OVER 40 YEARS OF ACADEMIC WISDOM

GAIN MORE KNOWLEDGE REACH GREATER HEIGHTS

PRESIDENCY UNIVERSITY

.NET Full stack Development-CSE3152

Department of Computer Science Engineering

School of Engineering

Objectives

- NET Framework Fundamentals, Visual Studio IDE Fundamentals
- Introduction to .NET
- NET and .NET Framework
- Common Language Infrastructure (CLI)
- Features & Versions
- CLR Architecture
- Class Library & Assembly Manifest
- Meta data & benefits
- Application Types & Security Concepts

Introduction to .NET

- Consistent object-oriented programming environment.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To make the developer experience consistent across widely varying types of applications, such as Windows-based applications and Web-based applications.
- To build all communication on industry standards to ensure that code based on the .NET Framework can integrate with any other code.

The .NET Framework provides a comprehensive programming model for building all kinds of applications on Windows, from mobile to web to desktop.

.NET and .NET Framework

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

.NET Framework

- Microsoft .NET Framework is a complex technology that provides the infrastructure for building, running, and managing next generation applications.
- In a layered representation, the .NET Framework is a layer positioned between the Microsoft Windows operating system and your applications.
- .NET is a platform but also is defined as a technology because it is composed of several parts such as libraries, executable tools, and relationships and integrates with the operating system.
- Can be described as Development platform or Execution environment which comprises tools and technologies, to develop distributed applications and distributed web services.
- Microsoft started development on the .NET Framework in the late 1990s originally under the name of Next Generation Windows Services.
- It consists of two major components: the **Common Language Runtime** (CLR), which provides memory management and other system services, and an extensive **Class Library**, which includes tested, reusable code for all major areas of application development

CLI

- The **Common Language Infrastructure (CLI)** is an open specification developed by Microsoft and standardized by ISO and ECMA
- It describes the executable code and runtime environment that form the core of the Microsoft .NET Framework and the free and open source implementations Mono and Portable.NET.
- The specification defines an environment that allows multiple high-level languages to be used on different computer platforms without being rewritten for specific architectures.
- The CLI specification describes the following four aspects:-

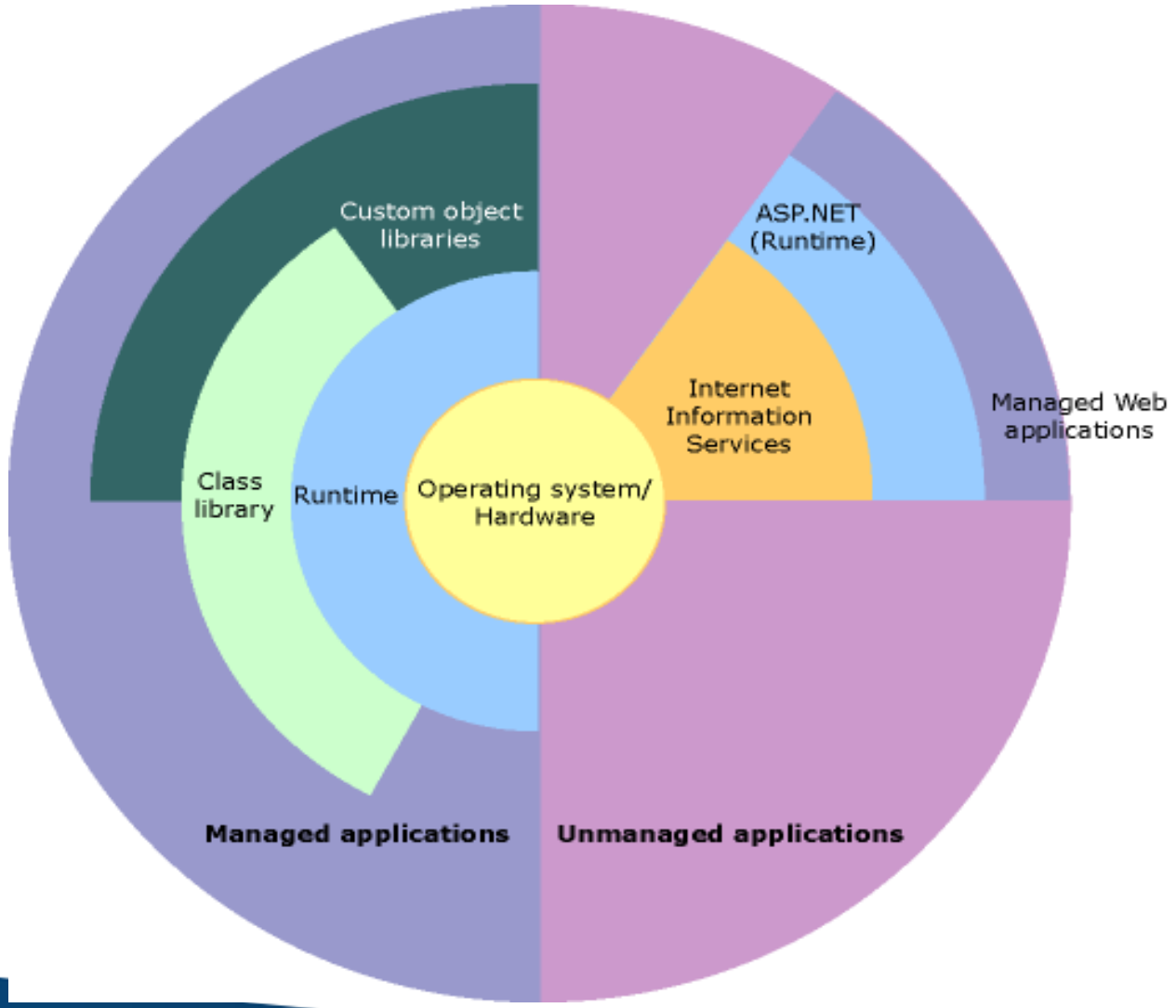
  CTS(Common Type System)

  Meta data

  CLS (Common Language specification)

  VES(Virtual Execution System)

- **CTS** : A set of data types and operations that are shared by all CTS-compliant programming languages.
- **METADATA:** It refers to certain data structures embedded within the Common Intermediate Language code that describes the high-level structure of the code.
- Metadata describes all classes and class members that are defined in the assembly, and the classes and class members that the current assembly will call from another assembly.
- The metadata for a method contains the complete description of the method, including the class (and the assembly that contains the class), the return type and all of the method parameters.
- **Common Language Specification (CLS)** A set of base rules to which any language targeting the CLI should conform in order to interoperate with other CLS-compliant languages. The CLS rules define a subset of the Common Type System.
- **VES :** The VES loads and executes CLI-compatible programs, using the metadata to combine separately generated pieces of code at runtime


Custom object libraries
ASP.NET (Runtime)
Internet Information Services
Managed Web applications
Class library
Runtime
Operating system/ Hardware
Managed applications
Unmanaged applications

Features

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

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

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

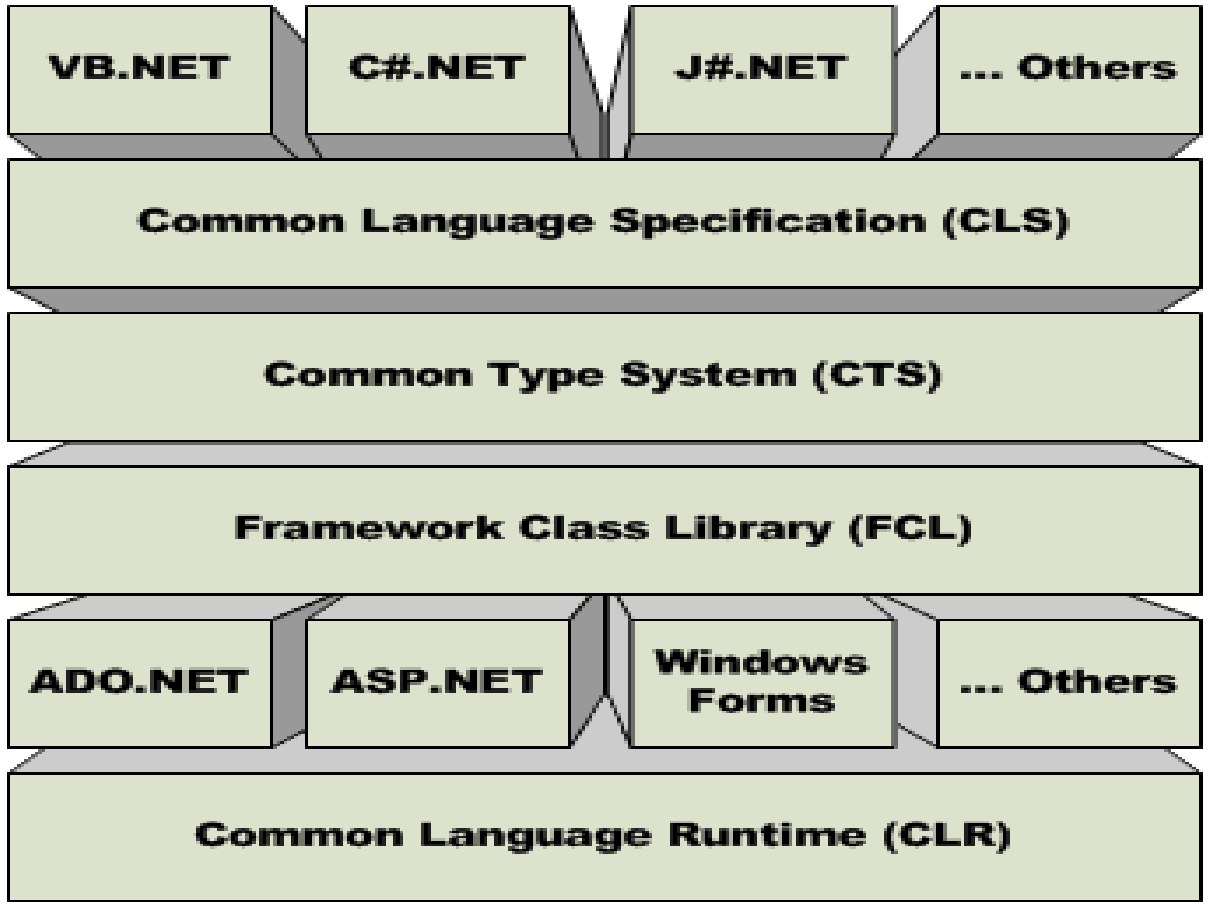
- **Development frameworks and technologies**. The .NET Framework includes libraries for specific areas of application development, such as ASP.NET for web applications, ADO.NET for data access, and Windows Communication Foundation for service-oriented applications.

- **Language interoperability**. Language compilers that target the .NET Framework emit an intermediate code named Common Intermediate Language (CIL), which, in turn, is compiled at run time by the common language runtime. With this feature, routines written in one language are accessible to other languages, and programmers can focus on creating applications in their preferred language or languages.
- **Version compatibility**. With rare exceptions, applications that are developed by using a particular version of the .NET Framework can run without modification on a later version.
- **Side-by-side execution**. The .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 applications can also coexist, and that an application can run on the version of the .NET Framework with which it was built.
- **Multitargeting.** By targeting the .NET Framework Portable Class Library, developers can create assemblies that work on multiple .NET Framework platforms, such as the .NET Framework, Silverlight, Windows Phone 7, or Xbox 360.

Versions

- Version 1.0 was the first release and was included in the Visual Studio .NET . – add on support for ODBC and Oracle
- Version 1.1 was released after the first release and Microsoft changed the Visual Studio to Visual studio .NET 2003 with the enhanced features. It was the default framework in Windows Server 2003 – Internet protocol V6 support
- Version 2.0 was released with the Visual Studio 2005 in the year 2005 – Generics
- 3.0 was released with the same visual studio of the previous one and was integrated with Windows Vista and Windows Server 2008 – WPF,WCF and WF
- 3.5 – AJAX - Visual Studio 2008
- 4.0 – Visual Studio 2010
- 4.5 – Visual Studio 2012
- 4.5.1 , 4.5.2 - Visual Studio 2013
- 4.6,4.6.1,4.6.2 – Visual studio 2015
- 4.7, 4.7.1, 4.7.2 - Visual studio 2017
- 4.8 - Visual Studio 2019 Visual studio 2022

.NET Architecture



The .NET Framework has two main components:

The .NET Framework has two main components: the common language runtime and the .NET Framework class library.

- The **common language runtime** is the foundation of the .NET Framework

- Runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness.

- In fact, the concept of **code management** is a fundamental principle of the runtime.

- Code that targets the runtime is known as managed code, while code that does not target the runtime is known as unmanaged code.

- The **class library**, the other main component of the .NET Framework, is a comprehensive, applications ranging from traditional command-line or graphical user interface (GUI) applications to applications based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

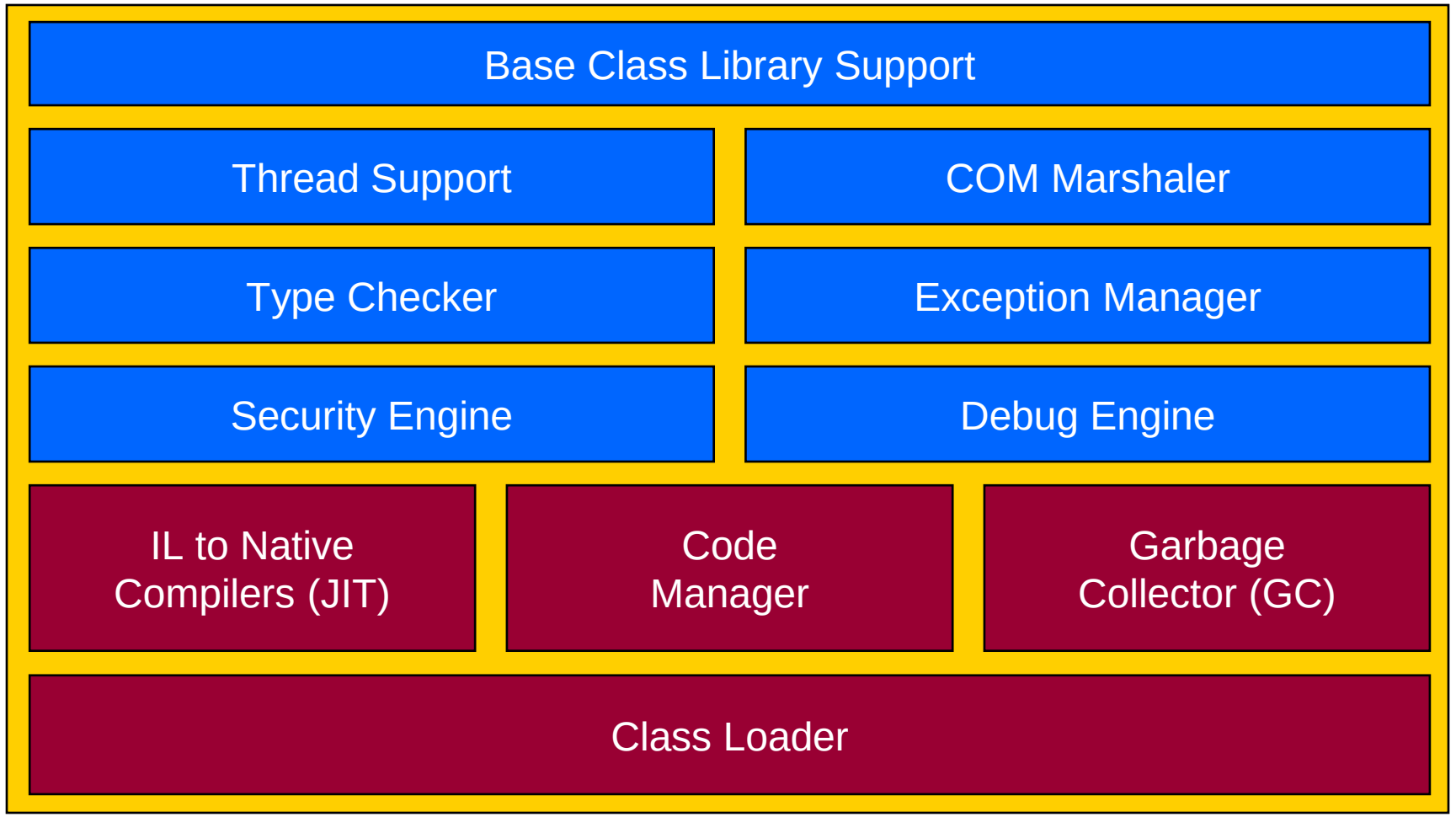
CLR

- The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services.
- It is an execution environment for program code defined by CLI.
- It lies between operating systems and applications written in .NET languages.
- The runtime enforces code access security
- The runtime also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing.
- Automatic memory management : CLR provides the garbage collection. The objects whose lifetime is managed by the garbage collection are called managed data.
- The runtime is designed to enhance performance.

- Platform independence: When you compile a program developed in a language that targets the CLR, the compiler translates the code into an intermediate language ie CPU-independent. The code can be executed from any platform that supports the .NET CLR.
- Security management : It is achieved through the code access Security model. In this, CLR enforces restrictions on managed code through the use of objects called permissions. It specifies what the code can access instead of specifying who can access the resources.
- Language interoperability: It is the ability of an application

  to interact with another application written in a different programming language. It helps maximize code reuse.

CLR Architecture



CLR Architecture

Different components :

- ***Class Loader*** : C# compiler of the framework generates assembly after compiling and this consists of CIL code and Meta data. The class loader is loading the assembly into the run time.
- ***Code Manager****:* This component is managing the above code during execution. This takes the responsibility of allocating memory to the objects also.
- ***Garbage collector***: This provides automatic garbage collection of the object when the object Is no longer in use. To achieve this, Garbage collector is performing the periodical checks in the heap from where the object gets memory.
- ***Security Checker****:* One of the important components of CLR is security checker. This engine restricts the access to system resources such as hard disk and enforcing the restriction on the MSIL code.
- ***Debug Engine*** : Debug means finding and removing the bug from the programs, The application written in any supported framework languages are debugged by this engine

- ***Type Checker***: This ensures the datatype checking of the variable. This is also checking the valid operations on the corresponding datatype. It means that integer value must be assigned to integer datatype and valid operations are allowed in that type.Otherwise it raises the exception.
- ***Thread support:*** Multithreading is a very important feature of any programming language. Threads are playing an very important role in developing the application in this framework The application can contain one or more threads. These threads are managed by the CLR.
- ***Exception Manager*** : The Net framework follows the structured exception handling in all its compliant languages. The exception might rise from managed code as well as unmanaged code. This manager provides the support to handle these type of Exceptions.
- ***Base class library support –*** It provides the types that the applications need at run time.

Class Library

- The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime.
- The class library is object oriented, providing types from which your own managed code can derive functionality. This not only makes the .NET Framework types easy to use, but also reduces the time associated with learning new features of the .NET Framework.
- In addition, third-party components can integrate seamlessly with classes in the .NET Framework.
- We can use the same set of classes for performing a specific task in VB as well as VC++
- Using these libraries, you can accomplish different tasks such as database connectivity, string handling, input/output functionality, numerical functions and file handling.
- Library comprises of namespaces and Assemblies.

- The entire library is split into two parts: Base Class Library(BCL) and the Framework Class Library(FCL).
- Superset of BCL library is called Framework Class library.
- BCL is a standard library which is available to all languages using the framework.
- The classes in **mscorlib.dll** and some of the classes

  System.dll and System.core.dll are considered to be a part of the BCL
- The BCL classes are available in both .NET framework as well as its alternative implementations including .NET Compact Framework, Microsoft Silverlight and Mono
- .NET includes the BCL in order to encapsulate a large number of common functions, such as file reading and writing, graphic rendering,database interaction, and XML document manipulation, which makes the programmer's job easier.
- The **Framework Class Library** (**FCL**) is a standard library and one of two core components of Microsoft NET Framework. The FCL is a collection of thousands of reusable classes (within hundreds of namespaces), interfaces and value types.

Assemblies

- An assembly is a single deployable unit that contains all the information about the implementation of classes, structures and interfaces.
- The new way of packaging the executable code in the .NET framework is Assemblies.
- It is a self describing unit and all the programs in .net constructed from these assemblies only.
- The .EXE or .DLL is called Assembly.
- It stores all the information about itself.
- This information is called assembly metadata and includes the name and version number of the assembly, security information, information about the dependencies and a list of the files that constitute the assembly.
- Namespaces are also stored in assemblies.

- Assemblies and metadata provide the CLR with the information required for executing application.
- Assemblies also play an important role in deployment and versioning.
- Assemblies can be .DLL or .EXE but they differ in their content.
- It also consists of assembly metadata known as Assembly Manifest.
- Every assembly has only one entry point.
- Information about the assembly version, identity (name), resources needed by the assembly and the scope of the assembly are contained in the manifest file.
- CLR loads and executes the IL code which is in portable executable file only if assembly contains the manifest.

- Assemblies can be private or shared.
- Private assembly can be accessible by a single application. On other way, Shared one is shared by multiple application.
- Depends upon the creation, Assemblies are classified into two namely static and dynamic, Static Assembly is created when we compile our code using compiler.
- It is stored in the hard disk in the form portable executable file. The dynamic one will be created in the fly during run time and it occupies only memory.
- The assembly can be stored in a single file or multiple files. Single file means that the CIL code, meta data, resources and Assembly metadata all are in single file only.
- On the other way, they can be separated in different file. That is called multiple file assembly.
- Assembly linker is used to link all the modules and resource files in the case of multiple structures.
- Sharing of the assemblies by multiple application is also possible by adding assemblies to the Global Assembly Cache.

Assembly Manifest

•Every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other.

• The assembly manifest contains this assembly metadata.

• An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes.

• It contains the assembly name, version number, culture, and strong name information that makes up the assembly's identity.

• It also contains list of all files in the assembly, type reference information and list of other assemblies that are referenced by the assembly.

• The assembly manifest can be stored in either a PE file (an .exe or .dll) with intermediate language (IL) code or in a standalone PE file that contains only assembly manifest information.

Functions of Manifest

- Enumerates the files that make up the assembly.
- Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.
- Enumerates other assemblies on which the assembly depends.
- Provides a level of indirection between consumers of the assembly and the assembly's implementation details.
- Renders the assembly self-describing.

Metadata

- In the past, a software component (.exe or .dll) that was written in one language could not easily use a software component that was written in another language. COM provided a step towards solving this problem.
- The .NET Framework makes component interoperation even easier by allowing compilers to emit additional declarative information into all modules and assemblies.
- This information, called metadata, helps components to interact seamlessly.
- Metadata is binary information describing your program that is stored either in a common language runtime portable executable (PE) file or in memory.
- Metadata is stored in one section of a .NET Framework portable executable (PE) file, while intermediate language (IL) is stored in another section of the PE file.
- Every type and member that is defined and referenced in a module or assembly is described within metadata.
- When code is executed, the runtime loads metadata into memory and references it to discover information about your code's classes, members, inheritance, and so on.

- The metadata portion of the file contains a series of table and heap data structures.
- The MSIL portion contains MSIL and metadata tokens that reference the metadata portion of the PE file.

Metadata stores the following information:

- Description of the assembly.
  - Identity (name, version, culture, public key).
  - The types that are exported.
  - Other assemblies that this assembly depends on.
  - Security permissions needed to run.
- Description of types.
  - Name, visibility, base class, and interfaces implemented.
  - Members (methods, fields, properties, events, nested types).
- Attributes.
  - Additional descriptive elements that modify types and members.

Benefits

- Metadata is the key to a simpler programming model, and eliminates the need for Interface Definition Language (IDL) files, header files, or any external method of component reference.
- Self-describing files -A module's metadata contains everything needed to interact with another module.
- Language interoperability and easier component-based design -Metadata provides all the information required about compiled code for you to inherit a class from a PE file written in a different language
- Attributes.- The .NET Framework lets you declare specific kinds of metadata, called attributes, in your compiled file.

Application Types

CLR supports the following type of applications:

- ASP.NET Web applications: These include dynamic and data driven browser based applications.
- Windows Form based applications: These refer to traditional rich client applications.
- Console applications: These refer to traditional DOS kind of applications like batch scripts.
- Component Libraries: This refers to components that typically encapsulate some business logic.

- Windows Custom Controls: As with traditional ActiveX controls, you can develop your own windows controls.
- Web Custom Controls: The concept of custom controls can be extended to web applications allowing code reuse and modularization.
- Web services: They are “web callable” functionality available via industry standards like HTTP, XML and SOAP.
- Windows Services: They refer to applications that run as services in the background. They can be configured to start automatically when the system boots up.

Security Concepts

- The Microsoft .NET Framework offers security transparency, code access security and role-based security to help address security concerns.
- It enables components to determine what users are authorized to do.
- These security mechanisms use a simple, consistent model so that developers familiar with code access security can easily use role-based security, and vice versa.
- Both code access security and role-based security are implemented using a common infrastructure supplied by the common language runtime.
- There are two kinds of Security permissions, and each has a specific purpose:

Code Access Permissions : represent access to a protected resource or the ability to perform a protected operation.

Role-Based Security Permissions : provide a mechanism for discovering whether a user (or the agent acting on the user's behalf) has a particular identity or is a member of a specified role.

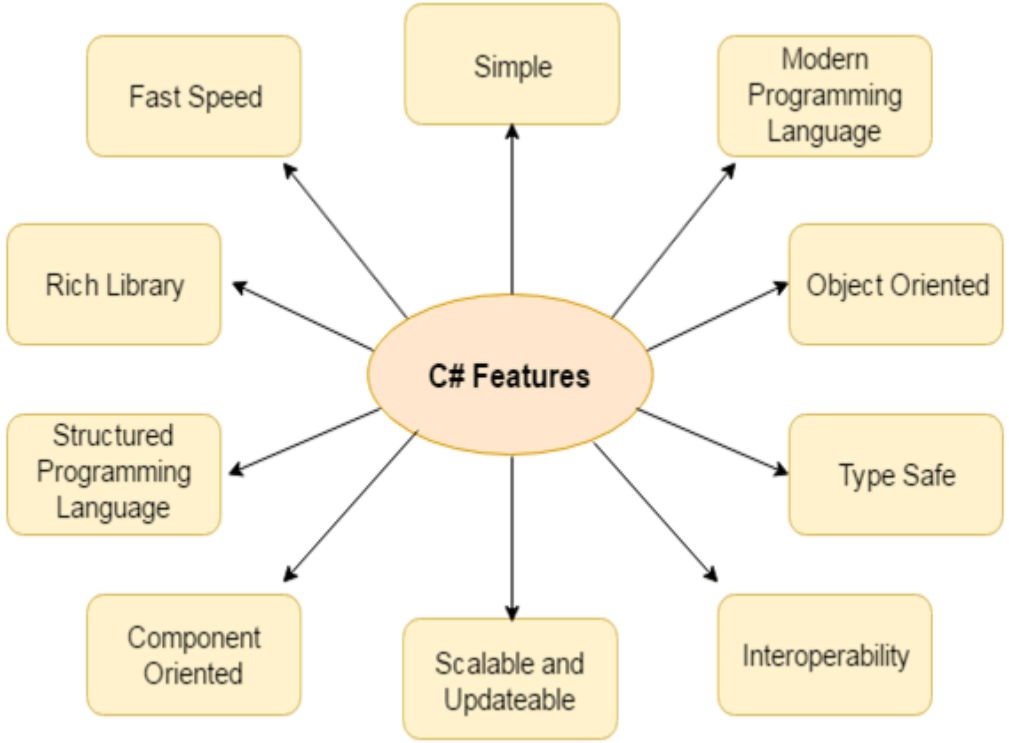
- Type-safe code accesses only the memory locations it is authorized to access. For example, type-safe code cannot read values from another object's private fields. It accesses types only in well-defined, allowable ways.

- Authentication is the process of discovering and verifying the identity of a principal by examining the user's credentials and validating those credentials against some authority.
- The information obtained during authentication is directly usable by your code.
- Authorization is the process of determining whether a principal is allowed to perform a requested action.
- Authorization occurs after authentication and uses information about the principal's identity and roles to determine what resources the principal can access.
- **For both cases, role based security can be used.**

C# Features

C# is object oriented programming language. It provides a lot of **features** that are given below.

1.Simple
2.Modern programming language
3.Object oriented
4.Type safe
5.Interoperability
6.Scalable and Updateable
7.Component oriented
8.Structured programming language
9.Rich Library
10.Fast speed



1) Simple
C# is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Modern Programming Language
C# programming is based upon the current trend and it is very powerful and simple for building scalable, interoperable and robust applications.

3) Object Oriented
C# is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grow.

4) Type Safe
C# type safe code can only access the memory location that it has permission to execute. Therefore it improves a security of the program.

5) Interoperability
Interoperability process enables the C# programs to do almost anything that a native C++ application can do.

6) Scalable and Updateable
C# is automatic scalable and updateable programming language. For updating our application we delete the old files and update them with new ones.

7) Component Oriented
C# is component oriented programming language. It is the predominant software development methodology used to develop more robust and highly scalable applications.

8) Structured Programming Language
C# is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

9) Rich Library
C# provides a lot of inbuilt functions that makes the development fast.

10) Fast Speed
The compilation and execution time of C# language is fast.

C# Arrays And Collections

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.** array index starts from 0. We can store only fixed set of elements in C# array.

EX: int[] arr = **new int**[5]; or **int**[] arr = **new int**[5]{ 10, 20, 30, 40, 50 }; or **int**[] arr = { 10, 20, 30, 40, 50 };

Advantages of C# Array

- Code Optimization (less code)
- Random Access
- Easy to traverse data
- Easy to manipulate data
- Easy to sort data etc.

Disadvantages of C# Array

- Fixed size

C# Array Types

There are 3 types of arrays in C# programming:

- Single Dimensional Array
- Multidimensional Array
- Jagged Array

C# Variables

Variables are containers for storing data values.
There are different types of variables

- int - stores integers (whole numbers), without decimals, such as 123 or -123
- double - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- string - stores text, such as "Hello World". String values are surrounded by double quotes
- bool - stores values with two states: true or false

- int myNum = 15;
  Console.WriteLine(myNum);
- string name = "John";
  Console.WriteLine(name);
- int myNum = 5;
- double myDoubleNum = 5.99D;
- char myLetter = 'D';
- bool myBool = true;
- string myText = "Hello";

Constants

This will declare the variable as "constant", which means unchangeable and read-only

The *'const'* keyword is useful when you want a variable to always store the same value, so that others (or yourself) won't mess up your code.

An example that is often referred to as a constant, is PI (3.14159...).

Note: You cannot declare a constant variable without assigning the value. If you do, an error will occur: A const field requires a value to be provided.

```
const int myNum = 15;
myNum = 20; // error
```

Operators

An operator is a program element that is applied to one or more operands in an expression or statement.

- Operators that take one operand, such as the increment operator (++) or new, are referred to as unary operators.
- Operators that take two operands, such as arithmetic operators (+,-,*,/), are referred to as binary operators.
- One operator, the conditional operator (?:), takes three operands and is the sole ternary operator in C#.

Postfix increment X++ will add 1 to x

```
var x = 42;
x++;
Console.WriteLine(x); // 43
```

Postfix decrement X-- will subtract one

```
var x = 42;
x--;
Console.WriteLine(x); // 41
```

? : Ternary Operator

Returns one of two values depending on the value of a Boolean expression.

Syntax:

condition ? expression_if_true : expression_if_false;

string name = "Frank"; Console.WriteLine(name == "Frank" ? "The name is Frank" : "The name is not Frank");

Assignment operator '='

The assignment operator = sets thr left hand operand's value to the value of right hand operand, and return that

value:

int a = 3; // assigns value 3 to variable a

int b = a = 5; // first assigns value 5 to variable a, then does the same for variable b

Console.WriteLine(a = 3 + 4); // prints 7

Decision and iteration statements

Programming in general often requires a decision or a branch within the code to account for how the code operates under different inputs or conditions.

- If-Else Statement

```
static void PrintPassOrFail(int score)
{
 if (score >= 50) // If score is greater or equal to 50
 {
 Console.WriteLine("Pass!");
 }
 else // If score is not greater or equal to 50
 {
 Console.WriteLine("Fail!");
 }
}
```

- If-Else-if Statement

Following on from the If-Else Statement example. The Else If statement follows directly after the If statement in the If-Else. If-Else structure, but intrinsically has a similar syntax as the If statement. It is used to add more branches to the code than what a simple If-Else statement can.

```
static void PrintPassOrFail(int score)
{
 if (score > 100) // If score is greater than 100
 {
 Console.WriteLine("Error: score is greater than 100!");
 }
 else if (score < 0) // Else If score is less than 0
 {
 Console.WriteLine("Error: score is less than 0!");
 }
 else if (score >= 50) // Else if score is greater or equal to 50
 {
 Console.WriteLine("Pass!");
 }
 else // If none above, then score must be between 0 and 49
 {
 Console.WriteLine("Fail!");
 }
}
```

switch case statement

is a selection statement. C# switch case statement executes code of one of the conditions based on a pattern match with the specified match expression. The C# switch statement is an alternative to using the C# if else statement when there are more than a few options.

SYNTAX:

```
switch (expression)
{
  case expression_value1:
    Statement
    break;
  case expression_value2:
    Statement
  break;
  case expression_value3:
      Statement
  break;
  default:
    Statement
  break;
}
```

// Generate a random value between 1 and 9

```
int caseSwitch = new Random().Next(1, 9);
switch (caseSwitch)
{
  case 1:
    Console.WriteLine("Case 1");
  break;
  case 2:
    Console.WriteLine("Case 2");
  break;
  case 3:
    Console.WriteLine("Case 3");
  break;
  default:
    Console.WriteLine("Value didn't match earlier.");
  break;
}
```

A class in C# is a blueprint or template that is used for declaring an object. However, there is no need to declare an object of the static class. A class Encapsulates member variables, functions, properties etc. A method is a block of code in C# programming.

- The function makes program modular and easy to understand.
- It provides reusability of code and makes c# programming more secure
- It is the basic building block of object-oriented programming

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
 namespace Creating_Class
{    class accept //Creating 1st. class
    {       public string name;
        public void acceptdetails()
        {  Console.Write("Enter your name:\t");
            name = Console.ReadLine();
        }
    }
```

```
class print // Creating 2nd class
    {       public void printdetails()
        {
            //Creating object of 1st. class
            accept a = new accept();
            //executing method of 1st class.
            a.acceptdetails();
            //Printing value of name variable
            Console.WriteLine("Your name is " + a.name);
        }
    }
```

```
class Program //Creating 3rd class
    {
        static void Main(string[] args)
        {
            print p = new print();
            p.printdetails();
            Console.ReadLine();
        }
    }
}
```

OUTPUT:

Enter your name: Kalpana.K.Raj

Your name is Kalpana.K.Raj

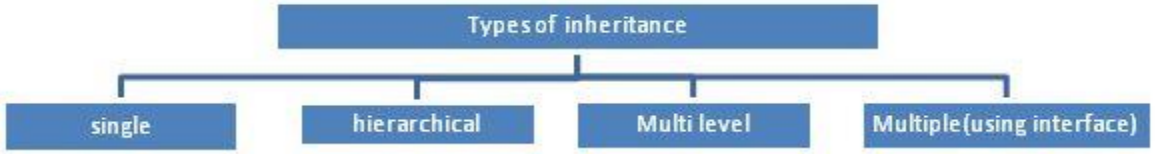
Constructor

- A constructor in C# is a member of a class.
- It is a method in the class which gets executed when a class object is created.

```
public class mySampleClass
  {
  public mySampleClass()
    {
      // This is the constructor method.
    }
  // rest of the class members goes here.
}
```

Inheritance

- Acquiring (taking) the properties of one class into another is called inheritance.
- Code reusability is one of the key features of OOPs, and it is achieved via inheritance.



Single inheritance in C#

```
class Shape //Parent class
{
    public void test()
    {
        Console.WriteLine("Parent Function");
    }
}
```

```
class Rectangle: Shape
{
    int length;
    int breadth;

    public Rectangle(int l, int b)
    {
        length = l;
        breadth = b;
    }
```

```
public int area()
  {
     return length * breadth;
  }

static void Main(string[] args)
  {
     Rectangle r = new
Rectangle(10, 10);
     r.test();
        Console.WriteLine("The area
of r is {0}", r.area());

  }
}
```

Access Modifiers

Modifier	Description
public	The code is accessible for all classes
private	The code is only accessible within the same class
protected	The code is accessible within the same class, or in a class that is inherited from that class.
internal	The code is only accessible within its own assembly, but not from another assembly.

C# Properties (Get and Set)

- The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, we must:

- declare fields/variables as '**private**'
- provide **public get and set** methods, through properties, to access and update the value of a private field

```
class Person
{
  private string name; // field
  public string Name   // property
  {
    get { return name; }
    set { name = value; }
  }
}
```

```
class Program
{
  static void Main(string[] args)
  {
    Person myObj = new Person();
    myObj.Name = "Kalpana";
    Console.WriteLine(myObj.Name);
  }
}
```

Events and Delegates:

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

Using Delegates with Events

- The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class.
- The class containing the event is used to publish the event. This is called the publisher class.
- other class that accepts this event is called the subscriber class.
- Events use the publisher-subscriber model.
- A publisher is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.
- A subscriber is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.

Declaring Events

To declare an event inside a class, first of all, you must declare a delegate type for the even as:

- **public delegate string BoilerLogHandler(string str);**

then, declare the event using the event keyword −

- **event BoilerLogHandler BoilerEventLog;**

```
using System;
namespace SampleApp
{
  public delegate string MyDel(string str);
  class EventProgram
 {
    event MyDel MyEvent;
    public EventProgram() {
      this.MyEvent += new MyDel(this.WelcomeUser);
    }
    public string WelcomeUser(string username) {
        return "Welcome To " + username;
    }
static void Main(string[] args) {
        EventProgram obj1 = new EventProgram();
        string result = obj1.MyEvent(“DNFS CLASSES");
        Console.WriteLine(result);
    }
  }
}
```

Delegates

- A Delegate is an abstraction of one or more function pointers. The .NET has implemented the concept of function pointers in the form of delegates. Delegates allow functions to be passed as parameters, returned from a function as a value, and stored in an array.
- A delegate can be defined as a delegate type. Its definition must be similar to the function signature.
- A delegate can be defined in a namespace and within a class. A delegate cannot be used as a data member of a class or local variable within a method. The prototype is:

  accessibility delegate return type delegatename(parameterlist);

  Delegate declarations look almost exactly like abstract method declarations, and replace the ‘abstract’ keyword with the ‘delegate’ keyword.

EXAMPLE:

```
using System;
namespace Delegates
{
    // Delegate Definition
    public delegate int operation(int x, int y);
    class Program
    {
        // Method that is passes as an Argument
        // It has same signature as Delegates
        static int Addition(int a, int b)
        {
            return a + b;
        }
        static void Main(string[] args)
        {
            // Delegate instantiation
            operation obj = new operation(Program.Addition);

            // output
            Console.WriteLine("Addition is={0}",obj(23,27));
            Console.ReadLine();
        }
    }
}
```

```
static void Main(string[] args)
        {
      // Delegate instantiation
       operation obj = new peration(Program.Addition);

            // output
       Console.WriteLine("Addition is={0}",obj(23,27));
            Console.ReadLine();
        }
    }
}
```

Anonymous types

- Allow us to create new types without defining them.
- The "type" of the type is decided by the compiler.
- This is way to defining read only properties into a single object without having to define type explicitly.
- The type of properties is also inferred by the compiler.

Ex:

```
var anonymousData = new {
    ForeName = “Kalpana",
        SurName = “KRaj"
};
Console.WriteLine("First Name : " + anonymousData.ForeName);
```

Anonymous Methods:

- An anonymous method in C# is a method without a name. Or We can say a code block without having a name.
- The Anonymous Methods in C# are defined using the delegate keyword and can be assigned to a variable of the delegate type.

Understand Anonymous Methods in C#:

The example shows here instead of binding the delegated with a named block (method or function), we are binding the delegate with an anonymous method (you can also unnamed code block).

```
using System;
namespace DelegateDemo
{
    public class AnonymousMethods
    {
        public delegate string GreetingsDelegate(string name);
        static void Main(string[] args)
        {
            GreetingsDelegate gd = delegate (string name)
            {
                return "Hello @" + name + " Welcome to Dotnet Classes";
            };
            string GreetingsMessage = gd.Invoke(“Kalpana");
            Console.WriteLine(GreetingsMessage);
```

Extension Method

- Extension method is a special kind of static method that is called as if it was an instance method on the extended type.
- Extension method is a static method of a static class, where the "this" modifier is applied to the first parameter.
- The type of the first parameter will be the type that is extended.
- Extension methods enable to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type

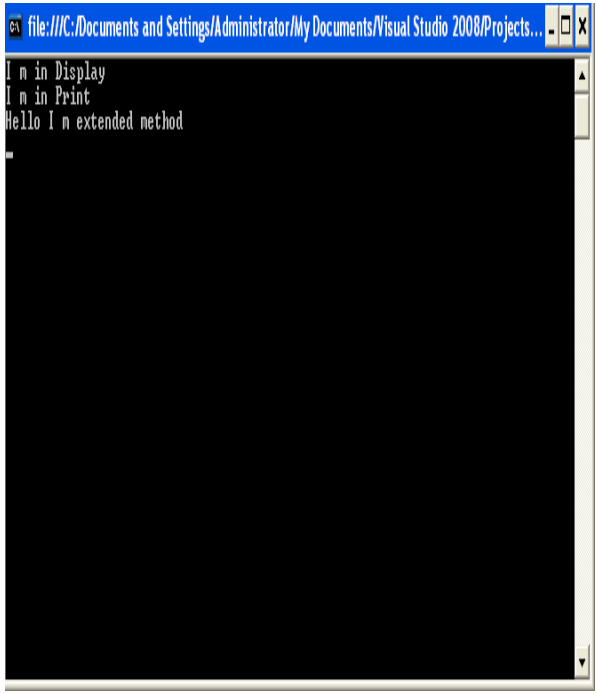
```
using System;
using System.Text;
using ClassLibExtMethod;

namespace ExtensionMethod1
{
    public static class XX
    {

        public static void NewMethod(this Class1 ob)
        {
            Console.WriteLine("Hello I m extended method");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Class1 ob = new Class1();
            ob.Display();
            ob.Print();
            ob.
                                                  y();
        }
    }
}
```

Display
Equals
GetHashCode
GetType
NewMethod
Print
ToString

(extension) void Class1.NewMethod()



Benefits of extension methods

- Extension methods allow existing classes to be extended without relying on inheritance or changing the class's source code.
- If the class is sealed, there is no concept of extending its functionality. For this, a new concept is introduced, in other words, extension methods.
- This feature is important for all developers, especially if you would like to use the dynamism of the C# enhancements in your class's design.

Sealed class:

- Sealed class is used to define the inheritance level of a class.
- class that cannot be inherited by any class but can be instantiated.
- The design intent of a sealed class is to indicate that the class is specialized and there is no need to extend it to provide any additional functionality through inheritance to override its behavior.
- We use sealed classes **to prevent inheritance**. As we cannot inherit from a sealed class, the methods in the sealed class cannot be manipulated from other classes. It helps to prevent security issues.
- Sealed class is the last class in the hierarchy.
- Sealed class can be a derived class but can't be a base class.
- A sealed class cannot also be an abstract class. Because abstract class has to provide functionality and here we are restricting it to inherit.

Sealed Methods

- Sealed method is used to define the overriding level of a virtual method.
- Sealed keyword is always used with override keyword.

Sealed Methods

- Sealed method is used to define the overriding level of a virtual method.
- Sealed keyword is always used with override keyword.

```
using System;
namespace SealedClass {
  class Animal {
    public virtual void makeSound() {
    Console.WriteLine("Animal Sound");
    }
  }
  class Dog : Animal {
    // sealed method
    sealed public override void makeSound() {
      Console.WriteLine("Dog Sound");
    }
  }
```

```
class Puppy : Dog {
    // trying to override sealed method
    public override void makeSound() {
      Console.WriteLine("Puppy Sound");
    }
  }
  class Program  {
    static void Main (string [] args) {
            // create an object of Puppy class
      Puppy d1 = new Puppy();
                          Console.ReadLine();
    }
  }
}
```

Sealed Method:

sealed keyword applies restrictions on the class and method. If you create a sealed class, it cannot be derived. If you create a sealed method, it cannot be overridden. It must be used with override keyword in method.

```
using System;
public class Animal{
    public virtual void eat() {
Console.WriteLine("eating..."); }
    public virtual void run() {
Console.WriteLine("running..."); }
 }
public class Dog: Animal
{
    public override void eat()
 { Console.WriteLine("eating bread..."); }
    public sealed override void run() {
    Console.WriteLine("running very fast...");
    }
}
```

```
public class BabyDog : Dog
{
    public override void eat()
{ Console.WriteLine("eating biscuits..."); }
    public override void run()
 { Console.WriteLine("running slowly..."); }
}
public class TestSealed
{
    public static void Main()
    {
        BabyDog d = new BabyDog();
        d.eat();
        d.run();
    }
}
```

Partial Classes:

It can break the functionality of a single class into many files. When the application is compiled, these files are then reassembled into a single class file. The partial keyword is used to build a partial class.

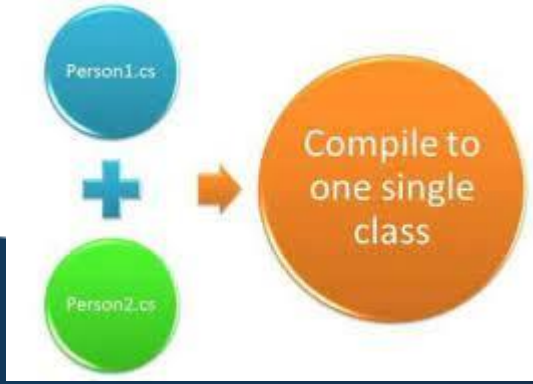
There are several situations when splitting a class definition is desirable:

- When working on large projects, spreading a class over separate files enables multiple programmers to work on it at the same time

- When working with automatically generated source, code can be added to the class without having to recreate the source file. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on.

- You can create code that uses these classes without having to modify the file created by Visual Studio.

- When using source generators to generate additional functionality in a class.



The **partial** keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the partial keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as public , private , and so on

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
        void Test2() { }
    }
}
```

Ex:

PartialClass1.cs

```
using System;

namespace PartialClasses
{
    public partial class PartialClass
    {
    public void HelloWorld()
    {
        Console.WriteLine("Hello, world!");
    }
    }
}
```

PartialClass2.cs

```
using System;

namespace PartialClasses
{
    public partial class PartialClass
    {
    public void HelloUniverse()
    {
        Console.WriteLine("Hello, universe!");
    }
    }
}
```

- Notice that both classes are defined with **the partial keyword** and have the same names.
- Also notice that each of them define a method - **HelloWorld() and HelloUniverse().**
- In this Program.cs we can now use this class as if it was defined in only one place, just like any other class:

```
using System;

namespace PartialClasses
{
    class Program
    {
    static void Main(string[] args)
    {
        PartialClass pc = new PartialClass();
        pc.HelloWorld();
        pc.HelloUniverse();
    }
    }
}
```

Threading.

Thread class is used for working with threads. It allows creating and accessing individual threads in a multithreaded application. The first thread to be executed in a process is called the main thread. When a program starts execution, the main thread is automatically created.

Thread Life Cycle

- The life cycle of a thread is started when instance of **System.Threading**. Thread class is created.
- When the task execution of the thread is completed, its life cycle is ended.

- There are following states in the life cycle of a Thread in C#.
- Unstarted
- Runnable (Ready to run)
- Running
- Not Runnable
- Dead (Terminated)
- Unstarted State

When the instance of Thread class is created, it is in unstarted state by default.

Runnable State
When start() method on the thread is called, it is in runnable or ready to run state.

Running State
Only one thread within a process can be executed at a time. At the time of execution, thread is in running state.

Not Runnable State
The thread is in not runnable state, if sleep() or wait() method is called on the thread, or input/output operation is blocked.

Dead State
After completing the task, thread enters into dead or terminated state.

Example of Thread:
One common example of use of thread is implementation of concurrent programming by modern operating systems. **Use of threads saves wastage of CPU cycle and increase efficiency of an application.**

```
using System;
using System.Threading;
namespace Demo {
   class Program {
      public static void ThreadFunc() {
         Console.WriteLine("Child thread starts");
      }
      static void Main(string[] args) {
         ThreadStart childref = new ThreadStart(ThreadFunc);
         Console.WriteLine("In Main: Creating the Child thread");
         Thread childThread = new Thread(childref);
         childThread.Start();
         Console.ReadKey();
      }
   }
}
```

Synchronous

- Synchronous represents a set of activities that starts happening together at the same time.

- A synchronous call waits for the method to complete before continuing with program flow.

How bad is it?

- It badly impacts the UI that has just one thread to run its entire user interface code.

- Synchronous behavior leaves end users with a bad user experience and a blocked UI whenever the user attempts to perform some lengthy (time-consuming) operation.

- A synchronous method call can create a delay in program execution that causes a bad user experience.
- **Hence, an asynchronous approach (threads) will be better.**

- An asynchronous method call (creation of a thread) will return immediately so that the program can perform other operations while the called method completes its work in certain situations.

What is Asynchronous Programming?

In asynchronous programming, the code gets executed in a thread without having to wait for an I/O-bound or long-running task to finish. For example, in the asynchronous programming model, the LongProcess() method will be executed in a separate thread from the thread pool, and the main application thread will continue to execute the next statement.

Microsoft recommends Task-based Asynchronous Pattern to implement asynchronous programming in the .NET Framework or .NET Core applications using async , await keywords
and Task or Task<TResult> class.

- **synchronous programming using the following console example.**

```
using System;

public class Program
{
        public static void Main(string[] args)
        {
                LongProcess();
                ShortProcess();
        }

static void LongProcess()
        {
        Console.WriteLine("LongProcess Started");
        //some code that takes long execution time
        System.Threading.Thread.Sleep(4000);
         // hold execution for 4 seconds

        Console.WriteLine("LongProcess Completed");
                }
        static void ShortProcess()
                {
        Console.WriteLine("ShortProcess Started");
                        //do something here
        Console.WriteLine("ShortProcess Completed");
                }
}
```

Asynchronous pattern using async keyword.

```
static async Task Main(string[] args)
{
    LongProcess();

    ShortProcess();
}

static async void LongProcess()
{
    Console.WriteLine("LongProcess Started");

    await Task.Delay(4000); // hold execution for 4 seconds

    Console.WriteLine("LongProcess Completed");

}
```

```
static void ShortProcess() {
    Console.WriteLine("ShortProcess Started");

    //do something here

    Console.WriteLine("ShortProcess Completed");
}
```

- **Async Method Returns Value**

```
static async Task Main(string[] args)
{
    Task<int> result = LongProcess();

    ShortProcess();

    var val = await result; // wait untill get the return value

    Console.WriteLine("Result: {0}", val);

    Console.ReadKey();
}
```

```
static async Task<int> LongProcess()
{
    Console.WriteLine("LongProcess Started");

    await Task.Delay(4000); // hold execution for 4 seconds

    Console.WriteLine("LongProcess Completed");

    return 10;   }

static void ShortProcess()
{
    Console.WriteLine("ShortProcess Started");

    //do something here

    Console.WriteLine("ShortProcess Completed");
}
```

Task based Asynchronous Pattern

The Microsoft .NET Framework 4.0 introduces a new Task Parallel Library (TPL) for parallel computing and asynchronous programming. **The namespace is "System.Threading.Tasks".**

A Task can represent an asynchronous operation and a Task provides an abstraction over creating and pooling threads.

```
Task<int> t = new Task<int>(() =>
{
    return calc(10);
});

t.Start(); Console.WriteLine("Task started");

// Wait task(s) to complete.
// After t is complete, get the result.
Task.WaitAll(t);

Console.WriteLine(t.Result);
```

- Asynchronous programming is a form of parallel programming where a set of statements run independently of the main programming flow.

- We use asynchronous programming when we have a blocking operation in the program and we want to continue with the execution of the program without waiting for the result. This allows us to implement tasks that can run at the same time.

- Asynchronous programming is about the asynchronous sequence of Tasks, while multithreading is about multiple threads running in parallel.

- Multithreading is a way of asynchrony in programming but we can also have single-threaded asynchronous tasks.

Exception Handling

- Exception handling is the method of catching and recording these errors in code so you can fix them.
- The try, catch, and finally statement in C# implements exception handling.
- The try encloses the code that might throw an exception, whereas the catch handles an exception if one exists.
- The finally is used for any cleanup work that needs to be done.

Exception Handling

```
try {
  // Statement which can cause an exception.
} catch (Type x) {
  // Statements for handling the exception
} finally {
  // Any cleanup code
}
```

Exception Handling

- If any exception occurs inside the try block, the control transfers to the appropriate catch block and later to the finally block.
- Exception Types
  - System.OutOfMemoryException
  - System.NullReferenceException
  - Syste.InvalidCastException
  - Syste.ArrayTypeMismatchException
  - System.IndexOutOfRangeException
  - System.ArithmeticException
  - System.DivideByZeroException
  - System.OverFlowException

Exception Handling

Multiple Catch Blocks

- A try block can throw multiple exceptions, which can be handled using multiple catch blocks.

Finally Block

- finally block: The finally block will always be executed whether an exception raised or not.
- Usually, a finally block should be used to release resources, e.g., to close any stream or file objects that were opened in the try block.

Collections

- Following are the Limitations of Array in C#:
  - The array size is fixed
  - We can never insert an element into the middle of an array
  - Deleting or removing elements from the middle of the array
- To overcome the above problems, the Collections are introduced
- The Collections in C# are a set of predefined classes that are present in the System.Collections namespace that provides greater capabilities and functionalities than the traditional arrays.

Collections

- Advantages
  - Size can be increased dynamically.
  - We can insert an element into the middle of a collection.
  - It also provides the facility to remove or delete elements from the middle of a collection.

Non-Generic Collections Classes in C#

- The **Non-Generic Collection Classes** come with C# 1.0 and are defined under the System.Collections namespace.
- The System.Collections namespace contains the following classes:
  - ArrayList: It Implements the System.Collections.IList interface using an array whose size is dynamically increased as required.
  - Stack: It represents a simple last-in-first-out (LIFO) non-generic collection of objects.
  - Queue: It represents a first-in, first-out collection of objects.

Collections

- Hashtable: It represents a collection of key/value pairs that are organized based on the hash code of the key.
- SortedList: It represents a collection of key/value pairs that are sorted by the keys and are accessible by key and by index.

- Syntax
  - collectionname obj_name=new collectionname;
- ArrayList
  - Arraylist is a class that is similar to an array, but it can be used to store values of various types.
  - An Arraylist doesn't have a specific size.
  - Any number of elements can be stored.

Collections

Generic Collections Classes in C#

- These collection classes 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.
- Syntax collectionname<type> obj_name=new collectionname<type>();

Working with Files

- C# File Class provides static methods for most file operations, including creating a file, copying and moving a file, deleting files, and working with FileStream to read and write streams.
- The File class is defined in the System.IO namespace.

Methods	Use
Create()	Create or overwrite a file in the specified path.
Open()	Opens a FileStream on the specified path with read / write access
WriteAllText()	Create a new file, writes the specified string to the file, and then closes the file
ReadAllText()	Opens a text file, reads all lines of the file, and then closes the file.
Copy()	Copies an existing file to a new file. Overwriting a file of the same name is not allowed.
AppendAllText()	Opens a file, appends the specified string to the file, and then closes the file. If the file does not exist, this method creates a file, writes the specified string to the file, then closes the file.

LINQ Overview

Advantages of LINQ

- LINQ offers an object-based, language-integrated way to query over data no matter where that data came from. So through LINQ we can query database, XML as well as collections.
- Compile time syntax checking
- It allows you to query collections like arrays, enumerable classes etc in the native language of your application, like VB or C# in much the same way as you would query a database using SQL

LINQ to Object

LINQ to Object provides functionality to query in-memory objects and collections. Any class that implements the IEnumerable<T> interface (in the System.Collections.Generic namespace) can be queried with SQO.

LINQ to ADO.NET

LINQ to ADO.NET deals with data from external sources, basically anything ADO.NET can connect to. Any class that implements IEnumerable<T> or IQueryable<T> (in the System.Query namespace) can be queried with SQO.

LINQ to SQL (DLinq) {Queries performed against the relation database only Microsoft SQL Server Supported}
LINQ to DataSet {Supports queries by using ADO.NET data sets and data tables}
LINQ to Entities
LINQ to XML (XLinq)

LINQ to XML is used to query XML data sources.