
   class Program
   {
      static void ComputeSum()
      {
         Console.WriteLine("ComputeSum invoked on thread = "
                     +
         Thread.CurrentThread.ManagedThreadId);
         double s = 0;
         for(double i=1;i<=1000000000;i++)
         {
            s = s + i;
         }
         Console.WriteLine("Sum = " + s);
      }
      static void Display()
      {
         Console.WriteLine("Display invoked on thread = "
         +Thread.CurrentThread.ManagedThreadId);

         for (int i=1;i<=10;i++)
         {
            Console.WriteLine("i = " +
            i); Thread.Sleep(1000);
         }
      }
```

```
static void Main(string[] args)
    {
       Console.WriteLine("Main invoked on thread = " +
       Thread.CurrentThread.ManagedThreadId);

       ThreadStart ts = new
       ThreadStart(ComputeSum); Thread t1 = new
       Thread(ts);
       //same as above statements
       Thread t2 = new
       Thread(Display);

       t1.Priority =
       ThreadPriority.Highest;
       t2.Priority =
       ThreadPriority.Highest;

       //invoke the
       thread t1.Start();
       t2.Start();
  //make the Main thread wait till the thread t1,t2 are
       finished t1.Join();
       t2.Join();

       Console.WriteLine("Back in Main..");
       Console.WriteLine("Press enter to
       terminate"); Console.ReadLine();
    }
```

Challenges in Multi-threading:

- **Race conditions** in multi-threading can lead to unpredictable and inconsistent results. This can be avoided using locks. But locks applied to long running code blocks can degrade performance. Also, locks are the main source of deadlocks.
- **Deadlocks** (due to multi-threading) can put a system to halt for an indefinite period of time.
- Threads normally share a common set of data.

Race Condition:

- It is normal for one thread to use the data of another thread. In such a case there will be a scenario when two or more threads will race to reach a code block first. As a result the data may get manipulated on its previous value producing unexpected results. This situation is called a Race Condition.
- Race conditions not only give unexpected results, but also provide different outputs at different points in time.
- Race conditions can be avoided by applying locks on the code block causing the race condition. Locks on long running code blocks result in performance degradation. Hence locks should be applied to code blocks running for a short period of time.

Synchronization Using the C# lock Keyword:

- The first technique you can use to synchronize access to shared resources is the C# **lock** keyword.
- The **lock** keyword allows you to define a scope of statements that must be synchronized between threads. By doing so, incoming threads cannot interrupt the current thread, preventing it from finishing its work.
- The **lock** keyword requires you to specify a token (an object reference) that must be acquired by a thread to enter within the lock scope.
- The C# **lock** statement is really just a shorthand notation for working with the **System.Threading.Monitor** class type.

Deadlocks

When each of the threads in a program is waiting for a lock to get released, the situation is termed a Deadlock.

Example: Here, deposit and withdraw is performed on a same account object. To ensure that both the operations occur in a synchronized manner we implement lock on the methods.

```
using System.Threading;
```

```
class Account
  {
     int balance;
     public
     Account()
     {
        balance = 1000;
     }
     public void Deposit()
     {
        lock (this)
        {
       Console.WriteLine("Deposit invoked on thread = "+
         Thread.CurrentThread.ManagedThreadId); Console.Write("Enter amount to deposit =
         ");
           int amt =
           Convert.ToInt32(Console.ReadLine());
           balance = balance + amt;
           Console.WriteLine("Existing balance after deposit = " + balance);
```

```
//here, we use Monitor class to implement lock
 public void Withdrawl()
      {
         Monitor.Enter(this);
    Console.WriteLine("Withdrawl invoked on thread = "
         +Thread.CurrentThread.ManagedThreadId); Console.Write("Enter amount to
         Withdraw = ");
         int amt = Convert.ToInt32(Console.ReadLine());

         balance = balance - amt;
         Console.WriteLine("Existing balance after Withdrawl = " + balance);

         Monitor.Exit(this);
      }
  }
```

Parallel Programming

Before understanding What is Parallel Computing, first let's take a look at the background of computations of a computer software and why it failed for the modern scenarios.

Computer software were written conventionally for serial computing. This meant that to solve a problem, an algorithm divides the problem into smaller instructions. These discrete instructions are then executed on Central Processing Unit of a computer one by one. Only after one instruction is finished, next one starts.

More about Parallel Programming:

- Today's personal computers and workstations have two or four cores (that is, CPUs) that enable multiple threads to be executed simultaneously.
- It is the use of multiple processing elements simultaneously for solving any problem. Problems are broken down into instructions and are solved concurrently as each resource which has been applied to work is working at the same time.
- Computers in the near future are expected to have significantly more cores.
- To take advantage of the hardware of today and tomorrow, you can parallelize your code to distribute work across multiple processors.
- However, many developers are not using the advantage of having multicore processors to develop better software applications. They follow the same as what 90s' developers did:
  creating single-threaded applications. In other words, they don't take advantage of all the extra processing power.
- In the past, parallelization required low-level manipulation of threads and locks.
- Visual Studio 2010 onwards and .NET Framework 4 enhance Parallel Programming support by providing a new runtime, new class library types, and new diagnostic tools.

Task Parallelism: Task parallelism employs the decomposition of a task into subtasks and then allocating each of the subtasks for execution. The processors perform execution of sub tasks concurrently.

Types of Parallelism

- **Coarse grain parallelism:**
  - **Large tasks, little scheduling or communication overhead.**
  - **Each service request executed in parallel, parallellism raised by many independent requests.**
  - **DoA, DoB, DoC and combine results. A, B, C large independent operations.**
  - **Applications:**
    - **Web Page rendering.**
    - **Servicing multiple web requests.**
    - **Calculating the payroll.**
- **Fine grain parallelism:**
  - **One activity broken down into a series of small co-operating parallel tasks:**
    - **F(n=0..X) can possibly be performed by many tasks in parallel for given values of n.**
    - **Input =>step 1 => step 2 =>Output.**
  - **Applications:**
    - **Route planning.**
    - **Graphical layout.**
    - **Sensor processing.**

Parallel Class

- Another abstraction of threads that is new with .NET 4 is the **Parallel** class.
- Namespace **System.Threading.Tasks**
- This class defines static methods for a ***parallel for and foreach*** .
- With the language defined for **for and foreach** , the loop is run from one thread.
- The **Parallel** class uses multiple tasks and, thus, multiple threads for this job.
- While the **Parallel.For**() and **Parallel.ForEach**() methods invoke the same method several times, **Parallel.Invoke**() allows invoking different methods concurrently.

Parallel Loops

Loops can be successfully parallelized if:

- The order of execution is not important.
- Each iteration is independent of each other.
- Each Iteration has sufficient work to compensate for scheduling overhead.

Parallel.For

Replacess a conventional for loop with a parallel loop:

- Restrictions:
  - **Only integer range.**
  - **Increments by only 1.**
  - **With the For() method, the first two parameters define the start and end of the loop.**

Parallel.For utilizes multiple tasks:

- The number of created tasks will vary based on number of cores and monitored throughput.
- A single task may execute many loop iterations.
- Each iteration of the loop results in a delegate invocation
- All the normal thread safety issues come into play.
- Heavy use of synchronization can negate the full benefit.

Parallel.ForEach

- Similar to Parallel.For but parallelizes the consumption of a IEnumerable<T>.
- IEnumerable<T> is not thread safe so access is guarded:
  - Initially one task is taken per task.
  - Attempts to learn the optionalamout to take per task,thus reducing contention.

Example: using Parallel.For and Parallel.ForEach

```
static void Display(int i)
    {
        Console.WriteLine("i={0} on thread : {1}", i,
        Thread.CurrentThread.ManagedThreadId);
     }
static void Main()
    {
      Console.WriteLine("Numbers from 1 to
      10:"); Parallel.For(1, 11, (i) => Display(i));

      string[] fruits = { "apple", "mango", "banana",
      "kiwi" }; Parallel.ForEach(fruits, f => {
        Console.Write("Thread : " +
        Thread.CurrentThread.ManagedThreadId); Console.WriteLine(" " + f);
      });
      Console.ReadLine();
    }
```

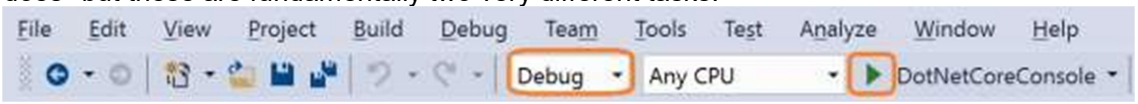
C# Debugging in Visual Studio 2019

Without fail, the code we write as software developers doesn't always do what we expected it to do. Sometimes it does something completely different! When this happens, the next task is to figure out why, and although we might be tempted to just keep staring at our code for hours, it's much easier and efficient to use a debugging tool, or debugger.

The term **debugging** can mean a lot of different things, but most literally, it means removing bugs from your code. Now, there are a lot of ways to do this. For example, you might debug by scanning your code looking for typos, or by using a code analyzer. You might debug code by using a performance profiler. Or, you might debug by using a debugger.

A **debugger** is a very specialized developer tool that attaches to your running app and allows you to inspect your code. In the debugging documentation for Visual Studio, this is typically what we mean when we say "debugging".

When you run your app in Visual Studio for the first time, you may start it by pressing the green arrow button **Start** Debugging in the toolbar (or F5). By default, the Debug value appears in the drop-down to the left. If you are new to Visual Studio, this can leave the impression that debugging your app has something to do with running your app--which it does--but these are fundamentally two very different tasks.



A Debug value indicates a debug configuration. When you start the app (press the green arrow or F5) in a debug configuration, you start the app in debug mode, which means you are running your app with a debugger attached. This enables a full set of debugging features that you can use to help find bugs in your app.

A debugger, unfortunately, isn't something that can magically reveal all the problems or "bugs" in our code. Debugging means to run your code step by step in a debugging tool like Visual Studio, to find the exact point where you made a programming mistake. You then understand what corrections you need to make in your code, and debugging tools often allow you to make temporary changes so you can continue running the program.

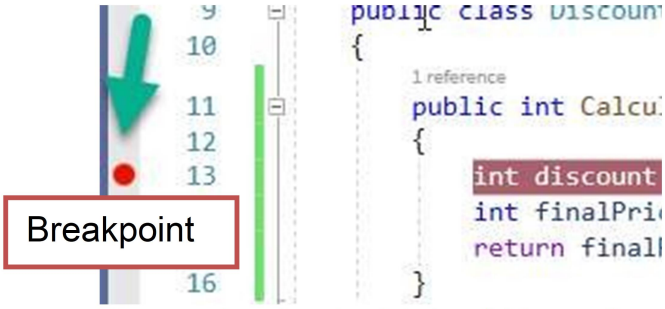
Using a debugger effectively is also a skill that takes time and practice to learn but is ultimately a fundamental task for every software developer.

Debugging comes in many forms: Stepping through the code with a debugger, investigating logs, unit testing, profiling, and analyzing dumps. As .NET developers, our main form of debugging is interactive debugging, with Visual Studio. Interactive debugging means attaching to a running
process with a debugger, and investigating the program's execution and state.

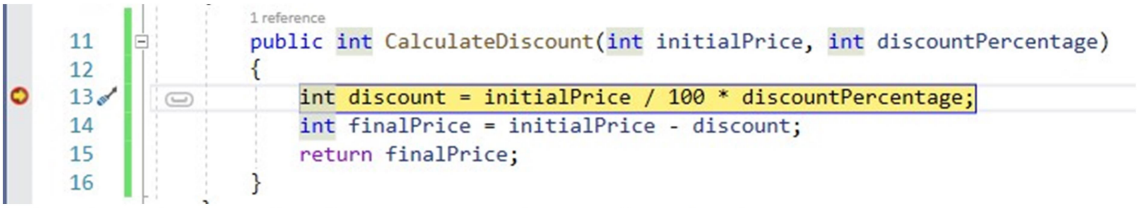
Steps to Debug the code:

1. The debugged process has to be in "Break Mode" for debugging. That means the program is currently paused on a specific line of code. More accurately, all the running threads are paused on a specific line of code.

Usually we use breakpoints because in most debugging scenarios we will want to debug when the program reaches a certain line of code.



You can place breakpoints by clicking on the margin, pressing F9, or Debug | Toggle Breakpoint. Once set, when your program reaches that line of code, Visual Studio's debugger will stop execution and enter "Break Mode":

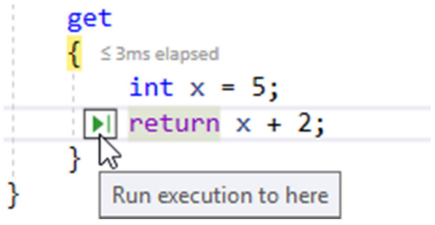


In break mode, the yellow line represents the next line of code to execute.

2. When in break mode, you can debug interactively and see how your execution of code progresses. The basic features of code navigation are:

    a. **Continue (F5)** will quit break mode and continue the program's execution until the next breakpoint is hit, entering break-mode again.
    b. **Step Over (F10**) will execute the current line and break on the next line of code.
    c. **Step Into (F11**) is used when the next execution line is a method or a property. When clicked, the debugger will step into the method's code first line. By default, properties are skipped. To enable stepping into property code go to **Tools | Options | Debugging** and uncheck Step over properties and operators.
    d. **Run** execution to here allows you to continue execution, and break in a specified location without a breakpoint. It's like creating a breakpoint and removing it after first break. You can do it in 3 ways:

        - Hover and click on the green arrow that appears on the start of each line



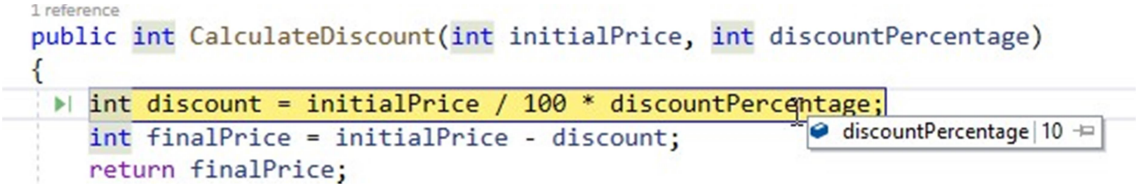
        - Stand on the desired line of code and click Ctrl + F10
        - Right click on the desired line of code and click on Set next statement

    e. Run to a cursor location allows you to forcefully set the next line of code to execute. The current (yellow) line will not be executed. It can be a line that was executed before or

after, but it's best for the new line of code to stay in the current scope. There are 2 ways to do this:

- Drag the yellow arrow to any line of code
- Stand on the desired line of code and click Ctrl+Shift+F10

3. When in break mode, you can investigate the value of local variables and class members. This is as easy as hovering over a variable:



4. Inspect variables in the Autos and Locals windows
   a. The Autos and Locals windows show variable values while you are debugging. The windows are only available during a debugging session.
   b. The Autos window shows variables used around the current breakpoint.
   c. The Locals window shows variables defined in the local scope, which is usually the current function or method.

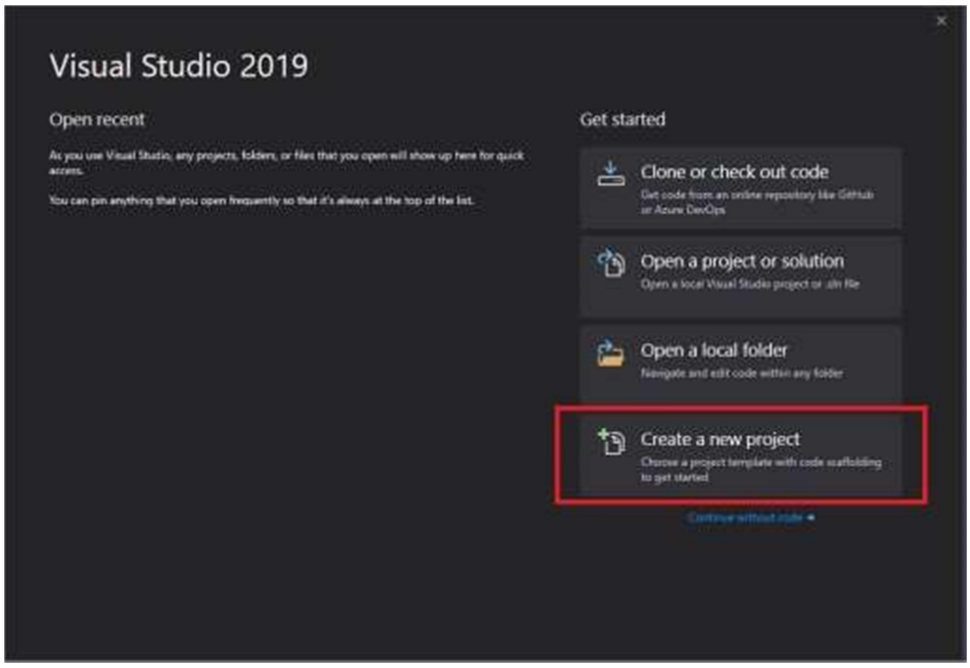
To open the Autos window, while debugging, select Debug > Windows > Autos
To open the Locals window, while debugging, select Debug > Windows > Locals, or press Alt+4.

Introduction to Windows Forms Applications

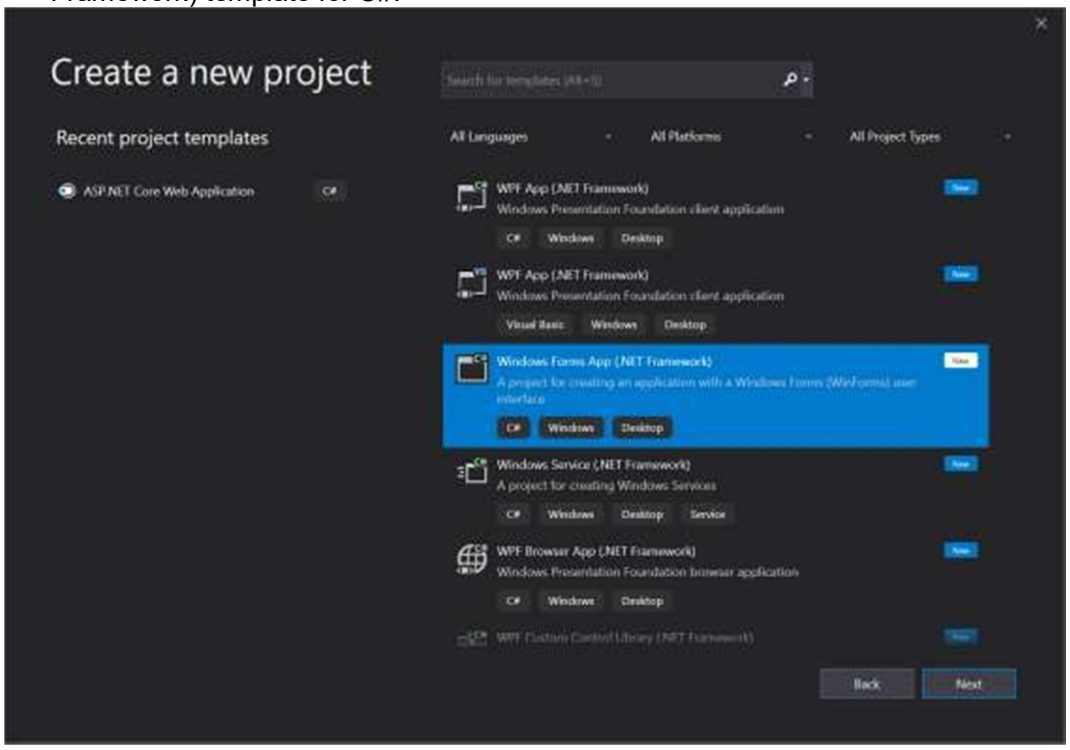
Windows Forms is a Graphical User Interface(GUI) class library which is bundled in .Net Framework. Its main purpose is to provide an easier interface to develop the applications for desktop, tablet, PCs. It is also termed as the WinForms. The applications which are developed by using Windows Forms or WinForms are known as the Windows Forms Applications that runs on the desktop computer. WinForms can be used only to develop the Windows Forms Applications not web applications. WinForms applications can contain the different type of controls like labels, list boxes, tooltip etc.

Creating a Windows Forms Application Using Visual Studio 2019

1. First, you'll create a C# application project. The project type comes with all the template files you'll need, before you've even added anything.
2. Open Visual Studio 2019.
3. On the start window, choose Create a new project.



4. On the Create a new project window, choose the Windows Forms App (.NET Framework) template for C#.



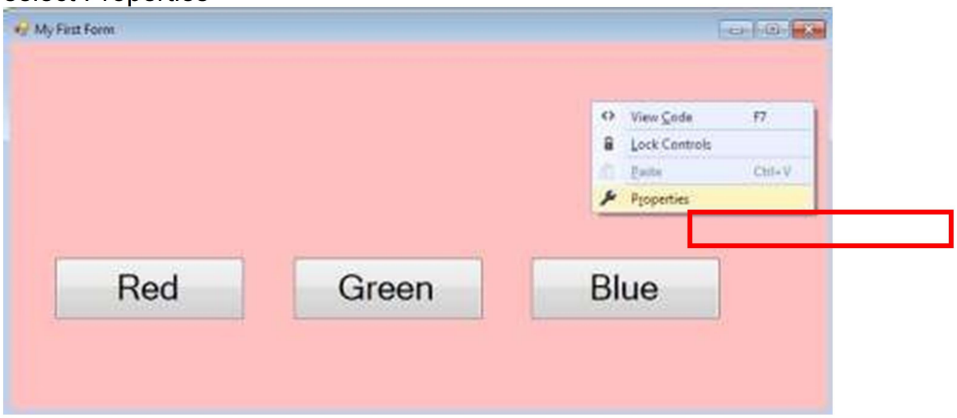
5. Provide the name for the project and click the Create button

Let's create an application which contains a Form-> Form1.cs and has 3 buttons. When the user clicks any of the buttons the backcolor of the Form needs to be changed accordingly.

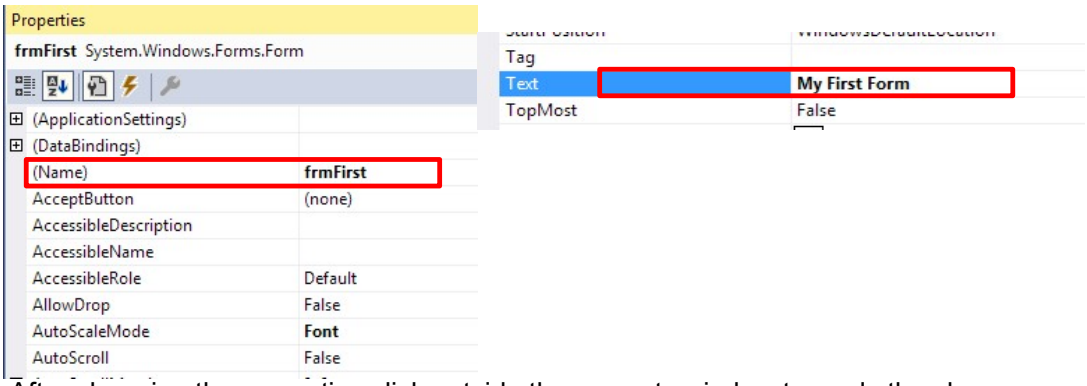
The design of the Form is as follows:

To change the properties of any control -> Right click on the control and select the Properties

- Change property of the Form: Right click on the empty area of the form and select Properties

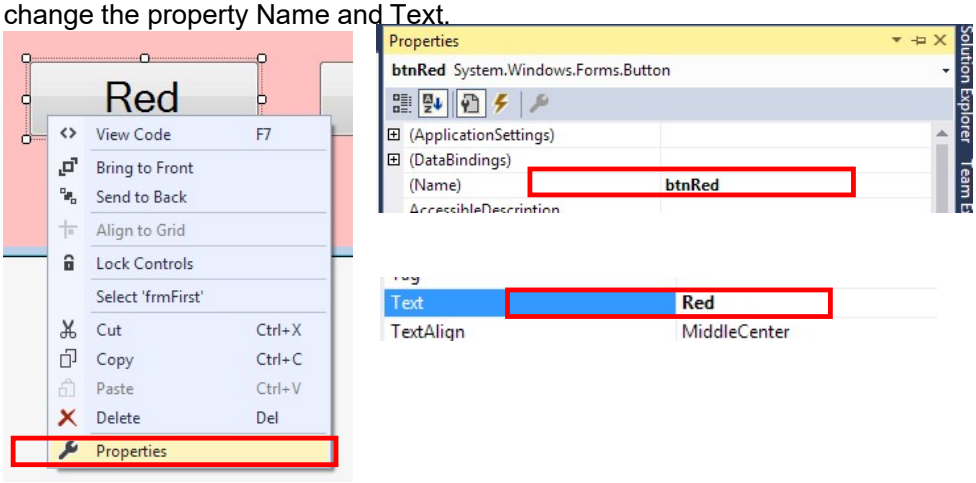


It will open the property window as follows: Change the Form Name and Text property as shown in the picture



After changing the properties click outside the property window to apply the changes.

- To change the property of the buttons, right click on respective button and change the property Name and Text.



- Do the same for the Green and Blue buttons
- Now to write the code to handle the button click event, double click on respective button and write the logic to change the background color of the form
- Example:
  Here, we double click the Red button and it will automatically generate a click event method in the code file as shown in the picture. Specify the following code in it as follows:

```
private void btnRed_Click(object sender, EventArgs e)
    {
      this.BackColor = Color.Red; //current form is referred with this keyword
    }
```

Similarly, double click the Green button and write the following code in it:

```
private void btnGreen_Click(object sender, EventArgs e)
    {
      this.BackColor = Color.Green;
    }
```

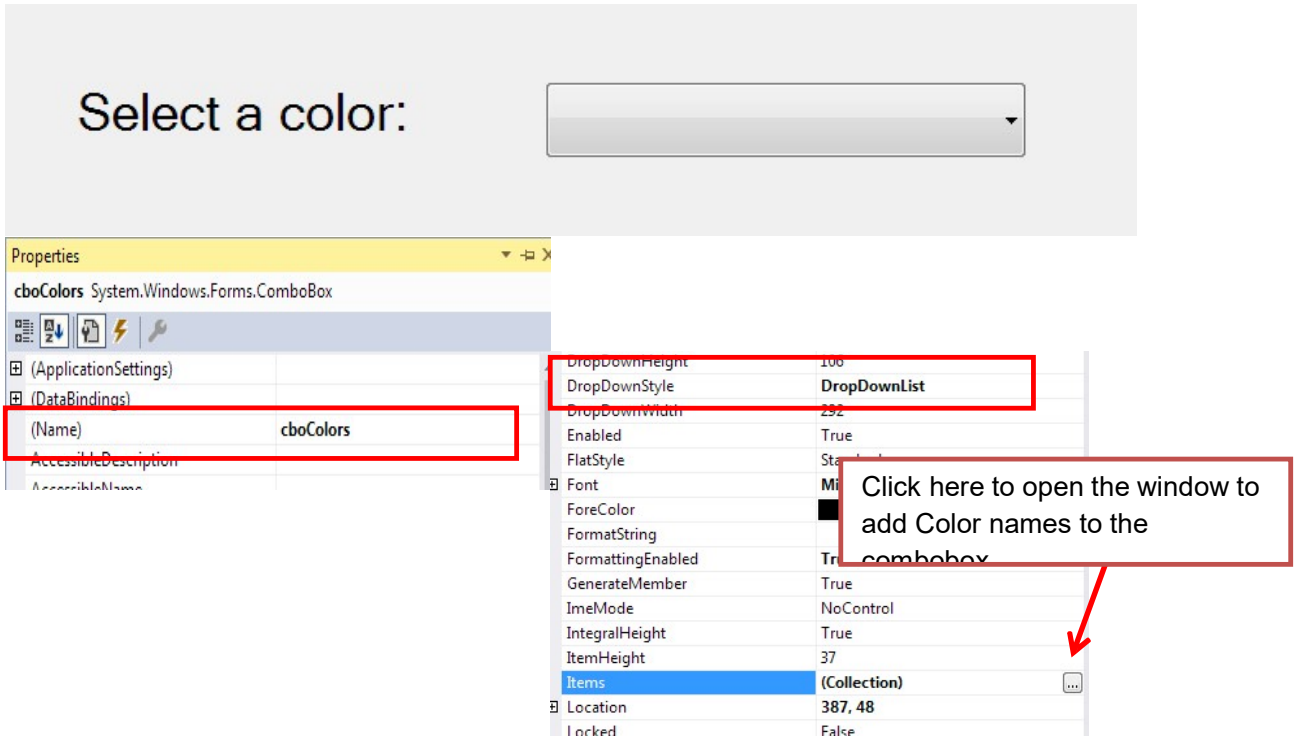
Double click the Blue button and write the following code:

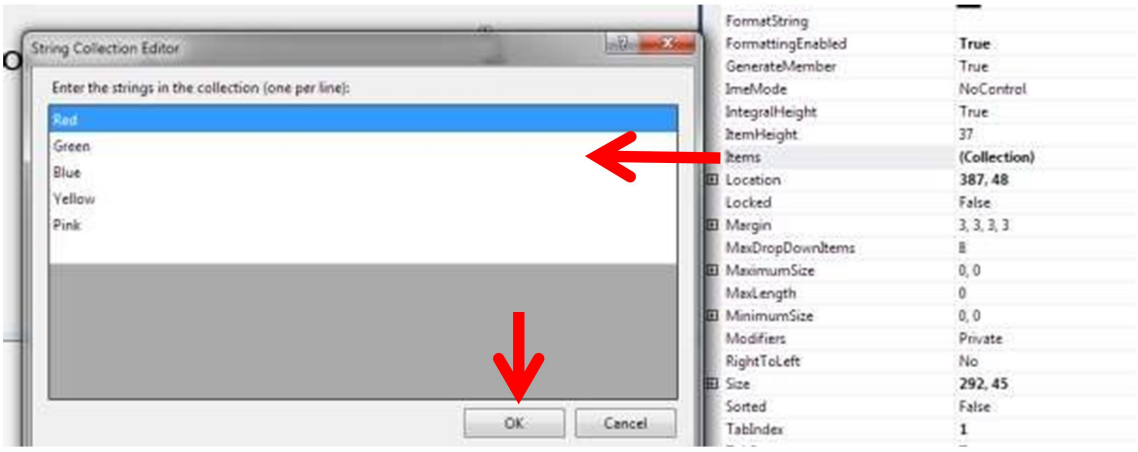
```
private void btnBlue_Click(object sender, EventArgs e)
    {
      this.BackColor = Color.Blue;
    }
```

- Now execute the form and click the buttons to change the background color of the Form

Example: **To apply background color using Combobox control**
In the Windows Forms Application project -> Project Menu -> Add Windows Forms -> Form2.cs

Add a **label** control to display a readonly text by changing the Text property and specify **Select a color.**
On the Form add a Combobox control and change its following properties





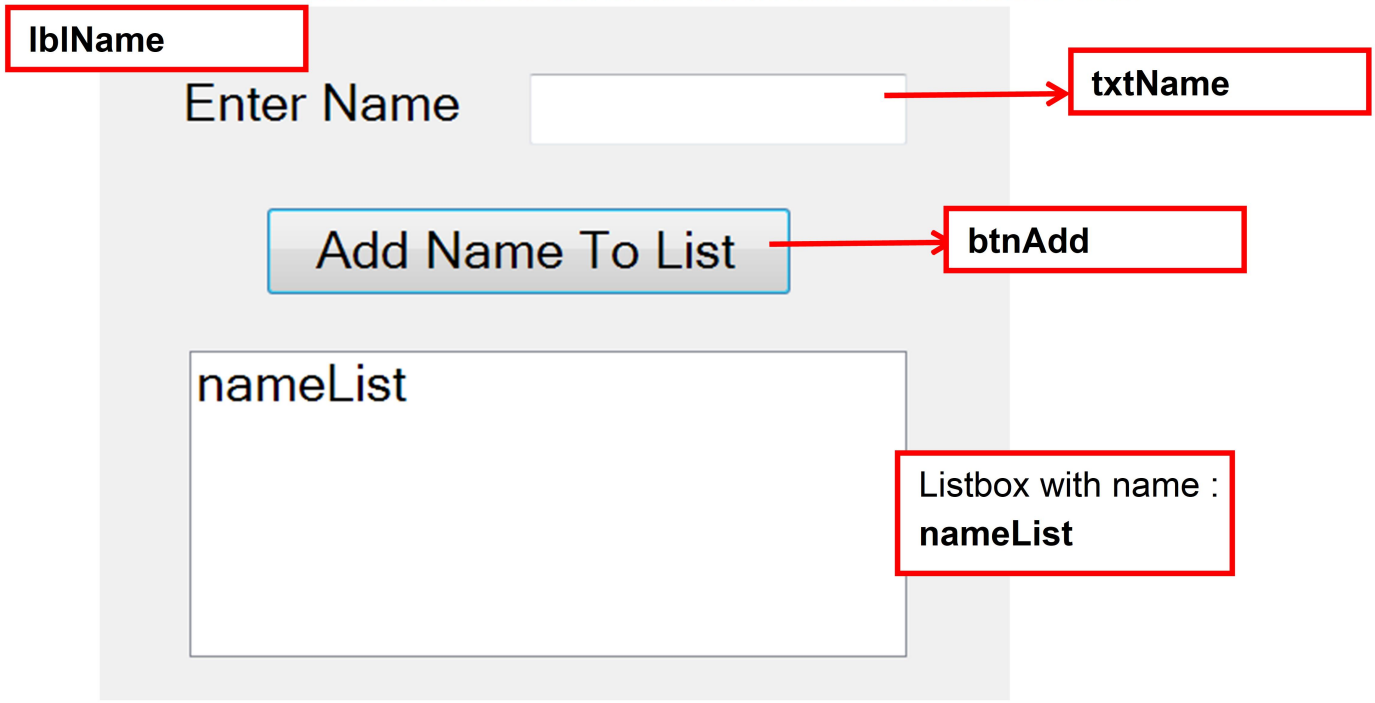
Click ok after entering all the required values.

Double click the Combobox and generate the comboBox_SelectedIndexChanged event and write the following code to retreive the selected color and apply it as the form background color.

```
private void cboColors_SelectedIndexChanged(object sender, EventArgs e)
    {
       //retreives the selected color
       name string c = cboColors.Text;
       this.BackColor =
       Color.FromName(c);
   //Here, Color is a .Net framework defined structure to assign the Color
```

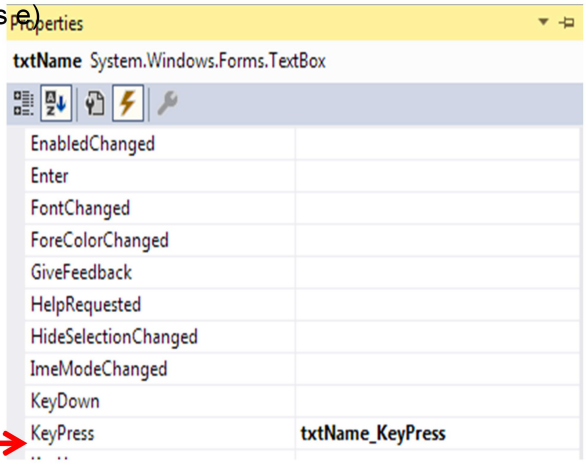
Run the form and select a color from the combobox and see the form background color changed accordingly.

Example : to allow the user to enter name in the textbox and after the user clicks the button the name should be added to the Listbox and the textbox data should be clear.



The code :

```
private void btnAdd_Click(object sender, EventArgs e)
    {
      if (txtName.Text == "")
      {
        MessageBox.Show("Provide the name..");
      }
      else
      {
        nameList.Items.Add(txtName.Text);
      }
      txtName.Clear()
      ;
      txtName.Focus(
      );
    }
 private void txtName  KeyPress(object sender,
KeyPressEventArgs e)
    {
      if (Char.IsDigit(e.KeyChar)) //not allow digit
      {

        e.Handled = true;
      }
    }
```



Double click the **KeyPress** event to **generate txtName_KeyPress** method