Namespace

The **namespace** keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements

They also provide assistance in avoiding name clashes Namespaces don't correspond to file, directory or assembly name. They could be written in separate files and/or separate assemblies and still belong to the same namespace.

Namespace alias directive. Sometimes you may encounter a long namespace and wish to have it shorter. This could improve readability and still avoid name clashes with similar named methods.

Namespaces are used in C# to keep one set of names separated from another. This is done to organize the classes so that they are easy to handle. If there are two classes with the same names in different namespaces, they do not conflict with one another.

Namespaces are heavily used in C# for two reasons:

- .NET uses namespaces to organize its many classes.
- Control the scope of class and method names in larger programming projects.

A namespace can have following types as its members:

- Namespaces (Nested Namespace)
- Classes
- Interfaces
- Structures
- Delegates

Define Namespace

```
namespace Namespace-Name
{
   //Body of namespace
}
```

Accessing members of namespace

The members of a namespace can be accessed using the dot(.) operator. The syntax for accessing the member of namespace

Syntax : Namespace-Name.Member-Name

Available C# Namespaces

- System (except these specific properties: DateTime.Now, DateTime.UtcNow, DateTime.Today)
- System.Collections
- Systems.Collections.Generic
- System.Linq
- System.Linq.Expressions
- System.Linq.Queryable
- System.Text.RegularExpressions

Nested Namespace

A namespace can contain another namespace it is called as nested namespace.

when two or more types or methods have the same name then it occur ambiguity

Two Methods to avoid ambiguity

- using fully qualified name
- using Alias Directives

Avoid ambiguity using fully qualified names

```
namespace PersonOne
{
    public class PersonDetails
    {
        public void PrintDetails
            ()
        { /* ... */ }
    }
}

namespace PersonTwo
{
    public class PersonDetails
    {
        public void PrintDetails()

        { /* ... */ }
    }
}
```

```
using System;

using PersonOne.PersonDetails;

using PersonTwo.PersonDetails;

class Demo
{
    public static void main()
    {

        //Avoid ambiguity by using fully qualified name

        PersonOne.PersonDetails();

        PersonOne.PersonDetails();
    }
}
```

Avoid ambiguity using alias directives

```
namespace PersonOne
{
    public class PersonDetails()
    {
        public void PrintDetails

        { /* ... */ }
    }
}

namespace PersonTwo
{
    public class PersonDetails
    {
        public void PrintDetails()

        { /* ... */ }
    }
}



using System;

using P1= PersonOne.PersonDetails;

using P2= PersonTwo.PersonDetails;

class Demo
{
    public static void main()
    {

        //Avoid ambiguity by using fully qualified name

        P1.PersonOne.PersonDetails();

        P2.PersonOne.PersonDetails();
    }
}
```

What is Class?

A **class** is used to create **objects** by serving as a blueprint or template, which plays a crucial role in defining the structure, behavior, and attributes.

- A class consists of data and behavior. **Data** is represented by **the class**'s fields and behavior is represented by **its** methods.
- The following access specifiers can be used to declare classes, ensuring that they are not accessible to other classes.
    - Public
    - Private
    - Protected
    - Default

a class is a reference type that defines a type of object. It can be thought of as a blueprint or template that defines the structure, behavior, and attributes of objects. Classes can also be used to create custom types by defining the kinds of information and methods included in a custom type.

A class can be thought of as a user-defined data type that encapsulates both data (attributes) and the actions (methods) that are applied to that data.

a class declaration contains only a keyword **class**, followed by an **identifier(name)** of the class. But there are some optional attributes that can be used with class declaration according to the application requirement.

In general, class declarations can include these components

- Modifiers
- Class keyword

Syntax for Declaring Classes

To declare a class in C#, use the **class** keyword, followed by the class name.

Class names should follow C# naming conventions .

The class definition is enclosed within curly braces {}.

```
public class MyClass
{
    // Fields, properties, and methods.
}
```

Example of class

```
namespace ClassDemo

{
    public class Program
    {
        static void Main(string[] args)
        {
            Vehicle vehicle = new Vehicle();
            vehicle.PrintDetails();
            Console.ReadLine();

        }
    }
    public class Vehicle
    {
        public string VehicleType { get; set; }
        public string VehicleName { get; set; }
        public string Color { get; set; }
        public string FuelType { get; set; }


        public Vehicle()
        {
            VehicleType = "Car";
            VehicleName = "xyz";
            Color = "red";
            FuelType = "petrol";
        }
        public void PrintDetails()
        {
            Console.WriteLine($"Vehicle type is :{VehicleType}");
            Console.WriteLine($"Vehicle Name is :{VehicleName}");
            Console.WriteLine($"Vehicle Color is :{Color}");
            Console.WriteLine($"Vehicle Fuel type : {FuelType}");



        }
    }
}
```

Constructor

A **constructor** is a special method that is automatically called when an instance of a class is created.

Constructors in class are used for initializing new objects. Fields are variables that provide the state of the class and its objects, and methods are used to implement the behavior of the class and its objects.

A constructor is a method which has a same name as it's containing class name.

There can be only one static constructor created per class. There is no return type, not even void, for a constructor. A parameterized constructor can't be a static constructor.

There can be any number of constructors in a class.

To limit access to a constructor, or which other class may call it, access modifiers can be used in the constructor declaration.

- Constructor is used to initialize data when declare fields
- Constructor does not have return type

Types of Constructor

- Default Constructor
- Parameterized Constructor

Default Constructor

Constructor without any parameters is called a default constructor**. i.e** Default type of constructor does not take Any parameters.

Default constructors are constructors that take no parameters. Every instance of the class must be initialized with identical values when using a default constructor. Inside a class, the default constructor sets all of the numeric values to zero and all of the string and object fields to null.

Parameterized Constructor

A parameterized constructor is a constructor that accepts one or more parameters. These parameters allow you to set initial values for object properties when you create a new instance of the class. By using parameterized constructors, you can ensure that objects are always created with valid initial values.

A constructor is considered parameterized if it accepts at least one parameter. Each instance of the class may have a different initialization value.

Example of default constructor

```
namespace Dayonepractice.constructor
{
    public class constructor
    {
        int id;
        string name;
        string city;


        public constructor()
        {
            id = 0;
            name = "test user";
            city = "test city";
       }

     }

}
```

Example of Parameterized Constructor

```
namespace Dayonepractice.constructor
{
    public class constructor
    {

        //parameterized  constructor
        public constructor(int id, string name, string city)
        {
            this.id = id;
            this.name = name;
            this.city = city;
        }
        public void PrintDetails()
        {
            Console.WriteLine($"id is {id}");
            Console.WriteLine($"name is {name}");
            Console.WriteLine($"city is {city}");

        }

     }

}
```

Destructor

Destructors are normally called when the C# garbage collector decides to clean the object from memory

Destructors are the opposite of constructors, which are called when an object is created to allocate memory. Destructors are important for resource management and are different from the GC, which automatically manages memory for managed objects.

The Destructor is called implicitly by the . NET Framework's Garbage collector and therefore programmer has no control as when to invoke the destructor.

Destructor is used to destroy objects of class when the scope of an object ends. It has the same name as the class and starts with a tilde ~

- A **Destructor** is used to release resources that the object has acquired during execution
- A destructor is defined using the tilde (~) followed by the class name.
- Destructor has no return type and has exactly the same name as the class name

When a Destructor called it automatically invokes Finalize Method. Finalize method in C# is used for cleaning memory from unused objects and instances. So, when you call the destructor, it actually translates into following code

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

Example of Destructor

```
namespace Dayonepractice.constructor
{
    public class constructor
    {

        int id;
        string name;
        string city;


        public constructor()
        {
            id = 0;
            name = "test user";
            city = "test city";
        }


        //it is a destructor same name as constructor

        ~constructor() { }
    }
}
```