

**GIT**

**1. Create Local Repository using Tortoise Git (Right Click ->Git Create Repository Here)**

**-> Add a text file to this Local Repository**

**-> Git Sync > Commit > Push**

**-> TortoiseGit -> ShowLog -> Compare With base. Master-> Revert Changes by this commit**

**2. Create Remote Repository in GitHub(Make it private)->Settings->Collaborators & Teams->Add Collaborators.**

**->Make Sure to add a SSH key in account**

**->Generate a key with puttygen**

**->Save public (.pub) & private (.ppk) key pairs in your computer.**

**3. Copy the HTTPS Link from GitHub Link and clone it in the Local repository, Load the generated putty key (SSH key) and add them in the account.**

**Tick Load Putty key (Don’t load putty key in HTTPS method).**

**4. In remote make a branch from main ->exam1-> Now pull it from local repository (Git Sync->pull)**

**->To switch Branches->TortoiseGit-> Switch/Checkout**

**->Switching from main-> exam1 will make a new branch in Local Repository.**

**->Modify Something and Commit->Push**

**->Now Merge from exam1 (This is on the Local Repository) -> Commit->Push.**

**5. Create a new branch in Local Repository, new branch exam2 based on exam1**

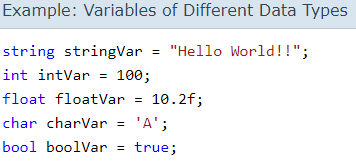
**->Always remember to Git Synch push/Pull and Commit**

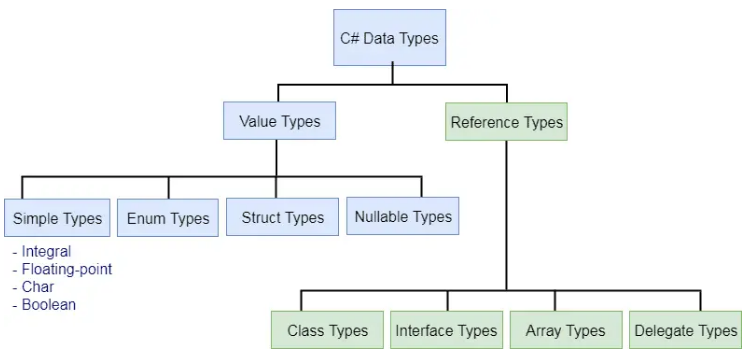
**->Stay in the exam2 branch when synching (in this case Push)**

**-> Switch to main -> Merge branches one by one (exam1 and exam2)**

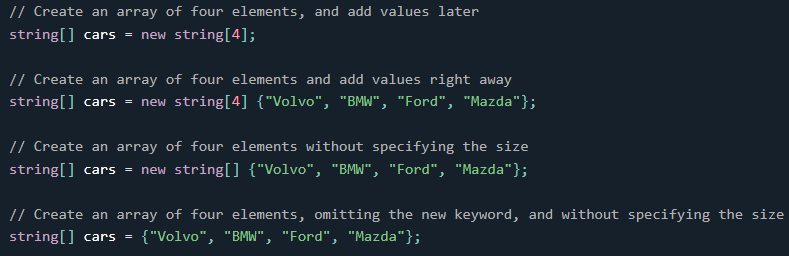
**->Resolve if Conflict Occurs**

****

****

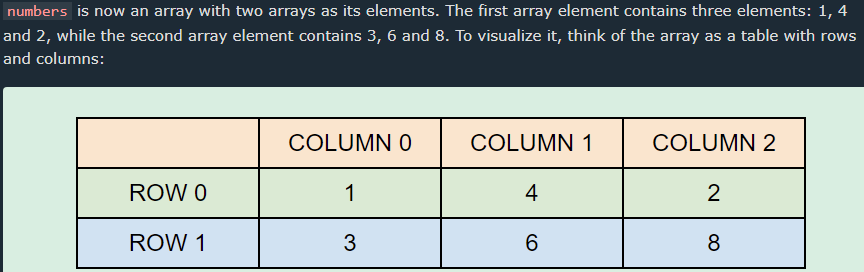
****

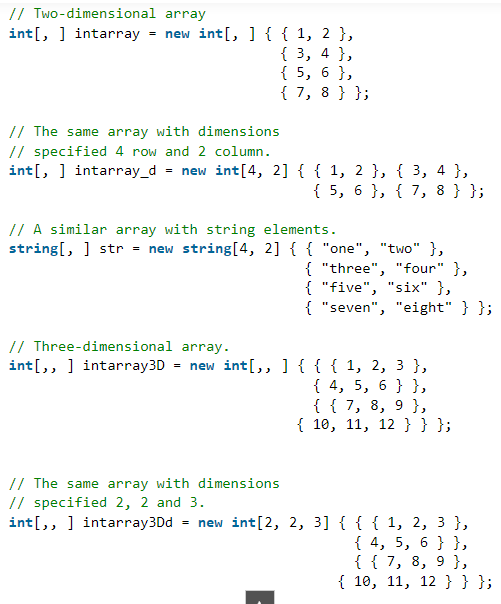
****

****

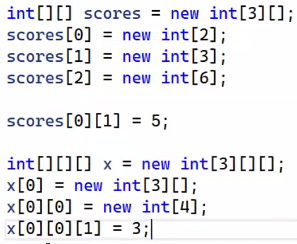
**Multidimensional/Rectangular Array**

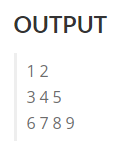
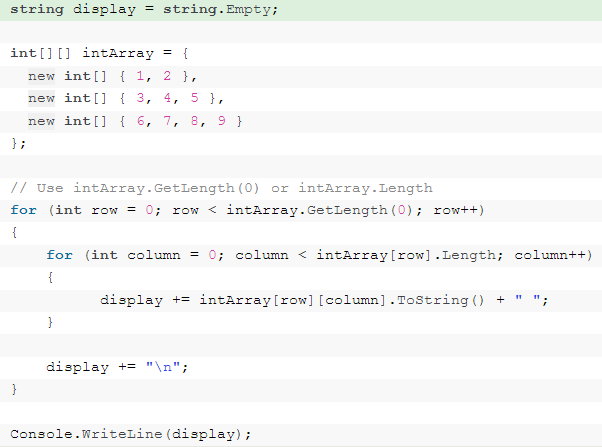
****

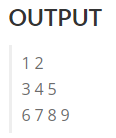
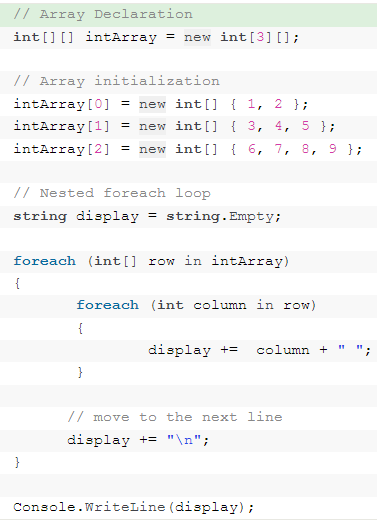
****

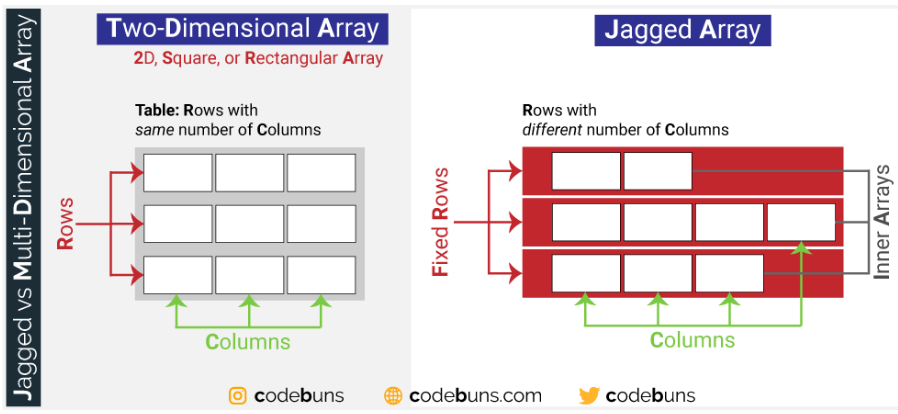
****

**Jagged Array**

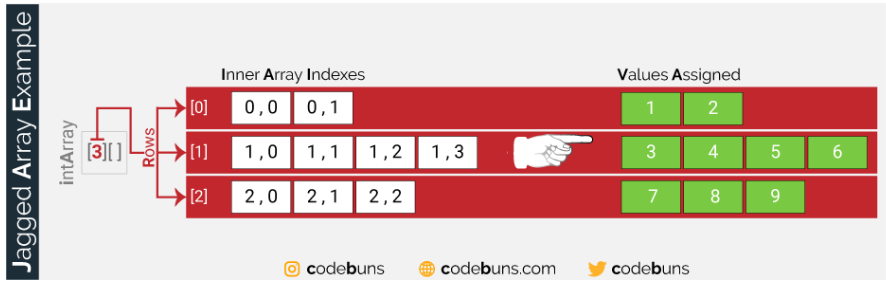
****

****

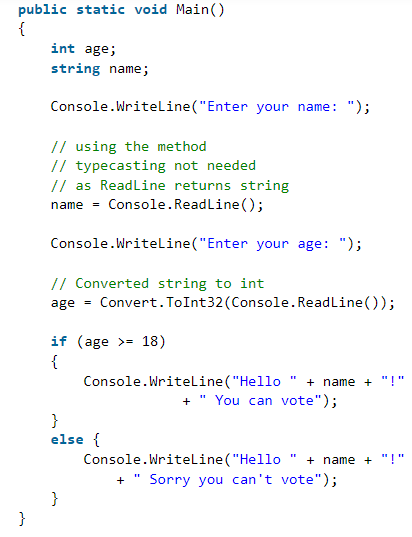
****

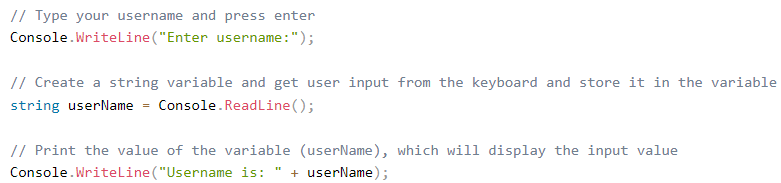
****

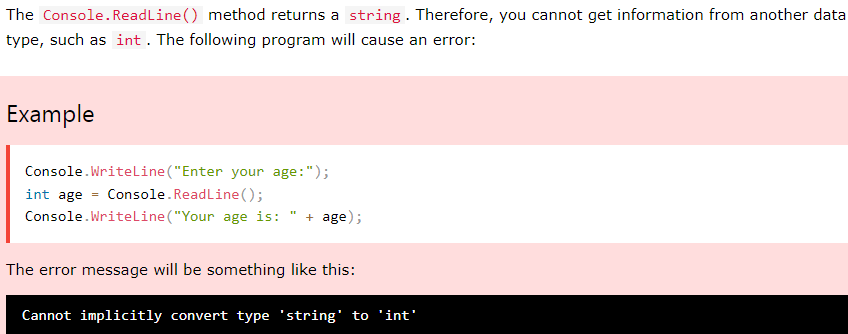
****

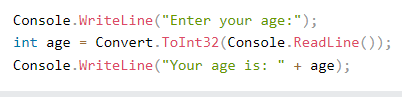
****

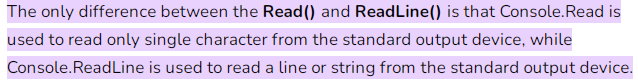
****

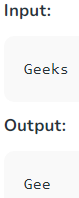
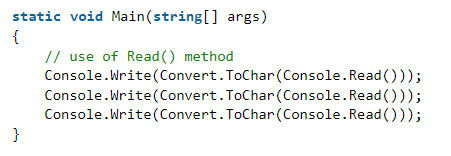
****

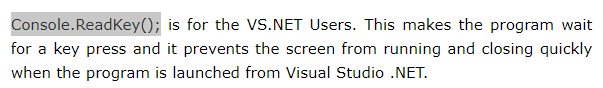
****

****

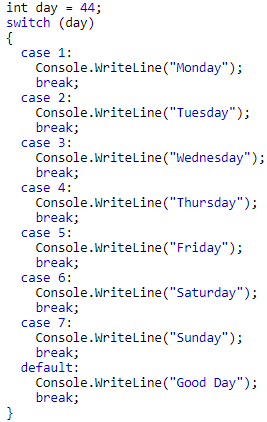
****

****

****

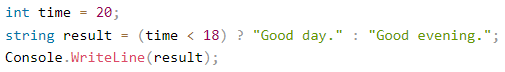
****

****

****

****

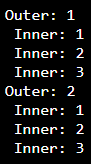
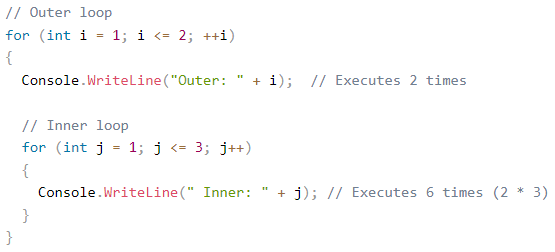
****

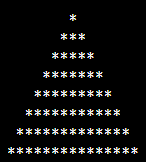
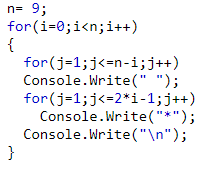
****

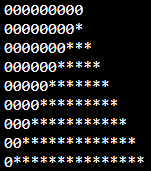
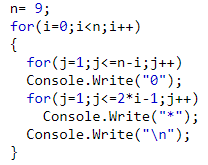
****

****

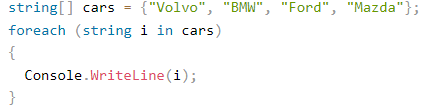
****

****

****

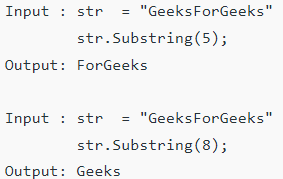
****

****

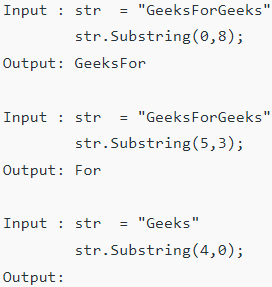
****

**String Formatting**

****

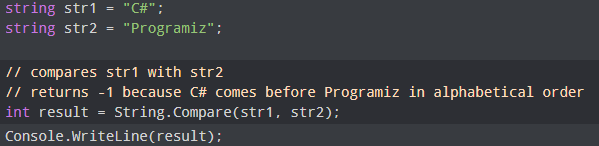
****

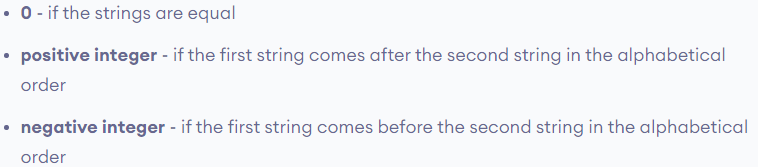
****

****

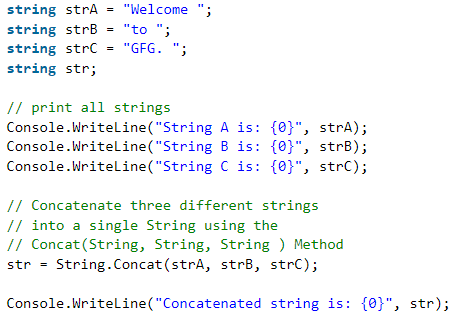
****

****

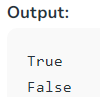
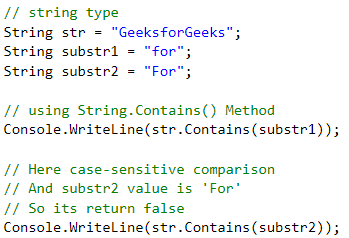
****

****

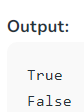
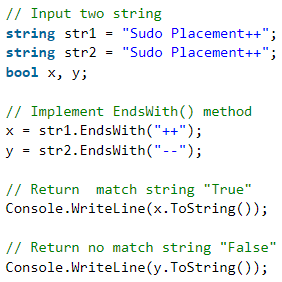
****

****

****

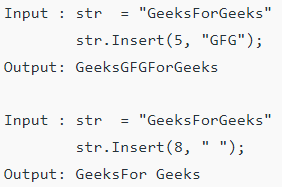
****

****

****

****

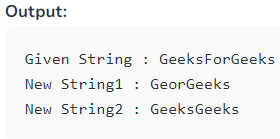
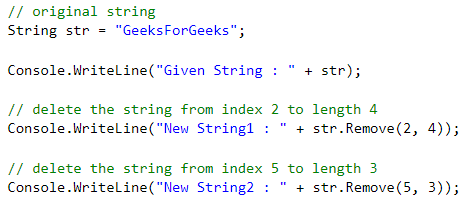
****

****

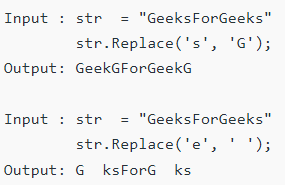
****

****

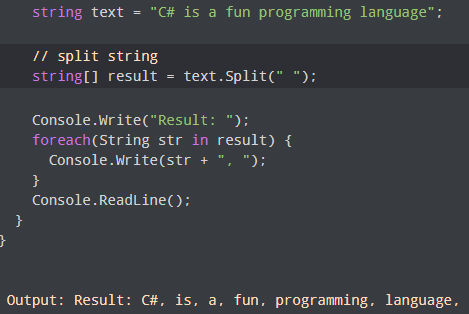
****

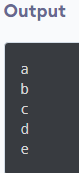
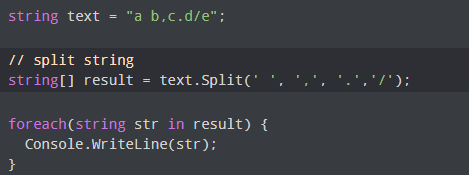
****

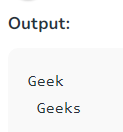
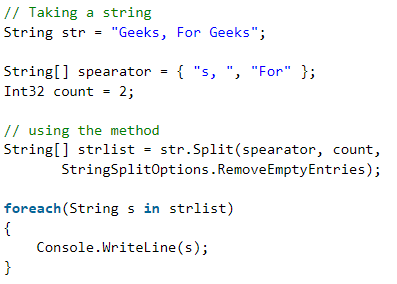
****

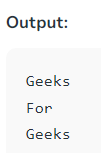
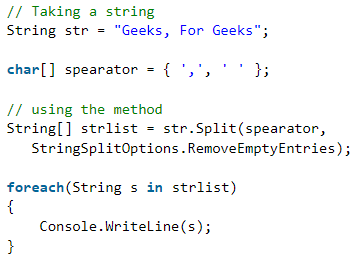
** (“For”, “ ”) is acceptable**

****

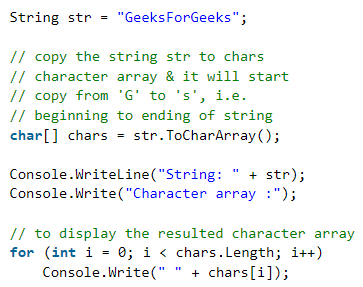
****

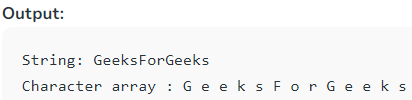
****

****

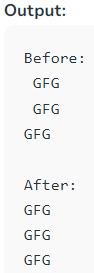
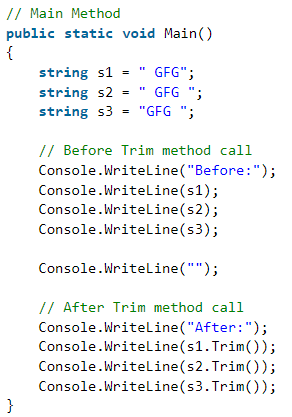
****

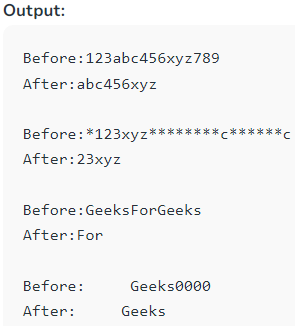
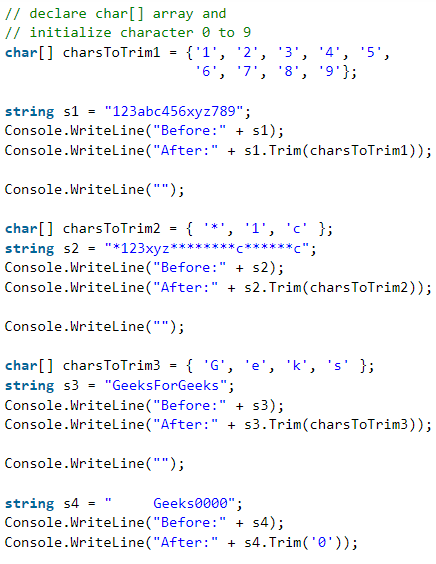
** (string constructor)**

** (Use String Constructor to reverse)**

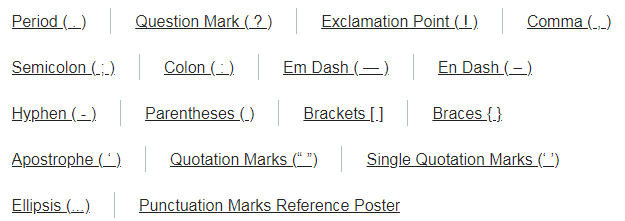
** string s = new string(chars);**

****

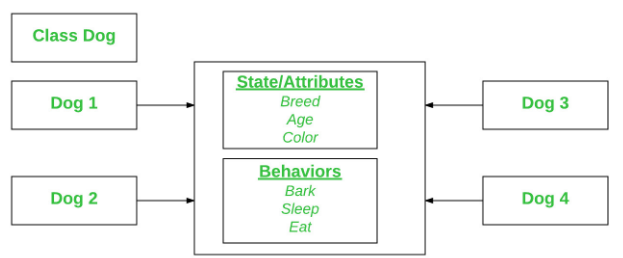
****

****

****

****

****

****

// C# program to illustrate the

// Initialization of an object

using System;

// Class Declaration

public class Dog {

    // Instance Variables

    String name;

    String breed;

    int age;

    String color;

    // Constructor Declaration of Class

    public Dog(String name, String breed, int age, String color)

    {

        this.name = name;

        this.breed = breed;

        this.age = age;

        this.color = color;

    }

    // Property 1

    public String GetName()

    {

        return name;

    }

    // Property 2

    public String GetBreed()

    {

        return breed;

    }

    // Property 3

    public int GetAge()

    {

        return age;

    }

    // Property 4

    public String GetColor()

    {

        return color;

    }

    // Method 1

    public String ToString()

    {

        return ("Hi my name is " + this.GetName()

                + ".\nMy breed, age and color are " + this.GetBreed()

                + ", " + this.GetAge() + ", " + this.GetColor());

    }

// Main Method

public static void Main(String[] args)

    {

        // Creating object

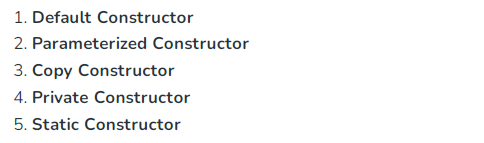
        Dog tuffy = new Dog("tuffy", "papillon", 5, "white");

        Console.WriteLine(tuffy.ToString());

    }

}

****

****

**Parameterized Constructor**

// C# Program to illustrate calling of

// parameterized constructor.

using System;

namespace ParameterizedConstructorExample {

class Geek {

    // data members of the class.

    String name;

    int id;

    // parameterized constructor would

    // initialized data members with

    // the values of passed arguments

    // while object of that class created.

    Public Geek(String name, int id)

    {

        this.name = name;

        this.id = id;

    }

    // Main Method

    public static void Main()

    {

        // This will invoke parameterized

        // constructor.

        Geek geek1 = new Geek("GFG", 1);

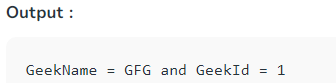
        Console.WriteLine("GeekName = " + geek1.name +

                         " and GeekId = " + geek1.id);

    }

}

}



### Copy Constructor

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

// C# Program to illustrate calling

// a Copy constructor

using System;

namespace copyConstructorExample {

class Geeks {

    private string month;

    private int year;

    // declaring Copy constructor

    public Geeks(Geeks s)

    {

        month = s.month;

        year = s.year;

    }

    // Instance constructor

    public Geeks(string month, int year)

    {

        this.month = month;

        this.year = year;

    }

    // Get details of Geeks

    public string Details

    {

        get

        {

            return "Month: " + month.ToString() +

                     "\nYear: " + year.ToString();

        }

    }

    // Main Method

    public static void Main()

    {

        // Create a new Geeks object.

        Geeks g1 = new Geeks("June", 2018);

        // here is g1 details is copied to g2.

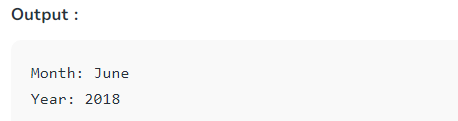
        Geeks g2 = new Geeks(g1);

        Console.WriteLine(g2.Details);

    }

}

}



**Private Constructor**

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

**Points To Remember:**

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

// a Private constructor

namespace privateConstructorExample {

public class Geeks {

    // declare private Constructor

    private Geeks()

    {

    }

    // declare static variable field

    public static int count\_geeks;

    // declare static method

    public static int geeks\_Count()

    {

        return ++count\_geeks;

    }

    // Main Method

    public static void Main()

    {

        // If you uncomment the following

        // statement, it will generate

        // an error because the constructor

        // is inaccessible:

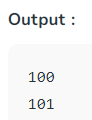
        // Geeks s = new Geeks(); // Error

        Geeks.count\_geeks = 99;

        // Accessing without any

        // instance of the class

        Geeks.geeks\_Count();

        Console.WriteLine(Geeks.count\_geeks);

        // Accessing without any

        // instance of the class

        Geeks.geeks\_Count();

        Console.WriteLine(Geeks.count\_geeks);

    }

}

}

**Static Constructor**

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

**Points To Remember:**

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

// C# Program to illustrate calling

// a Static constructor

using System;

namespace staticConstructorExample {

class geeks {

    // It is invoked before the first

    // instance constructor is run.

    static geeks()

    {

        // The following statement produces

        // the first line of output,

        // and the line occurs only once.

        Console.WriteLine("Static Constructor");

    }

    // Instance constructor.

    public geeks(int i)

    {

        Console.WriteLine("Instance Constructor " + i);

    }

    // Instance method.

    public string geeks\_detail(string name, int id)

    {

        return "Name:" + name + " id:" + id;

    }

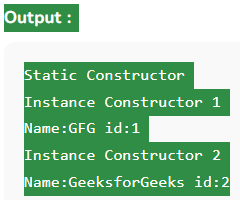
    // Main Method

    public static void Main()

    {

        // Here Both Static and instance

        // constructors are invoked for

        // first instance

        geeks obj = new geeks(1);

        Console.WriteLine(obj.geeks\_detail("GFG", 1));

        // Here only instance constructor

        // will be invoked

        geeks obj1 = new geeks(2);

        Console.WriteLine(obj1.geeks\_detail("GeeksforGeeks", 2));

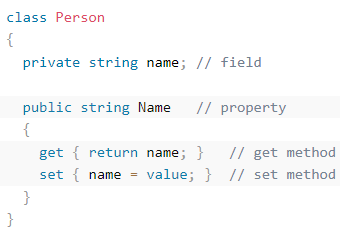
    }

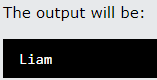
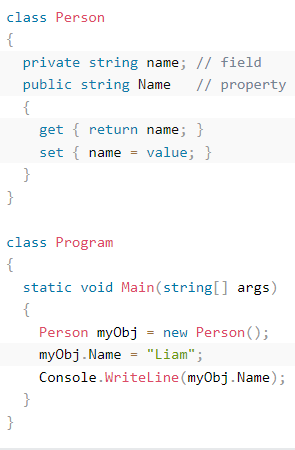
}

}

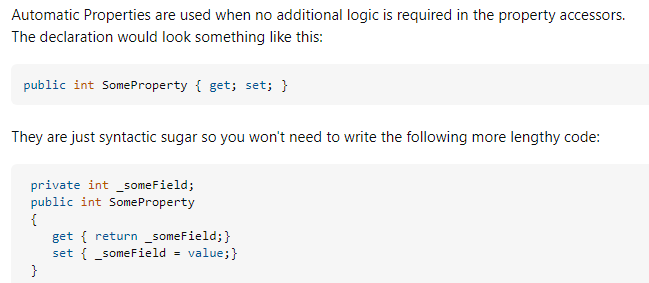
(Encapsulation)

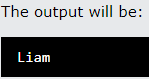
Private variables can only be accessed within the same class (an outside class has no access to it). However, sometimes we need to access them - and it can be done with properties. A property is like a combination of a variable and a method, and it has two methods: a get and a set method:













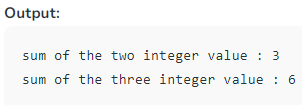
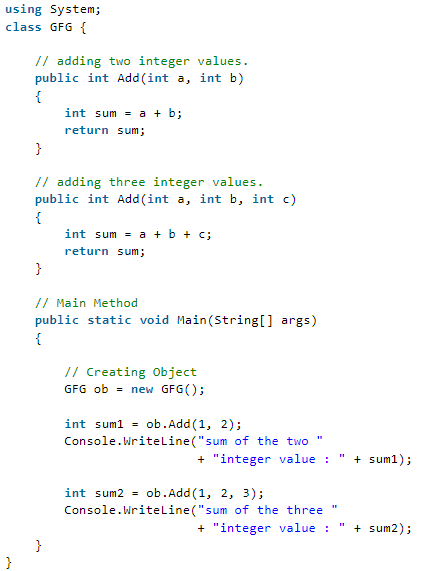
Method Overloading is a common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name. C# can distinguish the methods with **different method signatures**. i.e. the methods can have the same name but with different parameters list (i.e. the number of the parameters, order of the parameters, and data types of the parameters) within the same class.



**If both methods have the same parameter types, but different return type, then it is not possible.**

Overloading (compile-time polymorph, static) and Overriding (run-time polymorph, or dynamic because the type of the calling object is not known until runtime, and therefore the method implementation that runs is determined at runtime) both is a form of polymorphism.

**By changing the Number of Parameters**



**What happens when method signature is same and the return type is different?**  
The compiler will give error as the return value alone is not sufficient for the compiler to figure out which function it has to call. Only if both methods have different parameter types (so, they have the different signature), then Method overloading is possible.

**Access Modifiers (Encapsulation)**

**Public:** When we declare a type or type member **public**, it can be accessed from anywhere.

**Private:** When we declare a type member with the **private** access modifier, it can only be accessed within the same **class** or **struct.**

**Protected:** When we declare a type member as **protected**, it can only be accessed from the same **class** and its derived **classes**.

**Internal:** When we declare a type or type member as **internal**, it can be accessed only within the same **assembly**.

**Protected Internal:**The **protected internal** is a combination of **protected** and internal access modifiers. When we declare a member **protected internal**, it can be accessed from the same assembly and the derived class of the containing class from any other assembly.

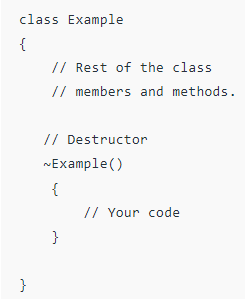
**Private Protected:** The **private protected** access modifier is a combination of **private** and **protected**. It is available from the C# version 7.2 and later.

When we declare a member private protected, it can only be accessed within the same class, and its derived class within the same assembly.

**Destructors**

Destructors in C# are methods inside the class used to destroy instances of that [class](https://www.geeksforgeeks.org/c-class-and-object/)when they are no longer needed. The Destructor is called implicitly by the [.NET Framework’s](https://www.geeksforgeeks.org/introduction-to-net-framework/) Garbage collector and therefore programmer has no control as when to invoke the destructor. An instance variable or an object is eligible for destruction when it is no longer reachable.

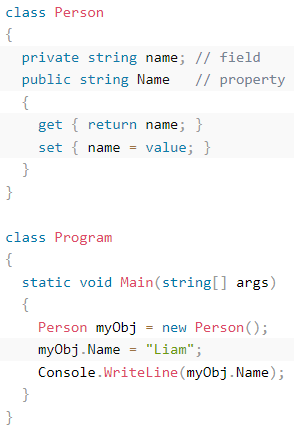
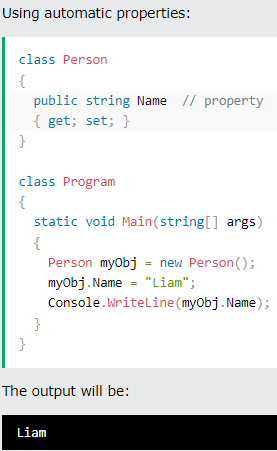
There cannot be more than one destructor in a class. A Destructor has no return type; it cannot be defined in Structures. It is only used with classes. It cannot be overloaded or inherited. It is called when the program exits.



**Properties (SET GET) (Encapsulation)**

Before we start to explain properties, you should have a basic understanding of "**Encapsulation**". The meaning of **Encapsulation** is to make sure that "sensitive" data is hidden from users. To achieve this, you must:

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

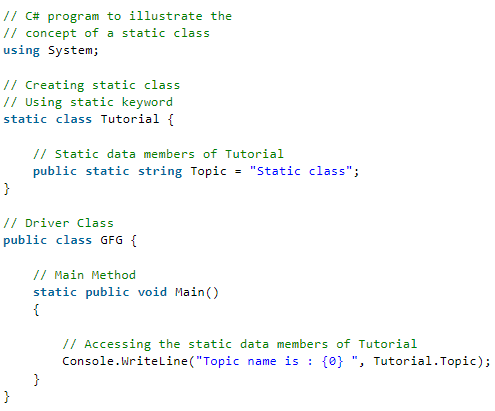
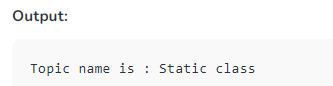
**Static**

**static**is a modifier in C# which is applicable for:

Classes, Variables, Methods, Constructor, properties, operators

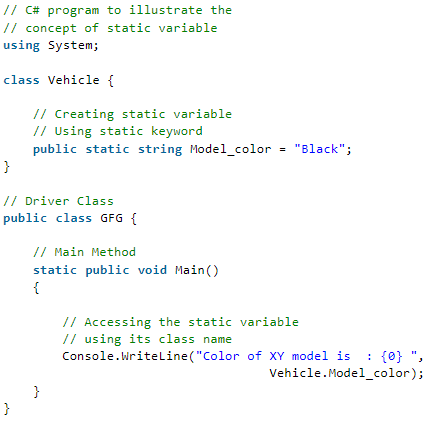
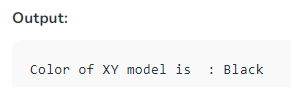
**Static Class**

A static class is declared with the help of ***static***keyword. A static class can only contain **static** **data members, static methods**, and a **static constructor**. It is not allowed to create objects of the **static class**. Static classes are [***sealed***](https://www.geeksforgeeks.org/c-sealed-class/), means one cannot inherit a static class from another class.

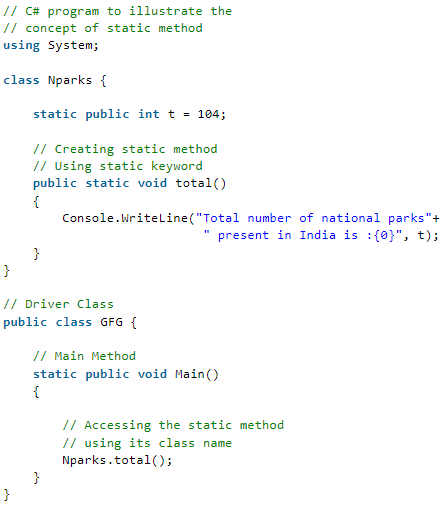
**Static Variable**

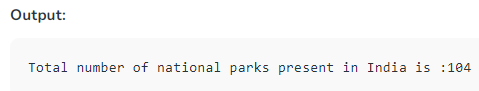
A static variable is declared with the help of static keyword. When a variable is declared as static, then a single copy of the variable is created and shared among all objects at the class level. Static variables are accessed with the name of the class; they do not require any object for access.

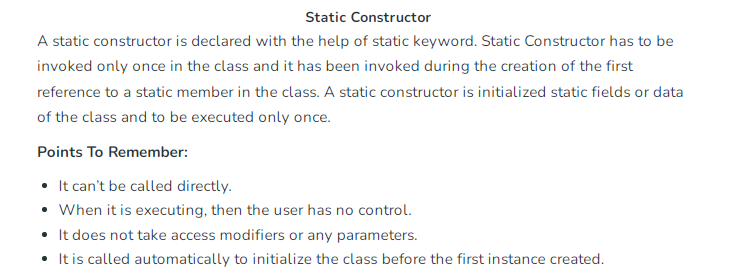
 

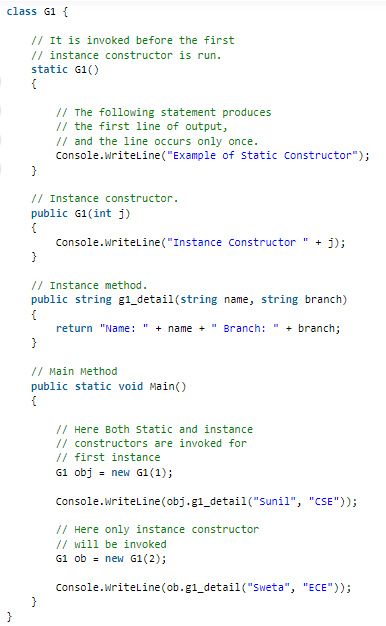
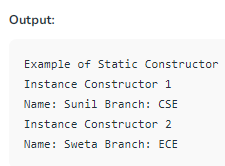
**Static Method**

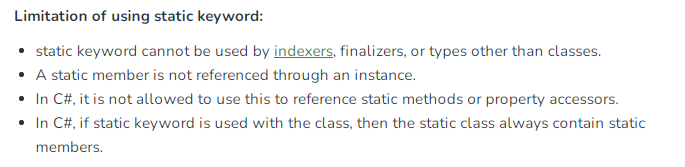
A static method is declared with the help of static keyword. Static methods are accessed with the name of the class. A static method can access static and non-static fields, static fields are directly accessed by the static method without class name whereas non-static fields require objects.







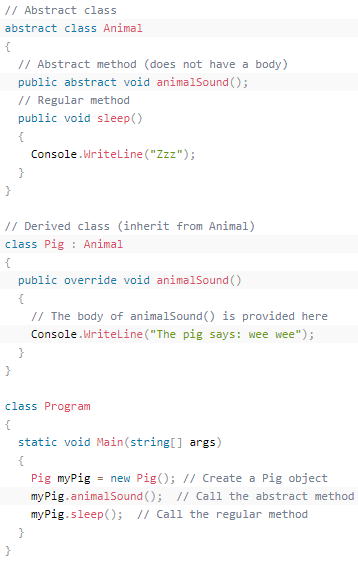
 



**Abstraction**

Data **abstraction** is the process of hiding certain details and showing only essential information to the user. Abstraction can be achieved with either [**abstract**](c-sharp-c-logo.png)[**classes**](c-sharp-c-logo.png) or [**interfaces**](https://www.w3schools.com/cs/cs_interface.php)

* **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
* **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the derived class (inherited to).



[**Abstraction in C#**](https://www.geeksforgeeks.org/c-abstraction/) is the process to hide the internal details and show only the functionality. The **abstract modifier** indicates the incomplete implementation. The keyword **abstract** is used before the class or method to declare the class or method as abstract. Also, the **abstract** modifier can be used with [**indexers**](https://www.geeksforgeeks.org/c-indexers/), events, and [**properties**](https://www.geeksforgeeks.org/c-properties/).

An abstract class is an incomplete class or special class that can't be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules what the derived classes must implement when they inherit an abstract class.

We can use an abstract class as a base class and all derived classes must implement abstract definitions. An abstract method must be implemented in all non-abstract classes using the override keyword. After overriding the abstract method is in the non-Abstract class. We can derive this class in another class and again we can override the same abstract method with it

## C# Abstract Class Features

1. An abstract class can inherit from a class and one or more interfaces.
2. An abstract class can implement code with non-Abstract methods.
3. An Abstract class can have modifiers for methods, properties etc.
4. An Abstract class can have constants and fields.
5. An abstract class can implement a property.
6. An abstract class can have constructors or destructors.
7. An abstract class cannot be inherited from by structures.
8. An abstract class cannot support multiple inheritances.

**Abstract Method:** A method that is declared abstract has no “body” and is declared inside the abstract class only. An abstract method must be implemented in all non-abstract classes using the override keyword. After overriding, the abstract method is in the non-Abstract class. We can derive this class in another class, and again we can override the same abstract method with it.

**Syntax:**

public abstract void geek();

// the method 'geek()' is abstract

**Abstract Class:** This is the way to achieve the abstraction in C#. An Abstract class is never intended to be instantiated directly. An abstract class can also be created without any abstract methods, we can mark a class abstract even if doesn’t have any abstract method. The Abstract classes are typically used to define a base class in the *class hierarchy*. Or in other words, an abstract class is an incomplete class or a special class that can’t be instantiated. The purpose of an abstract class is to provide a blueprint for derived classes and set some rules that the derived classes must implement when they inherit an abstract class. We can use an abstract class as a base class and all derived classes must implement abstract definitions.

**Syntax:**

abstract class gfg{}

// class 'gfg' is abstract

Abstract function and interface is almost same

**Interface (**Abstraction**)**

Another way to achieve [abstraction](https://www.w3schools.com/cs/cs_abstract.php) in C#, is with interfaces. It removes Coupling, Helps achieve Decouple in code.

Example: Suppose we are implementing internet networks. We will just add the interfaces to the new classes. And implement the function written in the interface.

**InternetService ->Interface {void connectToInternet(); }**

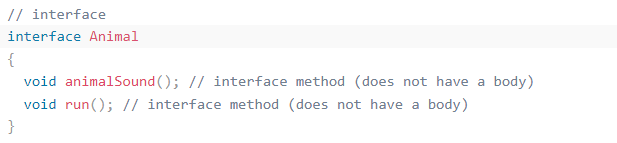
**Internet: IInternetService**

**Internet3G: IInternetService**

**Internet5G: IInternetService**

An interface is a completely "**abstract class**", which can only contain abstract methods and properties (with empty bodies)

* By convention, interface starts with I so that we can identify it just by seeing its name.
* We cannot use access modifiers inside an interface.
* All members of an interface are public by default.
* An interface doesn't allow fields.









**Advantages of C# interface**

Now that we know what interfaces are, let's learn about why interfaces are used in C#.

* Similar to abstract classes, interfaces help us to achieve **abstraction in C#**.  
  Here, the method calculateArea() inside the interface, does not have a body. Thus, it hides the implementation details of the method.
* Interfaces provide **specifications** that a class (which implements it) must follow.  
  In our previous example, we have used calculateArea() as a specification inside the interface IPolygon. This is like setting a rule that we should calculate the area of every polygon.  
  Now any class that implements the IPolygon interface must provide an implementation for the calculateArea() method.
* Interfaces are used to achieve multiple inheritances in C#.
* Interfaces provide **loose coupling** (having no or least effect on other parts of code when we change one part of a code).  
  In our previous example, if we change the implementation of calculateArea() in the Square class it does not affect the Rectangle class.

**Override (Polymorphism) Virtual**

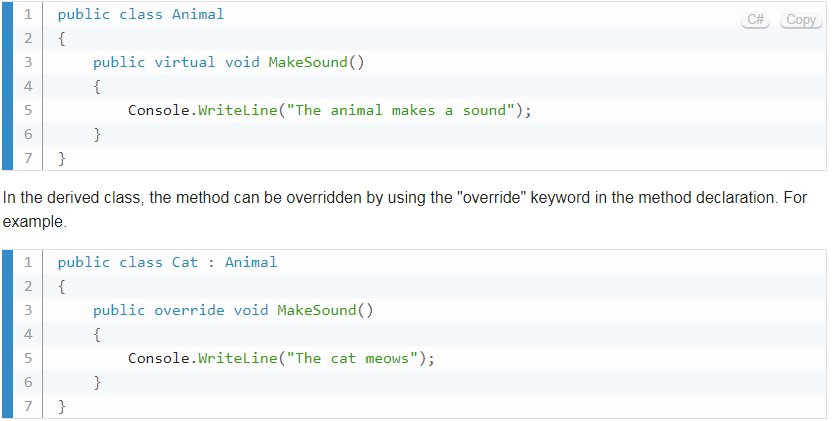
Method Overriding is a technique that allows the invoking of functions from another class (base class) in the derived class. Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.

Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes. When a method in a subclass has the same name, same parameters or signature and same return type (or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.

The method that is overridden by an override declaration is called the overridden base method. An override method is a new implementation of a member that is inherited from a base class. The overridden base method must be virtual, abstract, or override.

**In C# we can use 3 types of keywords for Method Overriding:**

* **Virtual keyword:** This modifier or keyword use within base class method. It is used to modify a method in *base class* for *overridden* that particular method in the derived class.
* **Override:** This modifier or keyword use with derived class method. It is used to modify a *virtual* or *abstract* method into *derived class* which presents in base class.



When an instance of the derived class is created and the overridden method is called, the implementation in the derived class will be executed instead of the implementation in the base class.

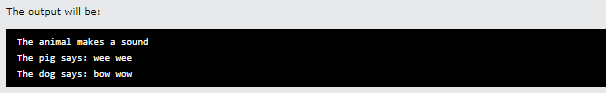
Using virtual methods can be useful in situations where you want to provide a base implementation in a base class, but allow derived classes to modify or extend the behavior of that method.

1. By default, methods are non-virtual. We can't override a non-virtual method.
2. We can't use the virtual modifier with the static, abstract, private, or override modifiers.

* Method overriding is possible only in derived classes. Because a method is overridden in the derived class from the base class.
* A non-virtual or a static method can’t be overridden.
* Both the override method and the virtual method must have the same access level modifier.

# We do this to Rewrite the method

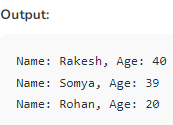
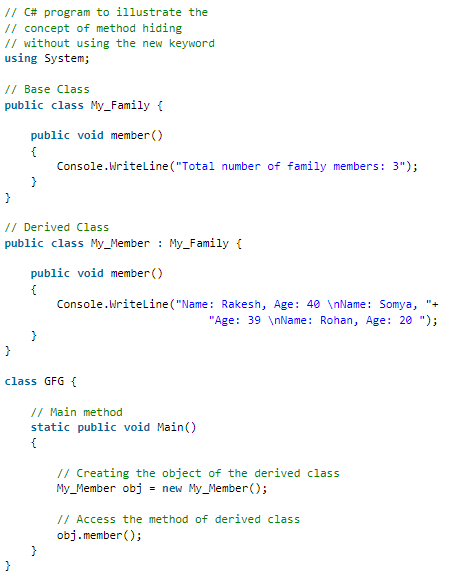
# We have to use new if no virtual

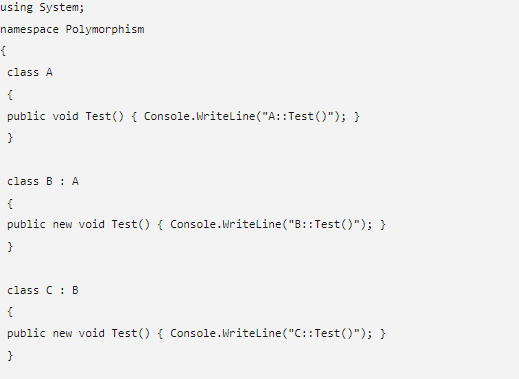


## Method Hiding (new keyword)

As you have seen in the above example the compiler generates the warnings since C# also supports method hiding. For hiding the base class method from the derived class simply declare the derived class method with a new keyword.

It is also known as **Method Shadowing**. In method hiding, you can hide the implementation of the methods of a base class from the derived class using the *new*keyword. Or in other words, in method hiding, you can redefine the method of the base class in the derived class by using the *new*keyword.





**Sealed**

Sealed classes are used to restrict the users from inheriting the class. A class can be sealed by using the ***sealed*** keyword. The keyword tells the compiler that the class is sealed, and therefore, cannot be extended. No class can be derived from a sealed class.

*A* ***method*** *can also be sealed*, and in that case, the method cannot be **overridden**. However, a method can be sealed in the classes in which they have been inherited. If you want to declare a method as sealed, then it has to be declared as **virtual** in its base class.

Sealed class can inherit any class but other class can’t inherit sealed class.

**NameSpaces**

Namespaces are used in C# to organize and provide a level of separation of codes. They can be considered as a container which consists of other namespaces, classes, etc.

A namespace can have following types as its members:

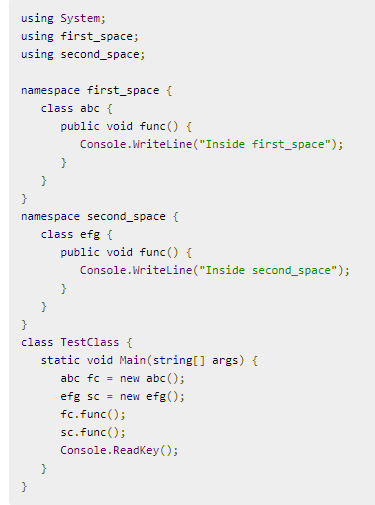
1. Namespaces (Nested Namespace)
2. Classes
3. Interfaces
4. Structures
5. Delegates

Let's understand the concept of namespace with a real life scenario. We have a large number of files and folders in our computer. Imagine how difficult it would be to manage them if they are placed in a single directory. This is why we put related files and folders in a separate directory. This helps us to manage our data properly.

The concept of namespace is similar in C#. It helps us to **organize** different members by putting related members in the same namespace.

Namespace also solves the problem of **naming conflict**. Two or more classes when put into different namespaces can have same name.

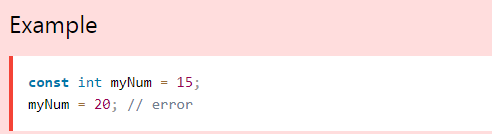




**Constant & Read-only**

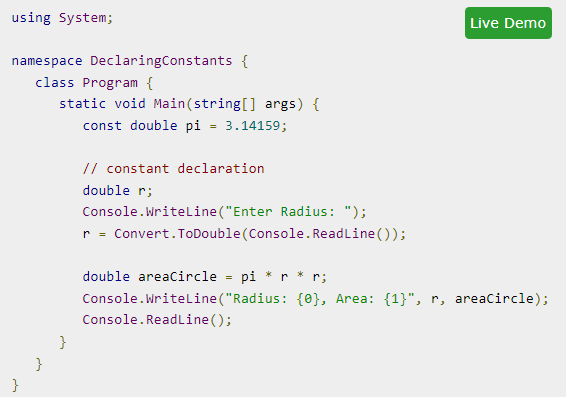
If you don't want others (or yourself) to overwrite existing values, you can add the const keyword in front of the variable type.

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

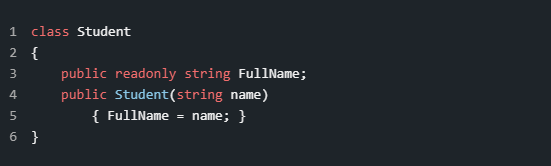


The const keyword is useful when you want a variable to always store the same value, so that others (or yourself) won't mess up your code. An example that is often referred to as a constant is PI (3.14159...).

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



**ReadOnly:**  It can be used on fields, but not on local variables. These fields can either be initialized when declared or at the constructor of our object. This keyword ensures that a variable instance or property of an object cannot be modified after initialization, and such attempts will result in an exception. While const is initialized at compile time, readonly keyword allow the variable to be initialized either at compile time or runtime.



This definition holds the FullName property of the class which is initialized at runtime, but the compilation still succeeds due to the nature of the readonly keyword.

**Difference:**

For the readonly keyword, the latest value is known by the runtime. For the const keyword, the value must be known by compile time. The common of these two keywords is that they both cover immutable data types, meaning the value cannot change throughout the life of the application. For the const keyword, the variable marked as such is placed by the compiler in the assembly metadata that defines the constant and metadata and embeds the value into the IL code after initialization. This means that there is no memory allocation for constants during runtime. Under the hood, the readonly value is not a constant; it's stored in the loader heap, which is a memory type that cannot be allocated until the type is loaded. The readonly fields are stored in dynamic memory; the values are accessible only at runtime. When an expression references a readonly field, the value is not obtained until runtime. Change in the value is reflected immediately, the assembly does not need to be recompiled.

**Conclusion**

The idea behind both keywords is the same. We would like to mark a variable to hold its value during the execution of the application and not allow modification. The first difference is the timeslot which initializes the const or readonly variables. The first, const, is initialized during compile-time and the latter, readonly, initialized is by the latest run-time. The second difference is that readonly can only be initialized at the class-level. Another important difference is that const variables can be referenced through "ClassName.VariableName", while readonly can be referenced through "InstanceName.VariableName". I have introduced the differences and similarities to the best of my abilities and I hope you found what you were looking for.

**Partial Class**

A partial class is a special feature of C#. It provides a special ability to implement the functionality of a single class into multiple files and all these files are combined into a single class file when the application is compiled. A partial class is created by using a ***partial***keyword. This keyword is also useful to split the functionality of methods, interfaces, or structure into multiple files.

**Syntax:**

public partial Class\_name

{

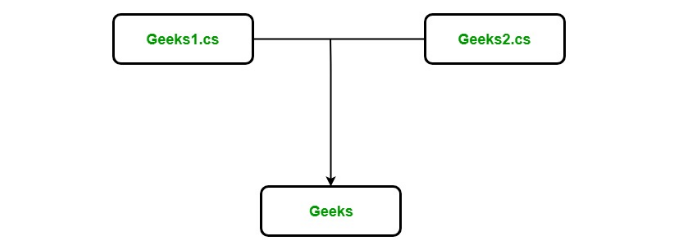
// code

}

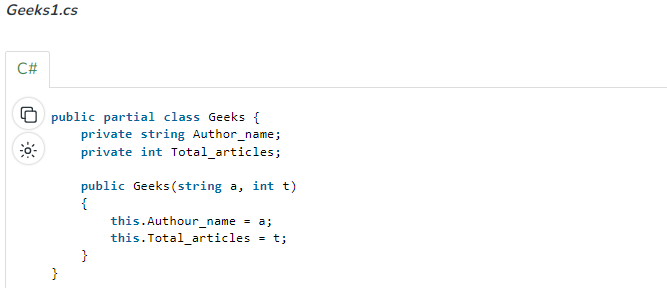
**Important points:**

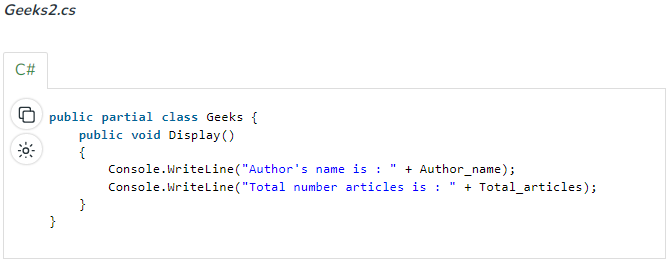
* When you want to chop the functionality of the class, method, interface, or structure into multiple files, then you should use *partial*keyword and all the files are mandatory to be available at compile time for creating the final file.
* The *partial*modifier can only present instantly before the keywords like struct, class, and interface.
* Every part of the partial class definition should be in the same assembly and [namespace](https://www.geeksforgeeks.org/c-namespaces/), but you can use a different source file name.
* Every part of the partial class definition should have the same accessibility as private, protected, etc.
* If any part of the partial class is declared as an abstract, sealed, or base, then the whole class is declared of the same type.
* The user is also allowed to use nested partial types.
* Dissimilar parts may have dissimilar base types, but the final type must inherit all the base types.

**Example:** Here, we are taking a class named *Geeks*and split the definition of Geeks class into two different files named Geeks1.cs, and Geeks2.cs as shown below:



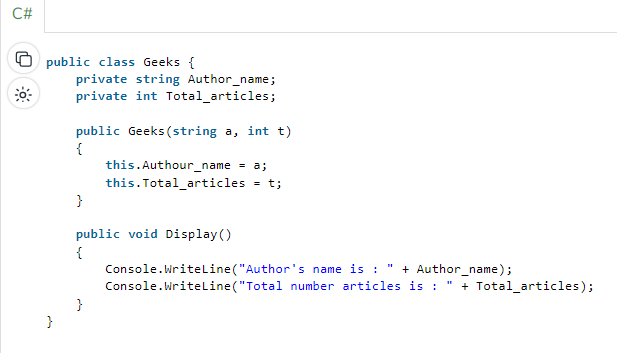
In *Geeks1.cs*, and *Geeks2.cs*, a partial class is created using the partial keyword and each file contains different functionality of Geeks class as shown below.





When we execute the above code, the compiler combines Geeks1.cs and Geeks2.cs into a single file, i.e. Geeks as shown below.  
***Geeks*** This class may contain the Main Method. For simplicity, here Main() method is not included.

**Without Partial Calss, it Will Look Like this**



**Advantages:**

* With the help of partial classes, multiple developers can work simultaneously in the same class in different files.
* With the help of a partial class concept, you can split the UI of the design code and the business logic code to read and understand the code.
* When you were working with automatically generated code, the code can be added to the class without having to recreate the source file like in Visual studio.
* You can also maintain your application in an efficient manner by compressing large classes into small ones.

**Structures**

Structure->Value Type, Always have default constructor, Can’t Inheritance

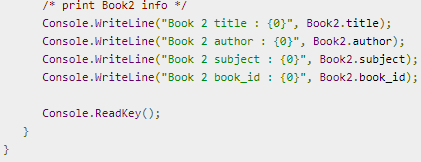
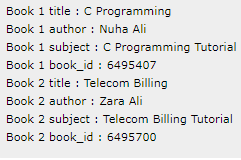
Class->Reference Type, May not have default constructor

In C#, a structure is a value type data type. It helps you to make a single variable hold related data of various data types. The **struct** keyword is used for creating a structure.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −

* Title
* Author
* Subject
* Book ID

****

****

**Features of C# Structures**

You have already used a simple structure named Books. Structures in C# are quite different from that in traditional C or C++. The C# structures have the following features −

* Structures can have methods, fields, indexers, properties, operator methods, and events.
* Structures can have defined constructors, but not destructors. However, you cannot define a default constructor for a structure. The default constructor is automatically defined and cannot be changed.
* Unlike classes, structures cannot inherit other structures or classes.
* Structures cannot be used as a base for other structures or classes.
* A structure can implement one or more interfaces.
* Structure members cannot be specified as abstract, virtual, or protected.
* When you create a struct object using the New operator, it gets created and the appropriate constructor is called. Unlike classes, structs can be instantiated without using the New operator.
* If the New operator is not used, the fields remain unassigned and the object cannot be used until all the fields are initialized.

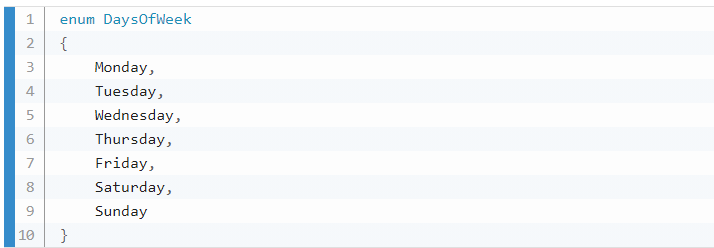
**Class versus Structure**

Classes and Structures have the following basic differences −

* classes are reference types and structs are value types
* structures do not support inheritance

**Enum**

Enums are powerful data types in C# that allow you to define a set of named constants. They are often used to represent a fixed number of possible values that a variable can take on.

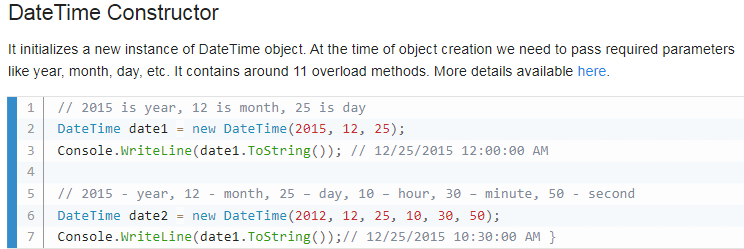


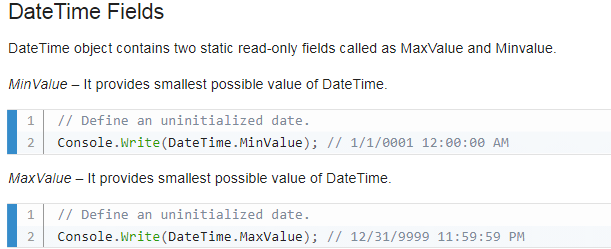


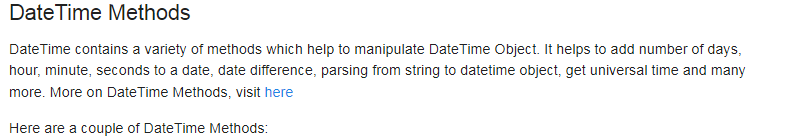


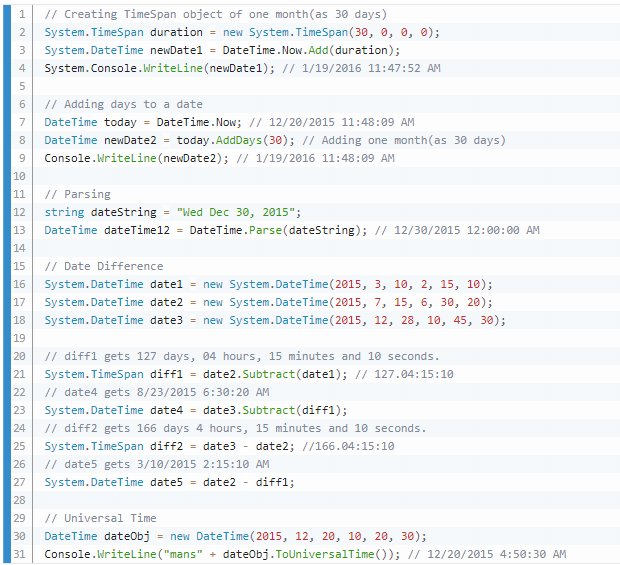
**DateTime**

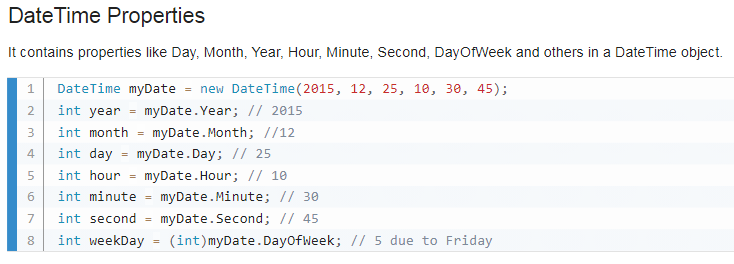
C# DateTime is a structure of value Type like int, double etc. It is available in System namespace and present in mscorlib.dll assembly. It implements interfaces like IComparable, IFormattable, IConvertible, ISerializable, IComparable, IEquatable. DateTime helps developer to find out more information about Date and Time like Get month, day, year, week day. It also helps to find date difference, add number of days to a date, etc.

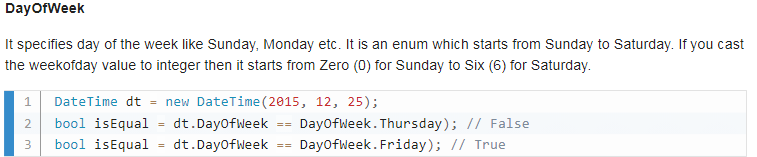


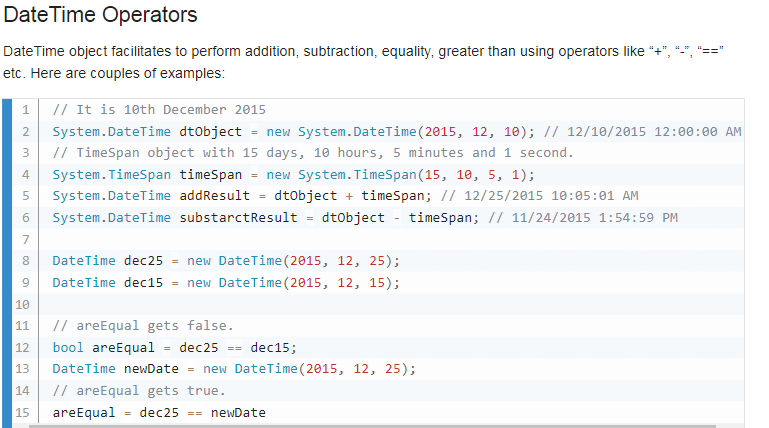




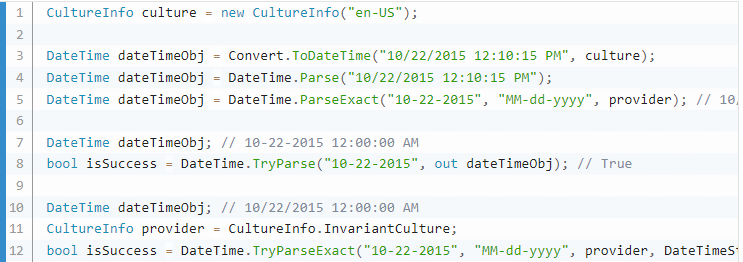








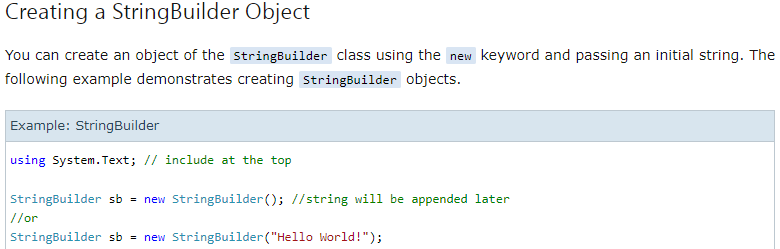


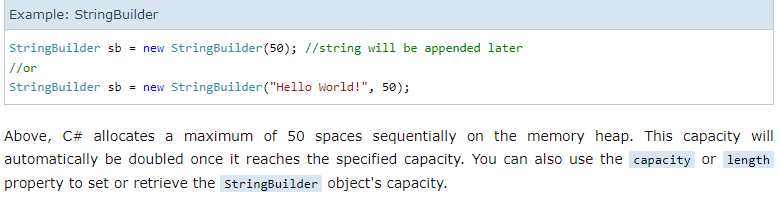


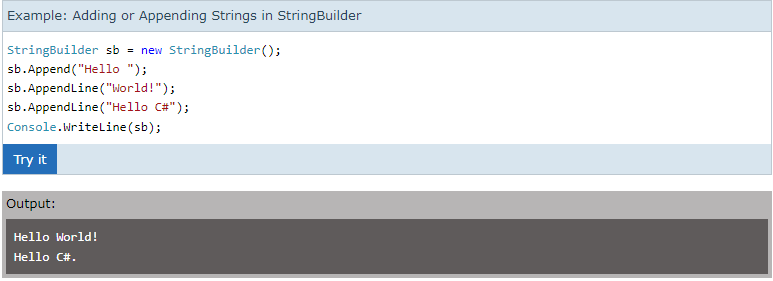
**String Builder**

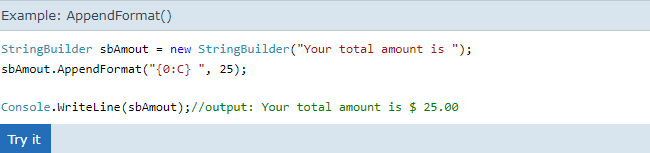
In C#, the string type is immutable. It means a string cannot be changed once created. For example, a new string, "Hello World!" will occupy a memory space on the heap. Now, by changing the initial string "Hello World!" to "Hello World! from Tutorials Teacher" will create a new string object on the memory heap instead of modifying an original string at the same memory address. This behavior would hinder the performance if the original string changed multiple times by replacing, appending, removing, or inserting new strings in the original string.

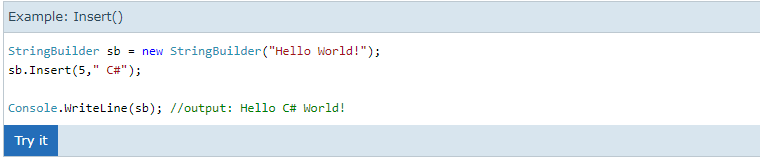
To solve this problem, C# introduced the StringBuilder in the [System.Text](https://docs.microsoft.com/en-us/dotnet/api/system.text" \t "_blank) namespace. The StringBuilder doesn't create a new object in the memory but dynamically expands memory to accommodate the modified string.

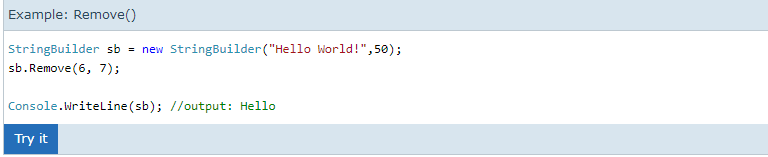


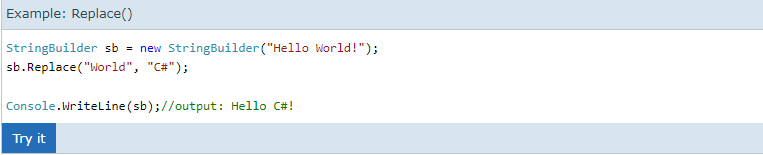


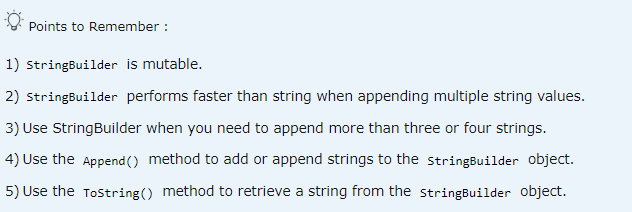








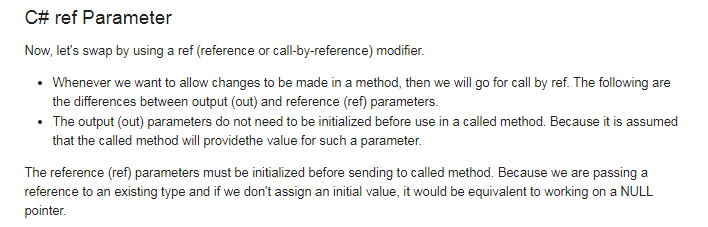


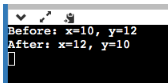
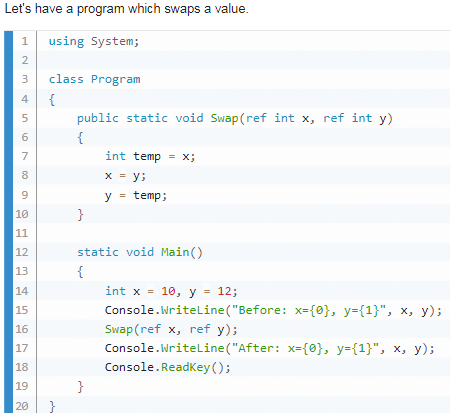


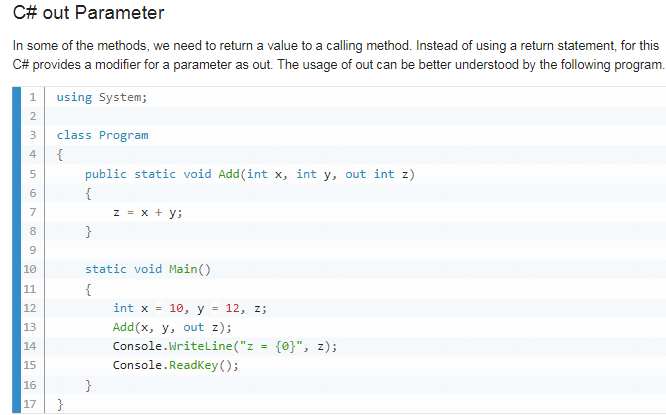
**Method Parameter Modifiers in C#**

Usually methods take parameters. There are many ways to pass the parameters and for this C# provides some parameter modifiers. Look at them below.

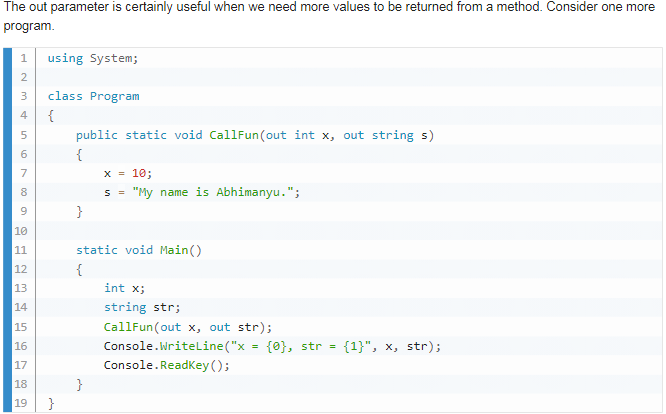
1. **None (or default) parameter:**If a parameter is not attached with any modifier, then the parameter's value is passed to the method. This is also known as call-by-value and it is the default for any parameter.
2. **Ref (reference) parameter:**If a parameter is attached with a ref modifier, then changes will be made in a method that affects the calling method. This is also known as call-by-reference.
3. **Out (output) parameter:**If a parameter is attached with an out modifier, then we can return a value to a calling method without using a return statement.
4. **Params (parameters) parameter:** If a parameter is attached with a params modifier, then we can send multiple arguments as a single parameter. Any method can have only one params modifier and it should be the last parameter for the method.



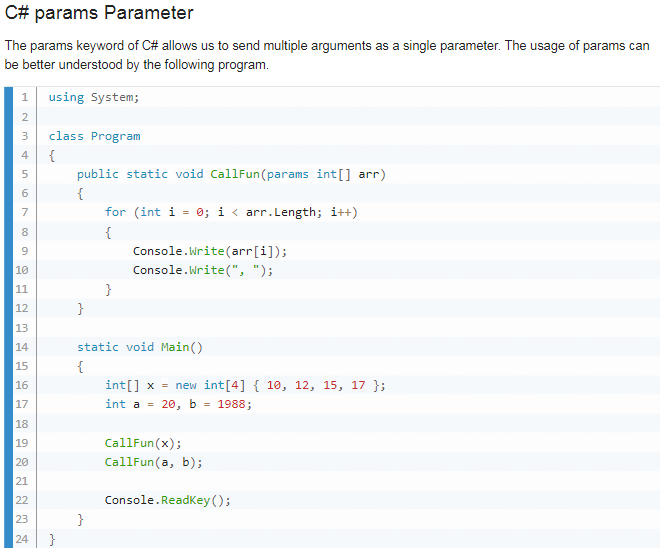




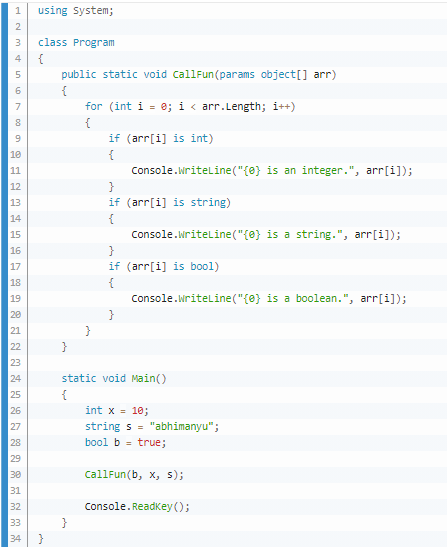




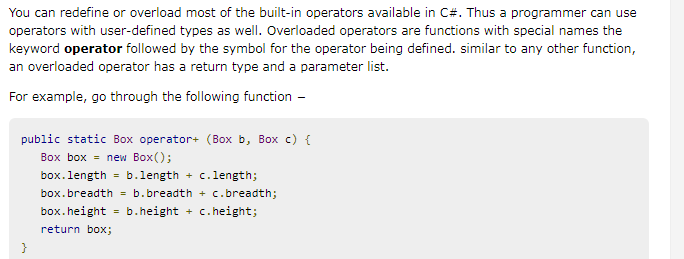




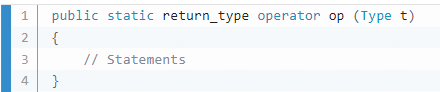


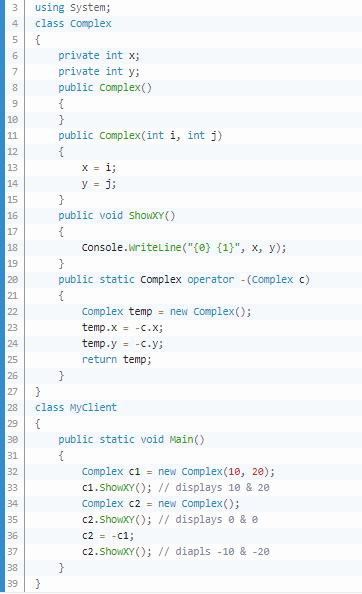


**Operator Overloading**



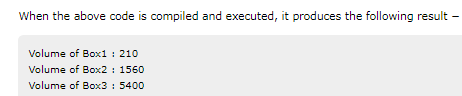


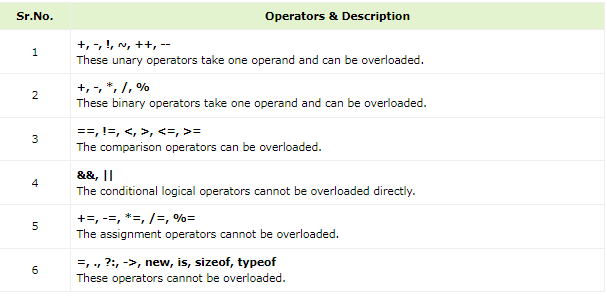


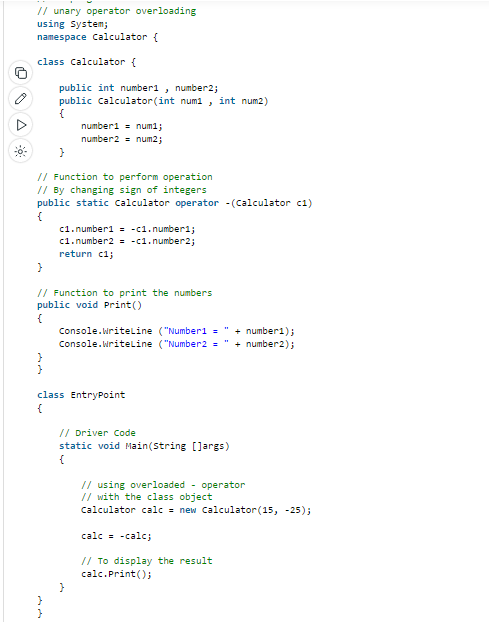
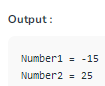


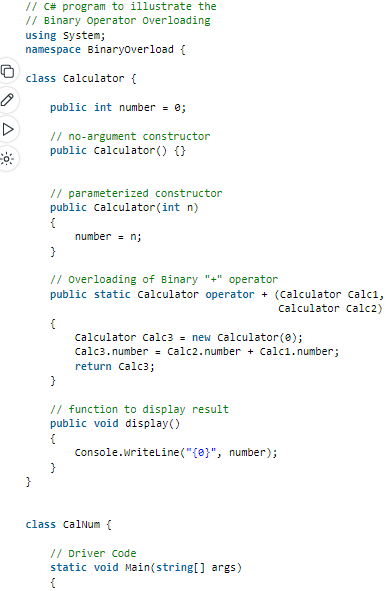


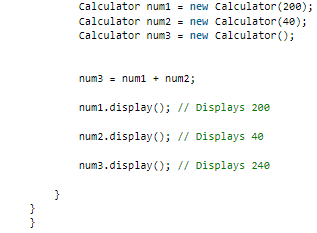
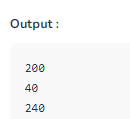






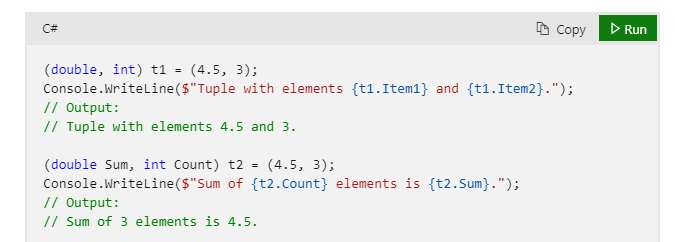
 

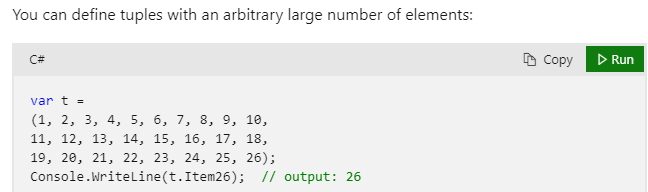


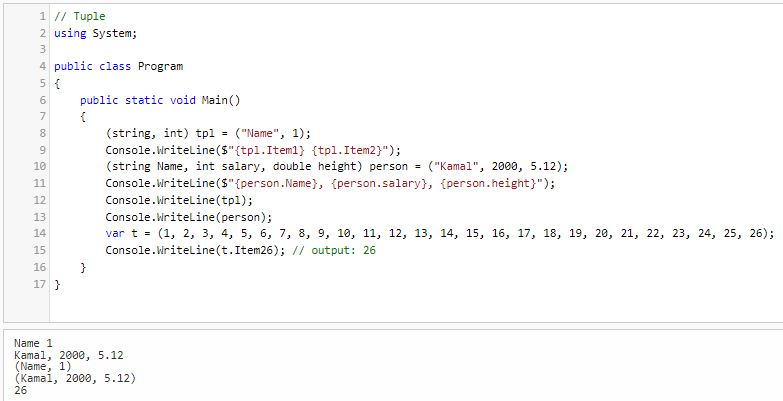
 

**Tuple**

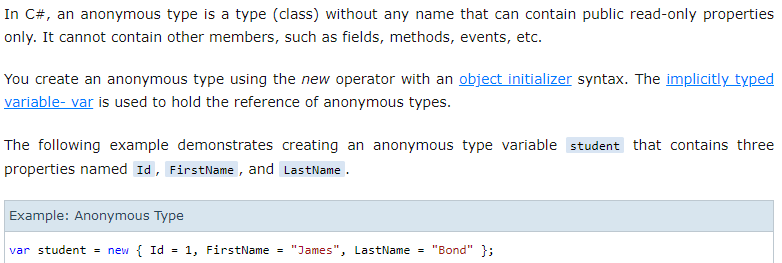
Making a custom data type while not making a separate class for it. It’s like not using a canon for killing a mosquito. Like making a mini class for variable. The *tuples* feature provides concise syntax to group multiple data elements in a lightweight data structure. We can learn more in Microsoft Website.

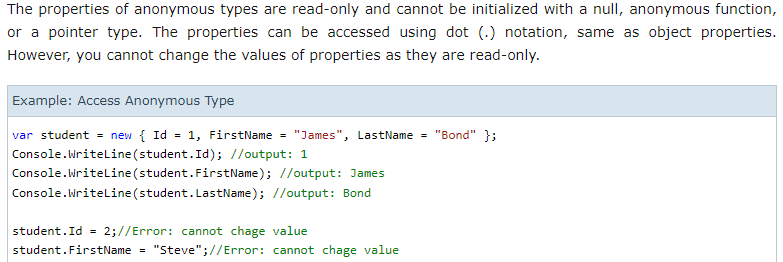


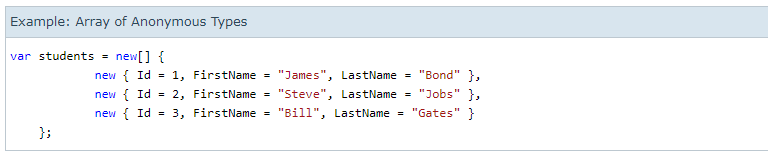


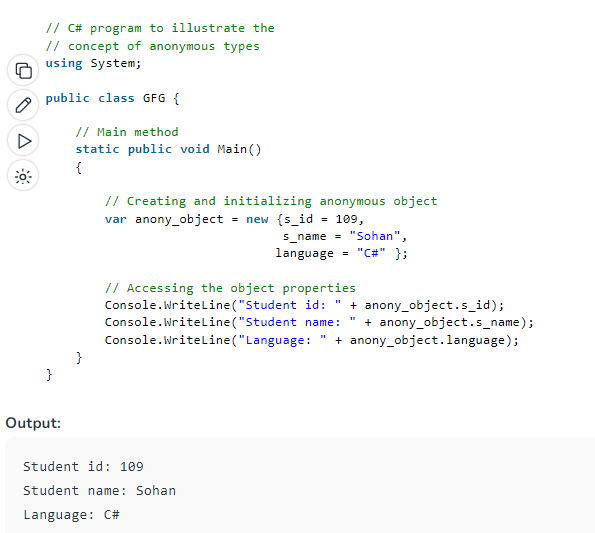


**Anonymous Object/Type**

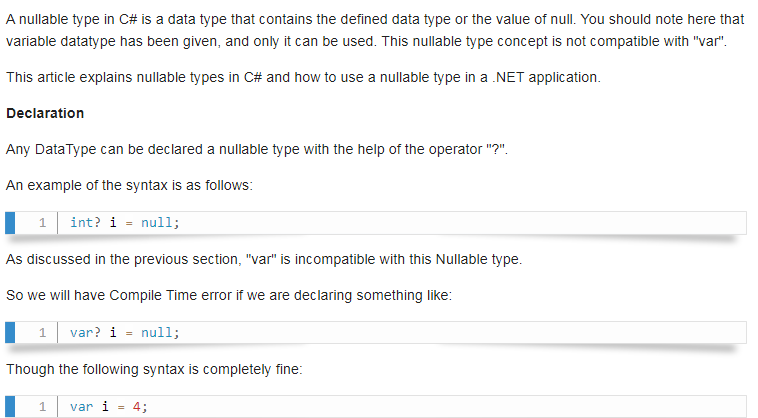


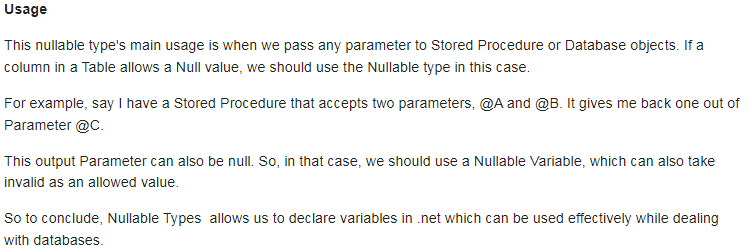


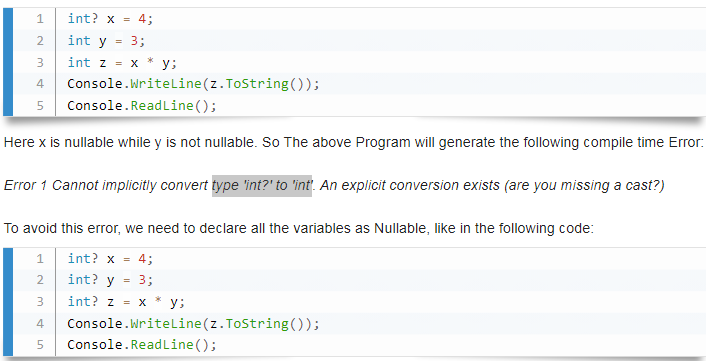


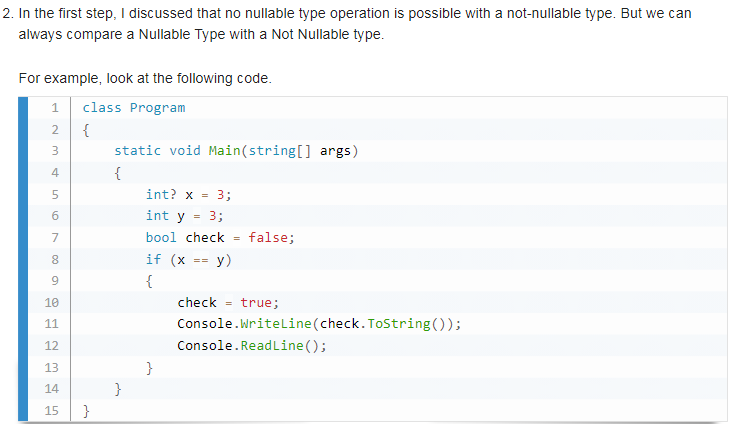


**Nullable Type**



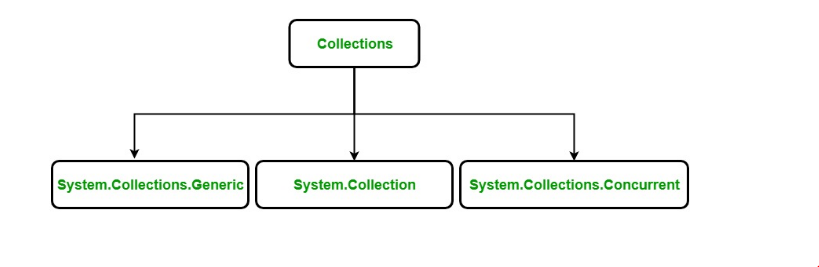


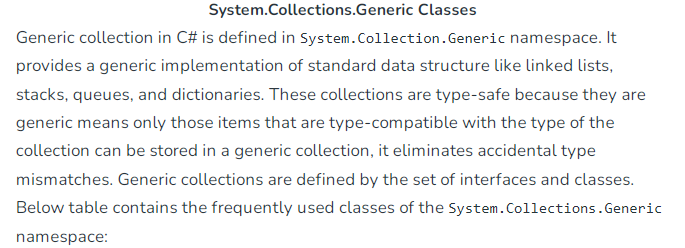


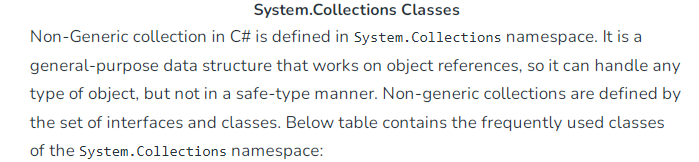


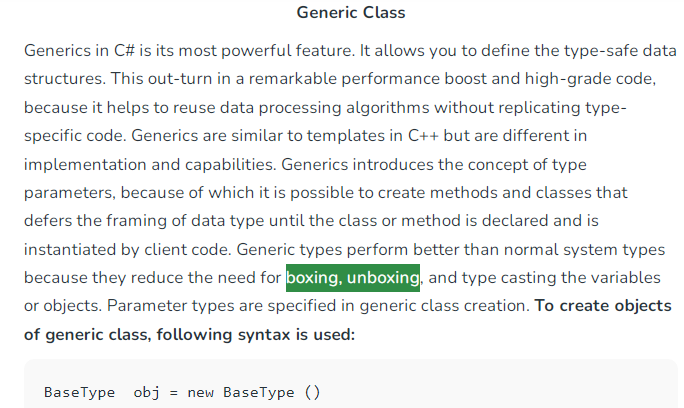
**Collections**

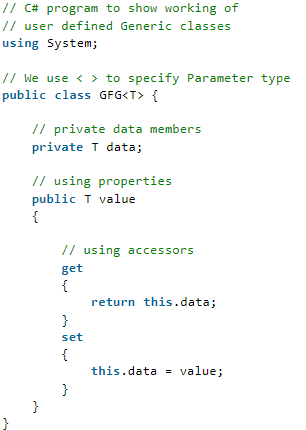
Collections standardize the way of which the objects are handled by your program. In other words, it contains a set of classes to contain elements in a generalized manner. With the help of collections, the user can perform several operations on objects like the store, update, delete, retrieve, search,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

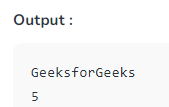


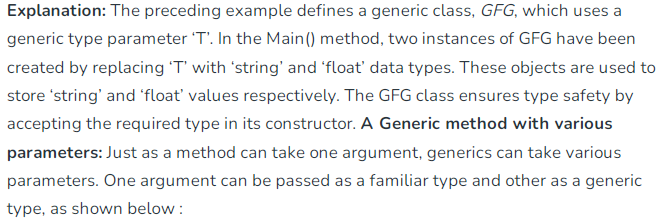




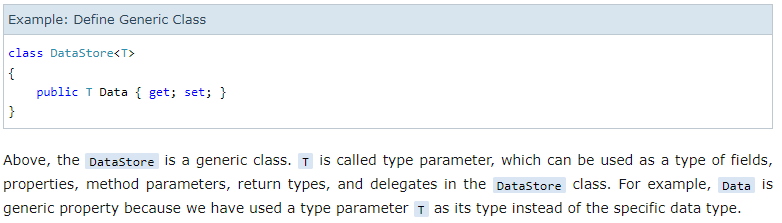


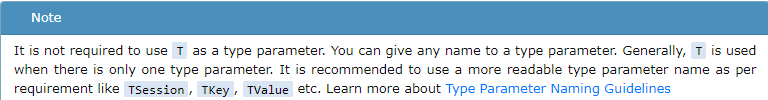


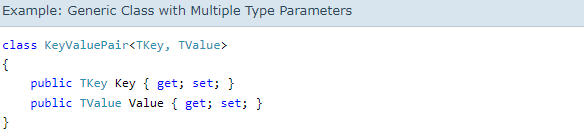
 



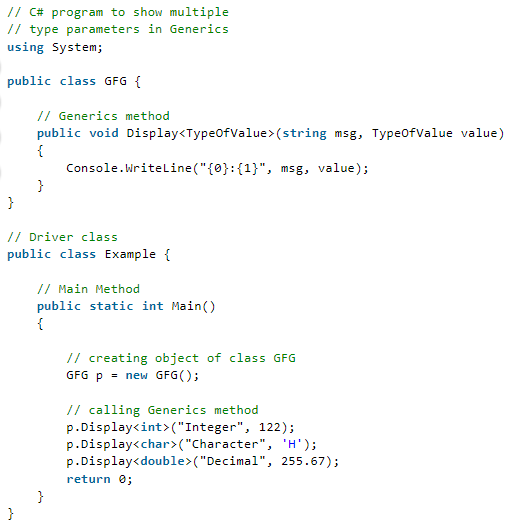
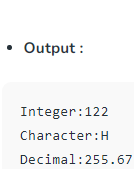


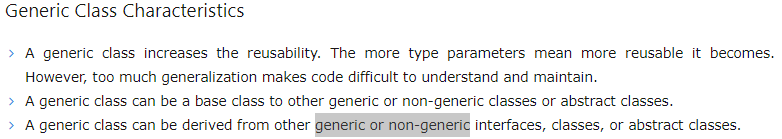


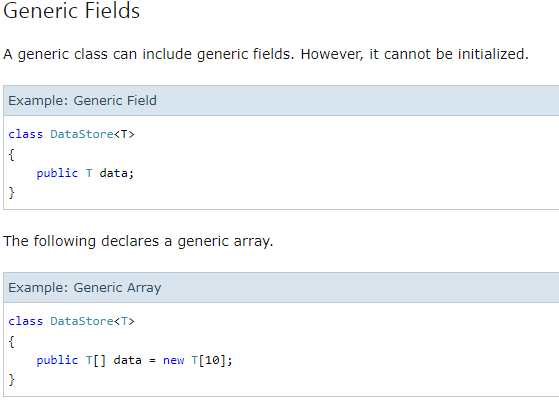


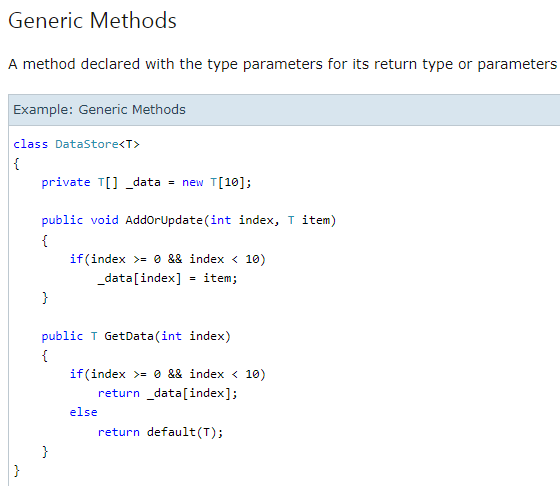




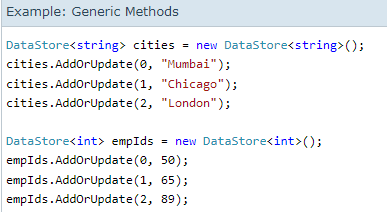
 



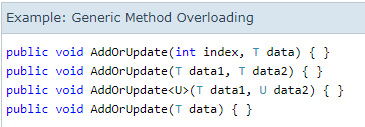


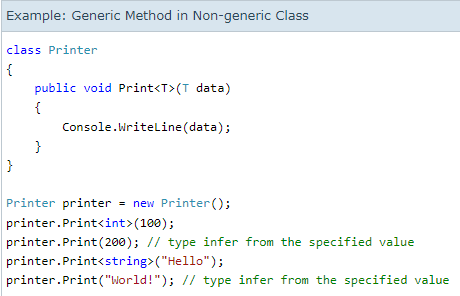






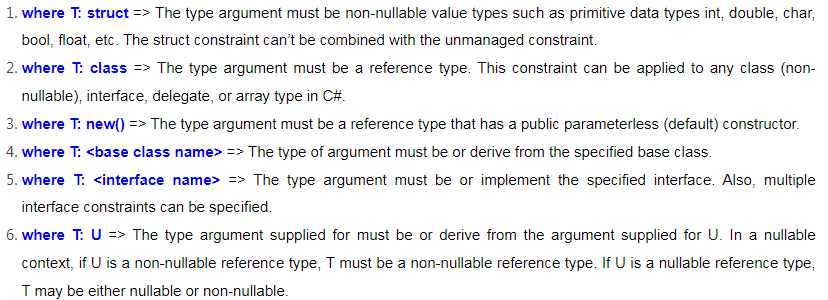






**Generic Constraints**

Constraints are validations that we can put on the generic type parameters. At the instantiation time of the generic class, if we provide an invalid type, then the compile will give an error. In C#, the generic constraints are specified by using the where keyword. The following are the list of different type of generic constraints available



**where T: class**

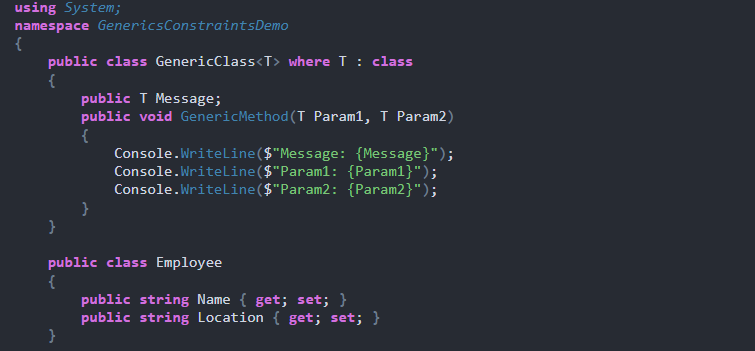
The type argument must be a reference type. We know a class is a reference type in C#. So “**where T: class**” is a reference type constraint. That means this constraint can be applied to any class (non-nullable), interface, delegate, or array.

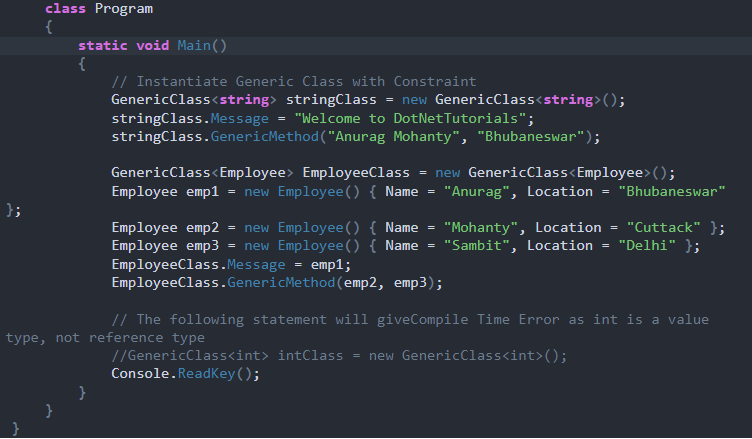
If you observe the above code, here, we defined the GenericClass with the “**where T: class**” constraint. That means, now the GenericClass will only accept reference-type arguments. Let’s create an instance of the Generic class by passing reference-type arguments as follows. In C#, the string is a reference type.  
**GenericClass<string> stringClass = new GenericClass<string>();**

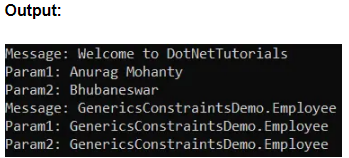
The following statement will give you a compile-time error as int is a value type, not a reference type.  
**GenericClass<int> intClass = new GenericClass<int>();**

**Example to understand where T: class Generic Constraint in C#**

When we created an instance of GenericClass using reference type arguments such as string and class, it works fine. But, when we try to create an instance with built-in types like int, bool, etc., we will get a compile-time error.







**where T: struct**

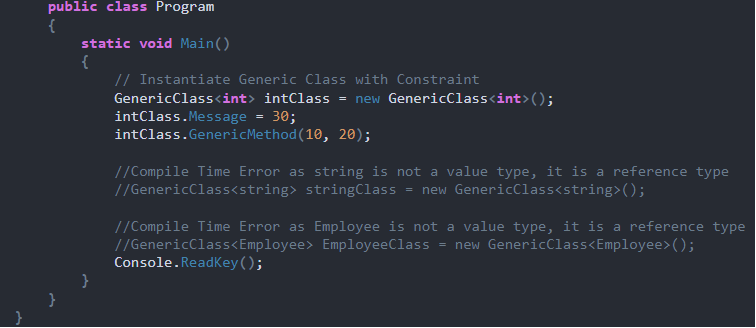
If you want the type argument to accept only the value type then you need to use where T: struct constraints in C#. In this case, the type argument must be non-nullable value types such as int, double, char, bool, float, etc. The struct constraint can’t be combined with the unmanaged constraint. Let us see an example to understand where T: struct constraint.

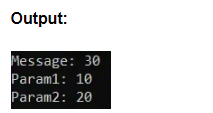
If you observe the above code, here, we defined the GenericClass with the “**where T: stru**ct” generic constraint. That means, now the GenericClass will only accept value-type arguments. Let us create an instance of GenericClass by passing value-type arguments as follows.  
**GenericClass<int> intClass = new GenericClass<int>();**

The following statement will give you a compile-time error as the string is a reference type, not a value type.  
**GenericClass<string> stringClass = new GenericClass<string>();**

When we created an instance of GenericClass using value-type arguments such as int, it works fine. But, when we try to create an instance with reference types such as String, Employee, etc., we will get a compile-time error.







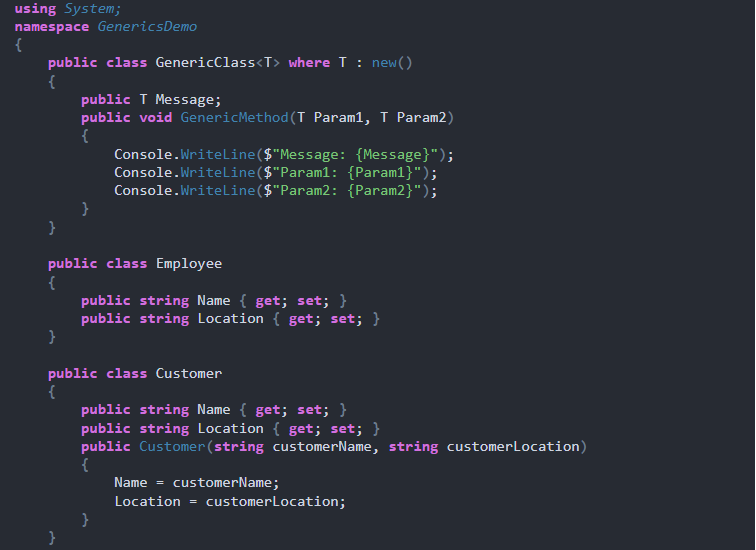
**where T: new()**

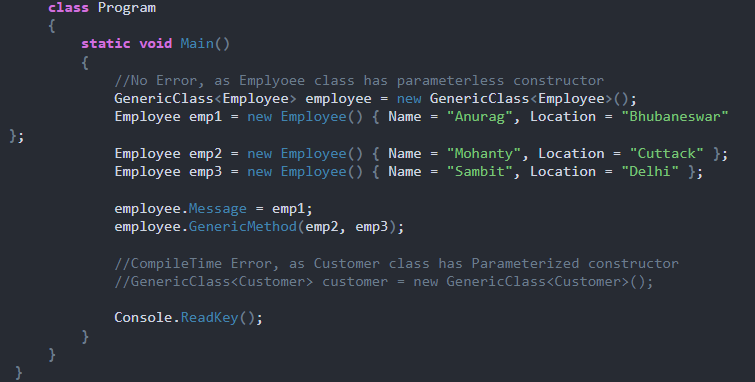
Here, the type argument must be a reference type that has a public parameterless (default) constructor. That means with the help of the new() constraint, we can only specify types which has a parameterless constructor.

As you can see in the above code, we have used **where T: new()** constraint which allows the type which has a parameterless default constructor(Employee). Now, let us create two more classes with one class having a parameterless default constructor and another class having parameterized constructor (Customer)

As you can see in the above code, we have not defined any constructor explicitly in the Employee class, so the compiler will provide a parameter-less default constructor. On the other hand, in the Customer class, we have defined one parameterized constructor explicitly. Now, Let us create an instance of GenericClass bypassing Employee type arguments as follows.  
**GenericClass<Employee> employee = new GenericClass<Employee>();**

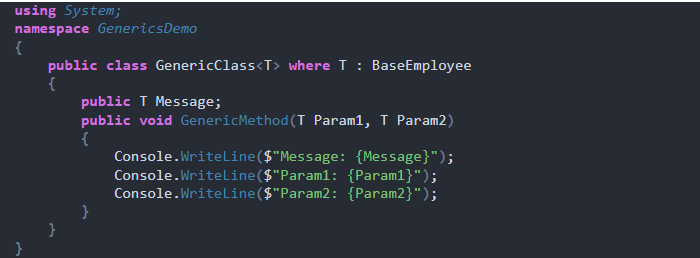
The following statement will give you a compile-time error as the Customer class has a Parameterized constructor or you can say the Customer class does not have a default parameterless constructor.  
**GenericClass<Customer> customer = new GenericClass<Customer>();**



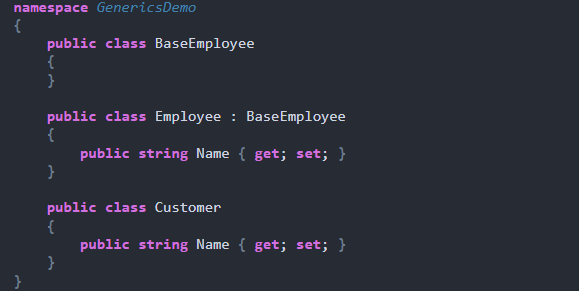


**where T: BaseClass**

Here, the type of argument must be derived from the specified base class. That means in the <base class> constraint, we can only specify types that are inherited from <base class>. The following example shows the base class constraint that restricts the type argument to be a derived class of the specified class.



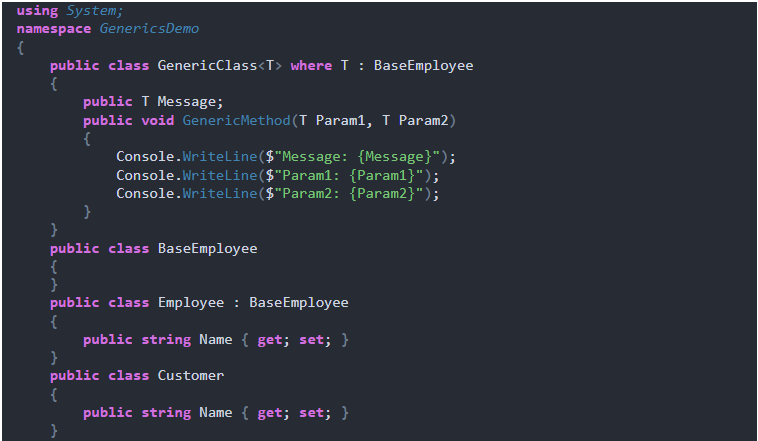
As you can see in the above code, here we have uses **where T: BaseEmployee** constraint which allows the type which is the derived class, abstract class, and interface of the BaseEmployee type. Now, let us create three more classes as follows.

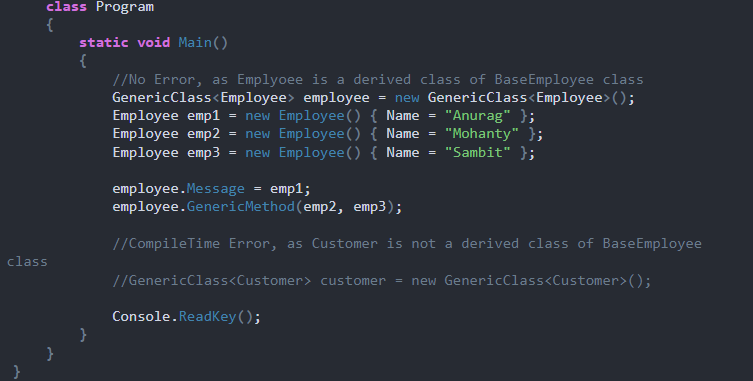


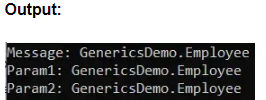
As you can see in the above code, the Employee class is inherited from the BaseEmployee class i.e. Employee is the derived class of the BaseEmployee class. On the other hand, Customer is not derived from the BaseEmployee class. Now, Let’s create an instance of GenericClass bypassing Employee type arguments as follows. It works fine because Employee is derived from the BaseEmployee class.  
**GenericClass<Employee> employee = new GenericClass<Employee>();**

The following statement will give you a compile-time error as the Customer class is not derived from the BaseEmployee type.  
**GenericClass<Customer> customer = new GenericClass<Customer>();**

When we created an instance of GenericClass using the Employee type argument, it works fine because Employee is the derived class of the BaseEmployee class. But, when we try to create an instance with Customer type, we will get a compile-time error because Customer is not a derived class of the BaseEmployee class.

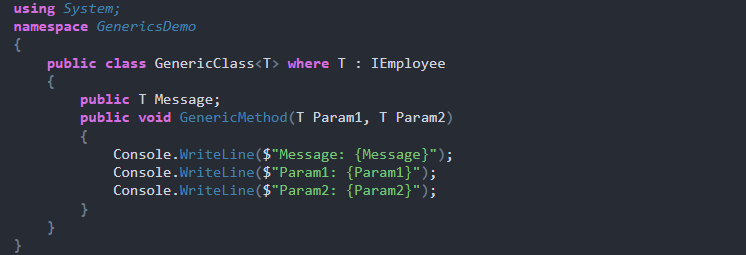




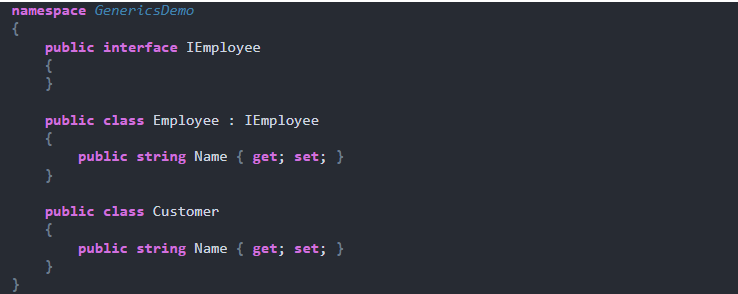


**where T: Interface**

Here, the type argument must be or implement the specified interface. Also, multiple interface constraints can be specified. That means in the <interface> constraint, we can only specify types that implement the <interface>



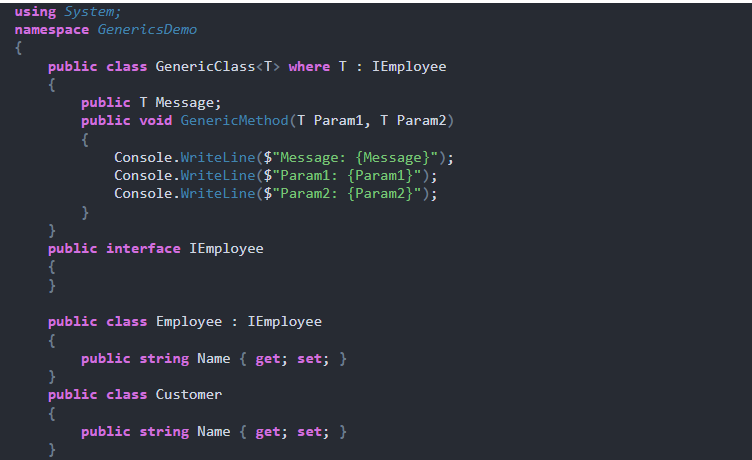
As you can see in the above code, here we have uses **where T: IEmployee** constraint which allows the type that should implement the IEmployee interface. Now, let us create one interface and two more classes as follows.

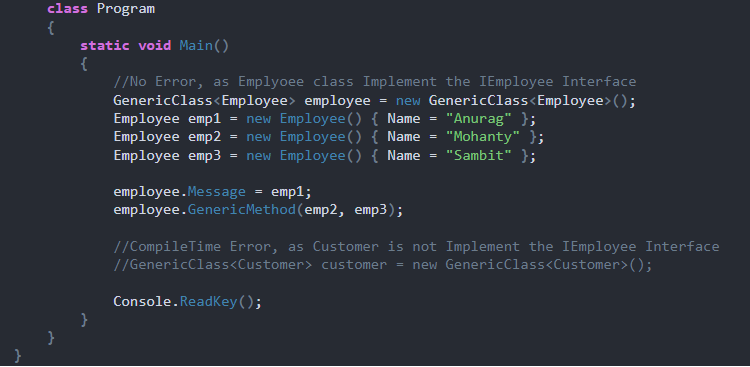


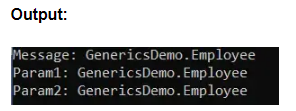
As you can see in the above code, the Employee class is implemented IEmployee interface. On the other hand, the Customer is not implementing the IEmployee interface. Now, let us create an instance of GenericClass bypassing Employee type arguments as follows. It works fine because the Employee class implements the IEmployee interface.  
**GenericClass<Employee> employee = new GenericClass<Employee>();**

The following statement will give you a compile-time error as the Customer class does not implement the IEmployee interface.  
**GenericClass<Customer> customer = new GenericClass<Customer>();**

When we created an instance of GenericClass using the Employee type argument, it works fine because the Employee class implements the IEmployee interface. But, when we try to create an instance with Customer type, we will get a compile-time error because the Customer class does not implement the IEmployee interface.

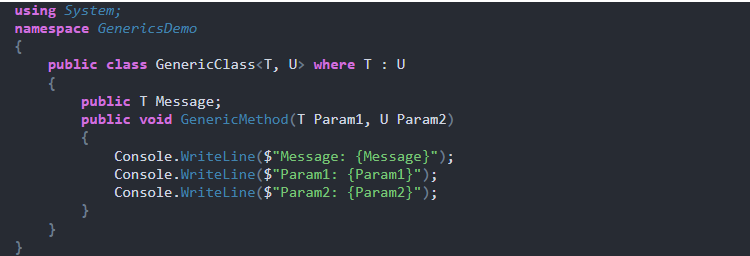




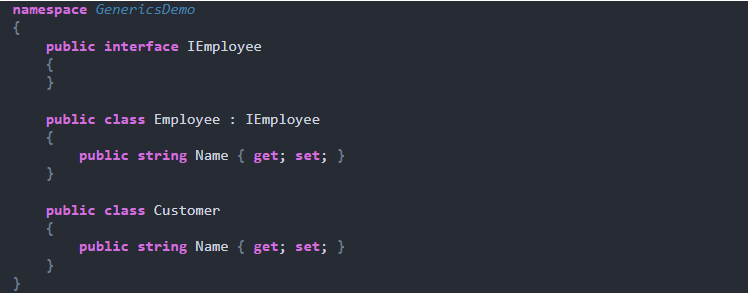


#### ****where T: U****

Here, the type argument supplied must be or derive from the argument supplied for U. In a nullable context, if U is a non-nullable reference type, T must be a non-nullable reference type. If U is a nullable reference type, T may be either nullable or non-nullable. So, in this constraint, there are two Type Arguments i.e. T and U. U can be an interface, abstract class, or simple class. T must inherit or implements the U class.

****

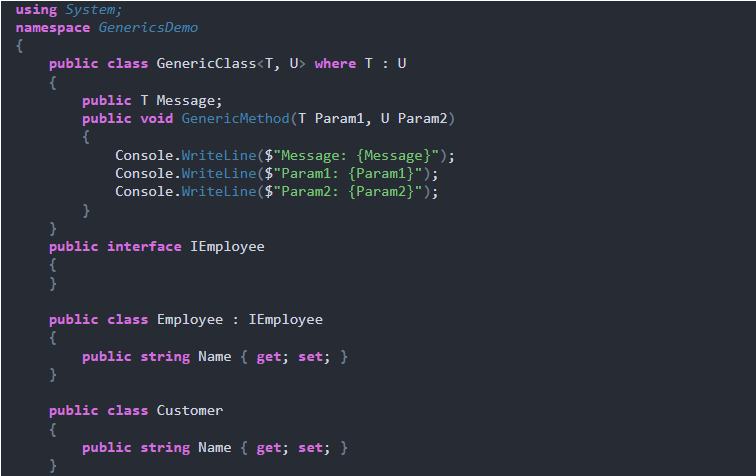
As you can see in the above code, here we have used **where T: U** constraint which allows the type (T) that must inherit or implements the U class. Now, let us create one interface and two more classes as follows.

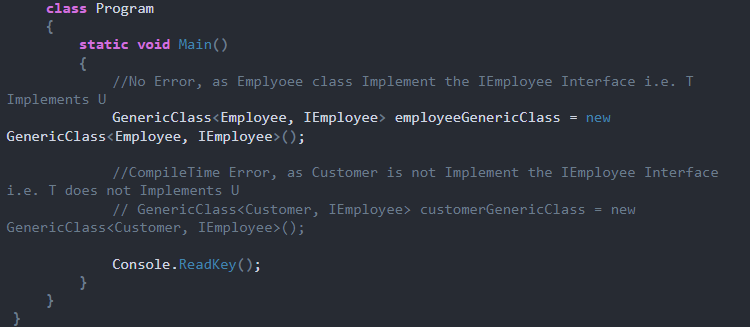
****

As you can see in the above code, the Employee class implements the IEmployee interface. On the other hand, the Customer class is not implementing the IEmployee interface. Now, Let’s create an instance of Genericclass bypassing Employee and IEmployee as type arguments for T and U as follows. It works fine because the Employee class implements the IEmployee interface.  
**GenericClass<Employee, IEmployee> employeeGenericClass = new GenericClass<Employee, IEmployee>();**

The following statement will give you a compile-time error as the Customer class does not implement the IEmployee interface i.e. T does not implement U.  
**GenericClass<Customer, IEmployee> customerGenericClass = new GenericClass<Customer, IEmployee>();**

When we created an instance of GenericClass using Employee and IEmployee type argument, it works fine because the Employee class implements the IEmployee interface. But, when we try to create an instance with Customer type, we will get a compile-time error because the Customer class does not implement the IEmployee interface.



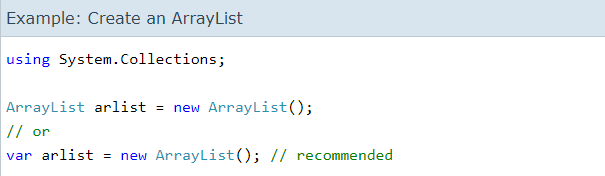


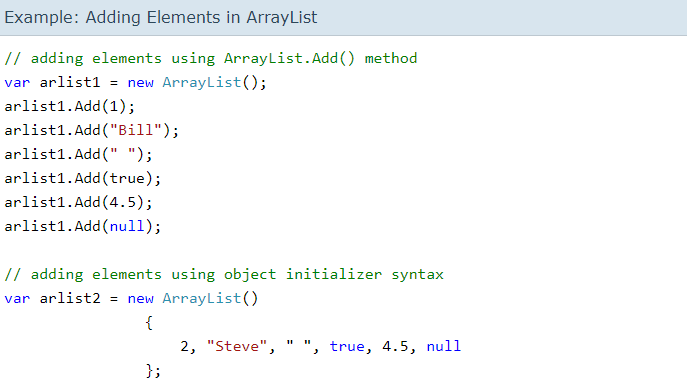
**Array List (**Non Generic**)**

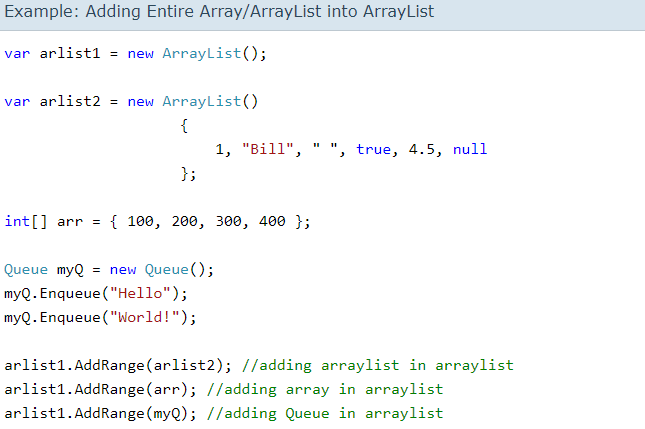
In C#, the ArrayList is a non-generic collection of objects whose size increases dynamically. It is the same as [Array](https://www.tutorialsteacher.com/csharp/array-csharp) except that its size increases dynamically.

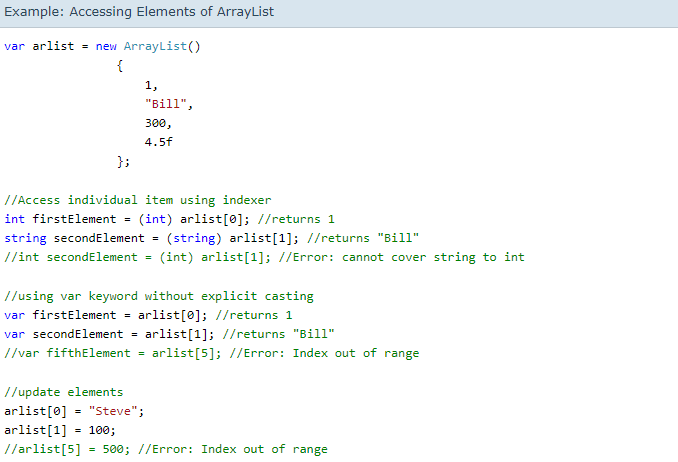
An ArrayList can be used to add unknown data where you don't know the types and the size of the data.

The ArrayList class included in the System.Collections namespace. Create an object of the ArrayList using the new keyword.





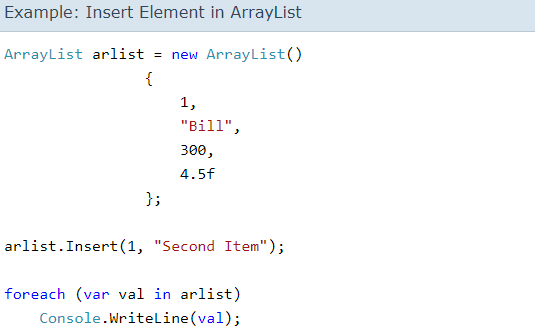




## Insert Elements in ArrayList

Use the Insert() method to insert an element at the specified index into an ArrayList.

Signature: *void Insert(int index, Object value)*





Use the Contains() method to determine whether the specified element exists in the ArrayList or not. It returns true if exists otherwise returns false.

Console.WriteLine(arList.Contains(300)); // true

Console.WriteLine(arList.Contains("Bill")); // true

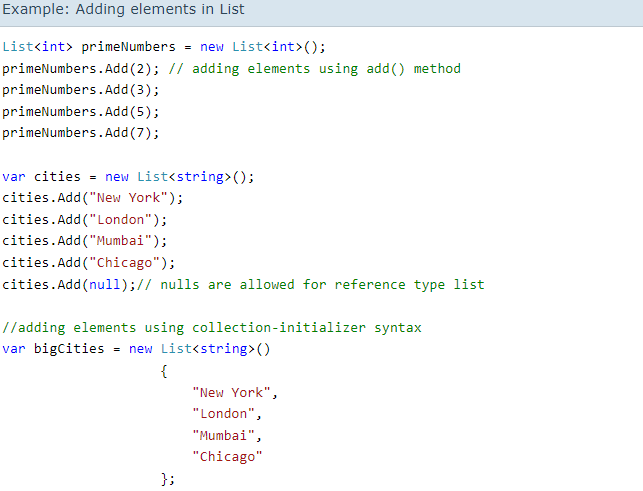
Console.WriteLine(arList.Contains(10)); // false

Console.WriteLine(arList.Contains("Steve")); // false

**List<T>(**Generic**)**

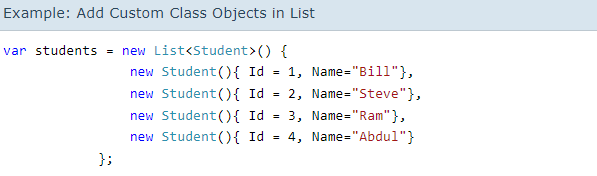
The List<T> is a collection of strongly typed objects that can be accessed by index and having methods for sorting, searching, and modifying list. It is the [generic](https://www.tutorialsteacher.com/csharp/csharp-generics) version of the [ArrayList](https://www.tutorialsteacher.com/csharp/csharp-arraylist) that comes under System.Collections.Generic namespace.

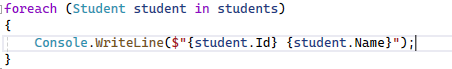
* List<T> equivalent of the [ArrayList](https://www.tutorialsteacher.com/csharp/csharp-arraylist), which implements [IList<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.ilist-1?view=netframework-4.8" \t "_blank).
* It comes under System.Collections.Generic namespace.
* List<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Add(), AddRange() methods or collection-initializer syntax.
* Elements can be accessed by passing an index e.g. myList[0]. Indexes start from zero.
* List<T> performs faster and less error-prone than the ArrayList.



In the above example, List<int> primeNumbers = new List<int>(); creates a list of int type. In the same way, cities and bigCities are string type list. You can then add elements in a list using the Add() method or the collection-initializer syntax.

You can also add elements of the custom classes using the collection-initializer syntax. The following adds objects of the Student class in the List<Student>.

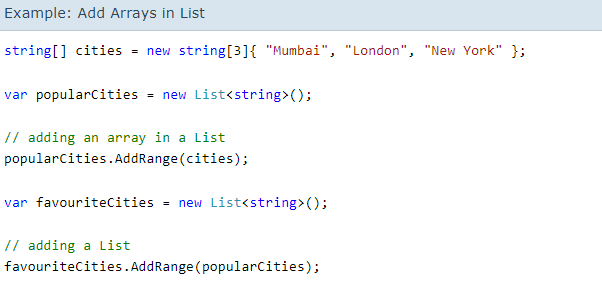


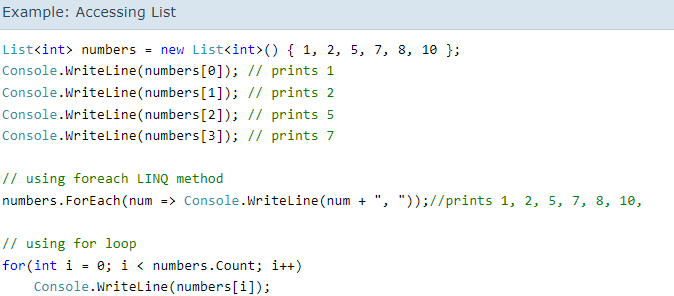


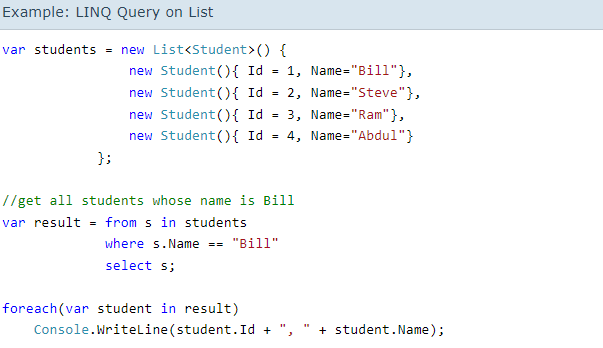
## Adding an Array in a List

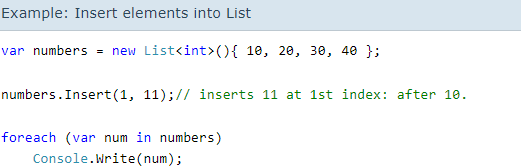
Use the AddRange() method to add all the elements from an array or another collection to List.

AddRange() signature: void AddRange(IEnumerable<T> collection)

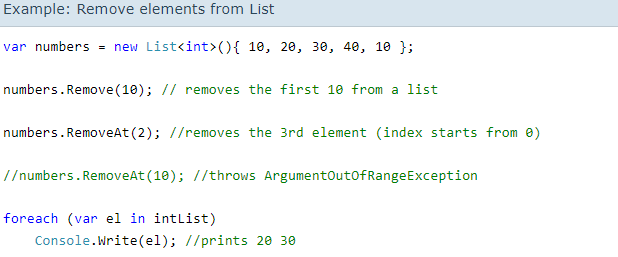


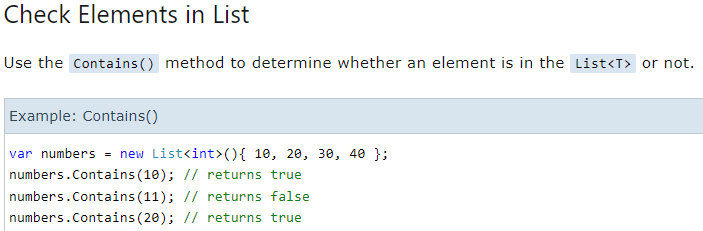


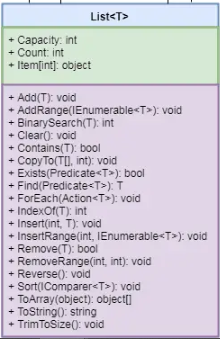
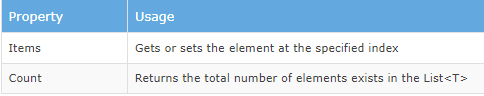


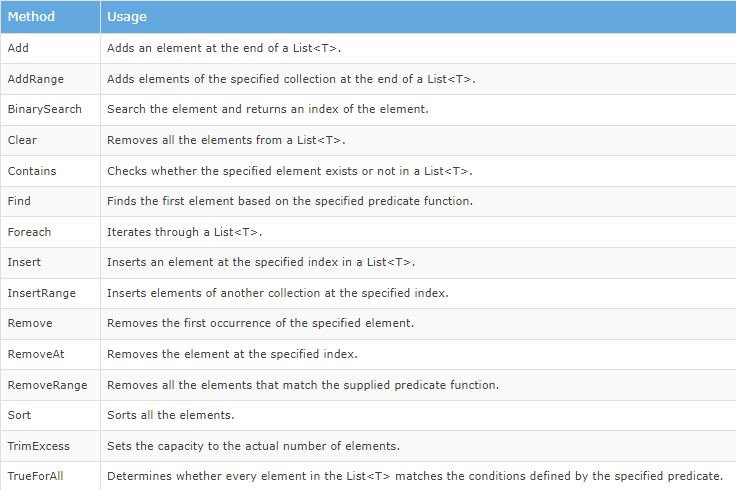






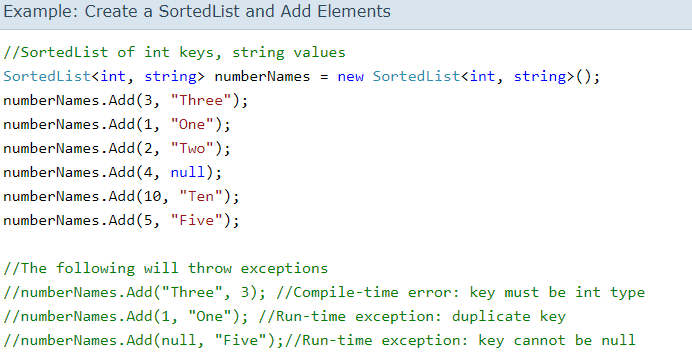


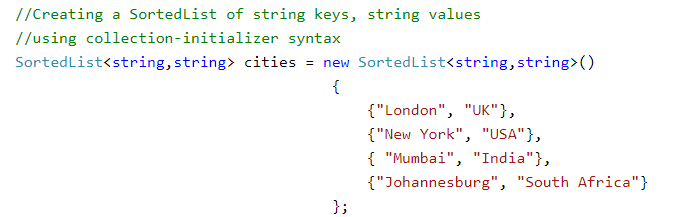
 



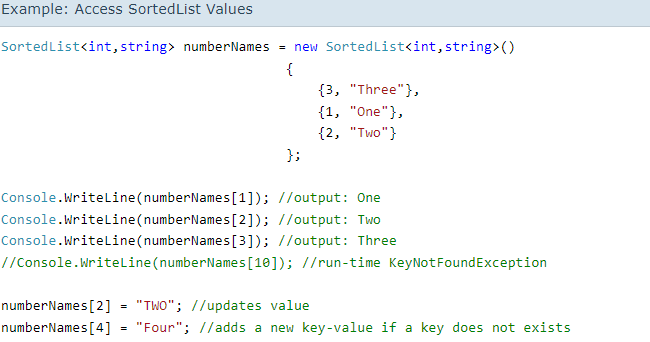
**SortedList<TKey, TValue>(**Generic**)**

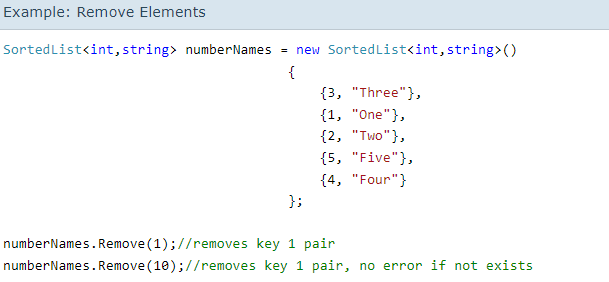
* SortedList<TKey, TValue> is an array of key-value pairs sorted by keys.
* Sorts elements as soon as they are added. Sorts primitive type keys in ascending order and object keys based on [IComparer<T>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.icomparer-1?view=netframework-4.8" \t "_blank).
* Comes under System.Collection.Generic namespace.
* A key must be unique and cannot be null.
* A value can be null or duplicate.
* A value can be accessed by passing associated key in the indexer mySortedList[key]
* Contains elements of type [KeyValuePair<TKey, TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.keyvaluepair-2?view=netframework-4.8" \t "_blank)
* It uses less memory than [SortedDictionary<TKey,TValue>.](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.sorteddictionary-2?view=netframework-4.8" \t "_blank)
* It is faster in the retrieval of data once sorted, whereas SortedDictionary<TKey, TValue> is faster in insertion and removing key-value pairs.







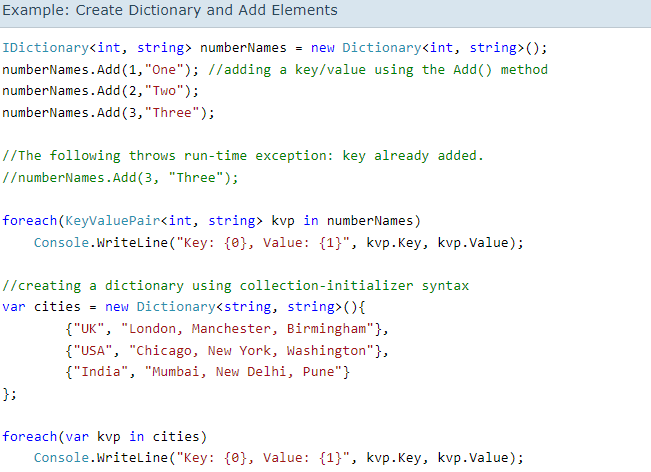




**Dictionary<TKey, TValue>(**Generic**)**

* Dictionary<TKey, TValue> stores key-value pairs.
* Comes under System.Collections.Generic namespace.
* Implements [IDictionary<TKey, TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.idictionary-2?view=netframework-4.8" \t "_blank) interface.
* Keys must be unique and cannot be null.
* Values can be null or duplicate.
* Values can be accessed by passing associated key in the indexer e.g. myDictionary[key]
* Elements are stored as [KeyValuePair<TKey, TValue>](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.keyvaluepair-2?view=netframework-4.8" \t "_blank) objects.



Example: Access Dictionary Elements

var cities = new Dictionary<string, string>(){

{"UK", "London, Manchester, Birmingham"},

{"USA", "Chicago, New York, Washington"},

{"India", "Mumbai, New Delhi, Pune"}

};

Console.WriteLine(cities["UK"]); //prints value of UK key

Console.WriteLine(cities["USA"]);//prints value of USA key

//Console.WriteLine(cities["France"]);

// run-time exception: Key does not exist

//use ContainsKey() to check for an unknown key

if(cities.ContainsKey("France")){

Console.WriteLine(cities["France"]);

}

//use TryGetValue() to get a value of unknown key

string result;

if(cities.TryGetValue("France", out result))

{

Console.WriteLine(result);

}

//use ElementAt() to retrieve key-value pair using index

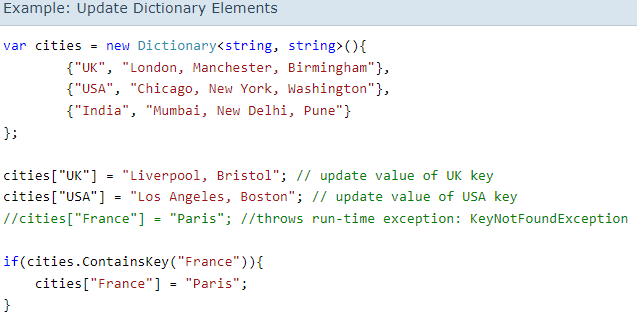
for (int i = 0; i < cities.Count; i++)

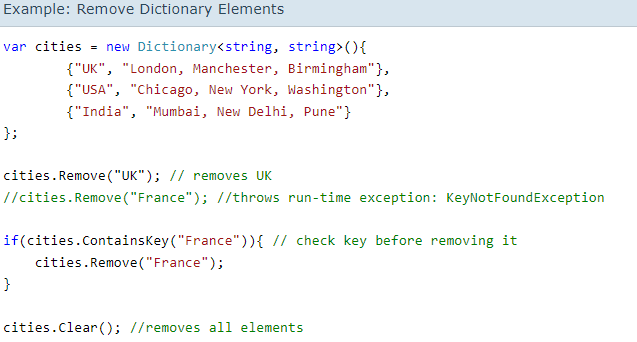
{

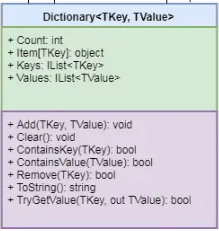
Console.WriteLine("Key: {0}, Value: {1}",

cities.ElementAt(i).Key, cities.ElementAt(i).Value);

}



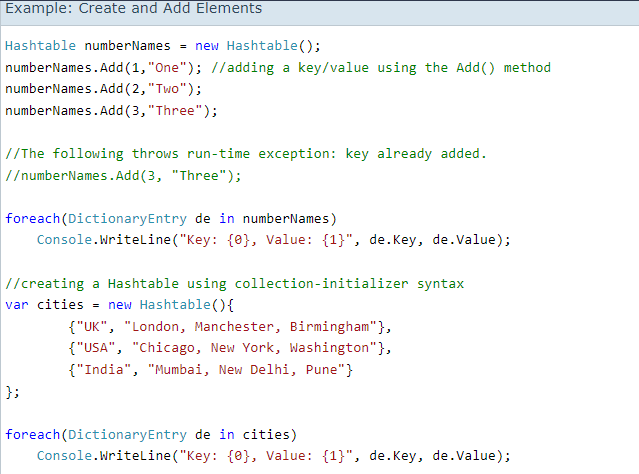


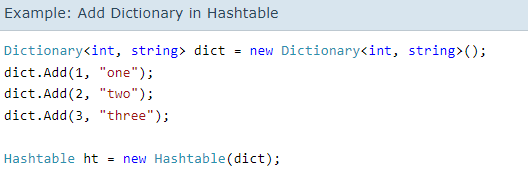


**HashTable**

The Hashtable is a non-generic collection that stores key-value pairs, similar to generic [Dictionary<TKey, TValue>](https://www.tutorialsteacher.com/csharp/csharp-dictionary) collection. It optimizes lookups by computing the hash code of each key and stores it in a different bucket internally and then matches the hash code of the specified key at the time of accessing values.

* Hashtable stores key-value pairs.
* Comes under System.Collections namespace.
* Implements [IDictionary](https://docs.microsoft.com/en-us/dotnet/api/system.collections.idictionary?view=netframework-4.8" \t "_blank) interface.
* Keys must be unique and cannot be null.
* Values can be null or duplicate.
* Values can be accessed by passing associated key in the indexer e.g. myHashtable[key]
* Elements are stored as [DictionaryEntry](https://docs.microsoft.com/en-us/dotnet/api/system.collections.dictionaryentry?view=netframework-4.8" \t "_blank) objects.

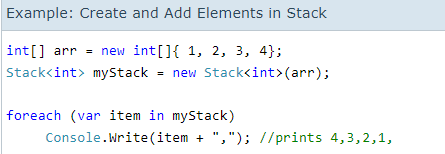


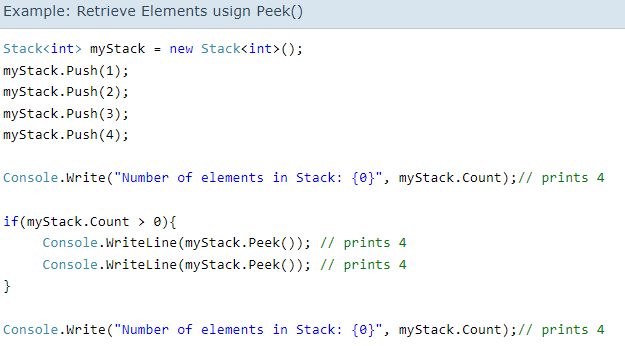




**Stack<T>**

* Stack<T> is Last In First Out collection.
* It comes under System.Collection.Generic namespace.
* Stack<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Push() method. Cannot use collection-initializer syntax.
* Elements can be retrieved using the Pop() and the Peek() methods. It does not support an indexer.
* Stack<T> is Last In First Out collection.
* It comes under System.Collection.Generic namespace.
* Stack<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Push() method. Cannot use collection-initializer syntax.
* Elements can be retrieved using the Pop() and the Peek() methods. It does not support an indexer.



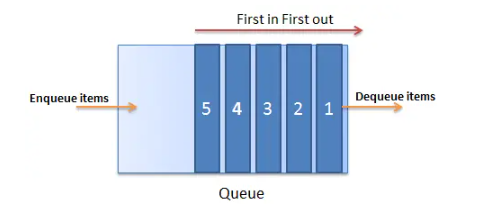


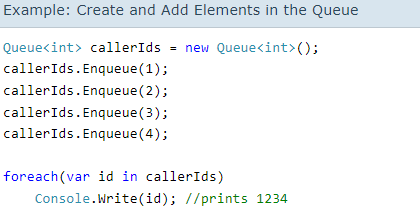
The Contains() method checks whether the specified element exists in a Stack collection or not. It returns true if it exists, otherwise false.

**Queue<T>**

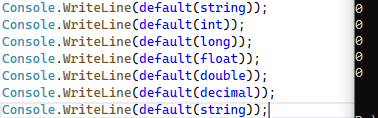
Queue is a special type of collection that stores the elements in FIFO style (First In First Out), exactly opposite of the [Stack<T>](https://www.tutorialsteacher.com/csharp/csharp-stack) collection. It contains the elements in the order they were added. C# includes generic Queue<T> and non-generic Queue collection. It is recommended to use the generic Queue<T> collection.

* Queue<T> is FIFO (First In First Out) collection.
* It comes under System.Collection.Generic namespace.
* Queue<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
* Elements can be added using the Enqueue() method. Cannot use collection-initializer syntax.
* Elements can be retrieved using the Dequeue() and the Peek() methods. It does not support an indexer.



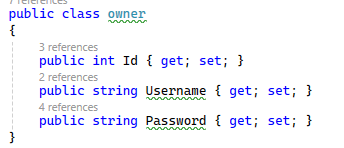
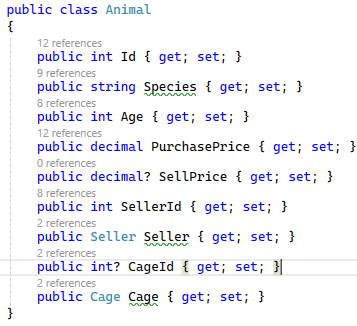


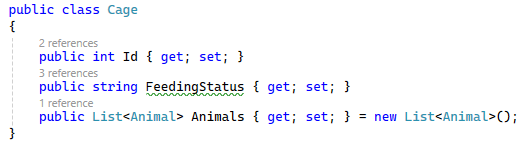
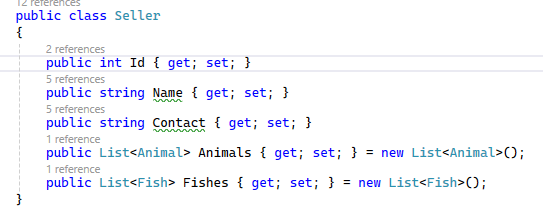
**Default**

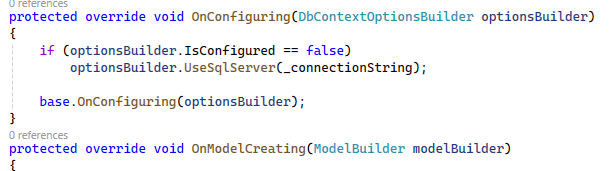


**Entity Framework Core**

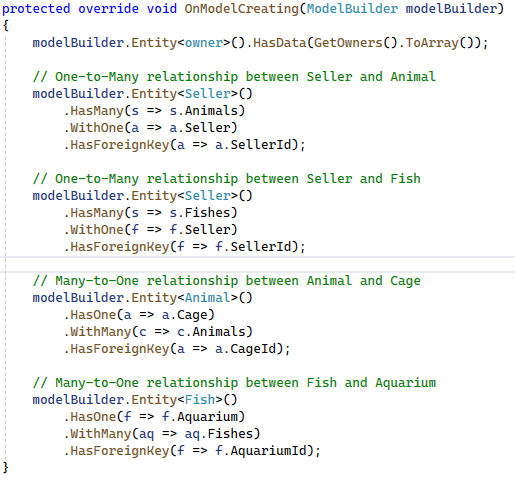
**Models:**

**** ****

**** ****

**APPDbcontext:** ****

**Fluent API:**

****

**DbSet<>:**

****

**Migration Commands(Package Manager Console)**

**tool install(error fix)**

dotnet tool install --global dotnet-ef --version 7.0

**Migration>>**

dotnet ef migrations add AddedStudentTable --project EntityPractice2 --context AppDbContext

**Database Update>>**

dotnet ef database update --project EntityPractice2 --context AppDbContext

**RemoveMigration>>**

dotnet ef database update "NameofMigrationToGoTo" --project EntityPractice2 --context AppDbContext |Removes From DB|

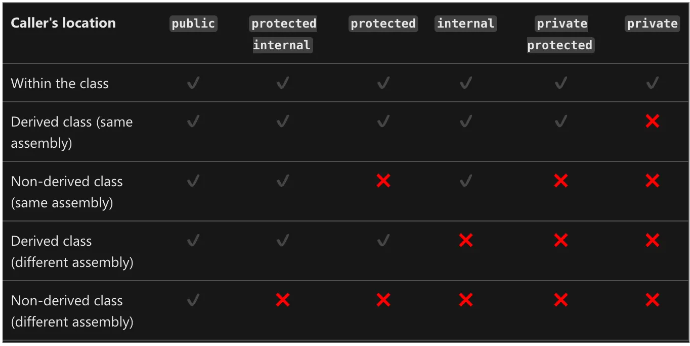
dotnet ef migrations remove --project EntityPractice2 --context AppDbContext |Removes From MIgration Folder In (VS)|

**Nuget Packages**

microsoft.entityframeworkcore.design\7.0.0\

microsoft.entityframeworkcore.sqlserver\7.0.0\

**Important Interfaces  
IEnumerable,**  **ICloneable, IDisposable, ICollection**

****

**Delegates**

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

[modifier] delegate [return\_type] [delegate\_name] ([parameter\_list]);

***modifier:****It is the required modifier which defines the access of delegate and it is optional to use.****delegate:****It is the keyword which is used to define the delegate.****return\_type:****It is the type of value returned by the methods which the delegate will be going to call. It can be void. A method must have the same return type as the delegate.****delegate\_name:****It is the user-defined name or identifier for the delegate.****parameter\_list:****This contains the parameters which are required by the method when called through the delegate.*

**Example:**

// "public" is the modifier

// "int" is return type

// "GeeksForGeeks" is delegate name

// "(int G, int F, int G)" are the parameters

public delegate int GeeksForGeeks(int G, int F, int G);

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

#### Instantiation & Invocation of Delegates

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

[delegate\_name] [instance\_name] = new [delegate\_name](calling\_method\_name);

**Example:**

GeeksForGeeks GFG = new GeeksForGeeks (Geeks);

// here,

// "GeeksForGeeks" is delegate name.

// "GFG" is instance\_name

// "Geeks" is the calling method.

// C# program to illustrate the use of Delegates

using System;

namespace GeeksForGeeks {

// declare class "Geeks"

class Geeks {

// Declaring the delegates

// Here return type and parameter type should

// be same as the return type and parameter type

// of the two methods

// "addnum" and "subnum" are two delegate names

public delegate void addnum(int a, int b);

public delegate void subnum(int a, int b);

    // method "sum"

    public void sum(int a, int b)

    {

        Console.WriteLine("(100 + 40) = {0}", a + b);

    }

    // method "subtract"

    public void subtract(int a, int b)

    {

        Console.WriteLine("(100 - 60) = {0}", a - b);

    }

// Main Method

public static void Main(String []args)

{

    // creating object "obj" of class "Geeks"

    Geeks obj = new Geeks();

    // creating object of delegate, name as "del\_obj1"

    // for method "sum" and "del\_obj2" for method "subtract" &

    // pass the parameter as the two methods by class object "obj"

    // instantiating the delegates

    addnum del\_obj1 = new addnum(obj.sum);

    subnum del\_obj2 = new subnum(obj.subtract);

    // pass the values to the methods by delegate object

    del\_obj1(100, 40);

    del\_obj2(100, 60);

    // These can be written as using

    // "Invoke" method

    // del\_obj1.Invoke(100, 40);

    // del\_obj2.Invoke(100, 60);

}

}

}

**Output:** 

(100 + 40) = 140

(100 - 60) = 40

**Explanation:**In the above program, there are two delegates *addnum*and *subnum*. We are creating the object *obj*of the class Geeks because both the methods(*addnum*and *subnum*) are instance methods. So they need an object to call. If methods are static then there is no need to create the object of the class.

// C# program to illustrate the

// Multicasting of Delegates

using System;

class rectangle {

// declaring delegate

public delegate void rectDelegate(double height,

                                  double width);

    // "area" method

    public void area(double height, double width)

    {

        Console.WriteLine("Area is: {0}", (width \* height));

    }

    // "perimeter" method

    public void perimeter(double height, double width)

    {

        Console.WriteLine("Perimeter is: {0} ", 2 \* (width + height));

    }

// Main Method

public static void Main(String []args)

{

    // creating object of class

    // "rectangle", named as "rect"

    rectangle rect = new rectangle();

    // these two lines are normal calling

    // of that two methods

    // rect.area(6.3, 4.2);

    // rect.perimeter(6.3, 4.2);

    // creating delegate object, name as "rectdele"

    // and pass the method as parameter by

    // class object "rect"

    rectDelegate rectdele = new rectDelegate(rect.area);

    // also can be written as

    // rectDelegate rectdele = rect.area;

    // call 2nd method "perimeter"

    // Multicasting

    rectdele += rect.perimeter;

    // pass the values in two method

    // by using "Invoke" method

    rectdele.Invoke(6.3, 4.2);

    Console.WriteLine();

    // call the methods with

    // different values

    rectdele.Invoke(16.3, 10.3);

}

}

**Output:**

Area is: 26.46

Perimeter is: 21

Area is: 167.89

Perimeter is: 53.2

**Binary Search**

Console.Write("Enter The Array:");

string[] T = Console.ReadLine().Split();

List<int> list = new List<int>();

for (int i = 0; i < T.Length; i++)

{

list.Add(int.Parse(T[i]));

}

int[] array = list.ToArray();

Array.Sort(array);

Console.Write("Sorted Array: ");

foreach (int l in array)

{

Console.Write(l + " ");

}

Console.Write("\nEnter Your Key To Find:");

int Key = int.Parse(Console.ReadLine());

static int binarySearch(int[] array, int Key)

{

int min = 0, max = array.Length - 1;

while (min <= max)

{

int mid = (min + max) / 2;

if (Key == array[mid])

{

return ++mid;

}

else if (Key < mid)

{

max = mid - 1;

}

else

{

min = mid + 1;

}

}

return -1;

}

int r = binarySearch(array, Key); Console.WriteLine(r);

**Events**

An event is a notification sent by an object to signal the occurrence of an action. Events in .NET follow the [observer design pattern](https://docs.microsoft.com/en-us/dotnet/standard/events/observer-design-pattern).

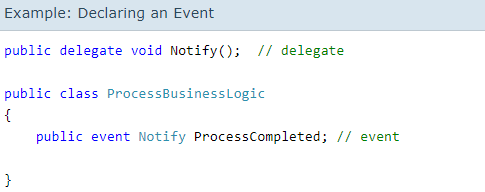
The class who raises events is called Publisher, and the class who receives the notification is called Subscriber. There can be multiple subscribers of a single event. Typically, a publisher raises an event when some action occurred. The subscribers, who are interested in getting a notification when an action occurred, should register with an event and handle it.

In C#, an event is an encapsulated [delegate](https://www.tutorialsteacher.com/csharp/csharp-delegates). It is dependent on the delegate. The [delegate](https://www.tutorialsteacher.com/csharp/csharp-delegates) defines the signature for the event handler method of the subscriber class.

## Declare an Event

## An event can be declared in two steps:

1. Declare a delegate.
2. Declare a variable of the delegate with event keyword.

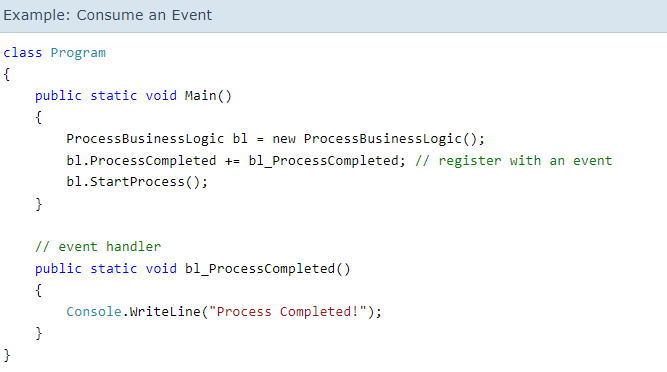


In the above example, we declared a delegate Notify and then declared an event ProcessCompleted of delegate type Notify using "event" keyword in the ProcessBusinessLogic class. Thus, the ProcessBusinessLogic class is called the publisher. The Notify delegate specifies the signature for the ProcessCompleted event handler. It specifies that the event handler method in subscriber class must have a void return type and no parameters.

Above, the StartProcess() method calls the method onProcessCompleted() at the end, which raises an event. Typically, to raise an event, protected and virtual method should be defined with the name On<EventName>. Protected and virtual enable derived classes to override the logic for raising the event. However, A derived class should always call the On<EventName> method of the base class to ensure that registered delegates receive the event.

The OnProcessCompleted() method invokes the delegate using ProcessCompleted?.Invoke();. This will call all the event handler methods registered with the ProcessCompleted event.

The subscriber class must register to ProcessCompleted event and handle it with the method whose signature matches Notify delegate, as shown below.



Above, the Program class is a subscriber of the ProcessCompleted event. It registers with the event using += operator. Remember, this is the same way we add methods in the invocation list of multicast delegate. The bl\_ProcessCompleted() method handles the event because it matches the signature of the Notify delegate.

## Built-in EventHandler Delegate

.NET Framework includes built-in delegate types [EventHandler](https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler" \t "_blank) and [EventHandler<TEventArgs>](https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler-1" \t "_blank) for the most common events. Typically, any event should include two parameters: the source of the event and event data. Use the EventHandler delegate for all events that do not include event data. Use [EventHandler<TEventArgs>](https://docs.microsoft.com/en-us/dotnet/api/system.eventhandler-1" \t "_blank) delegate for events that include data to be sent to handlers.

The example shown above can use EventHandler delegate without declaring a custom Notify delegate, as shown below.

**Example: Event Handler**

class Program

{

public static void Main()

{

ProcessBusinessLogic bl = new ProcessBusinessLogic();

bl.ProcessCompleted += bl\_ProcessCompleted; // register with an event

bl.StartProcess();

}

// event handler

public static void bl\_ProcessCompleted(object sender, EventArgs e)

{

Console.WriteLine("Process Completed!");

}

}

public class ProcessBusinessLogic

{

// declaring an event using built-in EventHandler

public event EventHandler ProcessCompleted;

public void StartProcess()

{

Console.WriteLine("Process Started!");

// some code here..

OnProcessCompleted(EventArgs.Empty); //No event data

}

protected virtual void OnProcessCompleted(EventArgs e)

{

ProcessCompleted?.Invoke(this, e);

}

}

**ASP.NET(I)**

**ASP.NET:** A collection of classes of reusable libraries given by Microsoft to be used in other .NET applications and to develop, build and deploy many types of applications on the Windows platform.

**CLR:**  Works a layer between Operating Systems and the applications written in .NET languages that conforms to the common Language Specification (CLS). The main function of Common Language Runtime (CLR) is to convert the Managed Code into native code and then execute the program.

When a .NET application is executed at that time the control will go to Operating System, then Operating System create a process to load **CLR.**

**CTS:** Common Type System (CTS). When we declare an int type in C# and VB.NET, then they are converted to int32. In other words, now both will have a common data type that provides flexible communication between these two languages.

**CLS:** Common Language Specification.

**Managed Code:** The code, which is developed in .NET framework, is known as managed code. This code is directly executed by CLR with help of managed code execution. Any language that is written in .NET Framework is managed code.