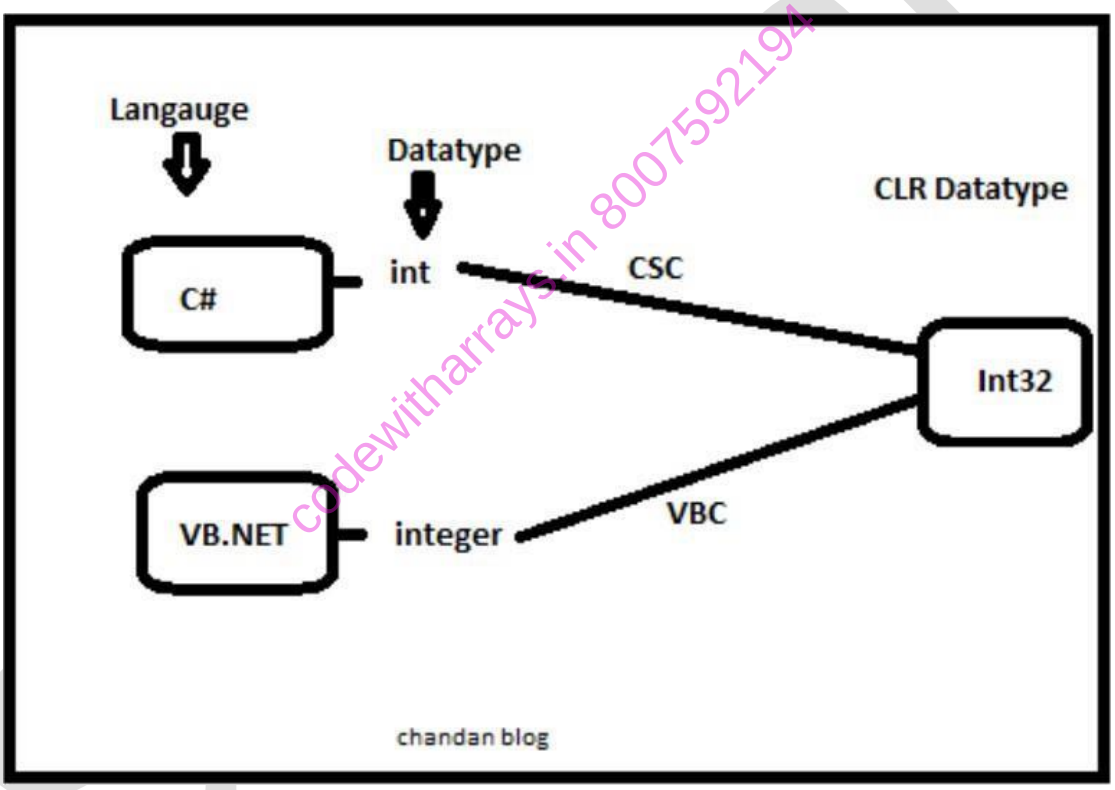
Framework Basics

CTS:-

Common Type System (CTS) describes the datatypes that can be used by managed code. CTS defines how these types are declared, used and managed in the runtime. It facilitates cross-language integration, type safety, and high-performance code execution. The rules defined in CTS can be used to define your own classes and values.

OR we can also understand like,

CTS deals with the data type. So here we have several languages and each and every language has its own data type and one language data type cannot be understandable by other languages but .NET Framework language can understand all the data types. C# has an **int** data type and VB.NET has **Integer** data type. Hence a variable declared as an int in C# and Integer in VB.NET, finally after compilation, uses the same structure Int32 from CTS.



Note

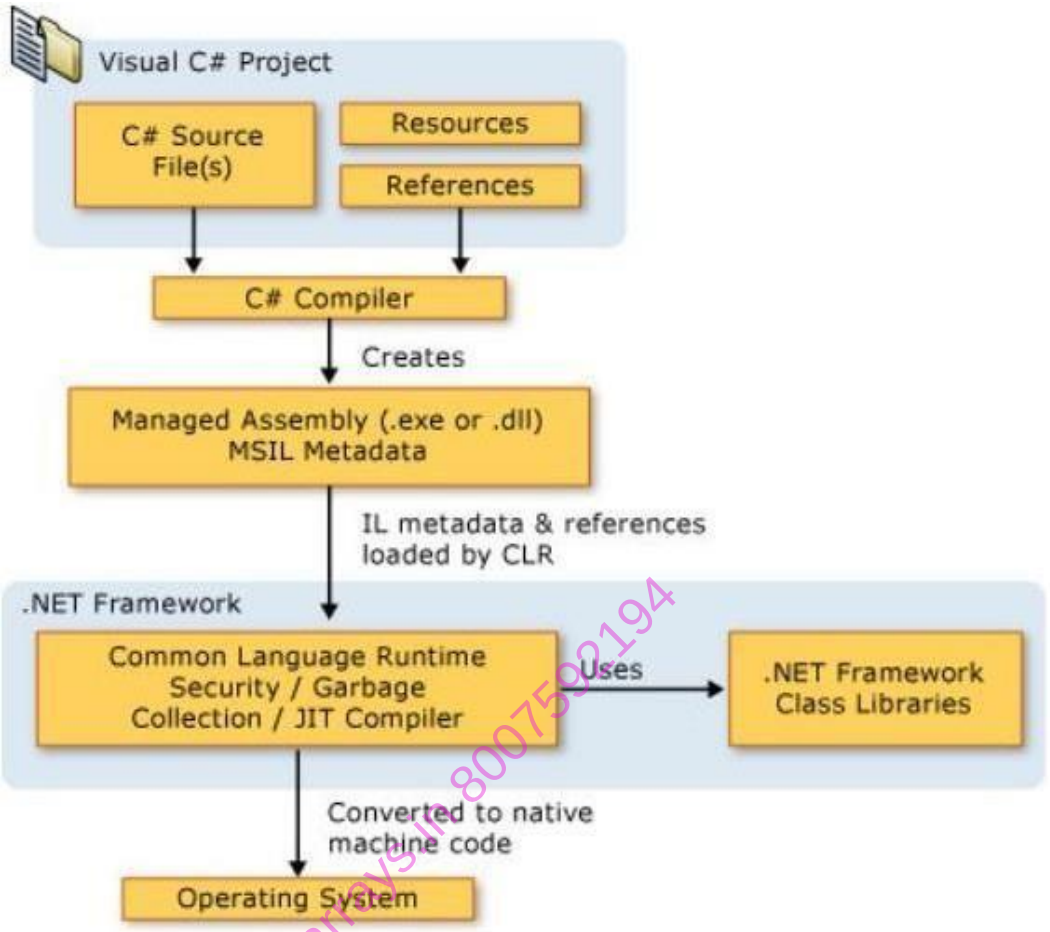
All the structures and classes available in CTS are common for all .NET Languages and the purpose of these is to support language independence in .NET. Hence it is called CTS.

Common Language Runtime (CLR) :-

The Common Language Runtime (CLR) is a core component of .NET Framework that manages the execution and the lifecycle of all .NET applications (code). It provides various services, including automatic memory management, exception handling, security, and type safety. When a .NET application is compiled, it generates an intermediate language code called Common Intermediate Language (CIL). The CLR is responsible for translating this CIL into machine code and managing the execution of the resulting program. The CLR also provides a platform for interoperability between different programming languages that target the .NET Framework. This means that a program written in one .NET language can easily use libraries written in another .NET language. Overall, the CLR is an essential component of the .NET Framework that enables developers to create robust, secure, and interoperable applications.

Key attributes of .NET CLR :-

- As part of the Microsoft .NET Framework, the Common Language Runtime (CLR) is the programming (Virtual Machine component) that manages the execution of programs written in any language that uses the .NET Framework, for example, C#, VB.Net, F# and so on.
- Programmers write code in any language, including VB.Net, C#, and F#, when they compile their programs into an intermediate form of code called CLI in a portable execution file (PE) that can be managed and used by the CLR. Then the CLR converts it into machine code to be will be executed by the processor.
- The information about the environment, programming language, its version, and what class libraries will be used for this code are stored as metadata with the compiler that tells the CLR how to handle this code.
- The CLR allows an instance of a class written in one language to call a method of the class written in another language.



As you can see from the above diagram, the CLR provides several services.

Functions of .NET CLR:-

- Convert code into CLI
- Exception handling
- Type safety
- Memory management (using the Garbage Collector)
- Security
- Improved performance
- Language independency
- Platform independency
- Architecture independency

Components of .NET CLR:-

- Performance improvements.
- The ability to easily use components developed in other languages.
- A class library provides extensible types.
- Language features such as inheritance, interfaces, and overloading for object-oriented programming.
- Support for explicit free threading that allows the creation of multithreaded, scalable applications.
- Support for structured exception handling.
- Support for custom attributes.
- Garbage collection.
- Use of delegates instead of function pointers for increased type safety and security.

IL Code:-

IL code stands for intermediate code.

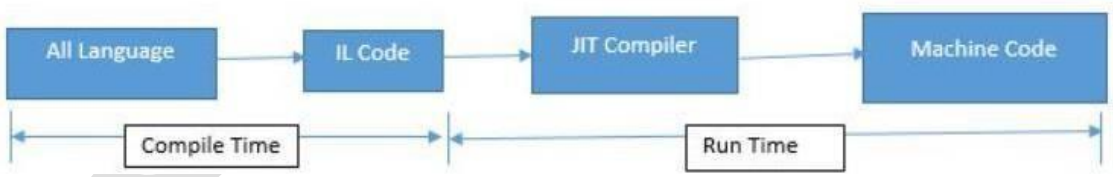
In Microsoft visual studio supports too many language. In same different user have different machine configuration and different operating system whose unknown to visual studio. That is the main problem. To avoid this problem Microsoft Creates a code that is called as IL Code.

It's called as IL code or half compiled code, it's creates at compile time.

Now why it is called as half compiled?

Visual studio does not know that what is system configuration of user, so in compile time it's compiled the language code and converted into IL code.

When this program will run at user system, this run time IL code is converted into machine code by JIT (just-in-time) compiler.



Summery

- IL code is half compiled code.
- IL Code is created at compile time.
- IL code is compiled by JIT Compiler.
- JIT Compiler converted IL Code into Machine Code.

What is a .Net Assembly?

The .NET assembly is the standard for components developed with the Microsoft.NET. Dot NET assemblies may or may not be executable, i.e., they might exist as the executable (.exe) file or dynamic link library (DLL) file. All the .NET assemblies contain the definition of types, versioning information for the type, meta-data, and manifest. The designers of .NET have worked a lot on the component (assembly) resolution.

An assembly can be a single file or it may consist of the multiple files. In the case of multi-file, there is one master module containing the manifest while other assemblies exist as non-manifest modules. A module in .NET is a subpart of a multi-file .NET assembly. Assembly is one of the most interesting and extremely useful areas of .NET architecture along with reflections and attributes.

.NET supports three kinds of assemblies:

1. private
2. shared
3. satellite

Private Assembly

Private assembly requires us to copy separately in all application folders where we want to use that assembly's functionalities; without copying, we cannot access the private assembly features and power. Private assembly means every time we have one, we exclusively copy into the BIN folder of each application folder.

Public Assembly

Public assembly is not required to copy separately into all application folders. Public assembly is also called Shared Assembly. Only one copy is required in system level, there is no need to copy the assembly into the application folder.

Public assembly should install in GAC.

Shared assemblies (also called strong named assemblies) are copied to a single location (usually the Global assembly cache). For all calling assemblies within the same application, the same copy of the shared assembly is used from its original location. Hence, shared assemblies

are not copied in the private folders of each calling assembly. Each shared assembly has a four-part name including its face name, version, public key token, and culture information. The public key token and version information makes it almost impossible for two different assemblies with the same name or for two similar assemblies with a different version to mix with each other.

GAC (Global Assembly Cache)

When the assembly is required for more than one project or application, we need to make the assembly with a strong name and keep it in GAC or in the Assembly folder by installing the assembly with the GACUtil command.

Satellite Assembly

Satellite assemblies are used for deploying language and culture-specific resources for an application.

What are the basic components of the .NET platform?

The basic components of .NET platform (framework) are:

.Net Applications *(Win Forms,Web Applications,Web Services)*
Data(ADO.Net) and XML Library
FrameWork Class Library(FCL) *(IO,Streams,Sockets,Security,Reflection,UI)*
Common Language Runtime(CLR) *(Debugger,Type Checker,JITer,GC)*
Operating System *(Windows,Linux,UNIX,Macintosh,etc.,)*

Common Language Runtime (CLR)

The most important part of the .NET Framework is the .Net Common Language Runtime (CLR) also called .Net Runtime in short. It is a framework layer that resides above the Operating System and handles/manages the execution of the .NET applications. Our .Net programs don't directly communicate with the Operating System but through CLR.

MSIL (Microsoft Intermediate Language) Code

When we compile our .Net Program using any .Net compliant language like (C#, VB.NET, C++.NET) it does not get converted into the executable binary code but to an intermediate code, called MSIL or IL in short, understandable by CLR. MSIL is an OS and H/w independent code. When the program needs to be executed, this MSIL or intermediate code is converted to binary

executable code, called native code. The presence of IL makes it possible for the Cross-Language Relationship as all the .Net compliant languages produce the similar standard IL code.

Just In Time Compilers

When our IL compiled code needs to be executed, CLR invokes JIT compilers which compile the IL code to native executable code (.exe or .dll) for the specific machine and OS. JITers in many ways is different from traditional compilers as they, as their name suggests, compile the IL to native code only when desired e.g., when a function is called, IL of function's body is converted to native code; just in time of need. So, the part of code that is not used by a particular run is not converted to native code. If some IL code is converted to native code then the next time when it's needed to be used, the CLR uses the same copy without re-compiling. So, if a program runs for some time, then it won't have any just in a time performance penalty. As JITers are aware of processor and OS exactly at runtime, they can optimize the code extremely efficiently resulting in very robust applications. Also, since JITer knows the exact current state of executable code, they can also optimize the code by in-lining small function calls (like replacing body of small function when its called in a loop, saving the function call time). Although Microsoft stated that C# and .Net are not competing with languages like C++ in efficiency, speed of execution, JITers can make your code even faster than C++ code in some cases when the program is run over an extended period of time (like web-servers).

Framework Class Library (FCL)

.NET Framework provides a huge set of Framework (or Base) Class Library (FCL) for common, usual tasks. FCL contains thousands of classes to provide them access to Windows API and common functions like String Manipulation, Common Data Structures, IO, Streams, Threads, Security, Network Programming, Windows Programming, Web Programming, Data Access, etc. It is simply the largest standard library ever shipped with any development environment or programming language. The best part of this library is they follow extremely efficient OO design (design patterns) making their access and use very simple and predictable. You can use the classes in FCL in your program just as you use any other class and can even apply inheritance and polymorphism on these.

Common Language Specification (CLS)

Earlier we used the term '.NET Compliant Language' and stated that all the .NET compliant languages can make use of CLR and FCL. But what makes a language '.NET compliant language'? The answer is the Common Language Specification (CLS). Microsoft has released a small set of specifications that each language should meet to qualify as a .NET Compliant Language. As IL is a very rich language, it is not necessary for a language to implement all the IL functionality, rather it meets the small subset of it, CLS, to qualify as a .NET compliant language, which is the reason why so many languages (procedural and OO) are now running under .Net umbrella. CLS basically addresses to language design issues and lays certain standards like there should be no global function declaration, no pointers, no multiple

inheritance and things like that. The important point to note here is that if you keep your code within the CLS boundary, your code is guaranteed to be usable in any other .Net language.

Common Type System (CTS)

.NET also defines a Common Type System (CTS). Like CLS, CTS is also a set of standards. CTS defines the basic data types that IL understands. Each .NET compliant language should map its data types to these standard data types. This makes it possible for the 2 languages to communicate with each other by passing/receiving parameters to/from each other. For example, CTS defines a type Int32, an integral data type of 32 bits (4 bytes) which is mapped by C# through int and VB.Net through its Integer data type.

Garbage Collector (GC)

CLR also contains Garbage Collector (GC) which runs in a low-priority thread and checks for un-referenced dynamically allocated memory space. If it finds some data that is no more referenced by any variable/reference, it re-claims it and returns the occupied memory back to the Operating System; so that it can be used by other programs as necessary. The presence of standard Garbage Collector frees the programmer from keeping track of dangling data.

C#

Basics

C# (C-Sharp) is a high-level, object-oriented programming language developed by Microsoft that runs on the .NET Framework.
C# is widely used to develop applications for web, desktop, mobile, games and much more.

Compilation

Compilation is the process of converting source code into a form that can be executed by a computer. The C# compiler converts C# source code into a Common Intermediate Language (CIL) assembly. The CIL assembly is then executed by the Common Language Runtime (CLR).

Roselyn Compiler Purpose

Roslyn is the open-source compiler platform for C# and Visual Basic.NET developed by Microsoft. It provides APIs for analyzing and manipulating code, enabling powerful code analysis and refactoring tools.

Command Line Compilation

To compile a C# program from the command line, you can use the csc.exe compiler. The csc.exe compiler is located in the .NET Framework installation directory.

Step 1: Open the text editor like Notepad or Notepad++, and write the code that you want to execute. Now save the file with **.cs** extension.

```
// C# program to print Hello World!
using System;
// namespace declaration
namespace HelloWorldApp {
    // Class declaration
    class Geeks {
        // Main Method
        static void Main(string[] args)
        {
            // statement
            // printing Hello World!
            Console.WriteLine("Hello World!");

            // To prevents the screen from
            // running and closing quickly
            Console.ReadKey();
        }
    }
}
```

Step 2: Compile your C# source code with the use of command:

csc File_name.cs

If your program has no error, then it will create a filename.exe file in the same directory where you have saved your program. Suppose you saved the above program as Hello.cs. So you will write **csc Hello.cs** on cmd. This will create a *Hello.exe* file.

Step 3: Now there are two ways to execute the Hello.exe. First, you have to simply type the filename i.e. Hello on the cmd and it will give the output. Second, you can go to the directory where you saved your program and there you find filename.exe. You have to simply double-click that file and it will give the output.
C: \users\Shubham>Hello
Hello world!
C: \users\Shubham>Hello.exe
Hello World!

Types

C# supports a variety of data types, including integers, floating-point numbers, strings, and Boolean. C# also supports user-defined types, such as classes and structs.

Loops

C# supports three types of loops: for loops, while loops, and do-while loops.

Basic Syntax

In C#, a basic program consists of the following:

- A Namespace Declaration
- Class Declaration & Definition
- Class Members (like variables, methods etc.)
- Main Method
- Statements or Expressions

Creating Assembly using C# On Windows Platform

EXE

An EXE file is an executable file that can be run on a Windows computer. To create an EXE file from a C# program, you need to compile the program with the csc.exe compiler.

DLL

A DLL file is a dynamic link library or class library that can be used by other programs. To create a DLL file from a C# program, you need to compile the program with the csc.exe compiler and specify the /target: library option.

Creating and Using DLL (Class Library) in C#

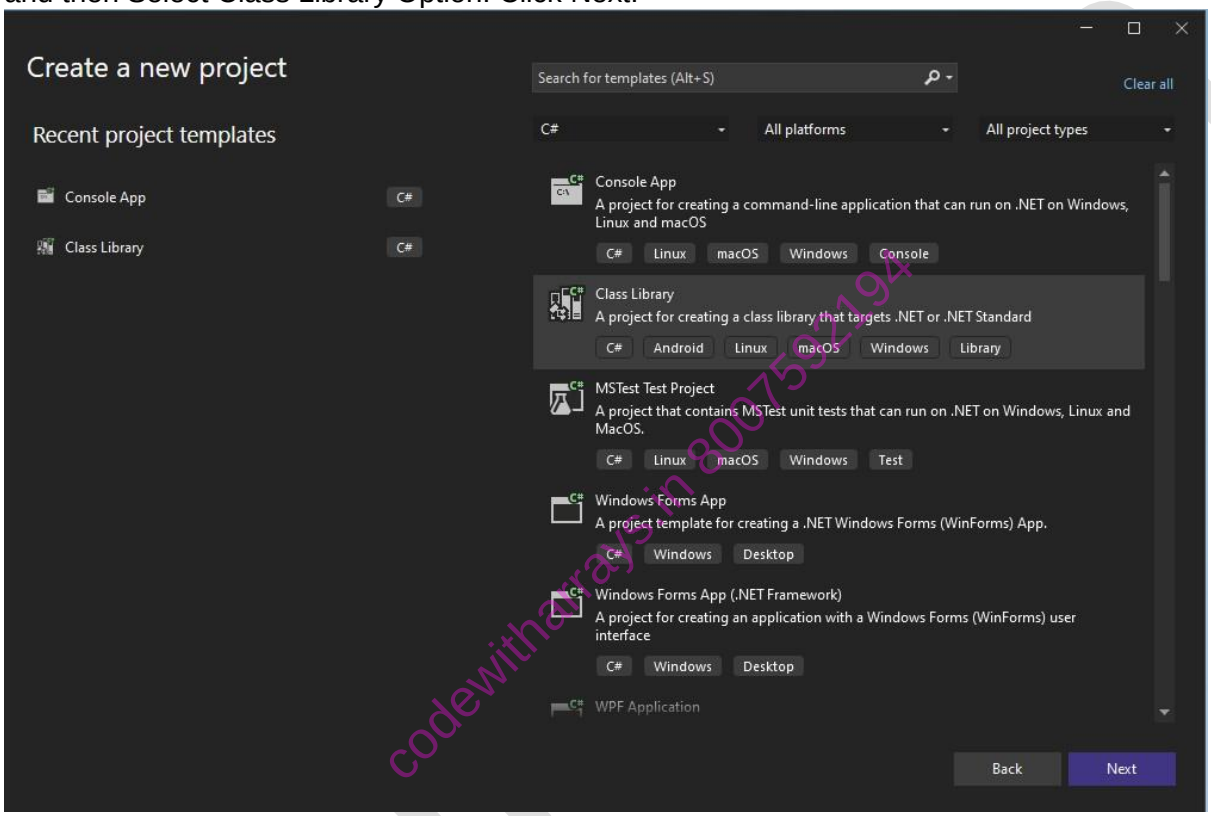
A class library file is a collection of classes and namespaces in C# without any entry point method like **Main**. Once we create a class library file it can be used in the C# project and classes inside it can be used as required. Class Library makes it convenient to use functionalities by importing DLL inside the program rather than redefining everything. So, Let's make our own class library in C#.

Terminologies

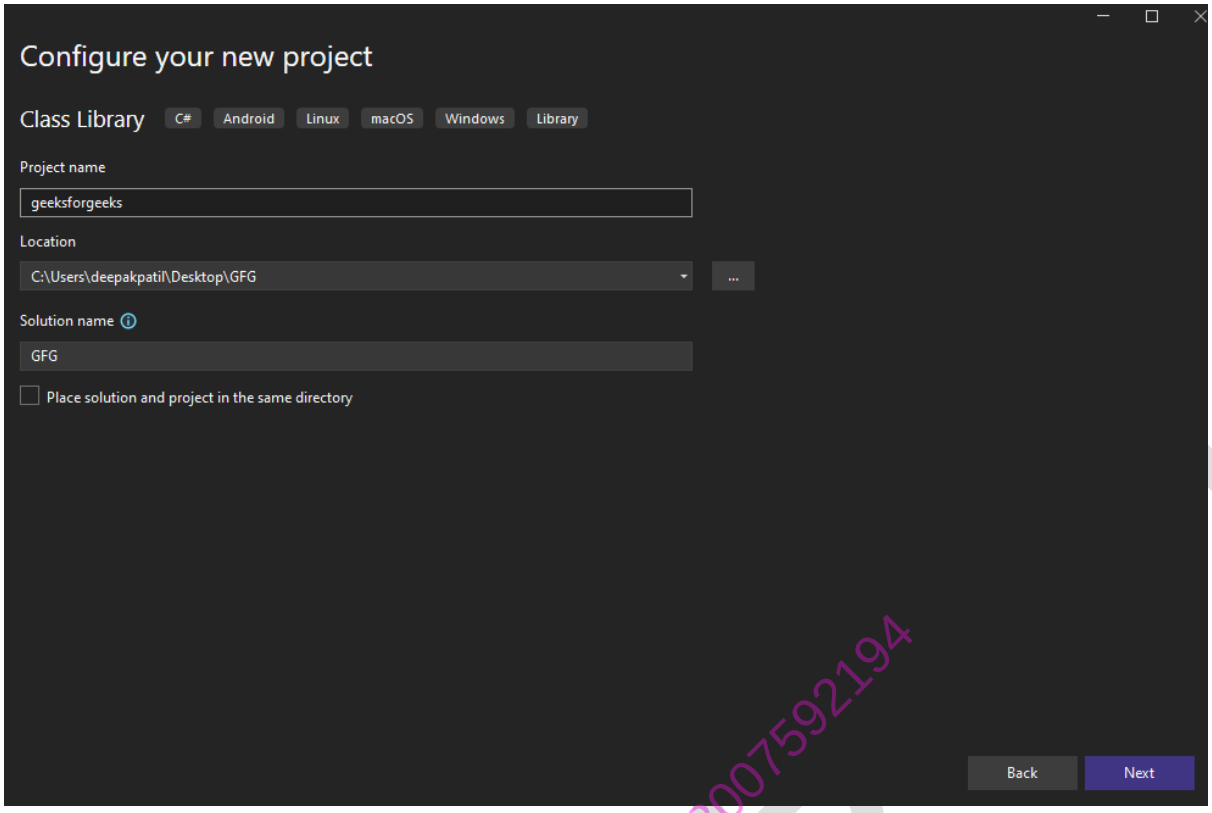
Class Library: It is a package or file that contains different namespaces and class definitions that are used by other programs.

Steps to Create and Use DLL in Visual Studio

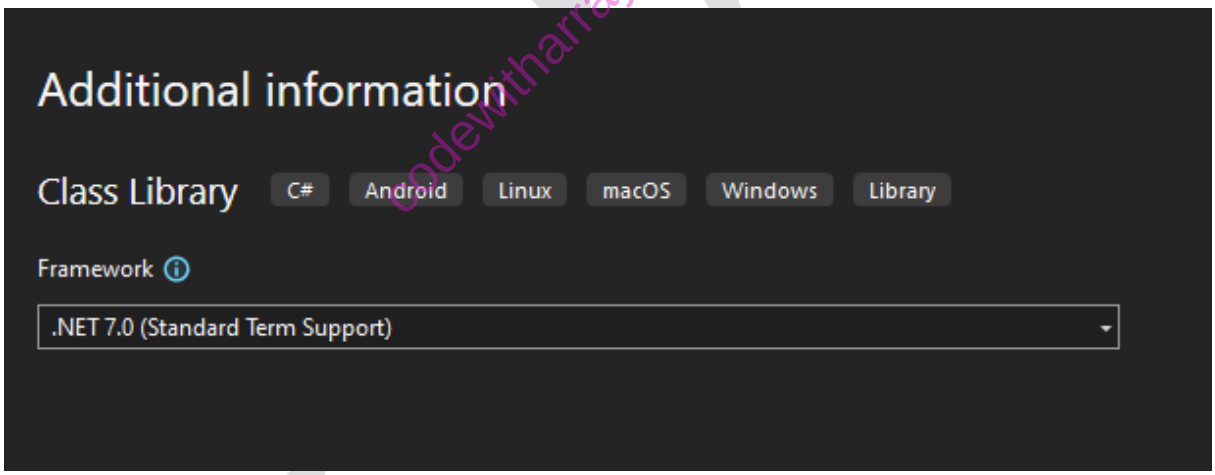
Step 1: Create a New Project in Visual Studio. Once You open visual studio it should open Create New Project OR You can click on the file and select the new project option. Select C# as language and then Select Class Library Option. Click Next.



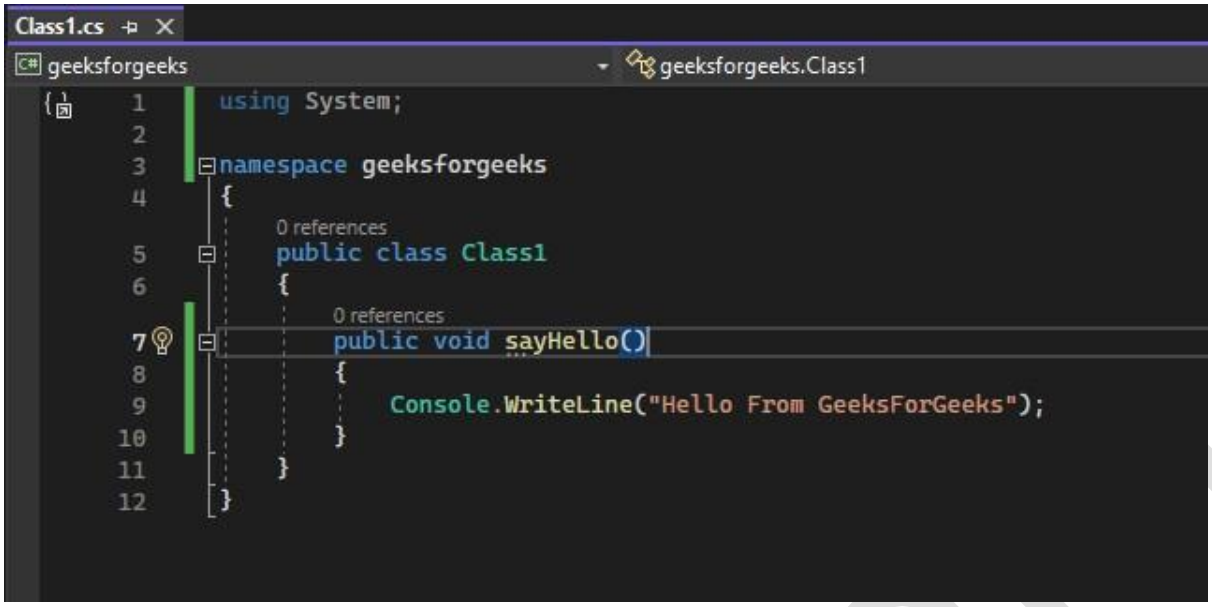
Step 2: On the next screen configure your class library project name. Make sure you give a different name for the Solution. Then click next.



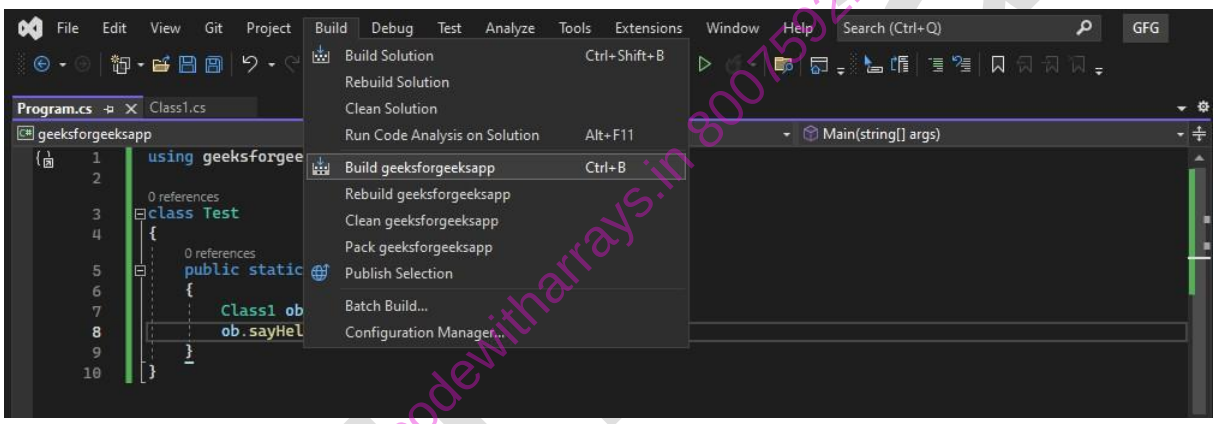
Step 3: On the next screen select the .NET version. I have selected 7.0. Then click create which will create a project.



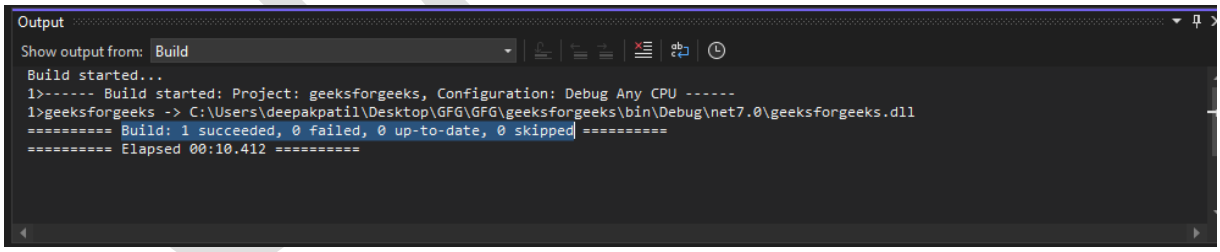
Step 4: Once the Project is created a C# file will already be created with namespace as project name and class Class1. Let's add some code inside the class that will print something when the method **sayHello()** inside the class is called.



Step 5: After writing the code click on the build in the menu bar and click build geeksforgeeks.

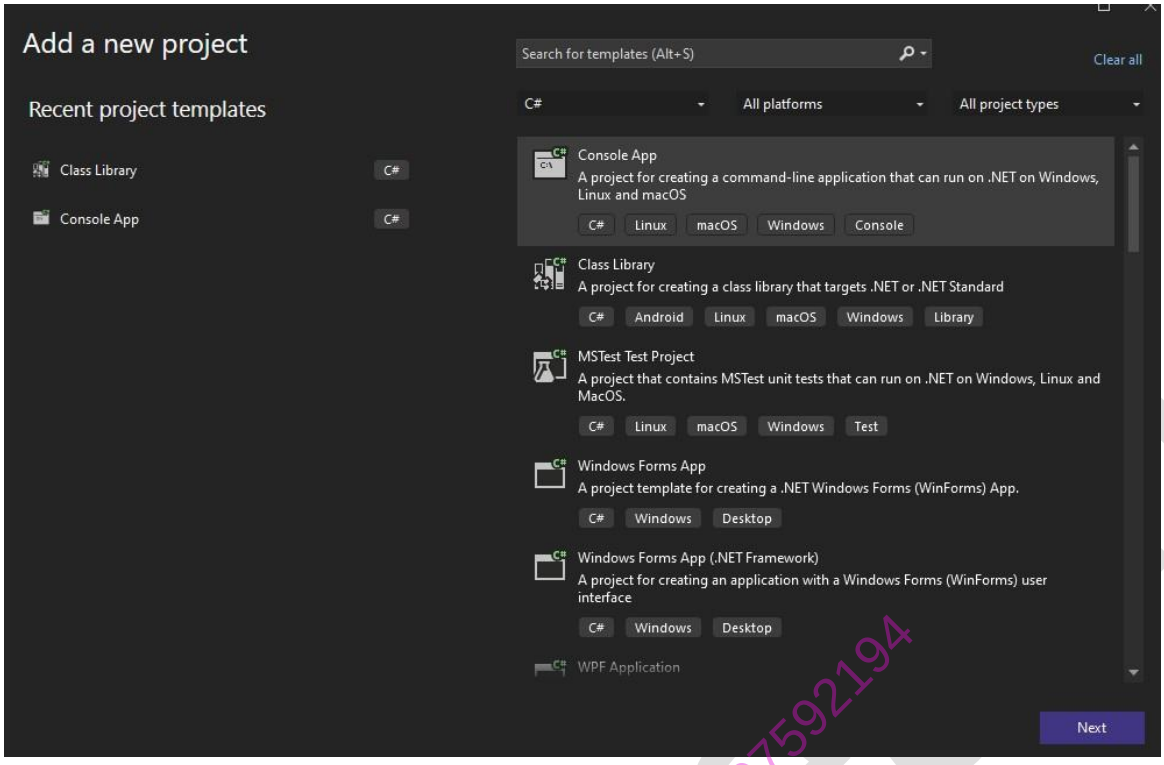


Step 6: If everything is correct you should get build success in the output below the editor.

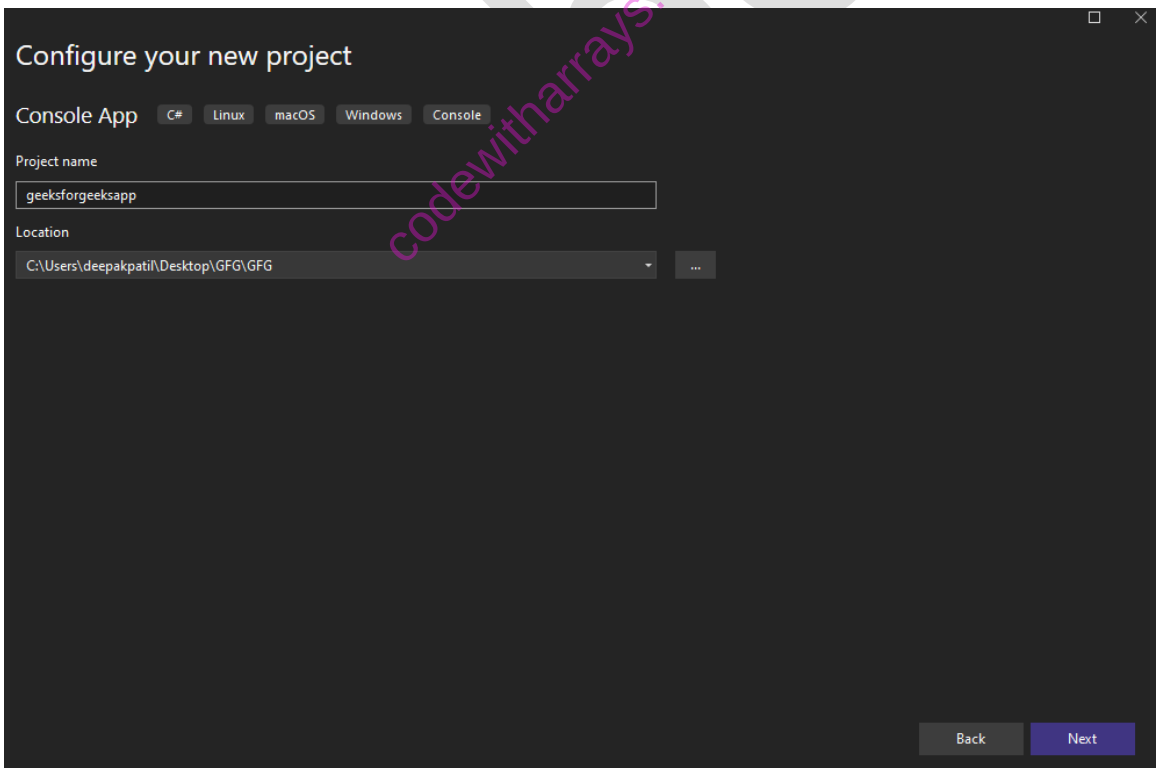


Step 7: Now the DLL file is created inside the project-folder/bin/Debug/net7.0 folder which can be used.

Step 8: Let's create a new project to use this DLL file. For the new project select Console App from the list and click next.



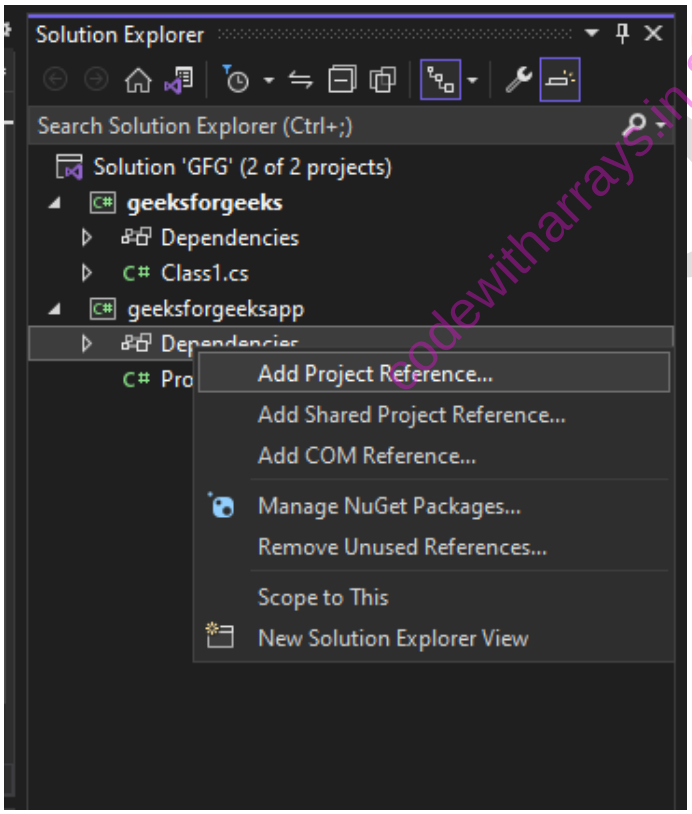
Step 9: Configure a new project with a name and give the same name for the solution as given for creating a Class library project OR just select the same folder for a solution.



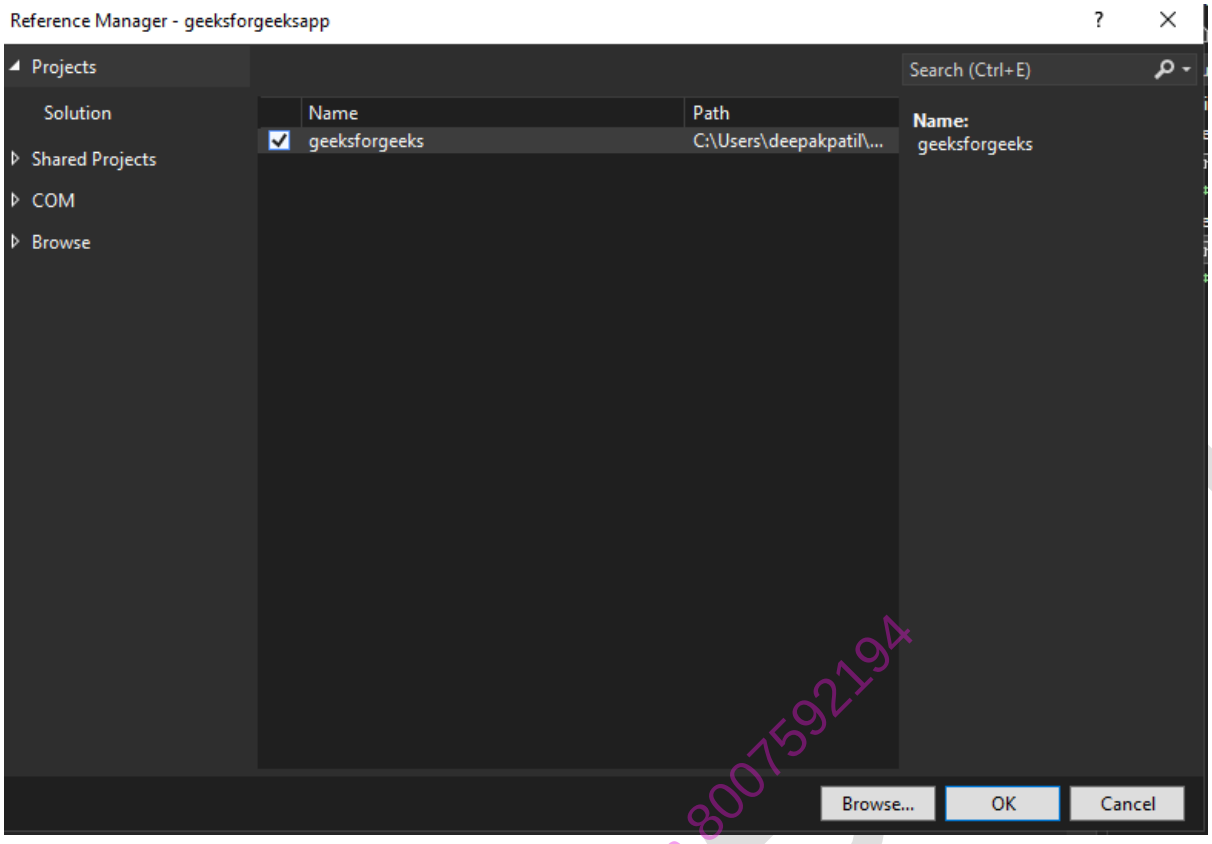
Step 10: Once the project is created it should have a program.cs file opened. Type the Code inside the program.cs file. import the DLL file inside the program by putting “using geeksforgeeks” at the top. Now we can use Class1 inside our program. Call the sayHello() method by creating an object of Class1.

```
Program.cs
geeksforgeeksapp                                    Test
1   using geeksforgeeks;
2
    0 references
3   class Test
4   {
        0 references
5       public static void Main(String[] args)
6       {
7           Class1 ob = new Class1();
8           ob.sayHello();
9       }
10  }
```

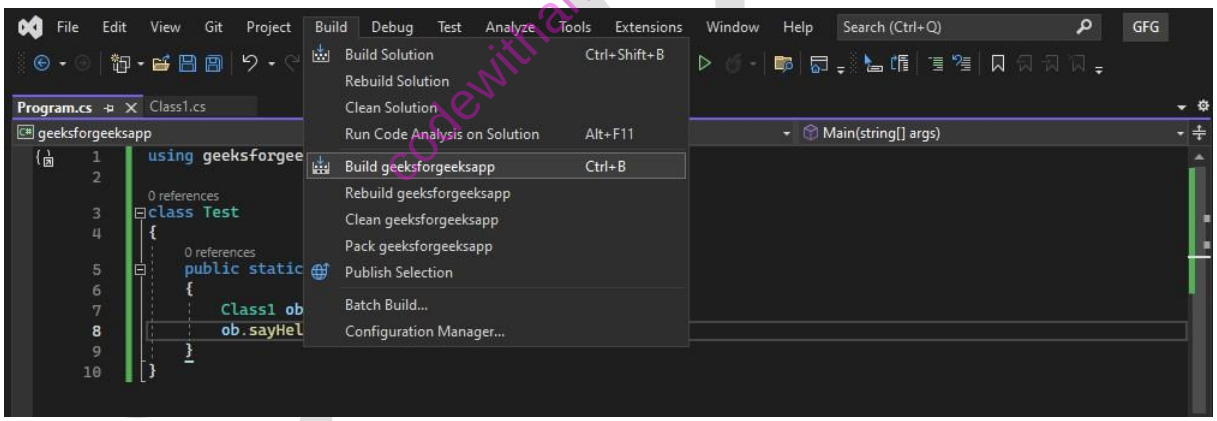
Step 11: Afterwards we have to add a reference of the DLL file to our project. For that in Solution Explorer select dependencies under our project name and right-click to select “Add Project reference”.



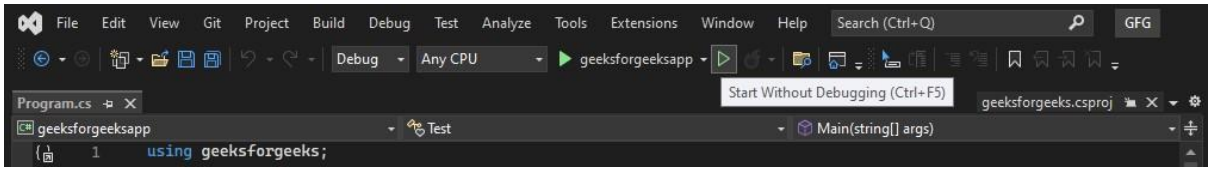
Step 12: Now select our Class Library project name from the list. Click Ok. Now we can build our project.



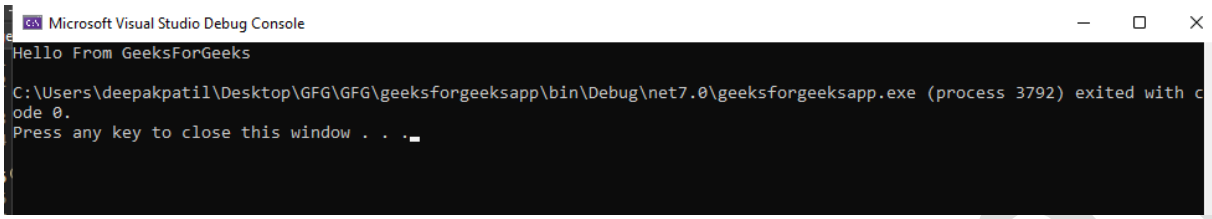
Step 13: From the menu click build and select build “geeksforgeeksapp” or your console project name. You should see the build succeeded at the output.



Step 14: Once Build is succeeded right click on Class Library Project from Solution Explorer and select unload the project. Then click on run without debugging on the sub-menu bar at the top (play button).



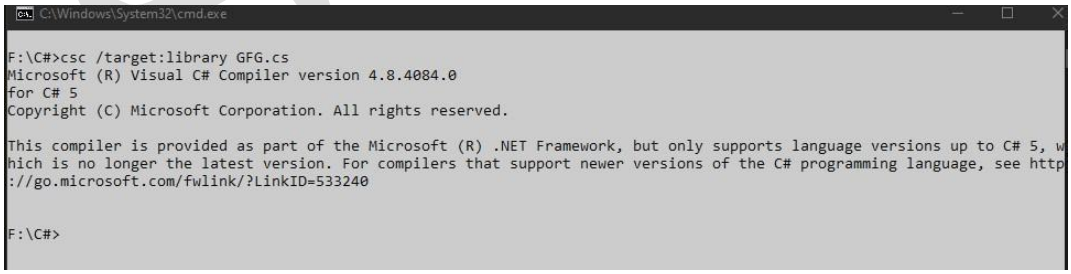
Step 15: You should see a console window opened with say hello message printed.



Steps to Create DLL file with C# Compiler

Step 1: Create a new blank file inside your favourite editor and save it as a **".cs"** file with the name you want for DLL. Add the Code to the file with namespace and class with any method. I have added the same method as above which prints "Hello From GeeksForGeeks".

Step 2: Open Command Prompt or Terminal where the CS file is saved. Compile the program with the CSC compiler and make sure you add the target file as a library which will generate a DLL file. If there is no error, then you should see a DLL file created inside a folder with the name as the filename.



Step 3: Now let's use this DLL. Create another file inside the same folder where DLL is located and save it as ".cs". Type your code to use the DLL. I have written the same code as above.

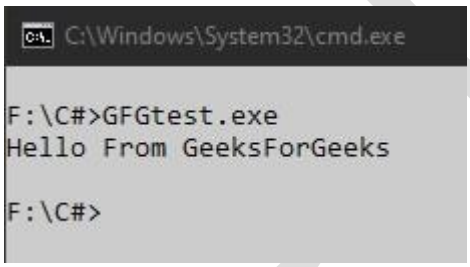
GFGtest.cs

```
using geeksforgeeks;

class Test{
    public static void Main(string[] args)
    {
        //Creating Object Of GFG Class defined inside DLL file
        GFG ob=new GFG();
        ob.sayHello();
    }
}
```

Step 4: Then save the file and compile it with CSC as follows. We have used /r to provide references for our DLL file.

Step 5: If no error compilation is successful and we can run our program type filename.exe to run the program. You should see the output printed.



Consuming DLL in EXE

To consume a DLL in an EXE file, you need to add a reference to the DLL file in the EXE project. You can do this by right-clicking on the References node in the Solution Explorer and selecting Add Reference.

Console | Convert Class & Methods

Convert class provides different methods to convert a base data type to another base data type. The base types supported by the Convert class are Boolean, Char, SByte, Byte, Int16, Int32, Int64, UInt16, UInt32, UInt64, Single, Double, Decimal, DateTime, and String. It also provides methods that support other conversions. This class is defined under System namespace.

Characteristics of Convert class:

- It provides methods that are used to convert every base type into every other base type.
- It provides methods that are used to convert integer values to the non-decimal string representation, also convert the string represent the non-decimal numbers to integer values.
- It provides methods that are used to convert any custom object to any base type.
- It provides a set of methods that supports base64 encoding.
- An OverFlowException can occur if a narrowing conversion results in a loss of data.

Field:

- DBNull: It is a constant which represents a database column that is absent of data i.e. database null.

Method	Description
ChangeType()	It returns an object of a specified type whose value is equivalent to a specified object.
FromBase64CharArray(Char[], Int32, Int32)	Converts a subset of a Unicode character array, which encodes binary data as base-64 digits, to an equivalent 8-bit unsigned integer array. Parameters specify the subset in the input array and the number of elements to convert.
FromBase64String(String)	Converts the specified string, which encodes binary data as base-64 digits, to an equivalent 8-bit unsigned integer array.
GetTypeCode(Object)	Returns the TypeCode for the specified object.
IsDBNull(Object)	Returns an indication whether the specified object is of type DBNull.
ToBase64CharArray()	Converts a subset of an 8-bit unsigned integer array to an equivalent subset of a Unicode character array encoded with base-64 digits.
ToBase64String()	Converts the value of an array of 8-bit unsigned integers to its equivalent string representation that is encoded with base-64 digits.
ToBoolean()	Converts a specified value to an equivalent Boolean value.

ToByte()	Converts a specified value to an 8-bit unsigned integer.
ToChar()	Converts a specified value to a Unicode character.
ToDateTime()	Converts a specified value to a DateTime value.
ToDecimal()	Converts a specified value to a decimal number.
ToDouble()	Converts a specified value to a double-precision floating-point number.
ToInt16()	Converts a specified value to a 16-bit signed integer.
ToInt32()	Converts a specified value to a 32-bit signed integer.
ToInt64()	Converts a specified value to a 64-bit signed integer.
ToSByte()	Converts a specified value to an 8-bit signed integer.
ToSingle()	Converts a specified value to a single-precision floating-point number.
ToUInt16()	Converts a specified value to a 16-bit unsigned integer.
ToUInt32()	Converts a specified value to a 32-bit unsigned integer.
ToUInt64()	Converts a specified value to a 64-bit unsigned integer.

Example :

```
// C# program to illustrate the
// use of ToDecimal(Int16) method
using System;

class GFG {

    // Main method
    static public void Main()
    {

        // Creating and initializing
        // an array
```

```
        short[] ele = {1, Int16.MinValue,
                       -00, 106, -32 };
        decimal sol;

        // Display the elements
        Console.WriteLine("Elements are:");
        foreach(short i in ele)
        {
            Console.WriteLine(i);
        }

        foreach(short num in ele)
        {

            // Convert the given Int16
            // values into decimal values
            // using ToDecimal(Int16) method
            sol = Convert.ToDecimal(num);

            // Display the elements
            Console.WriteLine("convert value is: {0}", sol);
        }
    }
}
```

Output:

```
Elements are:
1
-32768
0
106
-32
convert value is: 1
convert value is: -32768
convert value is: 0
convert value is: 106
convert value is
```


OOP Concepts

Object-Oriented Programming (OOP) is a programming paradigm based on the concept of "objects," which can contain data in the form of fields (attributes or properties) and code in the form of procedures (methods or functions). C# is a language that fully supports OOP principles. Here are the key OOP concepts in C#:

Classes and Objects:
Classes are blueprints or templates for creating objects. They define the structure and behavior of objects.

Objects are instances of classes. They represent real-world entities and contain data and behavior.

```
class Car
{
    public string Model { get; set; }
    public string Color { get; set; }
    public void StartEngine()
    {
        Console.WriteLine("Engine started!");
    }
}
Car myCar = new Car();
myCar.Model = "Toyota";
myCar.Color = "Red";
myCar.StartEngine();
```

Encapsulation:

Encapsulation is the bundling of data (fields) and methods that operate on the data within a single unit (class).
It helps in hiding the internal state of an object and restricting access to it through public methods.

Inheritance:

Inheritance allows a class (subclass or derived class) to inherit properties and behavior from another class (superclass or base class).
It promotes code reusability and establishes an "is-a" relationship between classes.

```
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}
class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("Dog is barking.");
    }
}
Dog myDog = new Dog();
myDog.Eat();
myDog.Bark();
```

Polymorphism:

Polymorphism allows objects of different types to be treated as objects of a common base type.It enables methods to be defined in a base class and overridden in derived classes to provide different implementations.

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine("Drawing a shape.");
    }
```

```
}
class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle.");
    }}
Shape myShape = new Circle();
myShape.Draw(); // Outputs: Drawing a circle.
```

Abstraction:
Abstraction is the process of hiding the complex implementation details and exposing only the necessary features of an object.
It allows you to focus on what an object does rather than how it does it.
Interfaces:
Interfaces define a contract for classes to implement. They specify the methods and properties that implementing classes must provide.
They support multiple inheritance of behavior and promote loose coupling between classes.

```
interface IShape
{
    void Draw();
}
class Rectangle : IShape
{
    public void Draw()
    {
        Console.WriteLine("Drawing a rectangle.");
    }
}
```

Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called a derived class or subclass) to inherit properties and behavior from another class (called a base class or superclass). In C#, inheritance is implemented using the : symbol followed by the name of the base class. Here's an overview of inheritance in C#:

Base Class and Derived Class:
A base class is a class that provides the foundation or common behavior that can be shared by one or more derived classes.A derived class is a class that inherits from a base class and can extend or specialize its behavior.

```
// Base class
public class Animal
{
    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}

// Derived class
public class Dog : Animal
{
```

```
    public void Bark()
    {
        Console.WriteLine("Dog is barking.");
    }
}
```

Syntax:
To declare a class as a derived class, use a colon (:) followed by the name of the base class.The derived class inherits all non-private members (fields, properties, methods) of the base class.

```
public class DerivedClass : BaseClass
{
    // Members of the derived class
}
```

Access Modifiers:
Inheritance respects access modifiers such as public, protected, internal, and private. Derived classes can access public and protected members of the base class.

Constructor Inheritance:
Constructors are not inherited by derived classes, but the derived class must invoke a constructor of the base class.
This can be done explicitly using the base keyword.

```
public class DerivedClass : BaseClass
{
    public DerivedClass() : base(/* parameters */) { }
}
```

Method Overriding:
Derived classes can override base class methods by providing a new implementation. Use the override keyword to indicate that a method overrides a base class method.

```
public class Dog : Animal
{
    public override void Eat()
    {
        Console.WriteLine("Dog is eating.");
    }
}
```

Sealed

In C#, the sealed modifier is used to prevent inheritance of a class or overriding of virtual methods. When a class is marked as sealed, it cannot be used as a base class for further derivation. Similarly, when a method is marked as sealed, it cannot be overridden in derived classes. Here's how sealed works in C#:

Sealed Classes:
When a class is marked as sealed, it means that the class cannot be used as a base class for inheritance. It prevents other classes from deriving from it.

```
public sealed class SealedClass
{
    // Class members
}
```

Sealed Methods:
When a method in a base class is marked as sealed, it means that the method cannot be overridden in derived classes.This is useful when you want to prevent further specialization of behavior defined in the base class method.

```
public class BaseClass
{
    public virtual void MyMethod()
    {
        // Method implementation
    }
}
public class DerivedClass : BaseClass
{
    public sealed override void MyMethod()
    {
        // Sealed method implementation
    }
}
```

Usage:
The sealed modifier is used to restrict inheritance and overriding in scenarios where further derivation or overriding is not desired.It can be applied to both classes and methods.

Benefits:
Helps in code optimization and performance improvements by preventing unnecessary virtual method calls in the inheritance hierarchy.Enhances code security by restricting unintended derivation or overriding of classes and methods.

Considerations:
Use sealed sparingly and only when necessary to prevent further derivation or overriding.Overuse of sealed can limit the flexibility and extensibility of the codebase.

Inheritance and Sealed Classes:
A class can be marked as sealed even if it doesn't have any virtual members.A sealed class cannot be used as a base class for inheritance, regardless of whether it has virtual members or not.

Static

In C#, the static keyword is used to define members (fields, methods, properties, constructors, and nested types) that belong to the type itself rather than to instances of the type. Here's how static works in various contexts:

Static Fields:
Static fields are shared across all instances of a class. They belong to the class rather than to any specific instance.Static fields are initialized once when the class is loaded into memory and are shared among all instances of the class.

```
public class MyClass
{
    public static int StaticField = 10;
}
```

Static Methods:
Static methods belong to the class rather than to instances of the class. They can be called directly

on the class without creating an instance.Static methods cannot access instance members directly but can access other static members.

```
public class MyClass
{
    public static void StaticMethod()
    {
        Console.WriteLine("Static method called.");
    }
}

MyClass.StaticMethod(); // Calling static method
```

Static Properties:

Static properties are similar to static fields but provide a getter and setter to access and modify the underlying static field.

```
public class MyClass
{
    private static int staticField;

    public static int StaticProperty
    {
        get { return staticField; }
        set { staticField = value; }
    }
}

MyClass.StaticProperty = 20; // Setting static property
int value = MyClass.StaticProperty; // Getting static property
```

Static Constructors:

Static constructors are called only once, when the class is first accessed or instantiated. They are used to initialize static fields or perform any one-time initialization for the class.

```
public class MyClass
{
    static MyClass()
    {
        // Static constructor
    }
}
```

Static Classes:

A static class is a class that only contains static members. It cannot be instantiated, and all its members are accessed directly through the class name.

```
public static class UtilityClass
{
    public static void DoSomething()
    {
        // Static method in a static class
    }
}
UtilityClass.DoSomething(); // Calling static method from a static class
```

Static Members in Interfaces:

Starting from C# 8.0, interfaces can contain static members.

```
public interface IMyInterface
{
    static void StaticMethod()
    {
        // Static method in an interface
    }
}
```

Static members are used when the behavior or data is shared across all instances of a class or when the behavior does not depend on the state of any specific instance. They provide a way to organize and access common functionality without requiring an instance of the class.

Abstract

In C#, the abstract keyword is used to define abstract classes and abstract members within those classes. Abstract classes cannot be instantiated directly, and they may contain abstract methods that must be implemented by non-abstract derived classes. Here's an overview of how abstract works:

Abstract Classes:
An abstract class is a class that cannot be instantiated directly. It serves as a base class for other classes and may contain abstract and non-abstract members.

```
public abstract class Animal
{
    public abstract void MakeSound(); // Abstract method
    public void Eat() // Non-abstract method
    {
        Console.WriteLine("Animal is eating.");
    }
}
```

Abstract Methods:
An abstract method is a method without a body, declared using the abstract keyword. It must be implemented by non-abstract derived classes.

```
public abstract void MakeSound();
```

Abstract Properties:

Abstract properties are similar to abstract methods but represent properties instead. They do not have an implementation and must be implemented by derived classes.

```
public abstract int MyProperty { get; set; }
```

Derived Classes:

Derived classes from abstract classes must provide concrete implementations for all abstract members declared in the base abstract class.

```
public class Dog : Animal
{
    public override void MakeSound()
    {
```

```
        Console.WriteLine("Woof!");
    }
}
```

Instantiation:

Abstract classes cannot be instantiated directly. However, they can be used as a base for other classes, which provide implementations for all abstract members.

```
Animal myAnimal = new Dog();
myAnimal.MakeSound(); // Outputs: Woof!
```

Use Cases:

Abstract classes are useful when you want to define a common interface for a group of related classes but don't want to provide a default implementation for some methods or properties. They are also handy when you want to enforce certain behaviors to be implemented by derived classes.

Abstract Members in Interfaces:

Starting from C# 8.0, interfaces can also contain abstract members. These members are implicitly abstract and must be implemented by classes that implement the interface.

```
public interface IMyInterface
{
    void MyMethod(); // Implicitly abstract
}
```

Abstract classes provide a way to define a common interface for a group of related classes while allowing for flexibility in their implementation. They are essential for building hierarchies of classes with shared behavior and contracts.

Interface - Concepts:

In C#, an interface is a reference type that defines a contract for classes to implement. It contains only the declaration of methods, properties, events, or indexers but not their implementation. Here's an overview of the key concepts related to interfaces in C#:

Interface Declaration:

An interface is declared using the interface keyword followed by the interface name and a list of members.

```
public interface IMyInterface
{
    void Method1();
    int Property1 { get; set; }
}
```

Interface Members:

Interface members include methods, properties, events, and indexers.These members are implicitly public and abstract, and they cannot contain any access modifiers.They define a contract that implementing classes must adhere to.

Implementing Interfaces:

A class implements an interface by providing concrete implementations for all the members declared in the interface.

```
public class MyClass : IMyInterface
{
    public void Method1()
    {
        // Implementation for Method1
    }

    public int Property1 { get; set; }
}
```

Multiple Interface Implementation:

A class can implement multiple interfaces by separating them with commas in the class declaration.

```
public class MyClass : IMyInterface1, IMyInterface2
{
    // Implementations for interface members
}
```

Implicit Interface Implementation:
Starting from C# 8.0, classes can provide implicit implementation of interface members.

```
public class MyClass : IMyInterface
{
    public void Method1()
    {
        // Implementation for Method1
    }

    public int Property1 { get; set; }
}
```

Interface Inheritance:
Interfaces can inherit from one or more other interfaces using the : syntax.

```
public interface IMyInterface2 : IMyInterface1
{
    // Interface members
}
```

Interface Segregation Principle (ISP):

ISP is a design principle that states that no client should be forced to depend on methods it does not use.It suggests splitting large interfaces into smaller, more specific ones.

Use Cases:

Interfaces are commonly used to define contracts for classes that have similar behavior but different implementations.They facilitate polymorphism and decoupling in software design by allowing objects to be treated uniformly based on their interfaces.Interfaces are a powerful tool in C# for designing flexible and extensible systems. They enable code reusability, maintainability, and testability by promoting loose coupling between components.

Events and Delegates :-

In C#, events and delegates are fundamental concepts used for implementing the observer design pattern, enabling communication between objects in an application. Here's an overview of each:

Delegates:

Delegates are a type-safe, object-oriented function pointers that allow methods to be passed as parameters to other methods. They define the signature and return type of the method(s) they can reference.

Declaration:

delegate returnType DelegateName(parameters);

public delegate void MyDelegate(string message);

Events:

Events are special types of delegates that encapsulate methods to be called when a certain action or notification occurs. They are used to implement the publisher-subscriber pattern.

Declaration:

public event DelegateName EventName;

public event EventHandler ButtonClicked;

Basic Usage:

Define Delegate: Define a delegate that matches the signature of the methods you want to encapsulate.

Declare Event: Declare an event based on the delegate.

Subscribe to Event: Subscribe methods to the event using the += operator.

Raise Event: Call the event, which will execute all subscribed methods.

```
using System;

public class Program
{
    // Step 1: Define delegate
    public delegate void EventHandler(string message);

    // Step 2: Declare event
    public event EventHandler Notify;

    // Step 3: Subscribe to event
    public void Subscribe(EventHandler method)
    {
        Notify += method;
    }
```

```
        // Step 4: Raise event
        public void RaiseEvent(string message)
        {
            Notify?.Invoke(message);
        }

        static void Main(string[] args)
        {
            Program program = new Program();

            // Subscribe methods to
            event
            program.Subscribe(Metho
            d1);
            program.Subscribe(Metho
            d2);

            // Raise event
            program.RaiseEvent("Event
            triggered!");

            Console.ReadLine();
        }

        static void Method1(string message)
        {
            Console.WriteLine("Method 1: " + message);
        }

        static void Method2(string message)
        {
            Console.WriteLine("Method 2: " + message);
        }
    }
```

This will output:

mathematica

Method 1: Event triggered!
Method 2: Event triggered!

Events and delegates are extensively used in C# for implementing various patterns like observer, command, etc., as well as in frameworks like Windows Forms and WPF for handling UI events.

File IO and Reflection

File IO :

File handling is a very crucial and important feature for many enterprise applications around us. To support this feature Microsoft .NET Framework offers the System.IO namespace, that provides various classes to enable the developers to do I/O.

Objectives:

Using the File class for reading and writing data.
Using the File and FileInfo class to manipulate files.
Using the DirectoryInfo and Directory classes to manipulate directories.

The File class of the System.IO namespace offers various static methods that enable a developer to do direct reading and writing of files. Typically, to read data from a file, you:

1. Get a hold on the file handle.
2. Open a stream to the file.
3. Buffer the file data into memory.
4. Release the hold of file handle once done.

Reading Data from Files : data is read from a file

The "ReadAllText" method reads the data of a file.

```
string filePath = @"C:\MyData\TestFile.txt";
string testData = File.ReadAllText(filePath);
```

The "ReadAllLines" method reads all the contents of a file and stores each line at a new index in an array of string type.

```
string filePath = @"C:\MyData\TestFile.txt";
string[] testDataLineByLine = File.ReadAllLines(filePath);
```

The "ReadAllBytes" method reads the contents of a file as binary data and stores the data in a byte array.

```
string filePath = @"C:\MyData\TestFile.txt";
byte[] testDataRawBytes = File.ReadAllBytes(filePath);
```

Each of these methods enable the developer to read the contents of a file and load into memory. The ReadAllText method will enable the developer to cache the entire file in memory via a single operation. Whereas the ReadAllLines method will read line-by-line into an array.

Writing Data to Files : data is written to a file. By default, all existing contents are removed from the file, and new content is written.

The "WriteAllText" method enables the developers to write the contents of string variable into a file. If the file exists, its contents will be overwritten. The following code example shows how to write the contents of a string named settings to a new file named settings.txt.

```
string filePath = @"C:\MyData\TestFile.txt";
string data = "C# Corner MVP & Microsoft MVP;";
File.WriteAllText(filePath, data);
```

The "WriteAllLines" method enables the developers to write the contents of a string array to a file. Each entry in the string array will be a new line in the new file.

```
String filePath = @"C:\MyData\TestFile.txt";
string[] data =  "MCT", "MCPD", "MCTS", "MCSD.NET", "MCAD.NET","CSM"};
File.WriteAllLines(filePath, data)
```

Appending Data to Files – writing information to a file. The only difference is that the existing data in a file is not overwritten. The new data to be written is added at the end of the file.
The "AppendAllText" method enables the developers to write the contents of a string variable at the end of an existing file.

```
String filePath = @"C:\MyData\TestFile.txt";
string data = "Also Certified from IIT Kharagpur";
File.AppendAllText(filePath, data);
```

The "AppendAllLines" method enables the developers to write the contents of a string array to the end of an existing file.

```
String filePath = @"C:\MyData\TestFile.txt";
string[] otherData = { "Worked with Microsoft", "Lived in USA" };
File.AppendAllLines(filePath, otherData);
```

File IO Steps :

1. Creating a file :

We use theCreate() method of theFile class to create a new file in C#. For example,

```
//create a file at patName
FileStream fs = File.Create(pathName);
```

Here, theFile class creates a file atpathName.

Note: If the file already exists, the Create() method overwrites the file.

Example:

```
using System;
using System.IO;
class Program
{
   static void Main()
   {
      // path of the file that we want to create
      string pathName = @"C:\Program\myFile.txt";

      // Create() creates a file at pathName
      FileStream fs = File.Create(pathName);

      // check if myFile.txt file is created at the specified path
      if (File.Exists(pathName))
      {
         Console.WriteLine("File is created.");
      }
      else
      {
         Console.WriteLine("File is not created.");
      }
   }
}
```

2. Open a file :

We use the Open() method of the File class to open an existing file in C#. The method opens aFileStream on the specified file.

Example:

```
using System;
using System.IO;
class Program
{
   static void Main()
   {
```

```
        string pathName = @"C:\Program\myFile.txt";

       // open a file at pathName
        FileStream fs = File.Open(pathName, FileMode.Open);
    }
}
```

In the above example, notice the code,

```
//opens the file at pathName
FileStream fs = File.Open(pathName,FileMode.Open);
```

Here, theOpen() method opensmyFile.txt file. Here,FileMode.Openspecifies -**open the existing file**.

3. Write to a file :

We use theWriteAllText() method of theFile class to write to a file. The method creates a new file and writes content to that file

Example:

```
using System;
using System.IO;
class Program
{
    static void Main()
    {
        string pathName = @"C:\Program\myFile.txt";

        // create a file at pathName and write "Hello World" to the file
        File.WriteAllText(pathName, "Hello World");
    }
}
```

In the above example, notice the code,

```
File.WriteAllText(pathName,"Hello World");
```

Here,the WriteAllText() method creates myFile.txt at c:\Program directory and writes “Hello Word” to the file.

4. Read a file :

We use theReadAllText() method of the File class to read contents of the file. The method returns a string containing all the text in the specified file.

Let's read the content of the filemyFile.txt where we had written"Hello World".

Exam
ple:

```
using
Syste
m;
using
System.IO;
class
Program
{
    static void Main()
    {
        string pathName = @"C:\Program\myFile.txt";

        // read the content of myFile.txt file
        string readText = File.ReadAllText(pathName);

        Console.WriteLine(readText);
    }
}
```

In the above example, notice the code,

```
//read the content of myFile.txt file
string readText= File.ReadAllText(pathName);
```

The readAllText() method read the file myFile.txt and returns “Hello World”.

Serialization :

serialization is the process of converting object into byte stream so that it can be saved to memory, file or database

Generics

Generics in C# allow you to write flexible and reusable code by creating classes, structures, interfaces, and methods that can work with any data type. They provide type safety without sacrificing flexibility. Here's an overview of how generics work:

Basic Syntax:

```
class ClassName<T>
{
    // Members, methods, properties, etc.
}
```

T is a type parameter, representing any data type. You can use any valid identifier instead of T.

Example:

```
public class GenericList<T>
{
    private T[] _items;
    private int _currentIndex = 0;

    public GenericList(int capacity)
    {
        _items = new T[capacity];
    }

    public void Add(T item)
    {
        if (_currentIndex < _items.Length)
        {
            _items[_currentIndex] = item;
            _currentIndex++;
        }
    }

    public T GetItem(int index)
    {
        if (index >= 0 && index < _items.Length)
        {
            return _items[index];
        }
        throw new IndexOutOfRangeException();
    }
}
```

Usage:

```
GenericList<int> intList = new GenericList<int>(5);
intList.Add(1);
intList.Add(2);
intList.Add(3);

int item = intList.GetItem(1); // item = 2
```

```
GenericList<string> stringList = new GenericList<string>(3);
stringList.Add("apple");
stringList.Add("banana");

string fruit = stringList.GetItem(0); // fruit = "apple"
```

Constraints:

You can apply constraints on generic type parameters to specify capabilities that generic types must have. Common constraints include where T : class, where T : struct, where T : interface, where T : new() (parameterless constructor), and custom constraints.

Example with Constraints:

```
public class MyClass<T> where T : IDisposable
{
    // Methods can use IDisposable methods on T
}
```

Generics are extensively used in .NET framework collections (like List<T>), LINQ, and various other scenarios where type safety and code reuse are essential. They help to avoid code duplication and increase the flexibility and maintainability of your codebase.

Boxing and Unboxing

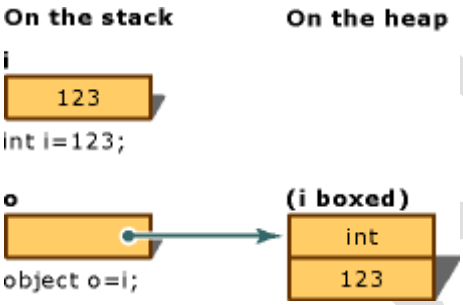
Boxing

Boxing is the process of converting a value type to the type object or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

The following statement implicitly applies the boxing operation on the variable i:

```
int i = 123;
// The following line boxes i.
object o = i;
```

The result of this statement is creating an object reference o, on the stack, that references a value of the type int, on the heap. This value is a copy of the value-type value assigned to the variable i. The difference between the two variables, i and o, is illustrated in the following image of boxing conversion



It is also possible to perform the boxing explicitly as in the following example, but explicit boxing is never required:

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Unboxing

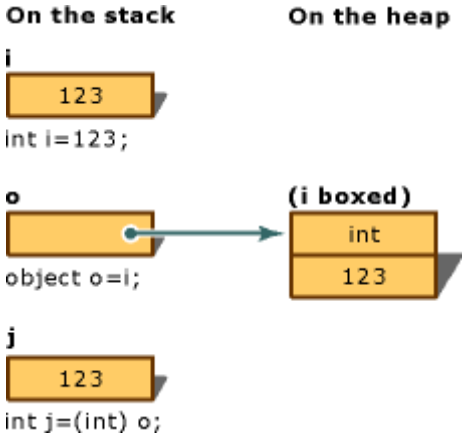
Unboxing is an explicit conversion from the type object to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

- Checking the object instance to make sure that it is a boxed value of the given value type.
- Copying the value from the instance into the value-type variable.

The following statements demonstrate both boxing and unboxing operations:

```
int i = 123;      // a value type
object o = i;     // boxing
```

```
int j = (int)o;    // unboxing
```



Arrays

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements. If you want the array to store elements of any type, you can specify object as its type.

```
type[] arrayName;
```

An array has the following properties:

- An array can be single-dimensional, multidimensional, or jagged.
- The number of dimensions are set when an array variable is declared. The length of each dimension is established when the array instance is created. These values can't be changed during the lifetime of the instance.
- A jagged array is an array of arrays, and each member array has the default value of null.
- Arrays are zero indexed: an array with n elements is indexed from 0 to n-1.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. All arrays implement IList and IEnumerable. You can use the foreach statement to iterate through an array. Single-dimensional arrays also implement IList and IEnumerable.

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = [1, 2, 3, 4, 5, 6];

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

```
// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = [1, 2, 3, 4];
```

Single-dimensional arrays

A single-dimensional array is a sequence of like elements. You access an element via its index. The index is its ordinal position in the sequence. The first element in the array is at index 0. You create a single-dimensional array using the new operator specifying the array element type and the number of elements.
The following example declares and initializes single-dimensional arrays:

```
int[] array = new int[5];
string[] weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
Thu
Fri
Sat
*/
```


he first declaration declares an uninitialized array of five integers, from array[0] to array[4]. The elements of the array are initialized to the default value of the element type, 0 for integers. The second declaration declares an array of strings and initializes all seven values of that array. A series of Console.WriteLine statements prints all the elements of the weekDay array.

Multidimensional arrays

Arrays can have more than one dimension. For example, the following declarations create four arrays: two have two dimensions, two have three dimensions. The first two declarations declare the length of each dimension, but don't initialize the values of the array. The second two declarations use an initializer to set the values of each element in the multidimensional array.

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,    5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```


For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are incremented first, then the next left dimension, and so on, to the leftmost index. The following example enumerates both a 2D and a 3D array:

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
```

```
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,  5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

In a 2D array, you can think of the left index as the row and the right index as the column. However, with multidimensional arrays, using a nested for loop gives you more control over the order in which to process the array elements:

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,  5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```


Jagged arrays

A jagged array is an array whose elements are arrays, possibly of different sizes. A jagged array is sometimes called an "array of arrays." Its elements are reference types and are initialized to null. The following examples show how to declare, initialize, and access jagged arrays. The first example, jaggedArray, is declared in one statement. Each contained array is created in subsequent statements. The second example, jaggedArray2 is declared and initialized in one statement. It's possible to mix jagged and multidimensional arrays. The final

example, jaggedArray3, is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes.

```
int[][] jaggedArray = new int[3][];

jaggedArray[0] = [1, 3, 5, 7, 9];
jaggedArray[1] = [0, 2, 4, 6];
jaggedArray[2] = [11, 22];

int[][] jaggedArray2 =
[
    [1, 3, 5, 7, 9],
    [0, 2, 4, 6],
    [11, 22]
];

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 =
[
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
];

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```


A jagged array's elements must be initialized before you can use them. Each of the elements is itself an array. It's also possible to use initializers to fill the array elements with values. When you use initializers, you don't need the array size.

Implicitly typed arrays

You can create an implicitly typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly typed variable also apply to implicitly typed arrays.

The following examples show how to create an implicitly typed array:

```
int[] a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
```

```
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
int[][] c =
[
    [1,2,3,4],
    [5,6,7,8]
];
// Looping through the outer array
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
string[][] d =
[
    ["Luca", "Mads", "Luke", "Dinesh"],
    ["Karen", "Suma", "Frances"]
];

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
```

```
        int j = 0;
        foreach (var element in subArray)
        {
            // Accessing each element and printing it to the console
            Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
            j++;
        }
        i++;
    }
    /* Outputs
    Element at d[0][0] is: Luca
    Element at d[0][1] is: Mads
    Element at d[0][2] is: Luke
    Element at d[0][3] is: Dinesh
    Element at d[1][0] is: Karen
    Element at d[1][1] is: Suma
    Element at d[1][2] is: Frances
    */
```

In the previous example, notice that with implicitly typed arrays, no square brackets are used on the left side of the initialization statement. Also, jagged arrays are initialized by using new [] just like single-dimensional arrays.

When you create an anonymous type that contains an array, the array must be implicitly typed in the type's object initializer. In the following example, contacts is an implicitly typed array of anonymous types, each of which contains an array named PhoneNumbers. The var keyword isn't used inside the object initializers.


```
var contacts = new[]
{
    new
    {
        Name = "Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new
    {
        Name = "Hanying Feng",
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

Collections and Data Structures

All collections provide methods for adding, removing, or finding items in the collection. In addition, all collections that directly or indirectly implement the ICollection interface or the ICollection interface share these features:

- The ability to enumerate the collection

.NET collections either implement System.Collections.IEnumerable or System.Collections.Generic.IEnumerable to enable the collection to be iterated through. An enumerator can be thought of as a movable pointer to any element in the collection. The foreach, in statement and the For Each...Next Statement use the enumerator exposed by the GetEnumerator method and hide the complexity of manipulating the enumerator. In addition, any collection that implements System.Collections.Generic.IEnumerable is considered a queryable type and can be queried with LINQ. LINQ queries provide a common pattern for accessing data. They're typically more concise and readable than standard foreach loops and provide filtering, ordering, and grouping capabilities. LINQ queries can also improve performance. For more information, see LINQ to Objects (C#), LINQ to Objects (Visual Basic), Parallel LINQ (PLINQ), Introduction to LINQ Queries (C#), and Basic Query Operations (Visual Basic).

- The ability to copy the collection contents to an array

All collections can be copied to an array using the CopyTo method. However, the order of the elements in the new array is based on the sequence in which the enumerator returns them. The resulting array is always one-dimensional with a lower bound of zero.

In addition, many collection classes contain the following features:

- Capacity and Count properties

The capacity of a collection is the number of elements it can contain. The count of a collection is the number of elements it actually contains. Some collections hide the capacity or the count or both.

Most collections automatically expand in capacity when the current capacity is reached. The memory is reallocated, and the elements are copied from the old collection to the new one. This design reduces the code required to use the collection. However, the performance of the collection might be negatively affected. For example, for List, if Count is less than Capacity, adding an item is an O(1) operation. If the capacity needs to be increased to accommodate the new element, adding an item becomes an O(n) operation, where n is Count. The best way to avoid poor performance caused by multiple reallocations is to set the initial capacity to be the estimated size of the collection.

A BitArray is a special case; its capacity is the same as its length, which is the same as its count.

- A consistent lower bound

The lower bound of a collection is the index of its first element. All indexed collections in the System.Collections namespaces have a lower bound of zero, meaning they're 0-indexed. Array has a lower bound of zero by default, but a different lower bound can be defined when creating an instance of the Array class using Array.CreateInstance.

- Synchronization for access from multiple threads (System.Collections classes only).

Non-generic collection types in the System.Collections namespace provide some thread safety with synchronization; typically exposed through the SyncRoot and IsSynchronized members. These collections aren't thread-safe by default. If you require scalable and efficient multi-threaded access to a collection, use one of the classes in the System.Collections.Concurrent namespace or consider using an immutable collection. For more information, see Thread-Safe Collections.

Choose a collection

In general, you should use generic collections. The following table describes some common collection scenarios and the collection classes you can use for those scenarios. If you're new to generic collections, the following table will help you choose the generic collection that works best for your task:

I want to...	Generic collection options	Non-generic collection options	Thread-safe or immutable collectio options
Store items as key/value pairs for quick look-up by key	Dictionary<TKey,TValue>	Hashtable (A collection of key/value pairs that are organized based on the hash code of the key.)	ConcurrentDictionary<TKey,TValue> ReadOnlyDictionary<TKey,TValue> ImmutableDictionary<TKey,TValue>
Access items by index	List<T>	Array ArrayList	ImmutableList<T> ImmutableArray
Use items first-in-first-out (FIFO)	Queue<T>	Queue	ConcurrentQueue<T> ImmutableQueue<T>
Use data Last-In-First-Out (LIFO)	Stack<T>	Stack	ConcurrentStack<T> ImmutableStack<T>
Access items sequentially	LinkedList<T>	No recommendation	No recommendation
Receive notifications when items are removed or added to the collection. (implements INotifyPropertyChanged and INotifyCollectionChanged)	ObservableCollection<T>	No recommendation	No recommendation
A sorted collection	SortedList<TKey,TValue>	SortedList	ImmutableSortedDictionary<TKey,TV ImmutableSortedSet<T>
A set for mathematical functions	HashSet<T> SortedSet<T>	No recommendation	ImmutableHashSet<T> ImmutableSortedSet<T>

ArrayList

Definition

- Namespace: System.Collections
- Assembly: System.Runtime.dll

Implements the IList interface using an array whose size is dynamically increased as required.

```
public class ArrayList : ICloneable, System.Collections.IList
```

Derived System.Windows.Forms.DomainUpDown.DomainUpDownItemCollection Implements ICollection IEnumerable IList ICloneable

Examples

The following example shows how to create and initialize an ArrayList and how to display its values.

```
using System;
using System.Collections;
public class SamplesArrayList  {

   public static void Main()  {

      // Creates and initializes a new ArrayList.
      ArrayList myAL = new ArrayList();
      myAL.Add("Hello");
      myAL.Add("World");
      myAL.Add("!");

      // Displays the properties and values of the ArrayList.
      Console.WriteLine( "myAL" );
      Console.WriteLine( "    Count:    {0}", myAL.Count );
      Console.WriteLine( "    Capacity: {0}", myAL.Capacity );
      Console.Write( "    Values:" );
      PrintValues( myAL );
   }

   public static void PrintValues( IEnumerable myList )  {
      foreach ( Object obj in myList )
         Console.Write( "   {0}", obj );
      Console.WriteLine();
   }
}


/*
This code produces output similar to the following:

myAL
    Count:    3
    Capacity: 4
```

```
    Values:   Hello   World   !

*/
```

Remarks

The ArrayList is not guaranteed to be sorted. You must sort the ArrayList by calling its Sort method prior to performing operations (such as BinarySearch) that require the ArrayList to be sorted. To maintain a collection that is automatically sorted as new elements are added, you can use the SortedSet class.
The capacity of an ArrayList is the number of elements the ArrayList can hold. As elements are added to an ArrayList, the capacity is automatically increased as required through reallocation. The capacity can be decreased by calling TrimToSize or by setting the Capacity property explicitly.

.NET Framework only: For very large ArrayList objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.

Elements in this collection can be accessed using an integer index. Indexes in this collection are zero-based.

The ArrayList collection accepts null as a valid value. It also allows duplicate elements.

Using multidimensional arrays as elements in an ArrayList collection is not supported.

Constructors

ArrayList()	Initializes a new instance of the ArrayList class that is empty and has the default initial capacity.
Array List(ICollection)	Initializes a new instance of the ArrayList class that contains elements copied from the specified collection and that has the same initial capacity as the number of elements copied.
ArrayList(Int32)	Initializes a new instance of the ArrayList class that is empty and has the specified initial capacity.

Properties

Capacity	Gets or sets the number of elements that the ArrayList can contain.
Count	Gets the number of elements actually contained in the ArrayList.
IsFixedSize	Gets a value indicating whether the ArrayList has a fixed size.
IsReadOnly	Gets a value indicating whether the ArrayList is read-only.
IsSynchronized	Gets a value indicating whether access to the ArrayList is synchronized (thread safe).
Item[Int32]	Gets or sets the element at the specified index.
SyncRoot	Gets an object that can be used to synchronize access to the ArrayList.

Methods

Adapter(IList)	Creates an ArrayList wrapper for a specific IList.
Add(Object)	Adds an object to the end of the ArrayList.
AddRange(ICollection)	Adds the elements of an ICollection to the end of the ArrayList.
BinarySearch(Int32, Int32, Object, IComparer)	Searches a range of elements in the sorted ArrayList for an element using the specified comparer and returns the zero-based index of the element.
BinarySearch(Object)	Searches the entire sorted ArrayList for an element using the default comparer and returns the zero-based index of the element.
BinarySearch(Object, IComparer)	Searches the entire sorted ArrayList for an element using the specified comparer and returns the zero-based index of the element.
Clear()	Removes all elements from the ArrayList.
Clone()	Creates a shallow copy of the ArrayList.
Contains(Object)	Determines whether an element is in the ArrayList.
CopyTo(Array)	Copies the entire ArrayList to a compatible one-dimensional Array, starting at the beginning of the target array.
CopyTo(Array, Int32)	Copies the entire ArrayList to a compatible one-dimensional Array, starting at the specified index of the target array.
CopyTo(Int32, Array, Int32, Int32)	Copies a range of elements from the ArrayList to a compatible one-dimensional Array, starting at the specified index of the target array.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
FixedSize(ArrayList)	Returns an ArrayList wrapper with a fixed size.
FixedSize(IList)	Returns an IList wrapper with a fixed size.
GetEnumerator()	Returns an enumerator for the entire ArrayList.
GetEnumerator(Int32, Int32)	Returns an enumerator for a range of elements in the ArrayList.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetRange(Int32, Int32)	Returns an ArrayList which represents a subset of the elements in the source ArrayList.
GetType()	Gets the Type of the current instance. (Inherited from Object)
IndexOf(Object)	Searches for the specified Object and returns the zero-based index of the first occurrence within the entire ArrayList.
IndexOf(Object, Int32)	Searches for the specified Object and returns the zero-based index of the first occurrence within the range of elements in the ArrayList that extends from the specified index to the last element.

IndexOf(Object, Int32, Int32)	Searches for the specified Object and returns the zero-based index of the first occurrence within the range of elements in the ArrayList that starts at the specified index and contains the specified number of elements.
Insert(Int32, Object)	Inserts an element into the ArrayList at the specified index.
InsertRange(Int32, ICollection)	Inserts the elements of a collection into the ArrayList at the specified index.

LastIndexOf(Object)	Searches for the specified Object and returns the zero-based index of the last occurrence within the entire ArrayList.
LastIndexOf(Object, Int32)	Searches for the specified Object and returns the zero-based index of the last occurrence within the range of elements in the ArrayList that extends from the first element to the specified index.
LastIndexOf(Object, Int32, Int32)	Searches for the specified Object and returns the zero-based index of the last occurrence within the range of elements in the ArrayList that contains the specified number of elements and ends at the specified index.
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
ReadOnly(ArrayList)	Returns a read-only ArrayList wrapper.
ReadOnly(IList)	Returns a read-only IList wrapper.
Remove(Object)	Removes the first occurrence of a specific object from the ArrayList.
RemoveAt(Int32)	Removes the element at the specified index of the ArrayList.
RemoveRange(Int32, Int32)	Removes a range of elements from the ArrayList.
Repeat(Object, Int32)	Returns an ArrayList whose elements are copies of the specified value.
Reverse()	Reverses the order of the elements in the entire ArrayList.
Reverse(Int32, Int32)	Reverses the order of the elements in the specified range.
SetRange(Int32, ICollection)	Copies the elements of a collection over a range of elements in the ArrayList.
Sort()	Sorts the elements in the entire ArrayList.
Sort(IComparer)	Sorts the elements in the entire ArrayList using the specified comparer.
Sort(Int32, Int32, IComparer)	Sorts the elements in a range of elements in ArrayList using the specified comparer.
Synchronized(ArrayList)	Returns an ArrayList wrapper that is synchronized (thread safe).
Synchronized(IList)	Returns an IList wrapper that is synchronized (thread safe).
ToArray()	Copies the elements of the ArrayList to a new Object array.
ToArray(Type)	Copies the elements of the ArrayList to a new array of the specified element type.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimToSize()	Sets the capacity to the actual number of elements in the ArrayList.

Hashtable

Sunbeam Institute of Information Technology Pune and Karad

- Namespace: System.Collections
- Assembly: System.Runtime.dll

Represents a collection of key/value pairs that are organized based on the hash code of the key.

```
public class Hashtable : ICloneable, System.Collections.IDictionary,
System.Runtime.Serialization.IDeserializationCallback,
System.Runtime.Serialization.ISerializable
```

- Derived : System.Configuration.SettingsAttributeDictionary System.Configuration.SettingsContext System.Data.PropertyCollection System.Printing.IndexedProperties.PrintPropertyDictionary
- Implements: ICollection IDictionary IEnumerable ICloneable IDeserializationCallback ISerializable

The following example shows how to create, initialize and perform various functions to a Hashtable and how to print out its keys and values.

```
using System;
using System.Collections;

class Example
{
    public static void Main()
    {
        // Create a new hash table.
        //
        Hashtable openWith = new Hashtable();

        // Add some elements to the hash table. There are no
        // duplicate keys, but some of the values are duplicates.
        openWith.Add("txt", "notepad.exe");
        openWith.Add("bmp", "paint.exe");
        openWith.Add("dib", "paint.exe");
        openWith.Add("rtf", "wordpad.exe");

        // The Add method throws an exception if the new key is
        // already in the hash table.
        try
        {
            openWith.Add("txt", "winword.exe");
        }
        catch
        {
            Console.WriteLine("An element with Key = \"txt\" already exists.");
        }

        // The Item property is the default property, so you
        // can omit its name when accessing elements.
        Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);
```

```
        // The default Item property can be used to change the value
        // associated with a key.
        openWith["rtf"] = "winword.exe";
        Console.WriteLine("For key = \"rtf\", value = {0}.", openWith["rtf"]);

        // If a key does not exist, setting the default Item property
        // for that key adds a new key/value pair.
        openWith["doc"] = "winword.exe";

        // ContainsKey can be used to test keys before inserting
        // them.
        if (!openWith.ContainsKey("ht"))
        {
            openWith.Add("ht", "hypertrm.exe");
            Console.WriteLine("Value added for key = \"ht\": {0}",
openWith["ht"]);
        }

        // When you use foreach to enumerate hash table elements,
        // the elements are retrieved as KeyValuePair objects.
        Console.WriteLine();
        foreach( DictionaryEntry de in openWith )
        {
            Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
        }

        // To get the values alone, use the Values property.
        ICollection valueColl = openWith.Values;

        // The elements of the ValueCollection are strongly typed
        // with the type that was specified for hash table values.
        Console.WriteLine();
        foreach( string s in valueColl )
        {
            Console.WriteLine("Value = {0}", s);
        }

        // To get the keys alone, use the Keys property.
        ICollection keyColl = openWith.Keys;

        // The elements of the KeyCollection are strongly typed
        // with the type that was specified for hash table keys.
        Console.WriteLine();
        foreach( string s in keyColl )
        {
            Console.WriteLine("Key = {0}", s);
        }

        // Use the Remove method to remove a key/value pair.
        Console.WriteLine("\nRemove(\"doc\")");
        openWith.Remove("doc");

        if (!openWith.ContainsKey("doc"))
        {
```

```
            Console.WriteLine("Key \"doc\" is not found.");
        }
    }
}

/* This code example produces the following output:

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
Value added for key = "ht": hypertrm.exe

Key = dib, Value = paint.exe
Key = txt, Value = notepad.exe
Key = ht, Value = hypertrm.exe
Key = bmp, Value = paint.exe
Key = rtf, Value = winword.exe
Key = doc, Value = winword.exe

Value = paint.exe
Value = notepad.exe
Value = hypertrm.exe
Value = paint.exe
Value = winword.exe
Value = winword.exe

Key = dib
Key = txt
Key = ht
Key = bmp
Key = rtf
Key = doc

Remove("doc")
Key "doc" is not found.
 */
```


- Each element is a key/value pair stored in a DictionaryEntry object. A key cannot be null, but a value can be.
- The objects used as keys by a Hashtable are required to override the Object.GetHashCode method (or the IHashCodeProvider interface) and the Object.Equals method (or the IComparer interface). The implementation of both methods and interfaces must handle case sensitivity the same way; otherwise, the Hashtable might behave incorrectly. For example, when creating a Hashtable, you must use the CaseInsensitiveHashCodeProvider class (or any case-insensitive IHashCodeProvider implementation) with the CaseInsensitiveComparer class (or any case-insensitive IComparer implementation).
- Furthermore, these methods must produce the same results when called with the same parameters while the key exists in the Hashtable. An alternative is to use a Hashtable constructor with an IEqualityComparer parameter. If key equality were simply reference equality, the inherited implementation of Object.GetHashCode and Object.Equals would suffice.

- Key objects must be immutable as long as they are used as keys in the Hashtable.
- When an element is added to the Hashtable, the element is placed into a bucket based on the hash code of the key. Subsequent lookups of the key use the hash code of the key to search in only one particular bucket, thus substantially reducing the number of key comparisons required to find an element.
- The load factor of a Hashtable determines the maximum ratio of elements to buckets. Smaller load factors cause faster average lookup times at the cost of increased memory consumption. The default load factor of 1.0 generally provides the best balance between speed and size. A different load factor can also be specified when the Hashtable is created.
- As elements are added to a Hashtable, the actual load factor of the Hashtable increases. When the actual load factor reaches the specified load factor, the number of buckets in the Hashtable is automatically increased to the smallest prime number that is larger than twice the current number of Hashtable buckets.
- Each key object in the Hashtable must provide its own hash function, which can be accessed by calling GetHash. However, any object implementing IHashCodeProvider can be passed to a Hashtable constructor, and that hash function is used for all objects in the table.
- The capacity of a Hashtable is the number of elements the Hashtable can hold. As elements are added to a Hashtable, the capacity is automatically increased as required through reallocation.
- .NET Framework only: For very large Hashtable objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.
- The foreach statement of the C# language (For Each in Visual Basic) returns an object of the type of the elements in the collection. Since each element of the Hashtable is a key/value pair, the element type is not the type of the key or the type of the value. Instead, the element type is DictionaryEntry.


```
foreach(DictionaryEntry de in myHashtable)
{
    // ...
}
```

- The foreach statement is a wrapper around the enumerator, which only allows reading from, not writing to, the collection.
- Because serializing and deserializing an enumerator for a Hashtable can cause the elements to become reordered, it is not possible to continue enumeration without calling the Reset method.

Constructors

Hashtable()	Initializes a new, empty instance of the Hashtable class using the default initial capacity, load factor, hash code provider, and comparer.
Hashtable(IDictionary)	Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the default load factor, hash code provider, and comparer.
Hashtable(IDictionary, IEqualityComparer)	Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to a new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the default load factor and the specified IEqualityComparer object.
Hashtable(IDictionary, IHash CodeProvider, IComparer)	**Obsolete.** Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the default load factor, and the specified hash code provider and comparer. This API is obsolete. For an alternative, see Hashtable(IDictionary, IEqualityComparer).
Hashtable(IDictionary, Single)	Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the specified load factor, and the default hash code provider and comparer.
Hashtable(IDictionary, Single, IEqualityComparer)	Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the specified load factor and IEqualityComparer object.
Hashtable(IDictionary, Single, IHashCodeProvider, IComparer)	**Obsolete.** Initializes a new instance of the Hashtable class by copying the elements from the specified dictionary to the new Hashtable object. The new Hashtable object has an initial capacity equal to the number of elements copied, and uses the specified load factor, hash code provider, and comparer.
Hashtable(IEqualityComparer)	Initializes a new, empty instance of the Hashtable class using the default initial capacity and load factor, and the specified IEqualityComparer object.
Hashtable(IHashCodeProvider, IComparer)	**Obsolete.** Initializes a new, empty instance of the Hashtable class using the default initial capacity and load factor, and the specified hash code provider and comparer.
Hashtable(Int32)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity, and the default load factor, hash code provider, and comparer.
Hashtable(Int32, IEquality Comparer)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity and IEqualityComparer, and the default load factor.
Hashtable(Int32, IHashCode	**Obsolete.**

Provider, IComparer)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity, hash code provider, comparer, and the default load factor.
Hashtable(Int32, Single)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity and load factor, and the default hash code provider and comparer.
Hashtable(Int32, Single, IEqualityComparer)	Initializes a new, empty instance of the Hashtable class using the specified initial capacity, load factor, and IEqualityComparer object.
Hashtable(Int32, Single, IHashCodeProvider, IComparer)	**Obsolete.** Initializes a new, empty instance of the Hashtable class using the specified initial capacity, load factor, hash code provider, and comparer.
Hashtable(SerializationInfo, StreamingContext)	**Obsolete.** Initializes a new, empty instance of the Hashtable class that is serializable using the specified SerializationInfo and StreamingContext objects.

Properties

comparer	**Obsolete.** Gets or sets the IComparer to use for the Hashtable.
Count	Gets the number of key/value pairs contained in the Hashtable.
EqualityComparer	Gets the IEqualityComparer to use for the Hashtable.
hcp	**Obsolete.** Gets or sets the object that can dispense hash codes.
IsFixedSize	Gets a value indicating whether the Hashtable has a fixed size.
IsReadOnly	Gets a value indicating whether the Hashtable is read-only.
IsSynchronized	Gets a value indicating whether access to the Hashtable is synchronized (thread safe).
Item[Object]	Gets or sets the value associated with the specified key.
Keys	Gets an ICollection containing the keys in the Hashtable.
SyncRoot	Gets an object that can be used to synchronize access to the Hashtable.
Values	Gets an ICollection containing the values in the Hashtable.

Methods

Add(Object, Object)	Adds an element with the specified key and value into the Hashtable.
Clear()	Removes all elements from the Hashtable.
Clone()	Creates a shallow copy of the Hashtable.
Contains(Object)	Determines whether the Hashtable contains a specific key.
ContainsKey(Object)	Determines whether the Hashtable contains a specific key.
ContainsValue(Object)	Determines whether the Hashtable contains a specific value.
CopyTo(Array, Int32)	Copies the Hashtable elements to a one-dimensional Array instance at the specified index.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an IDictionaryEnumerator that iterates through the Hashtable.
GetHash(Object)	Returns the hash code for the specified key.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	**Obsolete.** Implements the ISerializable interface and returns the data needed to serialize the Hashtable.
GetType()	Gets the Type of the current instance. (Inherited from Object)
KeyEquals(Object, Object)	Compares a specific Object with a specific key in the Hashtable.
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove(Object)	Removes the element with the specified key from the Hashtable.
Synchronized(Hashtable)	Returns a synchronized (thread-safe) wrapper for the Hashtable.
ToString()	Returns a string that represents the current object. (Inherited from Object)

Thread Safety

Hashtable is thread safe for use by multiple reader threads and a single writing thread. It is thread safe for multi-thread use when only one of the threads perform write (update) operations, which allows for lock-free reads provided that the writers are serialized to the Hashtable. To support multiple writers all operations on the Hashtable must be done through the wrapper returned by the Synchronized(Hashtable) method, provided that there are no threads reading the Hashtable object.

Generic Collections

- Namespace: System.Collections.Generic
- Contains interfaces and classes that define generic collections, which allow users to create strongly typed collections that provide better type safety and performance than non-generic strongly typed collections.

Classes

CollectionExtensions	Provides extension methods for generic collections.
Comparer<T>	Provides a base class for implementations of the IComparer<T> generic interface.
Dictionary<TKey,TValue>.KeyCollection	Represents the collection of keys in a Dictionary<TKey,TValue>. This class cannot be inherited.
Dictionary<TKey,TValue>.ValueCollection	Represents the collection of values in a Dictionary<TKey,TValue>. This class cannot be inherited.
Dictionary<TKey,TValue>	Represents a collection of keys and values.
EqualityComparer<T>	Provides a base class for implementations of the IEqualityComparer<T> generic interface.
HashSet<T>	Represents a set of values.
KeyedByTypeCollection<TItem>	Provides a collection whose items are types that serve as keys.
KeyNotFoundException	The exception that is thrown when the key specified for accessing an element in a collection does not match any key in the collection.
KeyValuePair	Creates instances of the KeyValuePair<TKey,TValue> struct.
LinkedList<T>	Represents a doubly linked list.
LinkedListNode<T>	Represents a node in a LinkedList<T>. This class cannot be inherited.
List<T>	Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.
PriorityQueue<TElement,TPriority>.UnorderedItemsCollection	Enumerates the contents of a PriorityQueue<TElement,TPriority>, without any ordering guarantees.
PriorityQueue<TElement,TPriority>	Represents a collection of items that have a value and a priority. On dequeue, the item with the lowest priority value is removed.
Queue<T>	Represents a first-in, first-out collection of objects.
ReferenceEqualityComparer	An IEqualityComparer<T> that uses reference equality (ReferenceEquals(Object, Object)) instead of value equality (Equals(Object)) when comparing two object instances.
SortedDictionary<TKey,TValue>.KeyCollection	Represents the collection of keys in a SortedDictionary<TKey,TValue>. This class cannot be inherited.
SortedDictionary<TKey,TValue>.ValueCollection	Represents the collection of values in a SortedDictionary<TKey,TValue>. This class cannot be

Collection	SortedDictionary<TKey,TValue>. This class cannot be inherited.
SortedDictionary<TKey,TValue>	Represents a collection of key/value pairs that are sorted on the key.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
SortedSet<T>	Represents a collection of objects that is maintained in sorted order.
Stack<T>	Represents a variable size last-in-first-out (LIFO) collection of instances of the same specified type.
SynchronizedCollection<T>	Provides a thread-safe collection that contains objects of a type specified by the generic parameter as elements.
SynchronizedKeyedCollection<K,T>	Provides a thread-safe collection that contains objects of a type specified by a generic parameter and that are grouped by keys.
SynchronizedReadOnlyCollection<T>	Provides a thread-safe, read-only collection that contains objects of a type specified by the generic parameter as elements.

Structs

Dictionary<TKey,TValue>.Enumerator	Enumerates the elements of a Dictionary<TKey,TValue>.
Dictionary<TKey,TValue>.KeyCollection.Enumerator	Enumerates the elements of a Dictionary<TKey,TValue>.KeyCollection.
Dictionary<TKey,TValue>.ValueCollection.Enumerator	Enumerates the elements of a Dictionary<TKey,TValue>.ValueCollection.
HashSet<T>.Enumerator	Enumerates the elements of a HashSet<T> object.
KeyValuePair<TKey,TValue>	Defines a key/value pair that can be set or retrieved.
LinkedList<T>.Enumerator	Enumerates the elements of a LinkedList<T>.
List<T>.Enumerator	Enumerates the elements of a List<T>.
PriorityQueue<TElement,TPriority>.UnorderedItemsCollection.Enumerator	Enumerates the element and priority pairs of a PriorityQueue<TElement,TPriority>, without any ordering guarantees.
Queue<T>.Enumerator	Enumerates the elements of a Queue<T>.
SortedDictionary<TKey,TValue>.Enumerator	Enumerates the elements of a SortedDictionary<TKey,TValue>.
SortedDictionary<TKey,TValue>.KeyCollection.Enumerator	Enumerates the elements of a SortedDictionary<TKey,TValue>.KeyCollection.
SortedDictionary<TKey,TValue>.ValueCollection.Enumerator	Enumerates the elements of a SortedDictionary<TKey,TValue>.ValueCollection.
SortedSet<T>.Enumerator	Enumerates the elements of a SortedSet<T> object.
Stack<T>.Enumerator	Enumerates the elements of a Stack<T>.

Interfaces

IAsyncEnumerable<T>	Exposes an enumerator that provides asynchronous iteration over values of a specified type.
IAsyncEnumerator<T>	Supports a simple asynchronous iteration over a generic collection.
ICollection<T>	Defines methods to manipulate generic collections.
IComparer<T>	Defines a method that a type implements to compare two objects.
IDictionary<TKey,TValue>	Represents a generic collection of key/value pairs.
IEnumerable<T>	Exposes the enumerator, which supports a simple iteration over a collection of a specified type.
IEnumerator<T>	Supports a simple iteration over a generic collection.
IEqualityComparer<T>	Defines methods to support the comparison of objects for equality.
IList<T>	Represents a collection of objects that can be individually accessed by index.
IReadOnlyCollection<T>	Represents a strongly-typed, read-only collection of elements.
IReadOnly Dictionary<TKey,TValue>	Represents a generic read-only collection of key/value pairs.
IReadOnlyList<T>	Represents a read-only collection of elements that can be accessed by index.
IReadOnlySet<T>	Provides a readonly abstraction of a set.
ISet<T>	Provides the base interface for the abstraction of sets.

- Many of the generic collection types are direct analogs of nongeneric types. Dictionary<TKey,TValue> is a generic version of Hashtable; it uses the generic structure KeyValuePair<TKey,TValue> for enumeration instead of DictionaryEntry. List is a generic version of ArrayList. There are generic Queue and Stack classes that correspond to the nongeneric versions. There are generic and nongeneric versions of SortedList<TKey,TValue>. Both versions are hybrids of a dictionary and a list. The SortedDictionary<TKey,TValue> generic class is a pure dictionary and has no nongeneric counterpart. The LinkedList generic class is a true linked list and has no nongeneric counterpart.

List Class

- Namespace: System.Collections.Generic
- Assembly: System.Collections.dll

Represents a strongly typed list of objects that can be accessed by index. Provides methods to search, sort, and manipulate lists.

```
public class List<T> : System.Collections.Generic.ICollection<T>,
System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IList<T>,
```

Type Parameters

T :The type of elements in the list

Inheritance : Object -> List

Implements : ICollection IEnumerable IList IReadOnlyCollection IReadOnlyList ICollection IEnumerable IList

- List class represents the list of objects which can be accessed by index. It comes under the System.Collections.Generic namespace. List class can be used to create a collection of different types like integers, strings etc. List class also provides the methods to search, sort, and manipulate lists.

Characteristics:

- It is different from the arrays. A List can be resized dynamically but arrays cannot.
- List class can accept null as a valid value for reference types and it also allows duplicate elements.
- If the Count becomes equals to Capacity, then the capacity of the List increased automatically by reallocating the internal array. The existing elements will be copied to the new array before the addition of the new element.
- List class is the generic equivalent of ArrayList class by implementing the IList generic interface.
- This class can use both equality and ordering comparer.
- List class is not sorted by default and elements are accessed by zero-based index.
- For very large List objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.

Constructors

List<T>()	Initializes a new instance of the List<T> class that is empty and has the default initial capacity.
List<T>(IEnumerable<T>)	Initializes a new instance of the List<T> class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied.
List<T>(Int32)	Initializes a new instance of the List<T> class that is empty and has the specified initial capacity.

Example

```
// C# program to create a List<T>
using System;
using System.Collections.Generic;
```

Properties

Capacity	Gets or sets the total number of elements the internal data structure can hold without resizing.
Count	Gets the number of elements contained in the List<T>.
Item[Int32]	Gets or sets the element at the specified index.

Methods

Add(T)	Adds an object to the end of the List<T>.
AddRange(IEnumerable<T>)	Adds the elements of the specified collection to the end of the List<T>.
AsReadOnly()	Returns a read-only ReadOnlyCollection<T> wrapper for the current collection.
BinarySearch(Int32, Int32, T, IComparer<T>)	Searches a range of elements in the sorted List<T> for an element using the specified comparer and returns the zero-based index of the element.
BinarySearch(T)	Searches the entire sorted List<T> for an element using the default comparer and returns the zero-based index of the element.
BinarySearch(T, IComparer<T>)	Searches the entire sorted List<T> for an element using the specified comparer and returns the zero-based index of the element.
Clear()	Removes all elements from the List<T>.
Contains(T)	Determines whether an element is in the List<T>.
ConvertAll<TOutput> (Converter<T,TOutput>)	Converts the elements in the current List<T> to another type, and returns a list containing the converted elements.
CopyTo(Int32, T[], Int32, Int32)	Copies a range of elements from the List<T> to a compatible one-dimensional array, starting at the specified index of the target array.
CopyTo(T[])	Copies the entire List<T> to a compatible one-dimensional array, starting at the beginning of the target array.
CopyTo(T[], Int32)	Copies the entire List<T> to a compatible one-dimensional array, starting at the specified index of the target array.
EnsureCapacity(Int32)	Ensures that the capacity of this list is at least the specified `capacity`. If the current capacity is less than `capacity`, it is increased to at least the specified `capacity`.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
Exists(Predicate<T>)	Determines whether the List<T> contains elements that match the conditions defined by the specified predicate.
Find(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the first occurrence within the entire List<T>.
FindAll(Predicate<T>)	Retrieves all the elements that match the conditions defined by the specified predicate.
FindIndex(Int32, Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the range of elements in the List<T> that starts at the specified index and contains the specified number of elements.
FindIndex(Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the

	range of elements in the List<T> that extends from the specified index to the last element.
FindIndex(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the first occurrence within the entire List<T>.
FindLast(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the last occurrence within the entire List<T>.
FindLastIndex(Int32, Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the range of elements in the List<T> that contains the specified number of elements and ends at the specified index.
FindLastIndex(Int32, Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the range of elements in the List<T> that extends from the first element to the specified index.
FindLastIndex(Predicate<T>)	Searches for an element that matches the conditions defined by the specified predicate, and returns the zero-based index of the last occurrence within the entire List<T>.
ForEach(Action<T>)	Performs the specified action on each element of the List<T>.
GetEnumerator()	Returns an enumerator that iterates through the List<T>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetRange(Int32, Int32)	Creates a shallow copy of a range of elements in the source List<T>.
GetType()	Gets the Type of the current instance. (Inherited from Object)
IndexOf(T)	Searches for the specified object and returns the zero-based index of the first occurrence within the entire List<T>.
IndexOf(T, Int32)	Searches for the specified object and returns the zero-based index of the first occurrence within the range of elements in the List<T> that extends from the specified index to the last element.
IndexOf(T, Int32, Int32)	Searches for the specified object and returns the zero-based index of the first occurrence within the range of elements in the List<T> that starts at the specified index and contains the specified number of elements.
Insert(Int32, T)	Inserts an element into the List<T> at the specified index.

InsertRange(Int32, IEnumerable<T>)	Inserts the elements of a collection into the List<T> at the specified index.
LastIndexOf(T)	Searches for the specified object and returns the zero-based index of the last occurrence within the entire List<T>.
LastIndexOf(T, Int32)	Searches for the specified object and returns the zero-based index of the last occurrence within the range of elements in the List<T> that extends from the first element to the specified index.
LastIndexOf(T, Int32, Int32)	Searches for the specified object and returns the zero-based index of the last occurrence within the range of elements in the List<T> that contains the specified number of elements and ends at the specified index.
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
Remove(T)	Removes the first occurrence of a specific object from the List<T>.
RemoveAll(Predicate<T>)	Removes all the elements that match the conditions defined by the specified predicate.
RemoveAt(Int32)	Removes the element at the specified index of the List<T>.
RemoveRange(Int32, Int32)	Removes a range of elements from the List<T>.
Reverse()	Reverses the order of the elements in the entire List<T>.
Reverse(Int32, Int32)	Reverses the order of the elements in the specified range.
Slice(Int32, Int32)	Creates a shallow copy of a range of elements in the source List<T>.
Sort()	Sorts the elements in the entire List<T> using the default comparer.
Sort(Comparison<T>)	Sorts the elements in the entire List<T> using the specified Comparison<T>.
Sort(IComparer<T>)	Sorts the elements in the entire List<T> using the specified comparer.
Sort(Int32, Int32, IComparer<T>)	Sorts the elements in a range of elements in List<T> using the specified comparer.
ToArray()	Copies the elements of the List<T> to a new array.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity to the actual number of elements in the List<T>, if that number is less than a threshold value.
TrueForAll(Predicate<T>)	Determines whether every element in the List<T> matches the conditions defined by the specified predicate.

Example

```
// C# Program to remove the element at
// the specified index of the List<T>
using System;
using System.Collections.Generic;

class Demo {

    // Main Method
    public static void Main(String[] args)
    {

        // Creating an List<T> of Integers
        List<int> firstlist = new List<int>();

        // Adding elements to List
        firstlist.Add(17);
        firstlist.Add(19);
        firstlist.Add(21);
        firstlist.Add(9);
        firstlist.Add(75);
        firstlist.Add(19);
        firstlist.Add(73);

        Console.WriteLine("Elements Present in List:\n");

        int p = 0;

        // Displaying the elements of List
        foreach(int k in firstlist)
        {
            Console.Write("At Position {0}: ", p);
            Console.WriteLine(k);
            p++;
        }

        Console.WriteLine(" ");

        // removing the element at index 3
        Console.WriteLine("Removing the element at index 3\n");

        // 9 will remove from the List
        // and 75 will come at index 3
        firstlist.RemoveAt(3);

        int p1 = 0;

        // Displaying the elements of List
        foreach(int n in firstlist)
        {
            Console.Write("At Position {0}: ", p1);
            Console.WriteLine(n);
            p1++;
```

```
        }
    }
}
/*
Elements Present in List:

At Position 0: 17
At Position 1: 19
At Position 2: 21
At Position 3: 9
At Position 4: 75
At Position 5: 19
At Position 6: 73

Removing the element at index 3

At Position 0: 17
At Position 1: 19
At Position 2: 21
At Position 3: 75
At Position 4: 19
At Position 5: 73
*/
```

Reference

Stack class

- Stack represents a last-in, first out collection of object. It is used when you need a last-in, first-out access to items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item. This class comes under System.Collections.Generic namespace.

Characteristics of Stack Class:

- Stack is implemented as an array.
- Stacks and queues are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Use Queue if you need to access the information in the same order that it is stored in the collection. Use System.Collections.Generic.Stack if you need to access the information in reverse order.
- Use the System.Collections.Concurrent.ConcurrentStack and System.Collections.Concurrent.ConcurrentQueue types when you need to access the collection from multiple threads concurrently.
- A common use for System.Collections.Generic.Stack is to preserve variable states during calls to other procedures.
- Three main operations can be performed on a System.Collections.Generic.Stack and its elements: - Push inserts an element at the top of the Stack. - Pop removes an element from the top of the Stack. - Peek returns an element that is at the top of the Stack but does not remove it from the Stack.

- The capacity of a Stack is the number of elements the Stack can hold. As elements are added to a Stack, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.
- Stack accepts null as a valid value for reference types and allows duplicate elements.
- The following example shows how to create and add values to a Stack and how to display its values.

```
using System;
using System.Collections.Generic;

class Example
{
    public static void Main()
    {
        Stack<string> numbers = new Stack<string>();
        numbers.Push("one");
        numbers.Push("two");
        numbers.Push("three");
        numbers.Push("four");
        numbers.Push("five");

        // A stack can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nPopping '{0}'", numbers.Pop());
        Console.WriteLine("Peek at next item to destack: {0}",
            numbers.Peek());
        Console.WriteLine("Popping '{0}'", numbers.Pop());

        // Create a copy of the stack, using the ToArray method and the
        // constructor that accepts an IEnumerable<T>.
        Stack<string> stack2 = new Stack<string>(numbers.ToArray());

        Console.WriteLine("\nContents of the first copy:");
        foreach( string number in stack2 )
        {
            Console.WriteLine(number);
        }

        // Create an array twice the size of the stack and copy the
        // elements of the stack, starting at the middle of the
        // array.
        string[] array2 = new string[numbers.Count * 2];
        numbers.CopyTo(array2, numbers.Count);

        // Create a second stack, using the constructor that accepts an
        // IEnumerable(Of T).
        Stack<string> stack3 = new Stack<string>(array2);

        Console.WriteLine("\nContents of the second copy, with duplicates and
```

```
nulls:");
        foreach( string number in stack3 )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nstack2.Contains(\"four\") = {0}",
            stack2.Contains("four"));

        Console.WriteLine("\nstack2.Clear()");
        stack2.Clear();
        Console.WriteLine("\nstack2.Count = {0}", stack2.Count);
    }
}

/* This code example produces the following output:

five
four
three
two
one

Popping 'five'
Peek at next item to destack: four
Popping 'four'

Contents of the first copy:
one
two
three

Contents of the second copy, with duplicates and nulls:
one
two
three




stack2.Contains("four") = False

stack2.Clear()

stack2.Count = 0
 */
```


Constructors

Stack()	Initializes a new instance of the Stack class that is empty and has the default initial capacity.
Stack(ICollection)	Initializes a new instance of the Stack class that contains elements copied from the specified collection and has the same initial capacity as the number of elements copied.
Stack(Int32)	Initializes a new instance of the Stack class that is empty and has the specified initial capacity or the default initial capacity, whichever is greater.

Properties

Count	Gets the number of elements contained in the Stack.
IsSynchronized	Gets a value indicating whether access to the Stack is synchronized (thread safe).
SyncRoot	Gets an object that can be used to synchronize access to the Stack.

Methods

Clear()	Removes all objects from the Stack.
Clone()	Creates a shallow copy of the Stack.
Contains(Object)	Determines whether an element is in the Stack.
CopyTo(Array, Int32)	Copies the Stack to an existing one-dimensional Array, starting at the specified array index.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an IEnumerator for the Stack.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object. (Inherited from Object)
Peek()	Returns the object at the top of the Stack without removing it.
Pop()	Removes and returns the object at the top of the Stack.
Push(Object)	Inserts an object at the top of the Stack.
Synchronized(Stack)	Returns a synchronized (thread safe) wrapper for the Stack.
ToArray()	Copies the Stack to a new array.
ToString()	Returns a string that represents the current object. (Inherited from Object)

Reference

Queue class

Namespace: System.Collections.Generic Inheritance: Object -> Queue Implements: IEnumerable IReadOnlyCollection ICollection IEnumerable

- This class implements a generic queue as a circular array. Objects stored in a Queue are inserted at one end and removed from the other. Queues and stacks are useful when you need temporary storage for information; that is, when you might want to discard an element after retrieving its value. Use Queue if you need to access the information in the same order that it is stored in the collection. Use Stack if you need to access the information in reverse order. Use ConcurrentQueue or ConcurrentStack if you need to access the collection from multiple threads concurrently.
- Three main operations can be performed on a Queue and its elements: - Enqueue adds an element to the end of the Queue. - Dequeue removes the oldest element from the start of the Queue. - Peek peek

returns the oldest element that is at the start of the Queue but does not remove it from the Queue.

- The capacity of a Queue is the number of elements the Queue can hold. As elements are added to a Queue, the capacity is automatically increased as required by reallocating the internal array. The capacity can be decreased by calling TrimExcess.
- Queue accepts null as a valid value for reference types and allows duplicate elements.

Examples

The following code example demonstrates several methods of the Queue generic class. The code example creates a queue of strings with default capacity and uses the Enqueue method to queue five strings. The elements of the queue are enumerated, which does not change the state of the queue. The Dequeue method is used to dequeue the first string. The Peek method is used to look at the next item in the queue, and then the Dequeue method is used to dequeue it.

The ToArray method is used to create an array and copy the queue elements to it, then the array is passed to the Queue constructor that takes IEnumerable, creating a copy of the queue. The elements of the copy are displayed.

An array twice the size of the queue is created, and the CopyTo method is used to copy the array elements beginning at the middle of the array. The Queue constructor is used again to create a second copy of the queue containing three null elements at the beginning.

The Contains method is used to show that the string "four" is in the first copy of the queue, after which the Clear method clears the copy and the Count property shows that the queue is empty.


```
using System;
using System.Collections.Generic;

class Example
{
    public static void Main()
    {
        Queue<string> numbers = new Queue<string>();
        numbers.Enqueue("one");
        numbers.Enqueue("two");
        numbers.Enqueue("three");
        numbers.Enqueue("four");
        numbers.Enqueue("five");

        // A queue can be enumerated without disturbing its contents.
        foreach( string number in numbers )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nDequeuing '{0}'", numbers.Dequeue());
        Console.WriteLine("Peek at next item to dequeue: {0}",
            numbers.Peek());
        Console.WriteLine("Dequeuing '{0}'", numbers.Dequeue());
```

```
        // Create a copy of the queue, using the ToArray method and the
        // constructor that accepts an IEnumerable<T>.
        Queue<string> queueCopy = new Queue<string>(numbers.ToArray());

        Console.WriteLine("\nContents of the first copy:");
        foreach( string number in queueCopy )
        {
            Console.WriteLine(number);
        }

        // Create an array twice the size of the queue and copy the
        // elements of the queue, starting at the middle of the
        // array.
        string[] array2 = new string[numbers.Count * 2];
        numbers.CopyTo(array2, numbers.Count);

        // Create a second queue, using the constructor that accepts an
        // IEnumerable(Of T).
        Queue<string> queueCopy2 = new Queue<string>(array2);

        Console.WriteLine("\nContents of the second copy, with duplicates and
nulls:");
        foreach( string number in queueCopy2 )
        {
            Console.WriteLine(number);
        }

        Console.WriteLine("\nqueueCopy.Contains(\"four\") = {0}",
            queueCopy.Contains("four"));

        Console.WriteLine("\nqueueCopy.Clear()");
        queueCopy.Clear();
        Console.WriteLine("\nqueueCopy.Count = {0}", queueCopy.Count);
    }
}

/* This code example produces the following output:

one
two
three
four
five

Dequeuing 'one'
Peek at next item to dequeue: two
Dequeuing 'two'

Contents of the first copy:
three
four
five
```

Constructor

Queue<T>()	Initializes a new instance of the Queue<T> class that is empty and has the default initial capacity.
Queue<T>(IEnumerable<T>)	Initializes a new instance of the Queue<T> class that contains elements copied from the specified collection and has sufficient capacity to accommodate the number of elements copied.
Queue<T>(Int32)	Initializes a new instance of the Queue<T> class that is empty and has the specified initial capacity.

Properties

Count	Gets the number of elements contained in the Queue<T>.

Methods

Clear()	Removes all objects from the Queue<T>.
Contains(T)	Determines whether an element is in the Queue<T>.
CopyTo(T[], Int32)	Copies the Queue<T> elements to an existing one-dimensional Array, starting at the specified array index.
Dequeue()	Removes and returns the object at the beginning of the Queue<T>.
Enqueue(T)	Adds an object to the end of the Queue<T>.
Ensure Capacity(Int32)	Ensures that the capacity of this queue is at least the specified `capacity`. If the current capacity is less than `capacity`, it is increased to at least the specified `capacity`.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the Queue<T>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetType()	Gets the Type of the current instance. (Inherited from Object)
Memberwise Clone()	Creates a shallow copy of the current Object. (Inherited from Object)
Peek()	Returns the object at the beginning of the Queue<T> without removing it.
ToArray()	Copies the Queue<T> elements to a new array.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity to the actual number of elements in the Queue<T>, if that number is less than 90 percent of current capacity.
TryDequeue(T)	Removes the object at the beginning of the Queue<T>, and copies it to the `result` parameter.
TryPeek(T)	Returns a value that indicates whether there is an object at the beginning of the Queue<T>, and if one is present, copies it to the `result` parameter. The object is not removed from the Queue<T>.

Dictionary class

- In C#, Dictionary is a generic collection which is generally used to store key/value pairs. The working of Dictionary is quite similar to the non-generic hashtable. The advantage of Dictionary is, it is generic type. Dictionary is defined under System.Collections.Generic namespace. It is dynamic in nature means the size of the dictionary is grows according to the need.

Important Points:

- The Dictionary class implements the - IDictionary<TKey,TValue> Interface - IReadOnlyCollection<KeyValuePair<TKey,TValue>> Interface - IReadOnlyDictionary<TKey,TValue> Interface - IDictionary Interface
- In Dictionary, the key cannot be null, but value can be.
- In Dictionary, key must be unique. Duplicate keys are not allowed if you try to use duplicate key then compiler will throw an exception.
- In Dictionary, you can only store same types of elements.
- The capacity of a Dictionary is the number of elements that Dictionary can hold.

Examples

The following code example creates an empty Dictionary<TKey,TValue> of strings with string keys and uses the Add method to add some elements. The example demonstrates that the Add method throws an ArgumentException when attempting to add a duplicate key.

The example uses the Item[] property (the indexer in C#) to retrieve values, demonstrating that a KeyNotFoundException is thrown when a requested key is not present, and showing that the value associated with a key can be replaced.

The example shows how to use the TryGetValue method as a more efficient way to retrieve values if a program often must try key values that are not in the dictionary, and it shows how to use the ContainsKey method to test whether a key exists before calling the Add method.

The example shows how to enumerate the keys and values in the dictionary and how to enumerate the keys and values alone using the Keys property and the Values property.

Finally, the example demonstrates the Remove method.


```
// Create a new dictionary of strings, with string keys.
//
Dictionary<string, string> openWith =
    new Dictionary<string, string>();

// Add some elements to the dictionary. There are no
// duplicate keys, but some of the values are duplicates.
openWith.Add("txt", "notepad.exe");
openWith.Add("bmp", "paint.exe");
openWith.Add("dib", "paint.exe");
openWith.Add("rtf", "wordpad.exe");

// The Add method throws an exception if the new key is
// already in the dictionary.
try
{
    openWith.Add("txt", "winword.exe");
}
catch (ArgumentException)
{
```

```
    Console.WriteLine("An element with Key = \"txt\" already exists.");
}

// The Item property is another name for the indexer, so you
// can omit its name when accessing elements.
Console.WriteLine("For key = \"rtf\", value = {0}.",
    openWith["rtf"]);

// The indexer can be used to change the value associated
// with a key.
openWith["rtf"] = "winword.exe";
Console.WriteLine("For key = \"rtf\", value = {0}.",
    openWith["rtf"]);

// If a key does not exist, setting the indexer for that key
// adds a new key/value pair.
openWith["doc"] = "winword.exe";

// The indexer throws an exception if the requested key is
// not in the dictionary.
try
{
    Console.WriteLine("For key = \"tif\", value = {0}.",
        openWith["tif"]);
}
catch (KeyNotFoundException)
{
    Console.WriteLine("Key = \"tif\" is not found.");
}

// When a program often has to try keys that turn out not to
// be in the dictionary, TryGetValue can be a more efficient
// way to retrieve values.
string value = "";
if (openWith.TryGetValue("tif", out value))
{
    Console.WriteLine("For key = \"tif\", value = {0}.", value);
}
else
{
    Console.WriteLine("Key = \"tif\" is not found.");
}


// ContainsKey can be used to test keys before inserting
// them.
if (!openWith.ContainsKey("ht"))
{
    openWith.Add("ht", "hypertrm.exe");
    Console.WriteLine("Value added for key = \"ht\": {0}",
        openWith["ht"]);
}

// When you use foreach to enumerate dictionary elements,
// the elements are retrieved as KeyValuePair objects.
```

```
Console.WriteLine();
foreach( KeyValuePair<string, string> kvp in openWith )
{
    Console.WriteLine("Key = {0}, Value = {1}",
        kvp.Key, kvp.Value);
}

// To get the values alone, use the Values property.
Dictionary<string, string>.ValueCollection valueColl =
    openWith.Values;

// The elements of the ValueCollection are strongly typed
// with the type that was specified for dictionary values.
Console.WriteLine();
foreach( string s in valueColl )
{
    Console.WriteLine("Value = {0}", s);
}

// To get the keys alone, use the Keys property.
Dictionary<string, string>.KeyCollection keyColl =
    openWith.Keys;

// The elements of the KeyCollection are strongly typed
// with the type that was specified for dictionary keys.
Console.WriteLine();
foreach( string s in keyColl )
{
    Console.WriteLine("Key = {0}", s);
}

// Use the Remove method to remove a key/value pair.
Console.WriteLine("\nRemove(\"doc\")");
openWith.Remove("doc");

if (!openWith.ContainsKey("doc"))
{
    Console.WriteLine("Key \"doc\" is not found.");
}

/* This code example produces the following output:

An element with Key = "txt" already exists.
For key = "rtf", value = wordpad.exe.
For key = "rtf", value = winword.exe.
Key = "tif" is not found.
Key = "tif" is not found.
Value added for key = "ht": hypertrm.exe

Key = txt, Value = notepad.exe
Key = bmp, Value = paint.exe
Key = dib, Value = paint.exe
Key = rtf, Value = winword.exe
Key = doc, Value = winword.exe
```

```
Key = ht, Value = hypertrm.exe

Value = notepad.exe
Value = paint.exe
Value = paint.exe
Value = winword.exe
Value = winword.exe
Value = hypertrm.exe

Key = txt
Key = bmp
Key = dib
Key = rtf
Key = doc
Key = ht

Remove("doc")
Key "doc" is not found.
*/
```

Remnarks

- The Dictionary<TKey,TValue> generic class provides a mapping from a set of keys to a set of values. Each addition to the dictionary consists of a value and its associated key. Retrieving a value by using its key is very fast, close to O(1), because the Dictionary<TKey,TValue> class is implemented as a hash table.
- As long as an object is used as a key in the Dictionary<TKey,TValue>, it must not change in any way that affects its hash value. Every key in a Dictionary<TKey,TValue> must be unique according to the dictionary's equality comparer. A key cannot be null, but a value can be, if its type TValue is a reference type.
- Dictionary<TKey,TValue> requires an equality implementation to determine whether keys are equal. You can specify an implementation of the IEqualityComparer generic interface by using a constructor that accepts a comparer parameter; if you do not specify an implementation, the default generic equality comparer EqualityComparer.Default is used. If type TKey implements the System.IEquatable generic interface, the default equality comparer uses that implementation.
- The capacity of a Dictionary<TKey,TValue> is the number of elements the Dictionary<TKey,TValue> can hold. As elements are added to a Dictionary<TKey,TValue>, the capacity is automatically increased as required by reallocating the internal array.
- .NET Framework only: For very large Dictionary<TKey,TValue> objects, you can increase the maximum capacity to 2 billion elements on a 64-bit system by setting the enabled attribute of the configuration element to true in the run-time environment.
- For purposes of enumeration, each item in the dictionary is treated as a KeyValuePair<TKey,TValue> structure representing a value and its key. The order in which the items are returned is undefined.

Constructor

Dictionary<TKey,TValue>()	Initializes a new instance of the Dictionary<TKey,TValue> class that is empty, has the default initial capacity, and uses the default equality comparer for the key type.
Dictionary<TKey,TValue> (IDictionary<TKey,TValue>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> and uses the default equality comparer for the key type.
Dictionary<TKey,TValue> (IDictionary<TKey,TValue>, IEquality Comparer<TKey>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IDictionary<TKey,TValue> and uses the specified IEqualityComparer<T>.
Dictionary<TKey,TValue>(IEnumerable<Key ValuePair<TKey,TValue>>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IEnumerable<T>.
Dictionary<TKey,TValue>(IEnumerable<Key ValuePair<TKey,TValue>>, IEquality Comparer<TKey>)	Initializes a new instance of the Dictionary<TKey,TValue> class that contains elements copied from the specified IEnumerable<T> and uses the specified IEqualityComparer<T>.
Dictionary<TKey,TValue>(IEquality Comparer<TKey>)	Initializes a new instance of the Dictionary<TKey,TValue> class that is empty, has the default initial capacity, and uses the specified IEqualityComparer<T>.
Dictionary<TKey,TValue>(Int32)	Initializes a new instance of the Dictionary<TKey,TValue> class that is empty, has the specified initial capacity, and uses the default equality comparer for the key type.
Dictionary<TKey,TValue>(Int32, IEquality Comparer<TKey>)	Initializes a new instance of the Dictionary<TKey,TValue> class that is empty, has the specified initial capacity, and uses the specified IEqualityComparer<T>.
Dictionary<TKey,TValue>(SerializationInfo, StreamingContext)	**Obsolete.** Initializes a new instance of the Dictionary<TKey,TValue> class with serialized data.

Properties

Comparer	Gets the IEqualityComparer<T> that is used to determine equality of keys for the dictionary.
Count	Gets the number of key/value pairs contained in the Dictionary<TKey,TValue>.
Item[TKey]	Gets or sets the value associated with the specified key.
Keys	Gets a collection containing the keys in the Dictionary<TKey,TValue>.
Values	Gets a collection containing the values in the Dictionary<TKey,TValue>.

Methods

OnDeserialization(Object)	Implements the ISerializable interface and raises the deserialization event when the deserialization is complete.
Remove(TKey)	Removes the value with the specified key from the Dictionary<TKey,TValue>.
Remove(TKey, TValue)	Removes the value with the specified key from the Dictionary<TKey,TValue>, and copies the element to the `value` parameter.
ToString()	Returns a string that represents the current object. (Inherited from Object)
TrimExcess()	Sets the capacity of this dictionary to what it would be if it had been originally initialized with all its entries.
TrimExcess(Int32)	Sets the capacity of this dictionary to hold up a specified number of entries without any further expansion of its backing storage.
TryAdd(TKey, TValue)	Attempts to add the specified key and value to the dictionary.
TryGetValue(TKey, TValue)	Gets the value associated with the specified key.

Add(TKey, TValue)	Adds the specified key and value to the dictionary.
Clear()	Removes all keys and values from the Dictionary<TKey,TValue>.
ContainsKey(TKey)	Determines whether the Dictionary<TKey,TValue> contains the specified key.
ContainsValue(TValue)	Determines whether the Dictionary<TKey,TValue> contains a specific value.
EnsureCapacity(Int32)	Ensures that the dictionary can hold up to a specified number of entries without any further expansion of its backing storage.
Equals(Object)	Determines whether the specified object is equal to the current object. (Inherited from Object)
GetEnumerator()	Returns an enumerator that iterates through the Dictionary<TKey,TValue>.
GetHashCode()	Serves as the default hash function. (Inherited from Object)
GetObjectData(SerializationInfo, StreamingContext)	**Obsolete.** Implements the ISerializable interface and returns the data needed to serialize the Dictionary<TKey,TValue> instance.
GetType()	Gets the Type of the current instance. (Inherited from Object)
MemberwiseClone()	Creates a shallow copy of the current Object.

(Inherited from Object)

Reflection :

Reflection objects are used for obtaining type information at runtime.
The classes that give access to the metadata of a running program are
in the System.Reflection namespace.

- Reflection has the following applications :
- It allows view attribute information at runtime.
- It allows examining various types in an assembly and instantiate these types.
- It allows late binding to methods and properties
- It allows creating new types at runtime and then performs some tasks using those types.

The MemberInfo object of the System.Reflection class needs to be initialized for discovering the attributes associated with a class.

System.Reflection.MemberInfo info = typeof(MyClass);

Reflection Attributes :

Attributes provide a powerful method of associating metadata, or declarative information, with code (assemblies, types, methods, properties, and so forth). After an attribute is associated with a program entity, the attribute can be queried at run time by using a technique called reflection.

Creating Custom Attributes :
Custom attributes are essentially traditional classes that derive directly or indirectly from System.Attribute

- Attributes are metadata extensions that give additional information to the compiler about the elements in the program code at runtime.
- Attributes are used to impose conditions or to increase the efficiency of a piece of code
- The primary steps to properly design custom attribute classes :
  1. Applying the AttributeUsageAttribute
  2. Declaring the attribute class
  3. Declaring constructors
  4. Declaring properties

1. Applying the AttributeUsageAttribute :

Finally, the class YourClass is inherited from the base class MyClass. The method MyMethod shows MyAttribute but not YourAttribute

```
public class YourClass : MyClass
{
// MyMethod will have MyAttribute but not YourAttribute.
public override void MyMethod()
{
//...
}
}
```

2. Declaring the attribute class :

```
[AttributeUsage(AttributeTargets.Method)]
public class MyAttribute : Attribute
{
// . . .
}
```

3. Declaring constructors :

```
public MyAttribute(bool myvalue)
{
this.myvalue = myvalue;
```

}

4. Declaring properties :

public bool MyProperty
{
get {return
this.myvalue;} set
{this.myvalue =
value;}
}

Loading Assembly at Runtime :

The act of loading external assemblies on demand is known as Dynamic Loading. Using the Assembly class, we can dynamically load both private and shared assemblies from the local location to a remote location as well as, explore its properties.

```
using System.Reflection;
namespace SessionLibDemo
{
    public class Program
    {
        static void Main(string[] args)
        {
            Assembly assembly = Assembly.LoadFile(@"E:\MSVS2022CDAC\SharedAssemblyDemo.dll");

            Type[] t = assembly.GetTypes();

            foreach (var i in t)
            {
                Console.WriteLine(i);
            }
        }
    }
}
```

Steps to create and use Shared Assembly:

Step 1 : Open VS.NET and Create a new Class Library

Step 2 : Generating Cryptographic Key Pair using the tool SN.Exe

- Make sure that you start Administrator “Developer Command Prompt for VS 2022”
- Write the following command on command prompt C:\> sn -k "C:\mynewkey.snk“

Step 3 : Sign the component with the key and build the class library project (Go to properties of the project in solution explorer -> select signing -> Check the checkbox of Sign the assembly and browse for the key).

Step 4 : Host the signed assembly in Global Assembly Cache
C:\>gacutil -i "E:\MyClassLibrary\bin\Debug\MyClassLibrary.dll"

Step 5 : Test the assembly by creating the client application. Just add the project reference and browse for the shared assembly recently created.

Step 6 : Execute the client program

Features in C#

- Partial Class
- Anonymous Methods
- Nullable Type
- Iterator
- Implicit Type
- Auto Property
- Object Initializer
- Anonymous Type
- Extension Methods
- Lambda Expression
- LINQ
- Partial Methods
- Dynamic Type
- Optional | Named Parameters
- Async | Await

Partial Class
In C#, a partial class is a class that can be split into multiple files. Each part of the partial class is declared with the partial keyword. When compiled, all the parts are combined into a single class definition.
Partial classes are often used in large projects or codebases to organize code more effectively, especially when multiple developers are working on different parts of the same class or when a class contains a significant amount of code. Here are some key points about partial classes:

1. **Syntax:** To declare a partial class, you use the partial keyword followed by the class keyword. For example.

```
public partial class MyClass
{
// Class members and methods
}
```

2. **Multiple Files:** Each part of the partial class can be defined in a separate file within the same namespace. All parts of the partial class must have the same access level modifier (public, internal, etc.) and be declared within the same assembly.
3. **Method Definitions:** Partial methods are a special feature of partial classes. A partial method is a method declaration without an implementation. One part of the partial class may declare a partial method, and another part may provide the implementation. If the implementation is not provided, the compiler removes the method call at compile time.

```
public partial class MyClass
{
partial void MyMethod();
}
// In another file:
public partial class MyClass
```

```
{
partial void MyMethod()
{
// Method implementation
}
}
```

4. **Benefits:** Partial classes can improve code organization, readability, and maintainability by allowing developers to focus on specific parts of a class without cluttering a single file with a large amount of code.

5. **Usage:** Partial classes are commonly used in GUI applications generated by visual designers (e.g., Windows Forms, WPF) to separate auto-generated code from developer-written code. They are also useful in frameworks and libraries where code generation or extension points are involved.

Anonymous Methods

Anonymous methods in C# allow you to define a method inline without specifying a name. They are particularly useful for defining event handlers or delegates where a simple, short method is required and defining a separate named method would be overkill. Here are some key points about anonymous methods:

1. **Syntax:** Anonymous methods are defined using the delegate keyword followed by a parameter list (if any) and a code block enclosed in curly braces. For example:

```
delegate(int x, int y)
{
 return x + y;
 };
```

2. **Usage with Delegates:** Anonymous methods are often used in conjunction with delegates. Instead of defining a separate named method and then passing it to a delegate, you can define the method inline where it's needed.
   For example:

```
Action<string> printMessage = delegate(string message)
{
    Console.WriteLine(message);
};

printMessage("Hello, world!");
```

3. **Capture of Outer Variables:** Anonymous methods can capture variables from the outer scope. This means they can access variables defined outside of their own scope. However, they can only capture variables by reference, not by value. This can lead to unexpected behavior if the outer variables are modified after the anonymous method is created.

4. **Replacement with Lambda Expressions:** Anonymous methods were largely replaced by lambda expressions in C# 3.0. Lambda expressions provide a more concise and expressive syntax for defining inline functions. The same functionality achieved with anonymous methods can usually be achieved more cleanly with lambda expressions.

```
Action<string> printMessage = (message) =>
{
    Console.WriteLine(message);
};
```

5. **Limitations:** Anonymous methods have some limitations compared to named methods or lambda expressions. For example, they cannot have a return type other than void, and they cannot be recursive

.

6. **Event Handlers:** Anonymous methods are commonly used for event handling in graphical user interfaces (GUIs). They allow you to define event handler logic inline, making the code more concise and easier to understand.

Anonymous methods provide a convenient way to define short, inline functions without the need for a separate named method. While they have largely been superseded by lambda expressions in modern C# code, they are still occasionally used in scenarios where lambda expressions are not applicable or where they would make the code less readable.

Nullable Type

In C#, a nullable type allows you to represent both a value type (like int, float, etc.) and the absence of a value (null). This is particularly useful when dealing with database fields or other scenarios where a value may or may not be present. Nullable types were introduced in C# 2.0.
The syntax for defining a nullable type is to append a question mark **?** to the type declaration. For example.

```
int? nullableInt;
float? nullableFloat;
```

Here are some key points about nullable types:

1. **Value or Null:** Nullable types allow a variable to hold either the underlying value type or a null reference. For example, int? can hold any integer value or be null.

2. **Nullable<T>:** Behind the scenes, nullable types are implemented using the Nullable<T> struct (also written as T?). This struct has two properties: HasValue which indicates whether the nullable type has a value, and Value which returns the underlying value if HasValue is true, or throws an exception if HasValue is false.

Iterator

In C#, an iterator is a block of code that enables you to iterate over a collection of items sequentially. It simplifies the process of iterating over collections like arrays, lists, or custom data structures by abstracting away the underlying implementation details of iteration. Iterators are commonly used with foreach loops.
Here's how iterators work in C#:

1. **Iterator Method:** To create an iterator, you define a method that returns an IEnumerable<T> or IEnumerator<T> object. This method uses the yield keyword to return each element of the collection one at a time.

2. **Yield Keyword:** The yield keyword is used within the iterator method to return each element of the collection. When the iterator method is called, execution starts from the beginning of the method until the first yield statement is reached. The method then returns the value specified by yield and suspends its execution. When the iterator is iterated again, execution resumes from where it left off until the next yield statement is encountered, and so on.

3. **IEnumerable<T> and IEnumerator<T>:** IEnumerable<T> represents a collection of items that can be enumerated, while IEnumerator<T> provides a way to iterate over the elements of the collection one at a time. The yield return statement is typically used with IEnumerable<T>, while the yield break statement can be used to prematurely end the iteration.

Implicit Type

Implicit typing in C# allows you to declare variables without explicitly specifying their data types. Instead, the compiler infers the data type based on the value assigned to the variable. This feature was introduced in C# 3.0 with the var keyword.

Here's how implicit typing works:

1. **Syntax:** Instead of explicitly specifying the data type, you use the var keyword followed by the variable name and an optional initializer. For example:

```
var number = 10;
var message = "Hello, world!";
```

2. **Type Inference:** When the compiler encounters a variable declared with var, it analyzes the expression on the right-hand side of the assignment operator (=) to determine its data type. The compiler then assigns the inferred type to the variable.

3. **Compile-Time Checking:** Although the type of the variable is not explicitly specified, the compiler performs type checking at compile time to ensure type safety. This means that the variable's type is determined at compile time, and the compiler enforces type rules accordingly.

4. **Static Typing:** Implicit typing does not change the statically-typed nature of C#. Once a variable's type is inferred, it cannot be changed. The variable is still statically typed, meaning its type is determined at compile time and cannot change during runtime.

5. **Readability:** Implicit typing can improve code readability by reducing verbosity, especially when variable names are descriptive and the inferred types are obvious from the assigned values.

6. **Limitations:** Implicit typing cannot be used in certain scenarios, such as when declaring fields, method parameters, or return types, where the data type must be explicitly specified. Additionally, using var with complex types or when the inferred type is not immediately obvious can reduce code clarity.

Here's an example illustrating the usage of implicit typing:

```
class Program
{
    static void Main()
```

```
{
    var number = 10;
    var message = "Hello, world!";

    Console.WriteLine($"Number: {number}, Type: {number.GetType()}");
    Console.WriteLine($"Message: {message}, Type: {message.GetType()}");
}}
```

In this example, the number variable is implicitly typed as an int, and the message variable is implicitly typed as a string. The GetType() method is used to retrieve the runtime type of each variable for demonstration purposes.

Auto Property

Auto-implemented properties in C# provide a shorthand syntax for defining properties of a class without explicitly declaring the backing field. This feature was introduced in C# 3.0.

Here's how auto-implemented properties work:

1. **Syntax:** Instead of explicitly declaring a backing field and writing separate getter and setter methods, you can use a simplified syntax to define the property directly within the class. The compiler automatically generates a private backing field for the property.

   ```
   public class MyClass
   {
       public int MyProperty { get; set; }
   }
   ```

2. **Access Modifiers:** You can specify access modifiers (public, private, protected, etc.) for auto-implemented properties just like regular properties. By default, auto-implemented properties are public.
3. **Initialization:** You can provide an initial value for an auto-implemented property directly within the property declaration.

   ```
   public class MyClass
   {
       public int MyProperty { get; set; } = 42;
   }
   ```

4. **Read-Only Auto Properties:** You can create read-only auto-implemented properties by omitting the setter. In this case, the property can only be assigned a value within the constructor or initializer.

   ```
   public class MyClass
   {
       public int MyReadOnlyProperty { get; } = 42;
   }
   ```

5. **Backing Field:** While auto-implemented properties do have a backing field generated by the compiler, you cannot directly access this field from within the class. It's purely an implementation detail.

Auto-implemented properties offer a concise and readable way to define properties in C#, especially for simple scenarios where custom getter/setter logic is not required. However, they are limited in flexibility compared to traditional properties, as you cannot directly manipulate the backing field or provide custom logic within the getter/setter methods.

Object Initializer

In C#, object initializers provide a concise syntax for initializing objects without explicitly invoking a constructor and setting individual properties one by one. This feature was introduced in C# 3.0 as part of the language enhancements.

Here's how object initializers work:

1. **Syntax:** Instead of using a constructor followed by property assignments, you can use the object initializer syntax to set properties directly within curly braces { } after the object creation expression.

   ```
   // Without object initializer
   MyClass obj = new MyClass();
   obj.Property1 = value1;
   obj.Property2 = value2;

   // With object initializer
   MyClass obj = new MyClass
   {
       Property1 = value1,
       Property2 = value2
   };
   ```

2. **Multiple Properties:** You can initialize multiple properties within the same object initializer block, separated by commas.

3. **Nested Object Initializers:** Object initializers can be nested to initialize properties of objects within objects.

   ```
   MyClass obj = new MyClass
   {
       NestedProperty = new NestedClass
       {
           Property1 = value1,
           Property2 = value2
       }
   };
   ```

4. **Anonymous Types:** Object initializers are often used with anonymous types to create objects with ad-hoc structures, especially in LINQ queries.

   ```
   var person = new { Name = "John", Age = 30 };
   ```

5. **Collection Initializers:** In addition to object initializers, C# also supports collection initializers, which allow you to initialize collections (such as lists, arrays, and dictionaries) using a similar syntax.

   ```
   List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
   ```

Object initializers provide a concise and readable way to initialize objects, especially when creating objects with many properties or nested objects. They contribute to cleaner and more expressive code, improving code maintainability and reducing verbosity.

Anonymous Type

In C#, an anonymous type is a type defined dynamically at compile-time without explicitly declaring a class structure. Anonymous types are useful when you need to create simple data structures on the fly without the overhead of defining a formal class.

Here are the key points about anonymous types:

1. **Syntax:** Anonymous types are defined using the new keyword followed by an object initializer within curly braces {}. The compiler infers the type of each property based on the assigned values.

   ```
   var person = new { Name = "John", Age = 30 };
   ```

2. **Properties:** Anonymous types have read-only properties. Once created, the properties cannot be modified. The property names and types are determined by the names and types of the assigned values.

3. **Use Cases:** Anonymous types are commonly used in LINQ queries when you want to project data into a temporary structure for immediate use. They are also useful for passing data between different parts of your code when defining a formal class would be overkill.

4. **Equality:** Anonymous types have value-based equality semantics. Two instances of the same anonymous type are considered equal if their corresponding properties have the same values.

   ```
   var person1 = new { Name = "John", Age = 30 };
   var person2 = new { Name = "John", Age = 30 };
   Console.WriteLine(person1.Equals(person2)); // Outputs: True
   ```

5. **Scope:** Anonymous types are generally scoped to the method or block in which they are defined. They cannot be used outside of their defining scope.

6. **Limitations:** Anonymous types cannot contain methods or events. They also cannot be used as method parameters or return types, as their type information is not available outside the scope where they are defined.

Extension Methods

Extension methods in C# allow you to add new methods to existing types (classes, structs, or interfaces) without modifying the original type. This feature enables you to "extend" the functionality of classes or interfaces that you don't control or can't modify, such as framework types or types from third-party libraries. Extension methods were introduced in C# 3.0.

Here are the key points about extension methods:

1. **Syntax:** Extension methods are defined as static methods within static classes. The first parameter of an extension method specifies the type that the method operates on, preceded by the this modifier. This indicates to the compiler that the method is an extension method.

```
public static class StringExtensions
{
    public static int WordCount(this string str)
    {
        return str.Split(new char[] { ' ', '.', '?' }, StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

2. **Usage:** Once defined, extension methods can be called as if they were instance methods of the extended type. The compiler automatically binds the call to the appropriate extension method based on the type of the first parameter

```
string text = "Hello world!";
int count = text.WordCount(); // Calls the extension method
```

3. **Visibility:** Extension methods are accessible wherever the containing namespace is visible. You don't need to import the namespace explicitly to use extension methods defined within it.

4. **Extension Method Chaining:** Extension methods can be chained together, allowing for fluent API design and concise code.

```
string text = "Hello world!";
int count = text.Trim().ToUpper().WordCount();
```

5. **Limitations:** Extension methods cannot access private or protected members of the extended type. They are also resolved statically, so they cannot be overridden in derived types.

6. **Naming:** It's a convention to name extension method classes with a suffix like "Extensions" to make their purpose clear.

Lambda Expression

Lambda expressions in C# provide a concise way to write anonymous functions or delegates. They are particularly useful for writing inline functions, especially when passing small pieces of code as arguments to methods or for defining event handlers, LINQ queries, and more. Lambda expressions were introduced in C# 3.0 as part of the language enhancements.

Here's how lambda expressions work:

1. **Syntax:** Lambda expressions consist of three parts: the parameter list, the lambda operator =>, and the expression or statement block. The parameter list can be empty or contain one or more parameters, enclosed in parentheses. The lambda operator separates the parameter list from the expression or statement block.

```
// Syntax: (parameters) => expression
Func<int, int, int> add = (x, y) => x + y;
```

2. **Type Inference:** Lambda expressions can be assigned to delegate types or used as arguments in method calls where the delegate type is expected. The compiler infers the delegate type based on the context in which the lambda expression is used.

3. **Single Statement:** If the body of the lambda expression consists of a single statement, you can omit the curly braces { }. The result of the expression is the result of the single statement.

   ```
   Func<int, int> square = x => x * x;
   ```

4. **Multiple Statements:** If the body of the lambda expression contains multiple statements, you must enclose them in curly braces { }. You can use the return keyword to return a value explicitly.

   ```
   Func<int, int> increment = x =>
   {
       int result = x + 1;
       return result;
   };
   ```

5. **Capture of Outer Variables:** Lambda expressions can capture variables from the outer scope (i.e., variables defined outside the lambda expression). The captured variables are kept alive as long as the delegate that holds the lambda expression is alive.

6. **Parameter Types:** Lambda expressions can have implicit or explicit parameter types. If the parameter types can be inferred from the context, you can omit them. Otherwise, you must specify them explicitly.

   ```
   // Implicit parameter types
   Func<int, int, int> add = (x, y) => x + y;

   // Explicit parameter types
   Func<int, int, int> add = (int x, int y) => x + y;
   ```

Lambda expressions provide a concise and expressive way to define inline functions in C#. They are widely used in modern C# programming, especially in LINQ queries, asynchronous programming with async and await, event handling, and functional programming paradigms.

LINQ

LINQ (Language Integrated Query) is a powerful feature in C# that enables developers to query and manipulate data using a SQL-like syntax directly within the C# language. LINQ was introduced in C# 3.0 as part of the .NET Framework 3.5.

Here are the key components and concepts of LINQ:

1. **Query Expressions:** LINQ allows you to write query expressions that resemble SQL queries. These expressions operate on collections of objects (e.g., arrays, lists, databases) and can include operations like filtering, sorting, grouping, joining, and projecting data.

```
var result = from person in people
             where person.Age > 18
             select person.Name;
```

2. **Standard Query Operators:** LINQ provides a set of standard query operators (methods) defined as extension methods in the System.Linq namespace. These operators allow you to perform various operations on collections, such as Where, Select, OrderBy, GroupBy, Join, Aggregate, and more.

```
var result = people.Where(person => person.Age > 18)
                   .Select(person => person.Name);
```

3. **Deferred Execution:** LINQ queries use deferred execution, meaning that the query is not executed immediately when it's defined. Instead, the query is executed when the results are enumerated (e.g., when iterating over the query result with a foreach loop).

4. **Integration with Lambda Expressions**: LINQ can be used with lambda expressions to provide more concise and flexible syntax for writing queries. Lambda expressions allow you to define inline functions for filtering, projecting, and sorting data.

```
var result = people.Where(person => person.Age > 18)
                   .Select(person => person.Name);
```

5. **LINQ to Objects:** LINQ can be used to query in-memory collections such as arrays, lists, and dictionaries. LINQ to Objects provides standard query operators for querying and manipulating these collections.

6. **LINQ to SQL, LINQ to Entities**: LINQ can be used to query relational databases using LINQ to SQL or LINQ to Entities. LINQ to SQL translates LINQ queries into SQL queries and executes them against the database, while LINQ to Entities works with Entity Framework to query and manipulate data stored in a database.

7. **LINQ to XML:** LINQ provides support for querying and manipulating XML documents using LINQ to XML. LINQ to XML allows you to write queries to search, filter, transform, and create XML documents.

LINQ is a versatile and powerful feature in C# that simplifies data querying and manipulation, improves code readability, and reduces the amount of boilerplate code needed for common data operations. It's widely used in various types of applications, including desktop, web, and mobile development.

Partial Methods

Partial methods in C# are methods that allow for separation of the method declaration (signature) from the method implementation. They enable you to define a method in one part of a partial class and provide the implementation in another part of the class, or optionally not provide an implementation at all. Partial methods are primarily used in code generation scenarios, such as when working with visual designers or auto-generated code.

Here are some key points about partial methods:

1. **Declaration:** Partial methods are declared using the partial keyword in one part of a partial class. The method signature is defined without specifying the method body.

```
public partial class MyClass
{
    partial void MyPartialMethod();
}
```

2. **Implementation:** The implementation of a partial method is provided in another part of the partial class. If an implementation is not provided, the compiler removes the method call at compile time.

```
public partial class MyClass
{
    partial void MyPartialMethod()
    {
        // Method implementation
    }
}
```

3. **Optional Implementation:** Partial methods are optional, meaning that you can define them in one part of the class and choose not to provide an implementation in another part. In this case, the method call is removed by the compiler, and any calls to the method are effectively no-ops.

4. **Usage:** Partial methods are commonly used in code generated by visual designers, such as Windows Forms or WPF designers in Visual Studio. They provide a way for developers to extend or customize the behavior of auto-generated code without modifying the generated code itself.

5. **Event Handlers:** Partial methods are also used as event handlers, especially in scenarios where the event may or may not have any subscribers. This allows for cleaner code and better performance by avoiding unnecessary event invocation.

6. **Return Type:** Partial methods can have void return type only. They cannot have any other return type, parameters, or access modifiers other than partial.

Partial methods provide a way to decouple method declaration from implementation, allowing for flexibility in extending or customizing code generated by tools or frameworks. They help improve code organization, readability, and maintainability, especially in large-scale projects or when working with code generated by automated tools.

Dynamic Type

In C#, the dynamic type allows you to declare variables whose types are resolved at runtime rather than compile time. This provides flexibility when working with types that are not known until runtime, such as objects from dynamic languages, COM objects, or objects whose types are determined dynamically.

Here are some key points about the dynamic type:

1. **Type Resolution at Runtime:** When a variable is declared as dynamic, the type of the variable is determined at runtime rather than compile time. This means that method and property resolution for dynamic variables is also deferred until runtime.

```
dynamic dynamicVariable = 10;
dynamicVariable = "Hello";
```

2. **No Compile-Time Type Checking:** Because the type of dynamic variables is determined at runtime, the compiler performs minimal type checking at compile time. This allows you to call methods and access properties that may not exist at compile time, but may be resolved at runtime.

   ```
   dynamic dynamicObject = GetDynamicObject();
   dynamicObject.SomeMethod(); // No compile-time error even if SomeMethod doesn't exist
   ```

3. **Late Binding:** Operations on dynamic variables are resolved at runtime through late binding, meaning that method calls and property accesses are resolved dynamically based on the actual type of the object at runtime.

4. **No IntelliSense Support:** Because the type of dynamic variables is determined at runtime, IDE features such as IntelliSense cannot provide compile-time information about members of dynamic objects. This can lead to reduced developer productivity and potential runtime errors.

5. **Performance Overhead:** Using dynamic can incur a performance overhead compared to statically typed code because of the late binding and runtime type resolution involved. However, this overhead is typically negligible in most scenarios.

6. **Use Cases:** The dynamic type is particularly useful when working with interoperability scenarios, such as calling dynamic languages like Python or JavaScript from C#, working with COM objects, or dealing with data from external sources where the schema is not known until runtime.

   ```
   dynamic excelApp = GetExcelApplication();
   dynamic worksheet = excelApp.Worksheets[1];
   var cellValue = worksheet.Cells[1, 1].Value;
   ```

While the dynamic type provides flexibility in certain scenarios, it should be used judiciously, as it bypasses compile-time type checking and can lead to harder-to-debug runtime errors if misused. It's typically best to use dynamic only when interacting with truly dynamic or unknown types, and to favor statically typed code wherever possible for improved safety and performance.

Optional | Named Parameters

Optional and named parameters in C# provide flexibility and readability when working with method parameters. Here's a breakdown of each feature:

Optional Parameters:

Optional parameters allow you to specify default values for parameters in method declarations. If a value is not provided when the method is called, the default value is used.
They are defined in method signatures by assigning a default value to parameters.
All optional parameters must be at the end of the parameter list.

Example:

```
public void SendMessage(string message, int priority = 1)
{
    Console.WriteLine($"Message: {message}, Priority: {priority}");
}

// Calling the method without specifying the optional parameter
SendMessage("Hello");  // Output: Message: Hello, Priority: 1
```

Named Parameters:

Named parameters allow you to specify arguments by name when calling methods, rather than by position.
This provides clarity and flexibility, especially when methods have many parameters or the order of parameters is not obvious.
Named parameters are specified by providing the parameter name followed by a colon (:) before the argument.
Example:

```
public void DisplayInfo(string name, int age)
{
    Console.WriteLine($"Name: {name}, Age: {age}");
}

// Calling the method with named parameters
DisplayInfo(age: 30, name: "Alice");   // Output: Name: Alice, Age: 30
```

Combining Optional and Named Parameters:

You can use optional and named parameters together.
Named parameters allow you to skip optional parameters or specify them out of order.

Example:

```
public void SendEmail(string to, string subject = "Hello", string cc = "")
{
    Console.WriteLine($"To: {to}, Subject: {subject}, CC: {cc}");
}

// Calling the method with named parameters and skipping optional parameters
SendEmail("example@example.com", cc: "copy@example.com");  // Output: To:
example@example.com, Subject: Hello, CC: copy@example.com
```

Optional and named parameters enhance code readability and maintainability by allowing for more expressive method calls. However, they should be used judiciously to avoid confusion, especially in public APIs where parameter changes may affect existing callers.

Async | Await

In C#, the async and await keywords are used to write asynchronous code in a more straightforward and readable manner. Here's how they work:

Async Methods:
The async modifier is used to define asynchronous methods. These methods can perform time-consuming operations without blocking the calling thread.

```
async Task MyAsyncMethod()
```

```
{
    // Asynchronous operations
}
```

Await Operator:
Inside an async method, the await operator is used to asynchronously wait for the completion of another asynchronous operation, such as a Task or Task<T>.

```
async Task<string> DownloadDataAsync()
{
    HttpClient client = new HttpClient();
    string data = await client.GetStringAsync("https://example.com");
    return data;
}
```

Awaiting Task:
When calling an async method, you use the await keyword before the method call. This allows the calling method to asynchronously wait for the completion of the async method.

```
string result = await DownloadDataAsync();
```

Exception Handling:
Async methods can use regular try-catch blocks to handle exceptions. Exceptions thrown during asynchronous operations are captured and propagated as if the code were synchronous.

```
try
{
    string result = await DownloadDataAsync();
    Console.WriteLine(result);
}
```

```
catch (Exception ex)
        {
            Console.WriteLine($"An error occurred: {ex.Message}");
        }
```

Async Event Handlers:
Async methods can be used as event handlers. For example, in UI applications, async event handlers can perform asynchronous operations without blocking the UI thread.

```
async void Button_Click(object sender, EventArgs e)
{
    string result = await
    DownloadDataAsync();
    MessageBox.Show(result);
}
```

Performance Benefits:
Asynchronous programming with async and await can improve the responsiveness and scalability of applications, especially in scenarios involving I/O-bound or CPU-bound operations.

```
Task.Run(() => MyMethod()); // Run CPU-bound operation asynchronously
```

Async and await keywords provide a more natural way to write asynchronous code in C#, making it easier to understand and maintain. They are widely used in modern C# applications for handling asynchronous operations efficiently and elegantly.

https://www.youtube.com/@codewitharrays

https://www.instagram.com/codewitharrays/

https://t.me/codewitharrays Group Link: https://t.me/ccee2025notes

+91 8007592194 +91 9284926333

codewitharrays@gmail.com

https://codewitharrays.in/project