C#(C-SHARP)

- It is a type-safe object-oriented language.
- It enables developers to build the applications that run on the .NET Framework.
- It relies on the runtime(CLR) to perform automatic memory management.
- Statements in C# execute sequentially
- The C# language is platform-neutral and works with a range of platform-specific compilers and frameworks.

OBJECT ORIENTATION

- C# is a rich implementation of the object-orientation paradigm, which includes encapsulation, inheritance, and polymorphism.
- The features of C# from an object oriented perspective are:

Unified type system:

- The fundamental building block in C# is an encapsulated unit of data and functions called a type.
- C# has a unified type system, where all types ultimately share a common base type.
- This means that all types, whether they represent business objects or are primitive types such as numbers, share the same basic set of functionality.

CONT…

- For example, any type can be converted to a string by calling its ToString method.

Classes and interfaces:

- In a traditional object-oriented paradigm, the only kind of type is a class.
- In C#, there are several other kinds of types, one of which is an interface.
- An interface is like a class except it is only a definition for a type, not an implementation.

CONT…

- It's particularly useful in scenarios where multiple inheritance is required (unlike languages such as C++, C# does not support multiple inheritance of classes).

Properties, methods, and events:

- In the pure object-oriented paradigm, all functions are methods.
- Methods are only one kind of function member, which also includes properties and events.
- Properties are function members that encapsulate a piece of an object's state, such as a button's color or a label's text.
- Events are function members that simplify acting on object state changes.

TYPE SAFETY

- C# is primarily a type-safe language,
- It means that types can interact only through protocols they define, thereby ensuring each type’s internal consistency.
- For instance, C# prevents you from interacting with a string type as though it were an integer type. .
- More specifically, C# supports static typing, meaning that the language enforces type safety at compile time.

MEMORY MANAGEMENT

- C# relies on the runtime to perform automatic memory management.
- The CLR has a garbage collector that executes as part of the program, reclaiming memory for objects that are no longer referenced.
- This frees programmers from explicitly deallocating the memory for an object, eliminating the problem of incorrect pointers encountered in languages such as C++.
- For performance-critical hotspots and interoperability, pointers and explicit memory allocation is permitted in blocks that are marked unsafe

C# SYNTAX

- C# syntax is based on C and C++ syntax.
- C# code is **case-sensitive.**
- C# code is made up of a series of statements, each statement is terminated with a **semicolon.**
- C# is a **block-structured language**, meaning that all statements are part of a block of code.
- These blocks which are delimited with curly brackets(**{}**), may contain any number of statements, or none at all.

IDENTIFIERS

- Identifiers are names that programmers use for their classes, methods, variables, arrays and so on.
- An identifier must be a whole word, essentially made up of Unicode characters starting with a letter or underscore and subsequent characters may be letter, underscore or number.
- C# identifiers are case-sensitive.
- By convention, parameters, local variables, and private fields should be in camel case (e.g., firstName), and all other identifiers should be in Pascal case (e.g., GetFullName).

ACCESS MODIFIERS

- All types and type members have an accessibility level, which controls whether they can be used from other code in assembly or other assemblies.

public:

- ➢ The type or member can be accessed by any other code in the same assembly or another assembly that references it.

private:

- ➢ The type or member can be accessed only by code in the same class or struct.

protected:

- ➢ The type or member can be accessed only by code in the same class, or in a class that is derived from that class.

VARIABLE

- A variable represents a storage location that has a modifiable value.
- To use variables, programmers have to declare them.
- This means that programmers have to assign them a name and a type.
- Once you have declared variables you can use them as storage units for the type of data that you declared them to hold.

Variable declaration:

<access_modifiers> <type> <variable_name>;

Example: int a=2; string myString;

STRING TYPE C#'S

- string type (aliasing the System.String type) represents an immutable (unchangeable) sequence of Unicode characters.
- A string literal is specified inside double quotes:

```
        string a = "Heat";
```

- string is a reference type, rather than a value type.
- To avoid this problem. A verbatim string literal is prefixed with @ and does not support escape sequences.

```
        string a1 = "\\\\server\\fileshare\\helloworld.cs";
OR      string a2 = @"\\server\fileshare\helloworld.cs";
```

String Concatenation

- The + operator concatenates two strings:

  string s = "a" + "b"; Or s += “c”;

- One of the operands may be a non string value, in which case ToString is called on that value.

  example: string s = "a" + 5; // a5

String Interpolation

- A string preceded with the $ character is called an interpolated string.

  int x = 4;

  Console.Write ($"A square has {x} sides"); // A square has 4 sides

- C# will convert the expression to a string by calling its ToString method or equivalent

Null and empty strings

- We can create empty string using the static **string.Empty** field or empty literal.

  Exampe**:** string str=string.Empty;   OR string str=“”;

- The static **string.IsNullOrEmpty** method is a useful shortcut for testing whether a given string is either null or empty.
- **IndexOf** is more powerful to search string. **IndexOfAny** is similar.
- It returns the first position of a given character or substring (or -1 if the substring isn’t found):

```
Console.WriteLine ("abcde".IndexOf ("cd")); // 2
```

- **LastIndexOf** is like IndexOf, but works backward through the string. **LastIndexOfAny** is similar.

Manipulating Strings

- String is immutable, all the methods that “manipulate” a string return a new one, leaving the original untouched (same for when reassign a string variable).
- **Substring** extracts a portion of a string:

```
string left3 = "12345".Substring (0, 3);        // left3 = "123";
string end3 = "12345".Substring (2);            // end3 = "345";
```

- **Trim** function remove whitespace characters (including spaces, tabs, new lines, and Unicode variations of these) from the beginning or end of a string;
- **Replace** replaces all (nonoverlapping) occurrences of a particular character or substring:
- **ToUpper** and **ToLower** return upper and lowercase versions of the input string.
- **Split** divides a string up into pieces: string[] words = "The quick brown fox".Split();

String.Format

- The static **Format** method provides a convenient way to build strings that embed variables.
- **Format** simply calls ToString on them.
- When calling **String.Format**, you provide a composite format string followed by each of the embedded variables.

Eg.

```
string composite = “Your username:{0} and password:{1} ";

string s = string.Format (composite,“user123”,“pass123");
```

Arrays

- An array represents a fixed number of variables (called elements) of a particular type.
- The elements in an array are always stored in a contiguous block of memory, providing highly efficient access.
- An array is denoted with square brackets after the element type.

```
type[ ] arrayName;
```

For example:

```
char[] vowels = new char[5];        // Declare an array of 5 characters
```

- The **Length** property of an array returns the number of elements in the array.

CONTD…

- An array initialization expression to declare and populate an array in a single step:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

OR

```
char[] vowels = {'a','e','i','o','u'};
```

- All arrays inherit from the System.Array class, providing common services for all arrays.
- These members include methods to get and set elements regardless of the array type.

Multidimensional Arrays

- Multidimensional arrays come in two varieties: **rectangular** and **jagged**.
- Rectangular arrays represent an n-dimensional block of memory, and jagged arrays are arrays of arrays.

Rectangular arrays:

- Rectangular arrays are declared using commas to separate each dimension.
- The following declares a rectangular two-dimensional array, where the dimensions are
- 3 by 3:

```
int[,] matrix = new int[3,3];
```

Jagged Arrays

- Jagged arrays are declared using successive square brackets to represent each dimension.
- Example of declaring a jagged two-dimensional array, where the outermost dimension is 3:

```
int[][] matrix = new int[3][];
```

- All array indexing is bounds-checked by the runtime.
- An IndexOutOfRangeException is thrown if we use an invalid index:

```
int[] arr = new int[3];
arr[3] = 2;          // IndexOutOfRangeException thrown
```

CLASS DECLARATION IN C#

- It is the most common kind of reference type.
- It is simply an abstract model used to define a custom data types.
- It is a blue print of object.
- It may contain any combination of encapsulated data (fields or members variables), operations(methods) and accessors to data(properties).
- A class in C# is declared using the keyword **class** and its members are enclosed in parenthesis.

EXAMPLE

```
<access_modifier>  <class> class_name
{
}
public class MyClass
{
            //code
}
```

CONTD..

Fields:

- A field is a variable that is a member of a class and can hold data of the class. For example:

```
class Student
{
        string name;
        int age = 10;
}
```

CONTD..

Properties:

- Provide access to a class attribute (a field). Useful for exposing fields in components. A property is declared like a field, but with a **get/set** block added.

```
public class Student
{
    string firstName;
    public string FirstName
    {
        get{     return firstName; }
        set{     firstName=value; }
    }
}
```

Methods(Functions)

- Methods implement some actions/operations that can be performed by an object on the data.
- It can receive input data from the caller by specifying parameters and output data back to the caller by specifying a return type.
- It can specify a void return type, indicating that it doesn't return any value to its caller.
- A method's signature must be unique within the type.
- A method's signature comprises its name and parameter types (but not the parameter names, nor the return type).

CONTD..

Function declaration:

<accessibilityLevel><returnType><functionName>(paramType paramName);

Implementation:

<accessibilityLevel><returnType><functionName>(paramType paramName, ...)

```
{           Code     }
public class Calculator
{
      public  int Sum(int a, int b)
      {
            return a+b;
      }
}
```

Constructor

- If method or function name same with the class name then it is called constructor. But it has no return type.
- It can initialize data member of new object. Constructor may be with parameter or without parameter. In C#, default constructor is automatically created if there is no any constructor.

```
public class Student
{
    stringfirstName;
    public Student(string fName)
    {
        firstName=fName;
    }
}
```

Object

- It is instance of class which hold values of object.
- The new operator is used to create an object or instantiate an object.
- The "new" operator can invoke the constructor.
- Any object of the reference type is assigned a null value unless it is declared using the new operator.
- The new operator assigns space in the memory to the object only during run time

```
Student objStudent = new Student();
```

objStudent is an object of Student class.

NAMESPACES

- A namespace is a domain within which type names must be unique.
- It is simply a logical collection of related classes in c#.
- Types are typically organized into hierarchical namespaces—both to avoid naming conflicts and to make type names easier to find.
- namespaces are independent of assemblies, which are units of deployment such as an .exe or .dll
- Namespaces also have no impact on member visibility—public, internal, private, and so on
- The **namepace** keyword defines a namespace for types within that block.

CONTD…

- The dots in the namespace indicate a hierarchy of nested namespaces.

```
namespace Outer
{
        namespace Inner
        {
                class Test{             }
        }
}
```

using Directive

- It includes namespace to use classes of that namespace.

```
using Outer.Inner;
```

Inheritance

- A class can inherit from another class to extend or customize the original class.
- Inheriting from a class lets you reuse the functionality in that class instead of building it from scratch.
- A class can inherit from only a single class, but can itself be inherited by many classes, thus forming a class hierarchy.

In this example,

```
public class Asset
{
        public string Name;
}
```

- Next, we define classes called Stock and House, which will inherit from Asset.
- Stock and House get everything an Asset has, plus any additional members that they define:

CONTD…

```
public class Stock :Asset // inherits from Asset
{
        public long SharesOwned;
}
public class House :Asset // inherits from Asset
{
        public decimal Mortgage;
}
```

Here’s how we can use these classes:

```
Stock stock = new Stock { Name="MSFT", SharesOwned=1000 };
Console.WriteLine (stock.Name); // MSFT
Console.WriteLine (stock.SharesOwned); // 1000
```

Polymorphism

- Generally, polymorphism is a combination of two words, poly, and another one is morphs.
- Here poly means “multiple” and morphs means “forms” so polymorphism means many forms.
- Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
- When a message can be processed in different ways is called polymorphism.

Polymorphism Provides Following Features:

- It allows us to invoke methods of derived class through base class reference during runtime.
- It has the ability for classes to provide different implementations of methods that are called through the same name.
- **Polymorphism is of two types:**
  - Compile time polymorphism/Overloading
  - Runtime polymorphism/Overriding

Compile Time Polymorphism

- Compile time polymorphism is method and operators overloading.
- It is also called early binding.
- In overloading method performs the different task at the different input parameters.

Runtime Time Polymorphism

- Runtime time polymorphism is done using inheritance and virtual functions.
- Method overriding is called runtime polymorphism.
- It is also called late binding.
- When overriding a method, you change the behavior of the method for the derived class.
- Overloading a method simply involves having another method with the same prototype.

Abstract Classes

- A class is declared as abstract using the **abstract** keyword.
- We can't create an instance of an abstract class. Instead, only its concrete sub classes can be instantiated.
- The purpose of an abstract class is to provide a blueprint for derived classes.
- An abstract class can implement code with non-Abstract methods.

Abstract Members

- Abstract classes are able to define abstract members.
- Abstract members are like virtual members, except they don’t provide a default implementation.
- Abstract members must be overridden in any non-abstract derived class.
- An abstract method is implicitly a virtual method.

CONTD…

```
public abstract class Animal
 {
          public abstract string Eat();          //abstract method with no implementation
 }


public class Cow :Animal
 {
       public override string Eat() //abstract class method implementation
      {
         return “Cow eats grass";
      }
 }
```

CONTD…

```
class Program

{

        static void Main(string[] args)

        {

          Cow myCow= new Cow(); // Create a Cow object

        Console.Write( myCow. Eat());  // Call the abstract method

       }

}
```

The Base Keyword

- The base keyword is similar to the this keyword.
- It serves two essential purposes:
  - Accessing an overridden function member from the subclass.
  - Calling a base-class constructor.

House uses the base keyword to access Asset’s implementation of Liability:

```
public class House :Asset
{
 public override decimal Liability => base.Liability + Mortgage;
}
```

Constructors And Inheritance

- A subclass must declare its own constructors.
- The base class's constructors are accessible to the derived class, but are never automatically inherited.
- For example, if we define Baseclass and Subclass as follows:

```
public class Baseclass
{             public int x;
             public Baseclass () { }
             public Baseclass (int x) { this.x = x; }
}
public class Subclass : Baseclass { }
             Subclass s = new Subclass (123);     //It is illegal:
```

CONTD…

- Subclass must hence "redefine" any constructors it wants to expose.
- however, it can call any of the base class's constructors with the base keyword:

```
public class Subclass : Baseclass
{
            public Subclass (int x) : base (x) { }
}
```

- The base keyword works rather like the this keyword, except that it calls a constructor in the base class.
- Base-class constructors always execute first; this ensures that base initialization

occurs before specialized initialization.

CONTD…

- **Implicit calling of the parameterless base-class constructor**
- If a constructor in a subclass omits the base keyword, the base type's parameterless constructor is implicitly called:

```
public class BaseClass
{
        public int X;
        public BaseClass() { X = 1; }
}
public class Subclass : BaseClass
{
        public Subclass() { Console.WriteLine (X); } // 1
}
```

Interfaces

- An interface is similar to a class, but it provides a specification rather than an implementation for its members.
- An interface declaration is like a class declaration, but it provides no implementation for its members.
- Interface members are by default **abstract** and **public**.
- A class (or struct) can implement multiple interfaces. In contrast, a class can inherit from only a single class.
- These members will be implemented by the classes and structs that implement the interface.
- An interface can contain only methods, properties, events, and indexers.
- An interface cannot contain a constructor (as it cannot be used to create objects)

CONTD…

```
using System;
public interface IShape
{
    void draw();
}
public class Rectangle : IShape
{
    public void draw()
    {
        Console.WriteLine("drawing rectangle...");
    }
}
```

CONTD…

```
public class Circle : IShape
{
    public void draw()
    {
        Console.WriteLine("drawing circle");
    }
}
```

CONTD…

```
class Program
{
  static void Main(string[] args)
  {
            IShape shape;
            shape = new Rectangle();  // Create a object
            shape. draw();        // drawing rectangle
            shape = new Circle();
            shape.draw();         // drawing circle
  }
}
```

Why And When To Use Interfaces

- To achieve security - hide certain details and only show the important details of an object (interface).
- C# does not support "multiple inheritance" (a class can only inherit from one base class).
- However, it can be achieved with interfaces, because the class can implement multiple interfaces.
- Note: To implement multiple interfaces, separate them with a comma .

```
interface IFirstInterface
{
  void myMethod(); // interface method
}
interface ISecondInterface
{
  void myOtherMethod(); // interface method
}
class DemoClass : IFirstInterface, ISecondInterface // Implement multiple interfaces
{
  public void myMethod()
  {
    Console.WriteLine("Some text..");
  }
  public void myOtherMethod()
  {
    Console.WriteLine("Some other text...");
  }
}
```

CONTD…

```
class Program
{
  static void Main(string[] args)
  {
    DemoClass myObj = new DemoClass();
    myObj.myMethod();
    myObj.myOtherMethod();
  }
}
```

Enums

- An **enum** is a special value type that lets you specify a group of named numeric constants.
- To define an enumeration type, use the **enum** keyword and specify the names of enum members:
- public enum BorderSide { Left, Right, Top, Bottom }

We can use this **enum** type as follows:

```
BorderSide topSide = BorderSide.Top;

bool isTop = (topSide == BorderSide.Top); // true
```

CONTD…

Enum Conversions

- You can convert an enum instance to and from its underlying integral value with an explicit cast:
- int i = (int) BorderSide.Left;
- BorderSide side = (BorderSide) i;

Nested Types

A nested type is declared within the scope of another type.

```
public class TopLevel
{
 public class Nested { } // Nested class
 public enum Color { Red, Blue, Tan } // Nested enum
}
```