

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).

Roslyn 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#

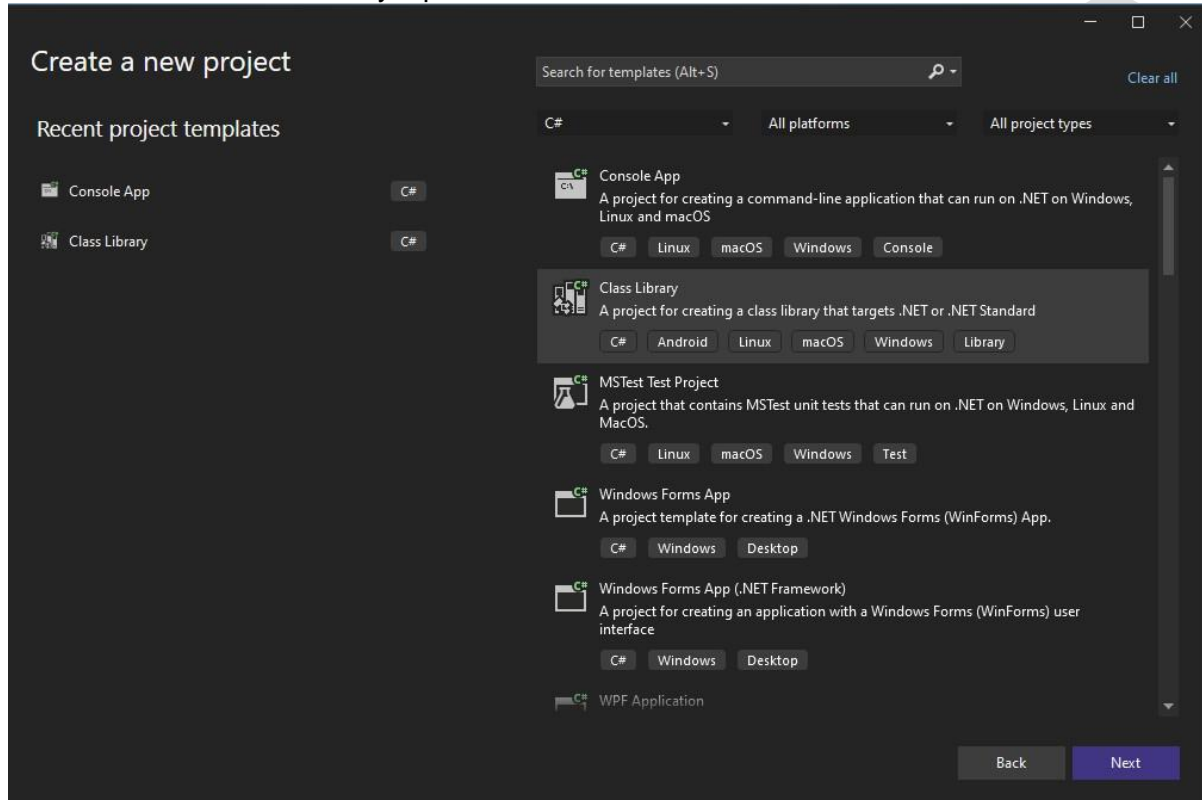
A class library file is a collection of classes and namespaces in C# without any entry point method like **Main**. Once we create a class library file it can be used in the C# project and classes inside it can be used as required. Class Library makes it convenient to use functionalities by importing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

Terminologies

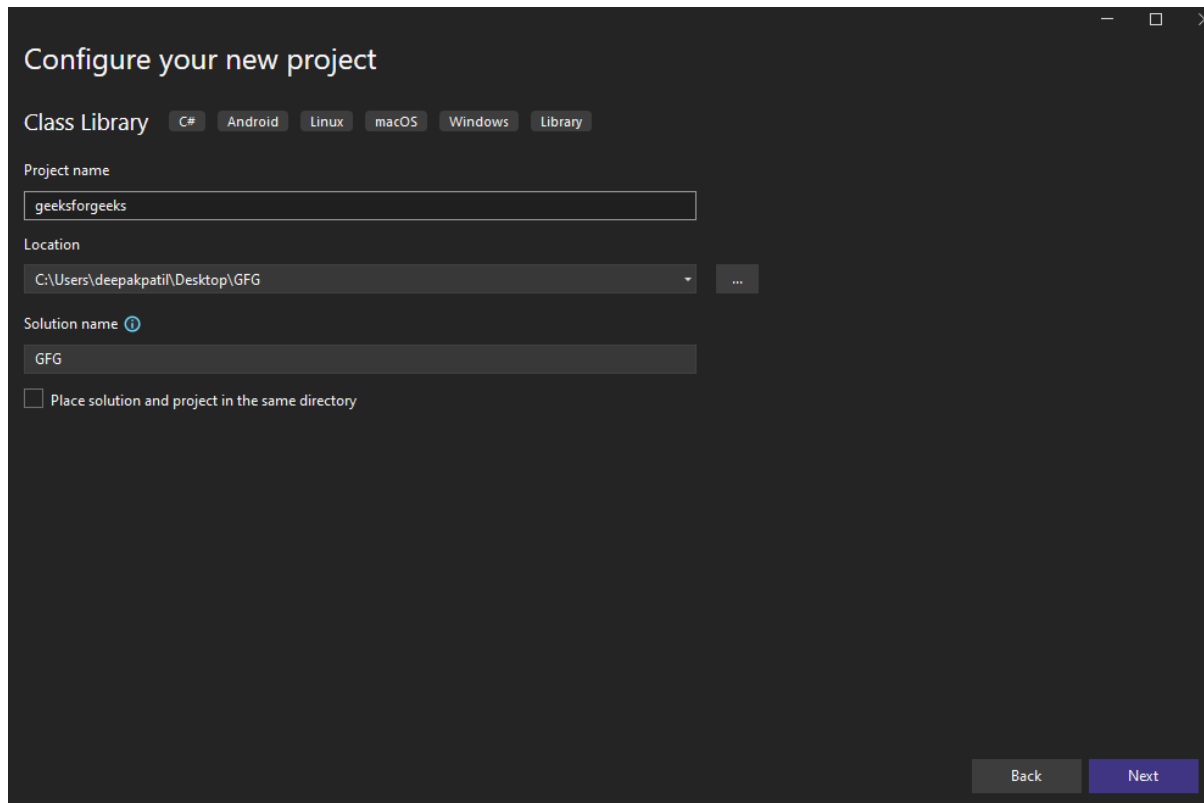
Class Library: It is a package or file that contains different namespaces and class definitions that are used by other programs.

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.



Configure your new project

Class Library C# Android Linux macOS Windows Library

Project name
geeksforgeeks

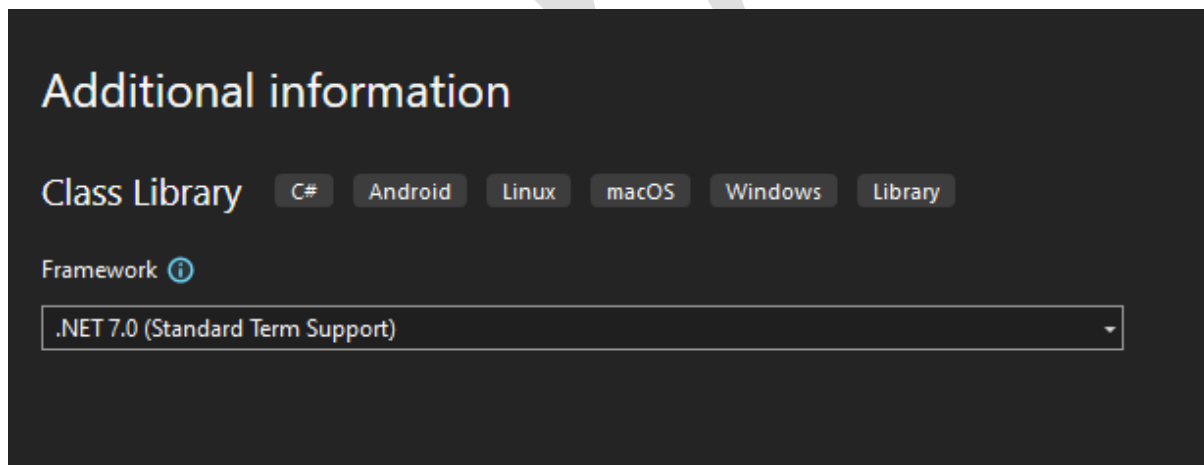
Location
C:\Users\deepakpatil\Desktop\GFG

Solution name ⓘ
GFG

☐ Place solution and project in the same directory

Back Next

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

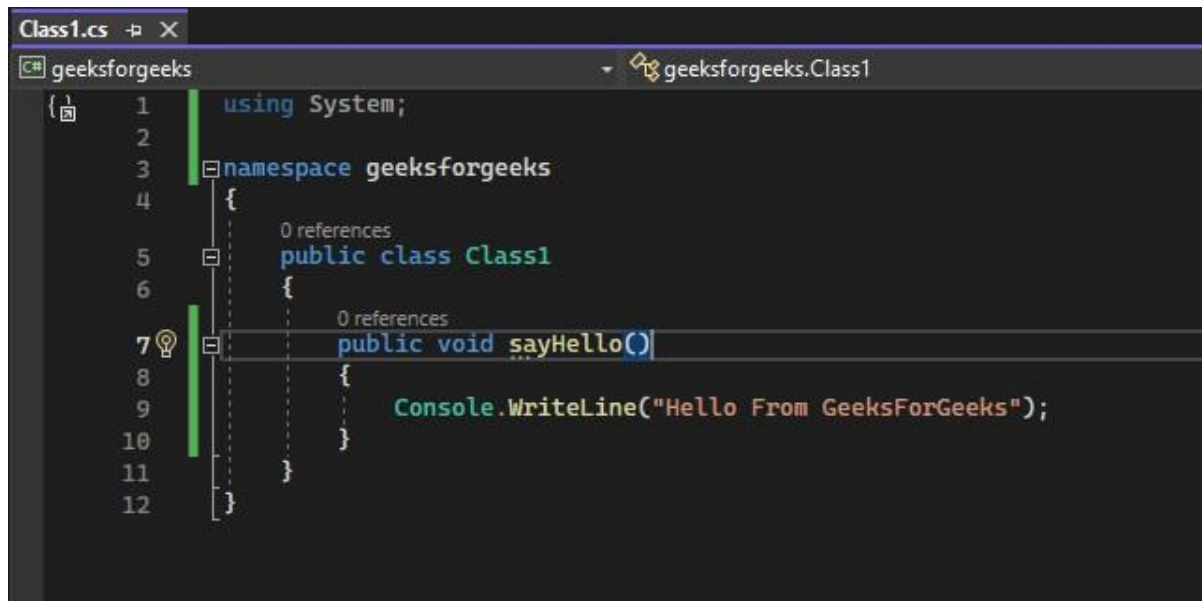


Additional information

Class Library C# Android Linux macOS Windows Library

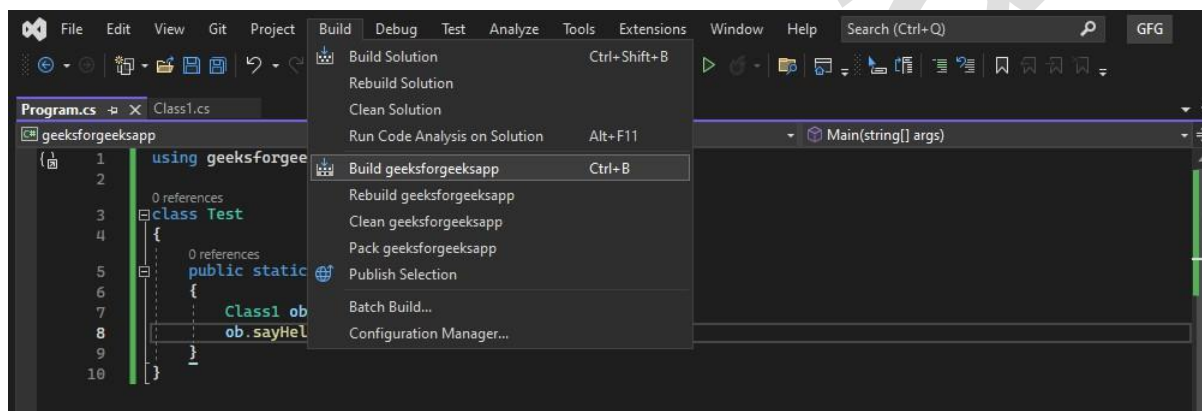
Framework ⓘ
.NET 7.0 (Standard Term Support)

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.

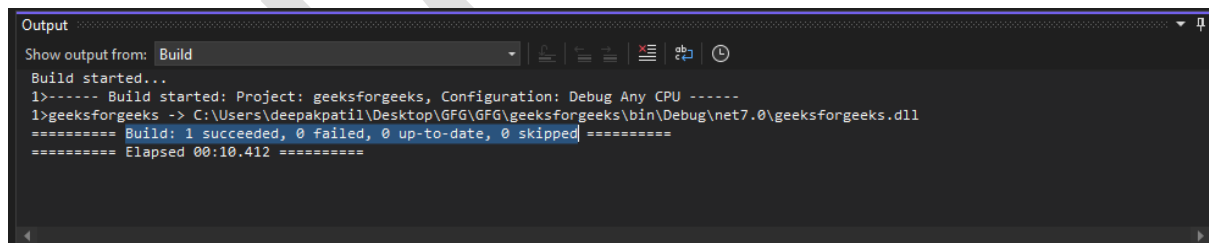


```
1 using System;
2
3 namespace geeksforgeeks
4 {
5     public class Class1
6     {
7         public void sayHello()
8         {
9             Console.WriteLine("Hello From GeeksForGeeks");
10        }
11    }
12 }
```

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



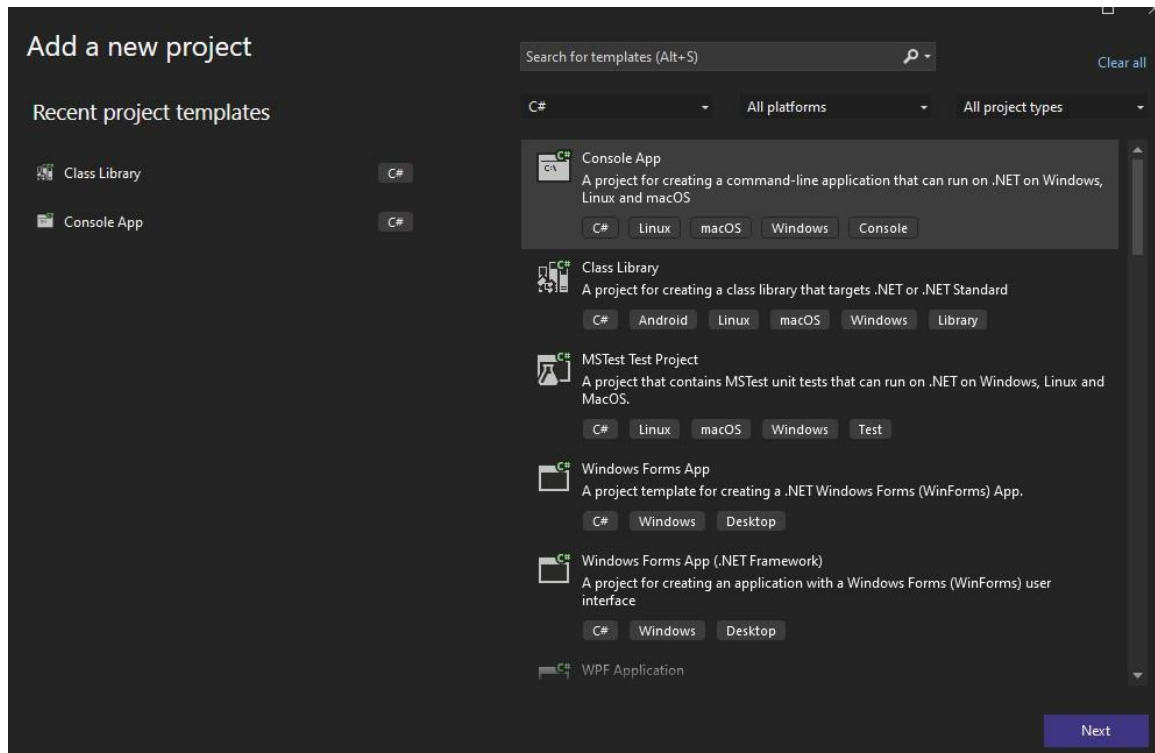
Step 6: If everything is correct you should get build success in the output below the editor.



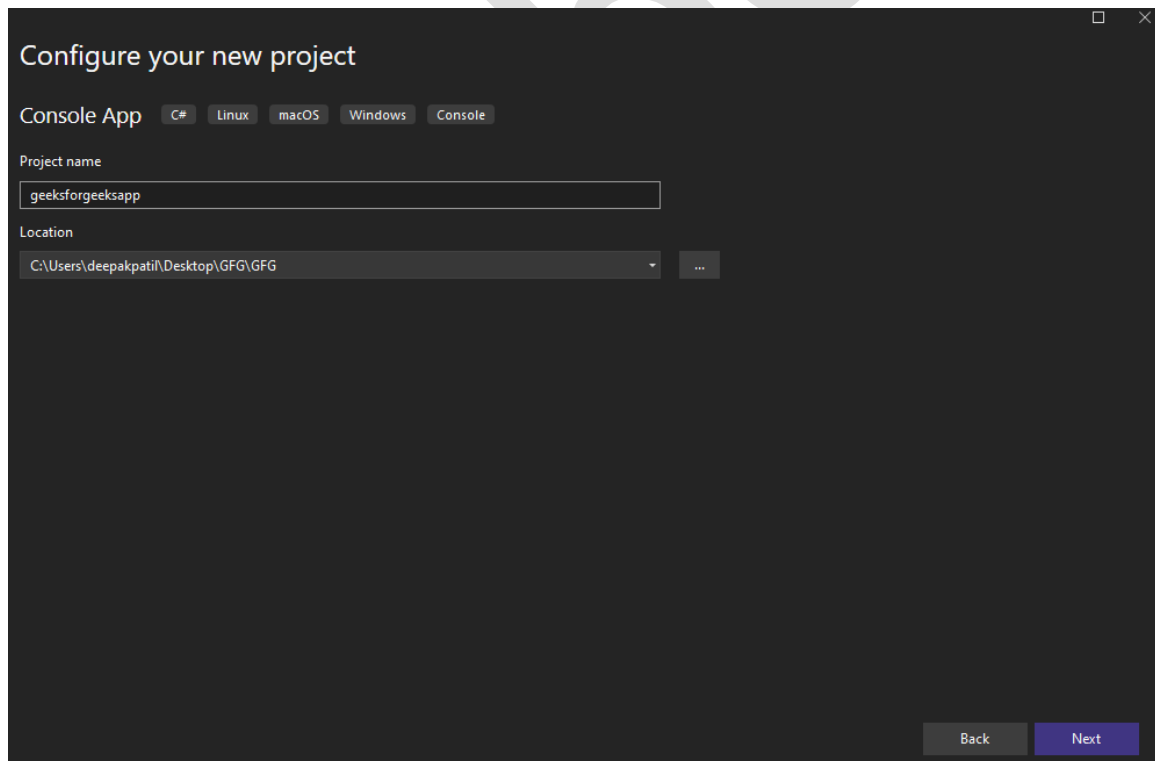
```
Output
Show output from: Build
Build started...
1>----- Build started: Project: geeksforgeeks, Configuration: Debug Any CPU -----
1>geeksforgeeks -> C:\Users\deepakpatil\Desktop\GFG\GFG\geeksforgeeks\bin\Debug\net7.0\geeksforgeeks.dll
===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
===== Elapsed 00:10.412 =====
```

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

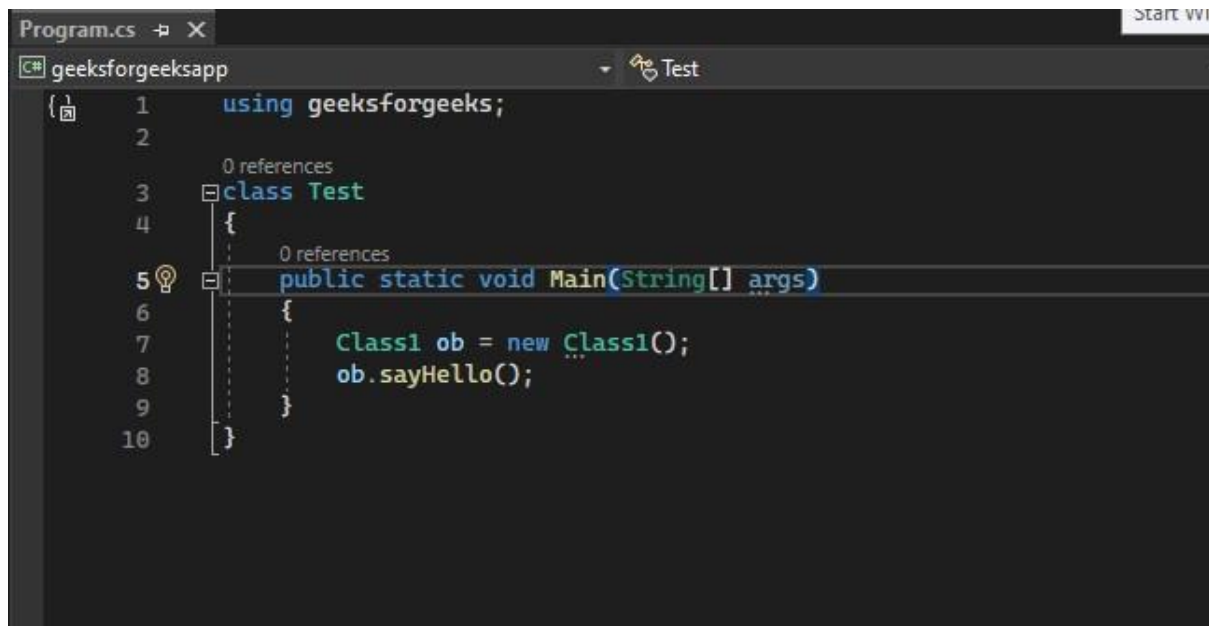
Step 8: Let's create a new project to use this DLL file. For the new project select Console App from the list and click next.



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.

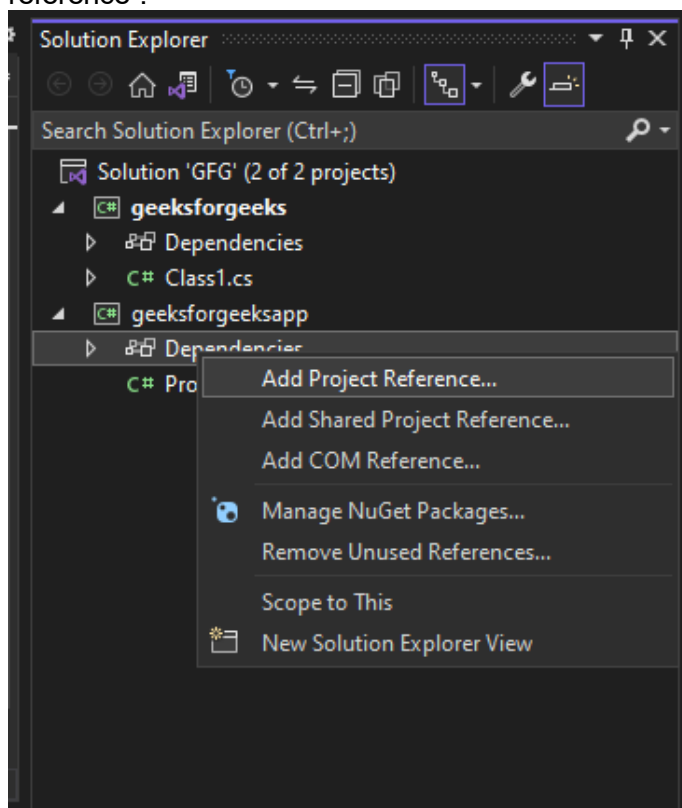


Step 10: Once the project is created it should have a program.cs file opened. Type the Code inside the program.cs file. import the DLL file inside the program by putting “using geeksforgeeks” at the top. Now we can use Class1 inside our program. Call the sayHello() method by creating an object of Class1.

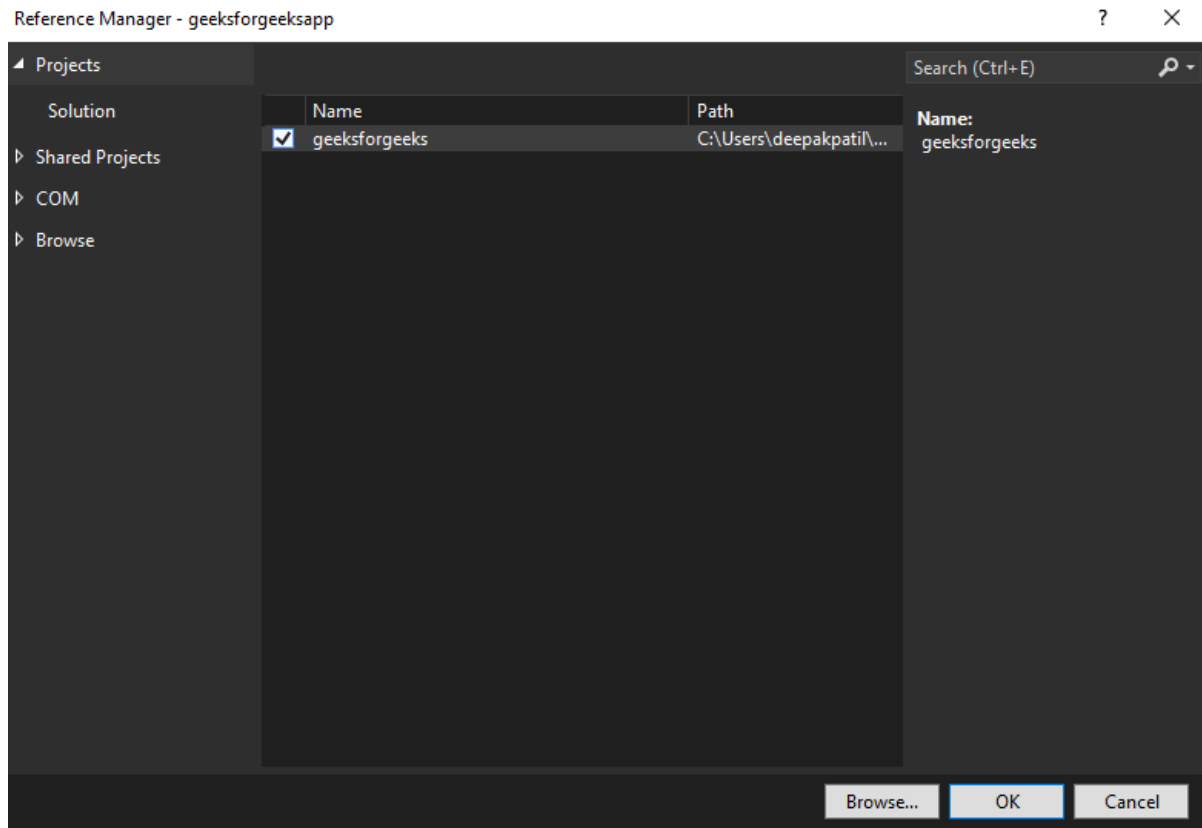


```
Program.cs → X
C# geeksforgeeksapp Test
1 using geeksforgeeks;
2
3 class Test
4 {
5     public static void Main(String[] args)
6     {
7         Class1 ob = new Class1();
8         ob.sayHello();
9     }
10 }
```

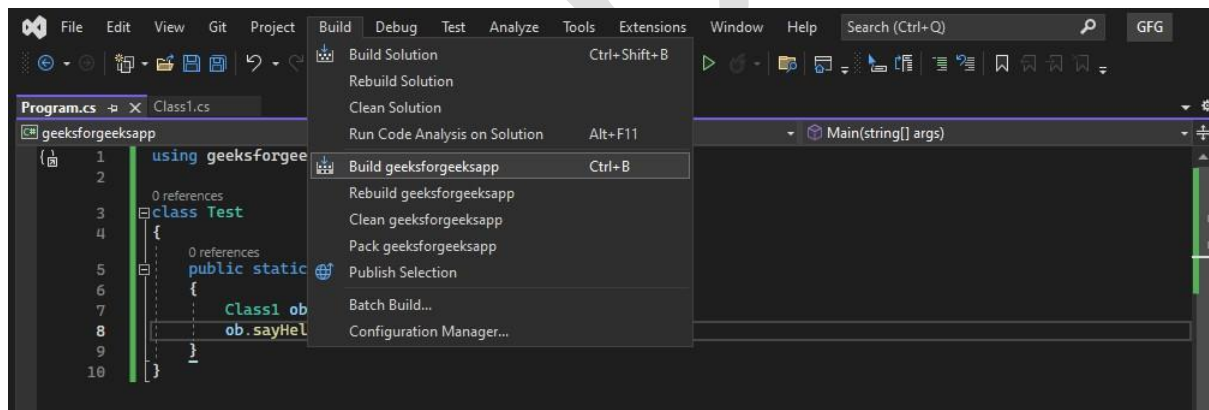
Step 11: Afterwards we have to add a reference of the DLL file to our project. For that in Solution Explorer select dependencies under our project name and right-click to select “Add Project reference”.



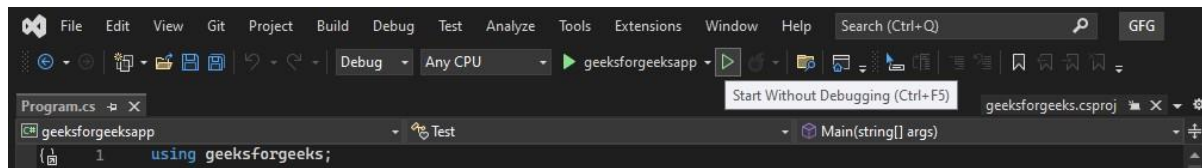
Step 12: Now select our Class Library project name from the list. Click Ok. Now we can build our project.



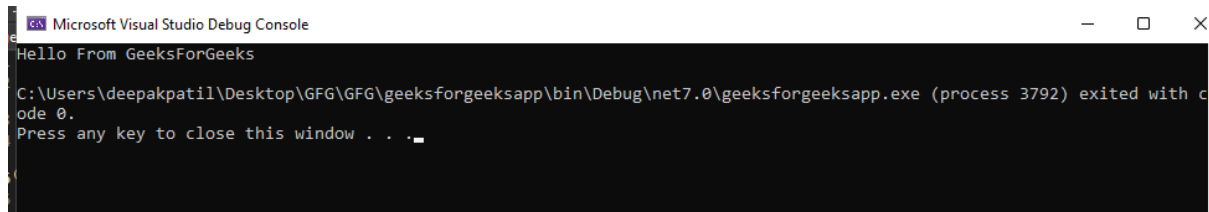
Step 13: From the menu click build and select build “geeksforgeeksapp” or your console project name. You should see the build succeeded at the output.



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).

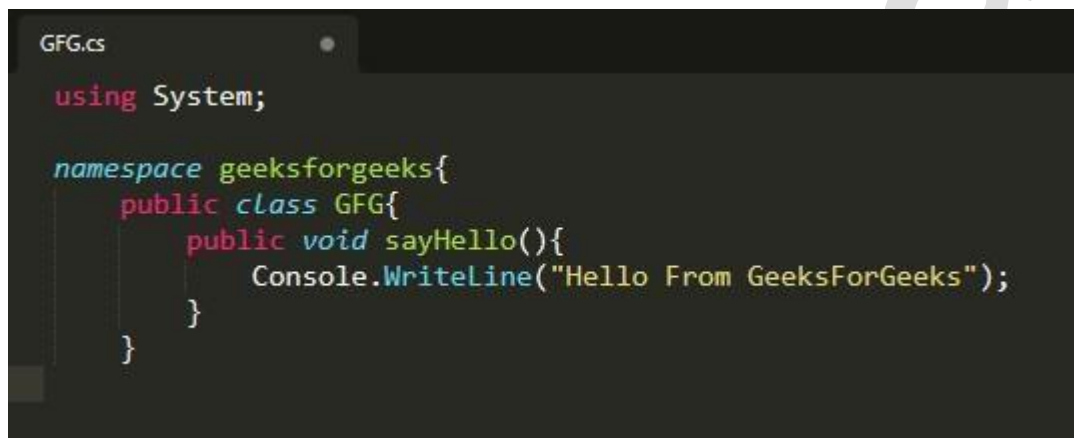


Step 15: You should see a console window opened with say hello message printed.

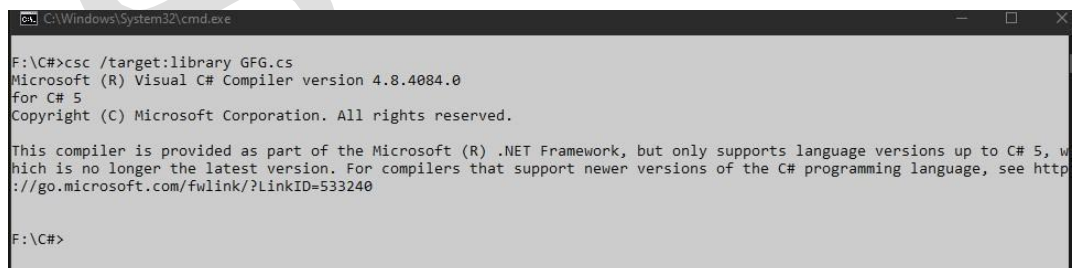


Steps to Create DLL file with C# Compiler

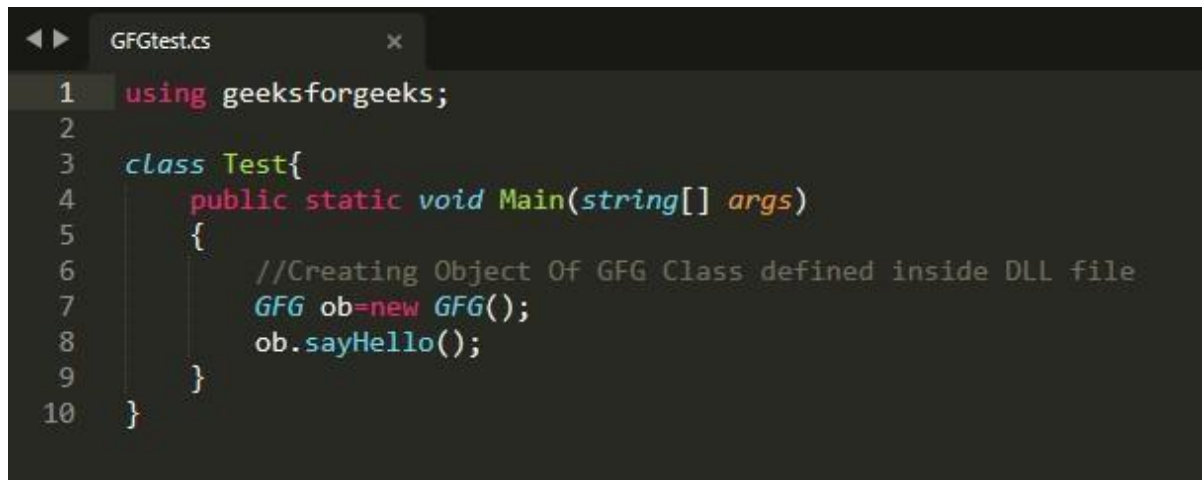
Step 1: Create a new blank file inside your favourite editor and save it as a “.cs” file with the name you want for DLL. Add the Code to the file with namespace and class with any method. I have added the same method as above which prints “Hello From GeeksForGeeks”.



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.



```

1  using geeksforgeeks;
2
3  class Test{
4      public static void Main(string[] args)
5      {
6          //Creating Object Of GFG Class defined inside DLL file
7          GFG ob=new GFG();
8          ob.sayHello();
9      }
10 }

```

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



```

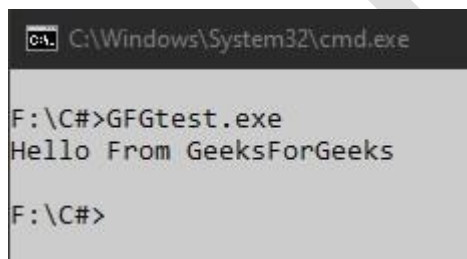
C:\Windows\System32\cmd.exe
F:\C#>csc /r:GFG.dll GFGtest.cs
Microsoft (R) Visual C# Compiler version 4.8.4084.0
for C# 5
Copyright (C) Microsoft Corporation. All rights reserved.

This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports language versions up to C# 5, which is no longer the latest version. For compilers that support newer versions of the C# programming language, see http://go.microsoft.com/fwlink/?LinkID=533240

F:\C#>

```

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.



```

C:\Windows\System32\cmd.exe
F:\C#>GFGtest.exe
Hello From GeeksForGeeks

F:\C#>

```

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.
DateTime()	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.