Unit -3
Creating Types in C#

Course Contents [12 Hrs.]

Classes; Constructors and Destructors; this Reference; Properties; Indexers; Static Constructors and Classes; Finalizers; Dynamic Binding; Operator Overloading; Inheritance; Abstract Classes and Methods; base Keyword; Overloading; Object Type; Structs; Access Modifiers; Interfaces; Enums; Generics

---

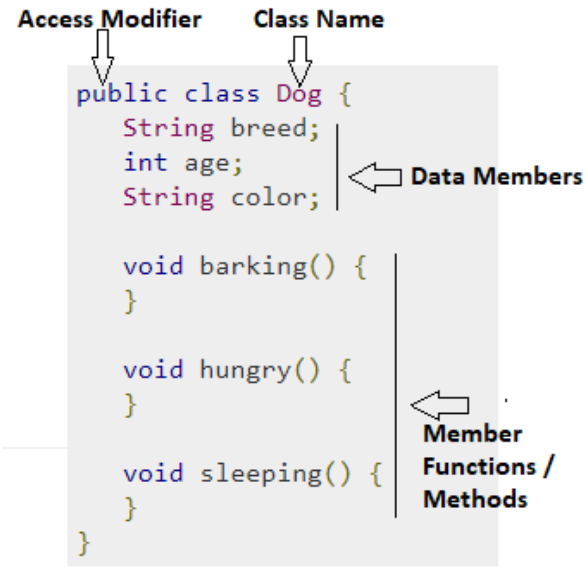
Classes

A class, in the context of C#, are templates that are used to create objects, and to define object data types and methods. Core properties include the data types and methods that may be used by the object. All class objects should have the basic class properties. Classes are categories, and objects are items within each category.

A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

1. **Modifiers**: A class can be public or has default access.
2. **Class name**: The name should begin with a initial letter (capitalized by convention).
3. **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the (:). A class can only extend (subclass) one parent.
4. **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the (:). A class can implement more than one interface.
5. **Body**: The class body surrounded by braces, { }.

General form of a class is shown below:



Object

An object is a combination of data and procedures working on the available data. An object has a state and behavior. The state of an object is stored in fields (variables), while methods (functions) display the object's behavior. Objects are created from templates known as classes. In C#, an object is created using the keyword "new". Object is an instance of a class. There are three steps to creating a C# object:

1. Declaration of the object
2. Instantiation of the object
3. Initialization of the object

When a object is declared, a name is associated with that object. The object is instantiated so that memory space can be allocated. Initialization is the process of assigning a proper initial value to this allocated space. The properties of C# objects include:

- One can only interact with the object through its methods. Hence, internal details are hidden.
- When coding, an existing object may be reused.
- When a program's operation is hindered by a particular object, that object can be easily removed and replaced.

A new object t from the class "tree" is created using the following syntax:

Tree t = new Tree ().

Fields

A field is a variable that is a member of a class or struct. For example:

```
class Octopus {
        string name;
        public int Age = 10;
        }
```

Fields allow the following modifiers:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifier	`new`
Unsafe code modifier	`unsafe`
Read-only modifier	`readonly`
Threading modifier	`volatile`

The read only modifier

The read only modifier prevents a field from being modified after construction. A read-only field can be assigned only in its declaration or within the enclosing type's constructor.

```
public int Age = 10;
static readonly int legs = 8, eyes = 2;
```

Methods

A method performs an action in a series of statements. A method can receive input data from the caller by specifying parameters and output data back to the caller by specifying a return type. A method can specify a void return type, indicating that it doesn't return any value to its caller.
A method can also output data back to the caller via ref/out parameters. A method's signature must be unique within the type. A method's signature comprises its name and parameter types in order (but not the parameter names, nor the return type).

Methods allow the following modifiers:

Static modifier	`static`
Access modifiers	`public internal private protected`
Inheritance modifiers	`new virtual abstract override sealed`
Partial method modifier	`partial`
Unmanaged code modifiers	`unsafe extern`
Asynchronous code modifier	`async`

Expression-bodied methods

A method that comprises a single expression, such as the following:

```
int Foo (int x) { return x * 2; }
```

can be written more tersely as an expression-bodied method. A fat arrow replaces the braces and return keyword:

```
int Foo (int x) => x * 2;
```

Expression-bodied functions can also have a void return type:

```
void Foo (int x) => Console.WriteLine (x);
```

Overloading methods

A type may overload methods (have multiple methods with the same name), as long as the signatures are different. For example, the following methods can all coexist in the same type:

```
void Foo (int x) {...}
void Foo (double x) {...}
void Foo (int x, float y) {...}
void Foo (float x, int y) {...}
```

However, the following pairs of methods cannot coexist in the same type, since the return type and the params modifier are not part of a method's signature:

```
void  Foo (int x) {...}
float Foo (int x) {...}          // Compile-time error

void  Goo (int[] x) {...}
void  Goo (params int[] x) {...}  // Compile-time error
```

Constructors

A constructor in C# is a block of code similar to a method that's called when an instance of an object is created. Here are the key differences between a constructor and a method:

- A constructor doesn't have a return type.
- The name of the constructor must be the same as the name of the class.
- Unlike methods, constructors are not considered members of a class.
- A constructor is called automatically when a new instance of an object is created. All classes have constructors, whether you define one or not, because C# automatically provides a default constructor that initializes all member variables to zero. However, once you define your own constructor, the default constructor is no longer used.

Default Constructor

The default constructor is a constructor that is automatically generated in the absence of explicit constructors (i.e. no user defined constructor). The automatically provided constructor is called sometimes a nullary constructor. Following is the syntax of a default constructor –

```
class ClassName {
   ClassName() {
   }
}
```

Instance Constructors

Constructors run initialization code on a class or struct. A constructor is defined like a method, except that the method name and return type are reduced to the name of the enclosing type:

```
public class Panda
{
  string name;                  // Define field
  public Panda (string n)       // Define constructor
  {
    name = n;                   // Initialization code (set up field)
  }
}
...

Panda p = new Panda ("Petey");   // Call constructor
```

Instance constructors allow the following modifiers:

public internal private protected

Overloaded Constructors

A class or struct may overload constructors. To avoid code duplication, one constructor may call another, using **this** keyword:

```
using System;

public class Wine
{
  public decimal Price;
  public int Year;
  public Wine (decimal price) { Price = price; }
  public Wine (decimal price, int year) : this (price) { Year = year; }
}
```

When one constructor calls another, the called constructor executes first.

Destructor

Destructors in C# are methods inside the class used to destroy instances of that class when they are no longer needed. The Destructor is called implicitly by the .NET Framework's 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.

Important Points:

- A Destructor is unique to its class i.e. there cannot be more than one destructor in a class.
- A Destructor has no return type and has exactly the same name as the class name (Including the same case).
- It is distinguished apart from a constructor because of the Tilde symbol (~) prefixed to its name.
- A Destructor does not accept any parameters and modifiers.
- 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.
- Internally, Destructor called the Finalize method on the base class of object.

Syntax

```
class Example
{
    // Rest of the class
    // members and methods.

   // Destructor
   ~Example()
    {
        // Your code
    }

}
```

Example:

```
class ConsDes
        {
                //constructor
                public ConsDes(string message)
                {
                        Console.WriteLine(message);
                }
                public void test()
```

```
        {
            Console.WriteLine("This is a method");
        }
        //Destructor
        ~ConsDes() {
            Console.WriteLine("This is a destructor");
            Console.ReadKey();
            }
        }
        class Construct
        {
            static void Main(string[] args)
            {
                string msg = "This is a constructor";
                ConsDes obj = new ConsDes(msg);
                obj.test();
            }
}
```

Output:

This is constructor.

This is a method.

This is a destructor.

Static Constructor

In C#, Static Constructor is used to perform a particular action only once throughout the application. If we declare a constructor as static, then it will be invoked only once irrespective of number of class instances and it will be called automatically before the first instance is created. Generally, in C# the static constructor will not accept any access modifiers and parameters. In simple words we can say it's a parameter less.

Following are the properties of static constructor in C# programming language.

- Static constructor in c# won't accept any parameters and access modifiers.
- The static constructor will invoke automatically, whenever we create a first instance of class.

- The static constructor will be invoked by CLR so we don't have a control on static constructor execution order in c#.
- In C#, only one static constructor is allowed to create.

C# Static Constructor Syntax

```
class User
{
    // Static Constructor
    static User() {
    // Your Custom Code
    }
}
```

Example

```
class User {
    // Static Constructor
    static User() {
    Console.WriteLine("I am Static Constructor");
    }
    // Default Constructor
    public User() {
    Console.WriteLine("I am Default Constructor");
    }
}
class Program
    {
        static void Main(string[] args)
        {
        // Both Static and Default constructors will invoke for first instance
        User user = new User();
        // Only Default constructor will invoke
        User user1 = new User();
        Console.WriteLine("\nPress Enter Key to Exit..");
        Console.ReadLine();
        } }
```