Phase 3

PREPARED BY BRIJESH KAMANI

Brijesh Kamani
FULL STACK DEVELOPER TRAINEE | RKIT

Table of Contents

14. Understanding Classes

14.1 Class types and usage

Class:

- *Class is a group of related methods and variables. We can say it is a blueprint of user-defined objects. We can create objects of a class.*
- *A class contains the field and methods of a particular object. We can create any amount of objects of any class.*
- *Classes are the main features of OOP programming languages. Because it gives us the ability to define any type of real-world entities and objects and work on objects.*
- *We can create a class by using class keyword and naming rules for the class identifier is the same as variable naming rules. The class have its body part where each property and method will be created.*

Syntax:

```
[access_specifire] [class_type] <class> <classname>
{
        [properties] …
        [methods] …
}
```

  - *If any variables or objects are created within the class body then it's called properties of that class.*
  - *IF any function is created within the class body then these functions are called methods of that class.*
  - *A type that is defined as a class is a* reference type*. At run time, when you declare a variable of a reference type, the variable contains the value null until you explicitly create an instance of the class by using the new operator, or assign it an object of a compatible type that may have been created elsewhere.*

There are other 4 types of classes in C#.

- *Abstract*
- *Partial*
- *Sealed*
- *Static*

14.2 static,sealed,abstract

Abstract Class:

Syntax:

```
[access_specifire] <abstract> <class> <classname>
{
        [properties]
                …
        [non-abstract method]
                …
        [abstract methods]
                …
}
```

- *An abstract class is used to define common methods for its subclasses. So there is no need to create its class.*
- *A method that does not have any definition is called the Abstract method.*
- *So We cannot create an object of an Abstract class because it may contain the Abstract method.*
- *We can create an abstract class using abstract keywords.*
- *If a class contains at least one abstract method then the whole class become an abstract class so the Class must be created with abstract keyword.*
- *For creating an object of abstract class we must have to inherit this class into subclass and all methods of superclass and subclass must have their definition.*
- *An Abstract can also contain non-abstract methods.*
- *We can also inherit another abstract or normal class into an abstract class.*

Partial Class:

Syntax*:*

```
[access_specifire] <partial> <class> <classname>
{
        [properties]
                …
        [non-abstract method]
                …
        [abstract methods]
                …
}
```

- *We can write some parts of the class in one file and other parts are other files using Partial class.*
- *Later during compile time, all parts combine into a single class.*
- *All parts of a partial class must be within the same assemble and namespace and they must be available at compile time.*
- *Class type and access_specifire of all same partial classes must be the same.*
- *Using partial class multiple developers can work on the same class using different files.*

Sealed class:

Syntax:

```
[access_specifier] <sealed> <class> <class_name>
{
        [properties]
                …
        [methods]
                …
}
```

- *Once a class is declared as Sealed it cannot be inherited. Al its variables, properties and methods will be sealed so no other class can inherit anything from this class. But we can create an instance of this class and access its variables and methods via its object.*
- *The main purpose of the sealed class is to withdraw the inheritance attribute from the user so that they can't attain a class from a sealed class. Sealed classes are used best when you have a class with static members.*
- *For example, a sealed class, DatabaseHelper, can be designed with properties and methods that can service the functionalities of database-related actions, including open- and closed-database connection, fetch and update data, etc. Because it performs crucial functions that should not be tampered with by overriding in its derived classes, it can be designed as a sealed class.*
- *Sealing restricts the benefit of extensibility and prevents customization of library types. Hence, a class has to be sealed after carefully weighing the impact of sealing it. The list of criteria to consider for sealing a class includes:*
    - *The class is static*
    - *The class contains inherited members that represent sensitive information*

- *The class is queried to retrieve its attributes through the reflection method*
- *The class inherits many virtual members that need to be sealed*

Static Class:

Syntax:

```
[access_specifier] <static> <class> <class_name>
{
        [static_properties]
                …
        [static_methods]
                …
}
```

- *We cannot create an instance of a static class that means a static class cannot be instantiated.*
- *A static class is also sealed that's mean we cannot also inherit a static class.*
- *Static classes only have static members, static methods and static constructors.*

15. Depth in Classes

15.1 Object, Properties, Methods, Events etc.

Object:

Syntatx:

```
<new> <constructor([parameters])>;
```

- *Object, in C#, is an instance of a class that is created dynamically. An object consists of instance members whose value makes it unique in a similar set of objects.*
- *When an object is instantiated, it is allocated with a block of memory and configured as per the blueprint provided by the class underlying the object. Objects of value type are stored in a stack, while those of reference type are allocated in the heap.*
- *The unified type system of C# allows objects to be defined. These can be user-defined, reference or value types, but they all inherit directly or indirectly from System. Object. This inheritance is implicit so that the type of the object need not be declared with System. An object is the base class.*
- *Since the execution of C# code is in the managed environment of .NET, wherein the garbage collector provides automatic memory reclamation, it is not necessary or possible to explicitly de-allocate memory that is allocated for objects. Objects of value type are destroyed when they go out of scope, while reference type objects are destroyed in a non-deterministic manner until the last reference to them is removed.*
- *The two operations related to objects created in C# are boxing and unboxing. While boxing implies the conversion of value type to object, unboxing refers to the conversion from an object to a value type. Boxing and unboxing operations need to be used carefully because they can put a drag on performance.*

Properties:

- *Properties are a member of a class. We can use properties for reading and writing private fields of a class.*
- *The property has 2 accessors. Called get and set.*
- *A get accessor returns a property value, and a set accessor assigns a new value. The value keyword represents the value of a property.*
- *Properties can be read-write, read-only, or write-only. The read-write property implements both, a get and a set accessor. A write-only property*

implements a set accessor, but no get accessor. A read-only property implements a get accessor, but no set accessor.

- *Usually, inside a class, we declare a data field as private and will provide a set of public SET and GET methods to access the data fields. This is a good programming practice since the data fields are not directly accessible outside the class. We must use the set/get methods to access the data fields.*

Methods:

- *A Method is a bunch of statements that perform some operation for a specific class/object.*
- *We just have to define a method for the specific type of option in class and later whenever we need to perform that operation we just need to call this method.*

Syntax of creating Method:-

```
[Access_specifire] <Return_type> <method_name> ([parameters , …])
{
        // Lines of code….
}
```

- *Here Access-Specifier represents the access area of Method.*
- *Return_Type represent which data type of value this method will return. If it returns nothing then its data type must be Void.*
- *Rules for providing methods name is same as Variable name.*
- *All parameters must be defined within parentheses. During calling We can pass data to our method using Parameters. Parameters specify how many numbers of data and which type of data method can accept after calling. The order of this data must be the same as already specified in a method declaration.*
- *Body of Method must be within Curly bracket. The body contains all the codes that will execute after the calling method.*

Syntax of Calling a Method:

```
        <Method_name>([Paramete rs, …]);
```

A Method may have 3 types of Arguments:

Value type:

- *Value type parameters are such parameters where they hold the only value of passed variables during Method Calling.*
- *It means it just copies the value of passed variables. Any changes done on parameters doesn't affect to passed variables of the Method Calling.*

Reference type:

- *It means its hold a memory address of the passed variables of Method Calling. It means any changes done on that parameter affect those variables that have been passed during Method Calling.*

Optional type:

- *An optional parameter has a default value as part of its definition. If* ***no*** *argument is sent for that parameter, the default value is used.*

Events:

- *Events enable a class or object to notify other classes or objects when something of interest occurs. The class that sends (or* raises*) the event is called the* publisher *and the classes that receive (or* handle*) the event are called* subscribers*.*
- *Events are user actions such as keypress, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.*
- *Following are the key points about the Event,*
    - *Event Handlers in C# return void and take two parameters.*
    - *The First parameter of Event - Source of Event means publishing object.*
    - *The Second parameter of Event - Object derived from EventArgs.*
    - *The publishers determine when an event is raised and the subscriber determines what action is taken in response.*
    - *An Event can have so many subscribers.*
    - *Events are used for the single user action like button click.*
    - *If an Event has multiple subscribers then event handlers are invoked synchronously.*

16. Scope & Accessibility Modifiers

Scope of variable:

- *Scope of variable define where a particular variable is acceptable is termed as the scope of that variable.*
- *A variable can be defined in a class, method, loop etc. In C/C++, all identifiers are lexically (or statically) scoped, i.e. Scope of a variable can be determined at compile-time and independent of the function call stack. But the C# programs are organized in the form of classes.*

- There are 3 types of Scope in C#.
  - *Class Leval Scope*
  - *Methods Leval Scope*
  - *Block Leval Scope*

Class Leval Scope:

- *Declaring the variables in a class but outside any method can be directly accessed anywhere in the class.*
- *These variables are also termed as the fields or class members.*
- *Class level scoped variables can be accessed by the non-static methods of the class in which it is declared.*
- *The access modifier of class level variables doesn't affect their scope within a class.*
- *Member variables can also be accessed outside the class by using the access modifiers.*

Method Level Scope

- *Variables that are declared inside a method have method-level scope. These are not accessible outside the method.*
- *However, these variables can be accessed by the nested code blocks inside a method.*
- *These variables are termed the local variables.*
- *There will be a compile-time error if these variables are declared twice with the same name in the same scope.*
- *These variables don't exist after the method's execution is over.*

Block Level Scope

- *These variables are generally declared inside the for, while statement etc.*
- *These variables are also termed as the loop variables or statements variable as they have limited their scope up to the body of the statement in which it is declared.*
- *Generally, a loop inside a method has three levels of nested code blocks(i.e. class level, method level, loop level).*
- *The variable which is declared outside the loop is also accessible within the nested loops. It means a class-level variable will be accessible to the methods and all loops. Method level variable will be accessible to loop and method inside that method.*
- *A variable that is declared inside a loop body will not be visible to the outside of loop body*

Access Specifier:

- *Access modifiers and specifiers specify the accessibility of a type and its members.*
- C# has 5 access specifier or access modifier keywords.

Private:

Private limits the accessibility of a member to within the defined type, for example, if a variable or a function is being created in a ClassA and declared as private then another ClassB can't access that.

Public*:*

Public has no limits, any members or types defined as the public can be accessed within the class, assembly even outside the assembly. Most DLLs are known to be produced by public class and members written in a .cs file.

Internal:

Internal plays an important role when you want your class members to be accessible within the assembly. An assembly is the produced .dll or .exe from your .NET Language code (C#). Hence, if you have a C# project that has ClassA, ClassB and ClassC then any internal type and members will become accessible across the classes within the assembly.

Protected:

Protected plays a role only when inheritance is used. In other words, any protected type or member becomes accessible when a child is inherited by the parent. In other cases (when no inheritance) protected members and types are not visible.

Protected internal:

Protected Internal is a combination of protected and internal both. A protected internal will be accessible within the assembly due to its internal flavour and also via inheritance due to its protected flavour.

17. Namespace & .Net Library

- *A Namespace is a collection of different classes which belong to the same type of groups or categories.*
- *Namespace gives us the ability to use different classes which may have the same names but different functionality or classes belonging to different parts of our projects.*
- *It also helps us to organize classes and control their scope. If we didn't have namespaces we'd have to (potentially) change a lot of code any time we added a library, or come up with tedious prefixes to make our function names unique. With namespaces, we can avoid the headache of naming collisions when mixing third-party code with our projects.*
- *In C#, Namespace doesn't have any access modifier because it has public access by default and we cannot change it.*
- *A namespace may also contain other namespaces.*
- *This is syntax of creating namespace.*

```
namespace <namespace_name> {
   // code declarations
}
```

- *We can either include classes of the namespace by using the "using" keyword or we can access all members of the namespace by using the dot(.) operator.*

Example:

```
using <namespace_name>[.][sub-namespace_name]    // by including classes
                                or
<namespace_name>.<member_name>              // by accessing directly
```

The following are some of the standard namespaces in the .NET framework.

- *System: Contain classes that implement basic functionalities like mathematical operations, data conversions etc.*
- **System.IO:** *Contains classes used for file I/O operations.*
- **System.Net***: Contains class wrappers around underlying network protocols.*
- **System.Collections:** *Contains classes that implement collections of objects such as lists, hashtable etc.*
- **System.Data:** *Contains classes that make up ADO.NET data access architecture.*
- **System.Drawing:** *Contains classes that implement GUI functionalities.*

- **System.Threading:** *Contains classes that are used for multithreading programming.*
- **System.Web:** *Classes that implement HTTP protocol to access web pages.*
- **System.Xml:** *Classes that are used for processing XML data.*

18. Creating and adding ref. to assemblies

Assemblies:

An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. It is an Output Unit, that is .exe or .dll file. It is a unit of Deployment and a unit of versioning and also it contains MSIL (Microsoft Intermediate Language) code. Assemblies are self-describing, it contains all the metadata about the modules, types, and other elements in the form of a manifest.

- Assemblies are of two types: Private and Shared Assemblies.
  - **Private Assemblies**

    A private assembly is intended only for one application. The files of that assembly must be placed in the same folder of the application.

  - **Shared Assemblies**

    A shared assembly is to be made into a Shared Assembly, then the naming conventions are very strict since it has to be unique across the entire system

19. Working with collections

- *If We Want to create and manage groups of related objects. There are two ways to group objects: by creating arrays of objects, and by creating collections of objects.*
- *Arrays are most useful for creating and working with a fixed number of strongly typed objects.*
- *Collections provide a more flexible way to work with groups of objects. Unlike arrays, the group of objects you work with can grow and shrink dynamically as the needs of the application change. For some collections, you can assign a key to any object that you put into the collection so that you can quickly retrieve the object by using the key.*
- *A collection is a class, so you must declare an instance of the class before you can add elements to that collection.*
- *If your collection contains elements of only one data type, you can use one of the classes in the System.Collections.Generic namespace. A generic collection enforces type safety so that no other data type can be added to it. When you*

retrieve an element from a generic collection, you do not have to determine its data type or convert it.

Many common collections are provided by. NET. Each type of collection is designed for a specific purpose.

Some of the common collection classes are described here:

- *System.Collections.Generic classes*
- *System.Collections.Concurrent classes*
- *System.Collections classes*

System.Collections.Generic Classes

- *You can create a generic collection by using one of the classes in the System.Collections.Generic namespace. A generic collection is useful when every item in the collection has the same data type. A generic collection enforces strong typing by allowing only the desired data type to be added.*
- *The following table lists some of the frequently used classes of the System.Collections.Generic namespace:*

Class	Description
Dictionary<TKey,TValue>	Represents a collection of key/value pairs that are organized based on the key.
List<T>	Represents a list of objects that can be accessed by index. Provides methods to search, sort, and modify lists.
Queue<T>	Represents a first in, first out (FIFO) collection of objects.
SortedList<TKey,TValue>	Represents a collection of key/value pairs that are sorted by key based on the associated IComparer<T> implementation.
Stack<T>	Represents a last in, first out (LIFO) collection of objects.

System.Collections.Concurrent Classes

- *In .NET Framework 4 and later versions, the collections in the System.Collections.Concurrent namespace provides efficient thread-safe operations for accessing collection items from multiple threads.*

- *The classes in the System.Collections.The concurrent namespace should be used instead of the corresponding types in the System.Collections.Generic and System.Collections namespaces whenever multiple threads are accessing the collection concurrently. For more information, see Thread-Safe-Collections and System.Collections.Concurrent.*
- *Some classes included in the System.Collections.Concurrent namespace are BlockingCollection<T>, ConcurrentDictionary<TKey,TValue>, ConcurrentQueue<T>, and ConcurrentStack<T>.*

System.Collections Classes

- *The classes in the System.Collections namespace do not store elements as specifically typed objects but as objects of type Object.*
- *Whenever possible, you should use the generic collections in the System.Collections.Generic namespace or the System.Collections.Concurrent namespace instead of the legacy types in the System.Collections namespace.*
- *The following table lists some of the frequently used classes in the System.Collections namespace:*

Class	Description
ArrayList	Represents an array of objects whose size is dynamically increased as required.
Hashtable	Represents a collection of key/value pairs that are organized based on the hash code of the key.
Queue	Represents a first-in, first-out (FIFO) collection of objects.
Stack	Represents a last-in, first-out (LIFO) collection of objects.