Net Centric Computing (dot net)

Old questions Solutions Chapter 1

BSC CSIT

1. Create class to showcase constructor, properties, indexers and encapsulation behavior of object-oriented language. [model question – 8 marks] [long question]

```
using System;

class Student
{
    // Encapsulation: private fields
    private string name;
    private int[] marks = new int[3]; // 3 subjects

    // Constructor
    public Student(string n)
    {
        name = n;
    }

    // Property
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    // Indexer
    public int this[int i]
    {
        get { return marks[i]; }
        set { marks[i] = value; }
    }

    // Display method
    public void Show()
    {
        Console.WriteLine("Name: " + Name);
        Console.WriteLine("Marks: " + marks[0] + " " + marks[1] + " " + marks[2]);
    }
}

class Program
{
    static void Main()
    {
        Student s1 = new Student("Adarsha");

        s1[0] = 85;
        s1[1] = 90;
        s1[2] = 80;
```

```
            s1.Show();
        }
    }
```

2. Differentiate between class, sealed class and interface. [model question -3 marks]

Class	**Sealed Class**	**Interface**
1. Can be inherited by other classes.	1. Cannot be inherited further.	1. Cannot be inherited but can be implemented by classes.
2. Can contain fields, methods, and constructors.	2. Same as class but inheritance is not allowed.	2. Contains only method declarations (no implementation).
3. Supports both abstraction and implementation.	3. Supports implementation only, no extension.	3. Supports 100% abstraction.
4. Declared using the class keyword.	4. Declared using the sealed class keyword.	4. Declared using the interface keyword.
5. Object can be created directly.	5. Object can be created directly but cannot extend it.	5. Object **cannot** be created directly.

3. Explain the process of compiling and executing .NET application. [model question -5 marks]

The compilation and execution of a .NET application happen in **multiple steps** as shown below:

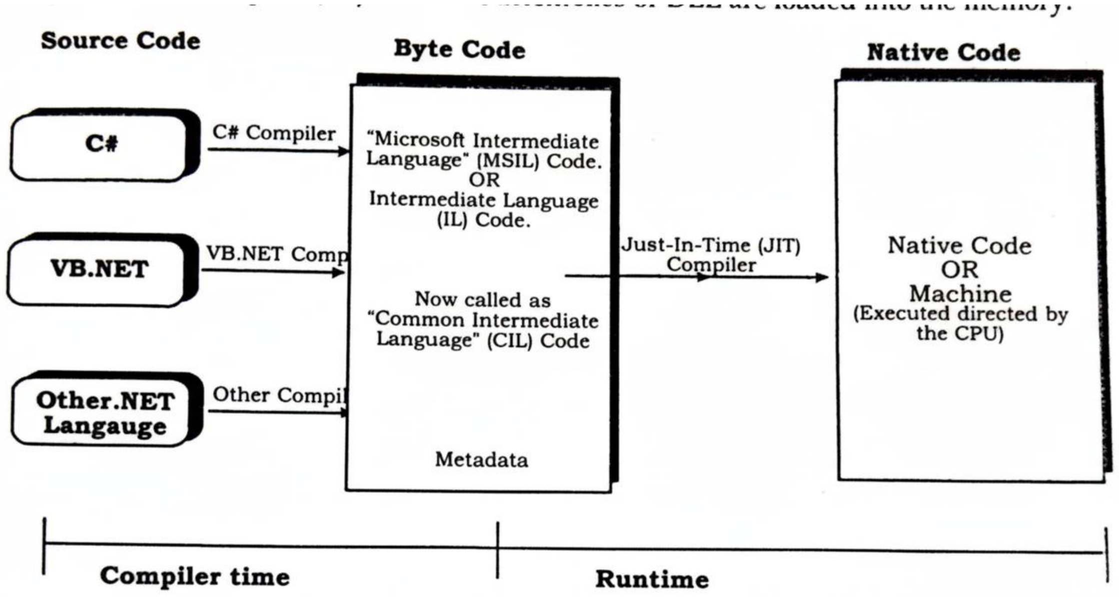

Source Code
Byte Code
Native Code
C#
C# Compiler
VB.NET
VB.NET Comp
Other.NET Langauge
Other Compil
"Microsoft Intermediate Language" (MSIL) Code. OR Intermediate Language (IL) Code.
Now called as "Common Intermediate Language" (CIL) Code
Metadata
Just-In-Time (JIT) Compiler
Native Code OR Machine (Executed directed by the CPU)
Compiler time
Runtime

Figure 3: Compilation and execution of .NET applications

1. Source Code

This is the first step in the compilation and execution of a .NET application. Before compiling and executing, programmers need to write the code for the application. This code can be written in any .NET-supported language, such as C#, VB.NET, or any other. This is known as source code.

2. Compilation by Language Compiler

Now, the source code is written first and then compiled by a compiler. If we have used C#, the C# compiler compiles the code, which is handled by csc.exe. Similarly, for other languages, their corresponding compilers compile the code. The compiled code is then converted into bytecode, also known as MSIL (Microsoft Intermediate Language) or simply Intermediate Language, and stored in an assembly file, such as .exe or .dll.

3. CLR and JIT Compilation

When the program runs (i.e., the .exe or .dll file), the CLR (Common Language Runtime) takes control. At this stage, our code is language-independent because it is in MSIL (Microsoft Intermediate Language) format, which is not tied to any specific programming language. The CLR can handle code written in any .NET-supported language. For the program to actually execute on a computer, the MSIL code must be converted into native machine code. This is done by the JIT (Just-In-Time) compiler, which converts the MSIL code into machine code whenever it is needed.

4. Execution

- The CLR executes the native code.
- During execution, CLR also provides important services like:
    - Memory management
    - Exception handling
    - Garbage collection
    - Security checks

5. Output

- Finally, the program runs and displays the output to the user.

4. Why do we need generics? What are the significances of MSIL? [model question – 5 marks].

Why do we need Generics?

1. **Type Safety:**
   Generics allow us to create classes, methods, and collections that work with any data type while maintaining compile-time type checking.
2. **Code Reusability:**
   We can write one generic class or method and use it for different data types without rewriting code.
3. **Performance:**
   Generics remove the need for boxing and unboxing (in case of value types), which improves performance.
4. **Readability and Maintainability:**
   Generic code is easier to read, manage, and less error-prone.
5. **Built in function:**There are built in function like ADD (), Remove (), Count () in generics which makes very easier to use without writing it from scratch.
6. **Example:** List<int> numbers = new List<int>(); // works safely with integers

Significance of MSIL (Microsoft Intermediate Language)

1. **Platform Independence:**
   MSIL makes .NET programs platform-independent because it runs on any system that has the **CLR**.
2. **Language Interoperability:**
   Programs written in different .NET languages (C#, VB.NET, etc.) are converted to the same MSIL, allowing them to work together.
3. **Security and Verification:**
   Before execution, CLR verifies the MSIL code to ensure it is safe and type-secure.
4. **Optimization:**
   MSIL is converted into optimized **native code** by the **JIT compiler** at runtime.
5. **Portability:**
   The same MSIL file (`.exe` or `.dll`) can run on different hardware and OS environments using .NET runtime.

5. Differentiate between struct and enum. Why do we need to handle the exception? Illustrate with an example with your own customized exception. [2078 -10 marks] [long question].

Difference between `struct` and `enum`

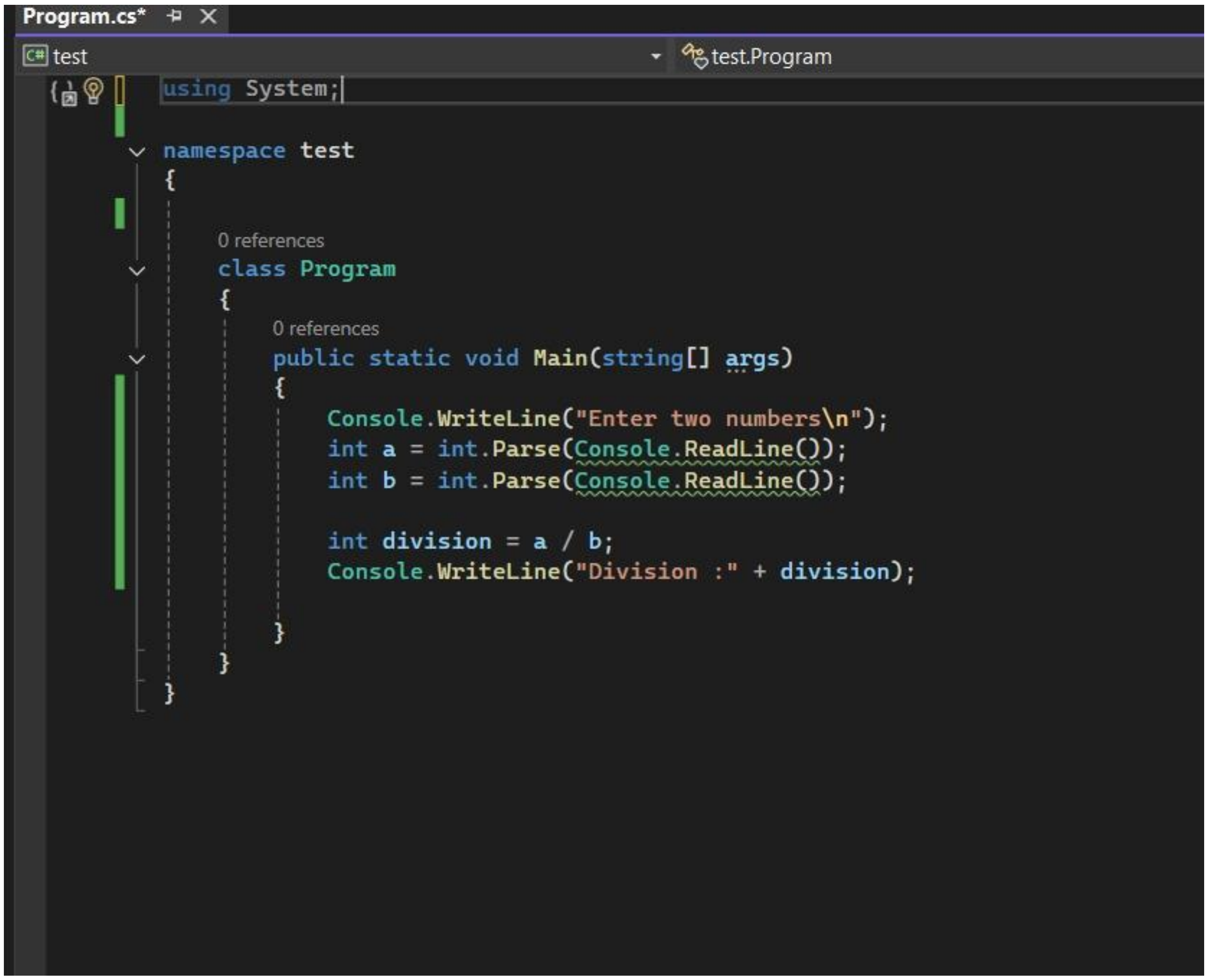
Struct	Enum
Used to define a collection of variables of different data types under one name.	Used to define a set of named constants (fixed values).
Declared using the struct keyword.	Declared using the enum keyword.

Struct	Enum
Can hold multiple data members (fields, methods, constructors).	Represents only symbolic names for constant values.
Can store data of different types.	Stores only integer constant values by default.
Example:	Example:

`struct Student`
`{`
`public int id;`
`public string name;`
`}`
	`enum Days`
	` {`
	`Sun,`
	` Mon,`
	` Tue,`
	` Wed,`
	` Thu,`
	` Fri,`
	` Sat`
	`}`

Why do we need to handle exceptions?

Lets see one example to see why we need to handle exceptions



```
using System;

namespace test
{
    class Program
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Enter two numbers\n");
            int a = int.Parse(Console.ReadLine());
            int b = int.Parse(Console.ReadLine());

            int division = a / b;
            Console.WriteLine("Division :" + division);

        }
    }
}
```

In this example there are no any errors but the error might be occurred during runtimes. Clearly we can see that we will be taking two integer inputs to perform division but instead of integer if user provide string or any other datatype then? Of course, our programs terminate also if the value of b becomes zero we will get an error. So to avoid this type error happens with user inputs and logic we need to handle exceptions.

Example: Custom Exception

```
using System;

// Custom exception class
class InvalidAgeException : Exception
{
    public InvalidAgeException(string message) : base(message)
    {
    }
}

class Program
{
    static void CheckAge(int age)
    {
        if (age < 0 || age > 150)
            throw new InvalidAgeException("Age must be between 0 and 150.");
        else
            Console.WriteLine("Valid Age: " + age);
    }

    static void Main()
    {
        try
        {
            CheckAge(180);   // invalid age
        }
        catch (InvalidAgeException e)
        {
            Console.WriteLine("Custom Exception Caught: " + e.Message);
        }
        finally
        {
            Console.WriteLine("Program ended successfully.");
        }
    }
}
```

Explanation:

- `InvalidAgeException` → user-defined exception class derived from `Exception`.
- `throw` → used to raise an exception.
- `try-catch` → used to handle exception.
- `finally` → executes always, even if an exception occurs.

6. What is named and positional attribute parameters? Describe the .NET architecture design and principles. [2078 - 5 marks]

Named and Positional Attribute Parameters:

Attributes provide metadata about program elements (classes, methods, properties, etc.). If we want to give some information for classes methods or properties we can use attributes. Obsolete is one of the example of built in attributes or standard attributes which is described below . Atrributes can be either standard or our own custom attributes.

They can take parameters in two ways: positional and named.

Positional Parameters

Positional parameters are the compulsory parameter used in the attributes . It should be in the same order like it is defined in attribute constructor.

[Obsolete("This method is obsolete")]

class Test { }

`"This method is obsolete"` is a **positional parameter**.

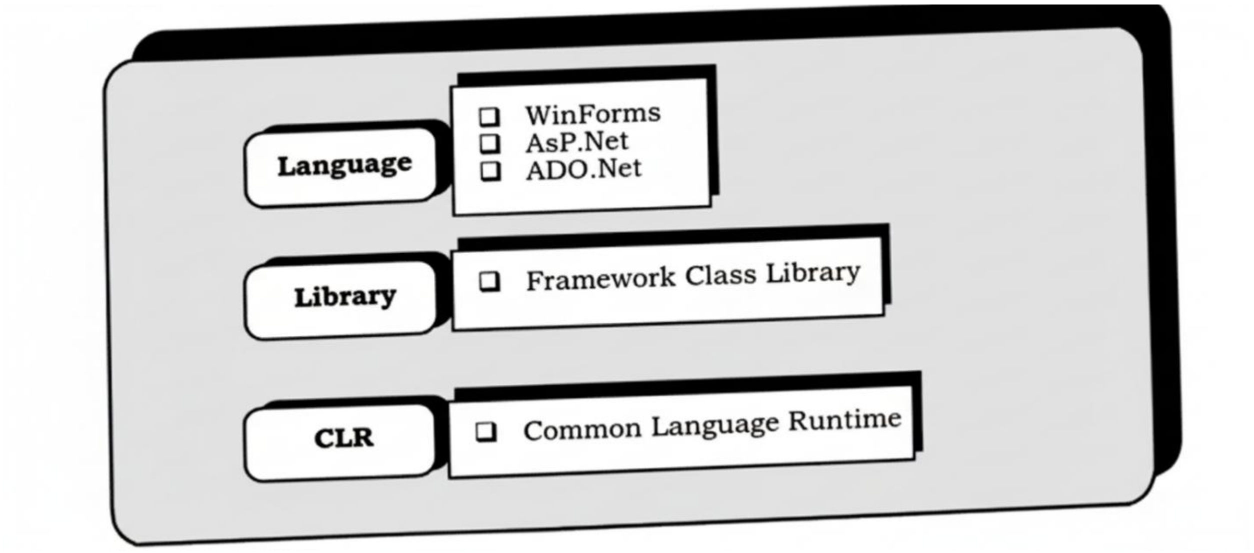
Named Parameters

Named parameter is an optional parameter and it is passed using the property name of the attribute.

```
[Obsolete("Old method", IsError = true)]

class Test { }
```

`IsError = true` is a **named parameter**.

.NET Architecture Design and Principles



A. Design

1. **Common Language Runtime (CLR)**

   When the program runs (i.e., the .exe or .dll file), the CLR (Common Language Runtime) takes control. At this stage, our code is language-independent because it is in MSIL (Microsoft Intermediate Language) format, which is not tied to any specific programming language. The CLR can handle code written in any .NET-supported language. For the program to actually execute on a computer, the MSIL code must be converted into native machine code. This is done by the JIT (Just-In-Time) compiler, which converts the MSIL code into machine code whenever it is needed.

- The CLR executes the native code.
- During execution, CLR also provides important services like:

- Memory management
- Exception handling
- Garbage collection
- Security checks

2. **Base Class Library (BCL)**

   Base class Library provides the built in classes methods and properties that we can use in our code. There are a lot of built in classes related to file handling exception handling and for other operations that we can use directly by importing without rewriting from scratch.

   For example using System.Collections we can use buit in class ArrayList.

3. **Application**

   There are different kind of applications we can create using dot net framework . For that there are built in frameworks and languages supports. Winforms can be used to create desktop apps, Asp.net is used to create beautiful and attractive websites or webapps,and Ado.net is used to create the application involving database and crud operations.

B. Principles

1. **Interoperability**
   Interoperability in .NET means that code written in one version of .NET can run on newer versions without any problem. For example, if you write a program in .NET 5, the same code can run on a newer .NET version. This backward compatibility is supported by the .NET platform itself.

2. **Portability**
   Portability in .NET means that the same program can run on different operating systems or platforms without needing major changes. For example, a .NET application written on Windows can also run on Linux or macOS. This is possible because .NET uses MSIL code and the CLR, which handle platform-specific details.

3. **Memory management**
   Memory management in .NET is handled by the CLR (Common Language Runtime). The CLR keeps track of resources that are no longer used by the running program and frees them for reuse. This is done by a program called the Garbage Collector (GC), which runs at intervals to manage memory efficiently by cleaning up unused resources.

4. **Simplified deployment**
   Simplified deployment in .NET means that applications are easy to install and run. .NET reduces dependency issues because most required libraries come with the framework or are packaged with the application. This makes deploying .NET programs faster and less complicated.

5. **Security**
   Security in .NET ensures that applications are safe from unauthorized access or harmful code. The CLR enforces security by checking permissions, validating code, and restricting unsafe operations, so programs run in a controlled and protected environment.

7. LINQ (short notes). [2078 -2.5 marks]

LINQ stands for Language Integrated Query. It allows you to write queries directly in C# (or other .NET languages) to work with data, similar to how you use SQL for databases. Using LINQ, you can perform operations like retrieving, creating, updating, and deleting data. It provides query keywords like from, select, where, order by, and group by, which make querying easier and more readable. LINQ works not only with databases but also with collections, XML, objects, and other data sources, making data handling simpler and type-safe within your code.

Example:

```
int[] numbers = { 1, 2, 3, 4, 5 };

var evenNumbers =     from n in numbers

                       where n % 2 == 0

                      select n;


foreach(var num in evenNumbers)

   Console.WriteLine(num);
```

- Output: 2 4

8. Write a program to demonstrate the concept of collection and generics. [2079 -8 marks] [Long question]

Program to demonstrate collections:

```
using System;
using System.Collections; // Required for ArrayList
```

```
class Program
{
    static void Main()
    {
        // Creating a collection using ArrayList
        ArrayList students = new ArrayList();

        // Adding elements of different types
        students.Add("Adarsha");
        students.Add("Rimal");
        students.Add(101); // Roll number

        Console.WriteLine("Students ArrayList:");

        foreach (var item in students)
        {
            Console.WriteLine(item);
        }

        // Removing an element
        students.Remove(101);

        Console.WriteLine("\nAfter removing roll number 101:");
        foreach (var item in students)
        {
            Console.WriteLine(item);
        }
    }
}
```

In the above program, we have used ArrayList, which is a collection that can store objects of different types. ArrayList is non-generic, meaning it does not enforce type safety, and we can add integers, strings, or any object. It provides methods like Add() and Remove() to manipulate the collection.

Program to demonstrate the Genrics :

```
using System;
using System.Collections.Generic; // Required for List<T>

class Program
{
    static void Main()
    {
        // Creating a generic collection using List<int>
        List<int> marks = new List<int>();

        // Adding elements
        marks.Add(85);
        marks.Add(90);
        marks.Add(78);

        Console.WriteLine("Marks List:");

        foreach (int m in marks)
        {
            Console.WriteLine(m);
        }

        // Removing an element
        marks.Remove(90);

        Console.WriteLine("\nAfter removing 90:");
        foreach (int m in marks)
        {
            Console.WriteLine(m);
        }
    }
}
```

Here, we have used List<T>, which is a generic collection. Generics allow us to store only a specific data type (here int), ensuring type safety at compile time. It also avoids boxing/unboxing overhead and makes the code cleaner. Methods like Add(), Remove(), and accessing elements by index are simple and efficient.

9. Lambda expression (short notes) [2079 -2.5 marks]

A lambda expression in C# is an anonymous function that can be used to define inline methods without creating a separate named method. It is mainly used with delegates and LINQ to make code shorter and more readable. The lambda operator => separates the input parameters from the body of the function. Lambda expressions are useful for simple operations and make code more concise and easy to understand.

Example:

```
using System;

class Program
{
    static void Main()
    {
        // Lambda expression to calculate square
        Func<int, int> square = x => x * x;
        Console.WriteLine(square(5)); // Output: 25
    }
}
```

Here, x => x * x is a lambda expression that takes an integer x and returns its square. The delegate Func<int, int> is used to store the lambda expression. This avoids creating a separate method for a simple operation.

10. Delegate and events (short notes) [2079 -2.5 marks]

A **delegate** in C# is a **type-safe pointer to a method**, meaning it can refer to any method with a **matching signature**. Delegates are used to **pass methods as arguments**, enabling flexible and reusable code.

An **event** is a **special kind of delegate** that allows a class to notify other classes or objects when something happens. Events are widely used in **GUI programming and real-time applications** to handle actions like button clicks or data updates.

Example:

```
using System;

// Delegate declaration
delegate void Notify(string message);

class Program
{
    // Event using the delegate
    static event Notify OnNotify;

    static void Main()
    {
        // Subscribe method to event
        OnNotify += ShowMessage;

        // Trigger the event
        OnNotify("Hello! Event triggered.");
    }

    static void ShowMessage(string msg)
    {
        Console.WriteLine(msg);
    }
}
```

Here, Notify is a delegate that can point to any method taking a string parameter. OnNotify is an event based on the delegate. When we trigger OnNotify, it calls the subscribed method ShowMessage. This shows how delegates and events work together to implement callback functionality in C#.

11. What are the needs for partial class and sealed class? How do you relate delegate with events? [2080 -5 marks] [Long question]

A partial class allows the definition of a class to be split into multiple files, which is helpful when working in large projects or with auto-generated code. For example, in Windows Forms or ASP.NET, the designer generates part of the class automatically, and the programmer can write another part without affecting the generated code. This improves code organization and maintainability.

Example (Partial Class):

```
// File1.cs
partial class Student
{
    public string Name;
}

// File2.cs
partial class Student
{
    public void Display()
    {
        Console.WriteLine("Name: " + Name);
    }
}

// Usage
class Program
{
    public static void Main(string[] args)
    {
        Student s = new Student();
        s.Name = "Adarsha";
        s.Display();
    }
}
```

A sealed class is a class that cannot be inherited, which is useful to prevent modification or extension of a class. It ensures that the class behavior remains unchanged and is often used for security or design reasons.

Example (Sealed Class):

```
sealed class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }
}

// class AdvancedCalculator : Calculator {} // Not allowed
```

Relationship between Delegate and Events :

A delegate is a type-safe reference to a method, while an event is based on a delegate and allows a class to notify other classes when something happens. In other words, events use delegates internally to maintain a list of subscriber methods. Delegates provide the mechanism, and events provide a controlled way of triggering methods externally.

Example:

```
delegate void Notify(string msg);
class Program
{
    static event Notify OnNotify;  // Event

    static void Main()
    {
        OnNotify += ShowMessage;   // Delegate subscribes method
        OnNotify("Hello! Event triggered.");
    }

    static void ShowMessage(string message)
    {
        Console.WriteLine(message);
    }
}
```
Here, the delegate Notify defines the method signature, and the event OnNotify uses the delegate to call the subscribed method ShowMessage when triggered. This shows how delegates and events are closely related.

12. Differentiate between generic and non-generic collections. Write a simple program to create generic class with generic constructor, generic member variable, generic property and generic method. [2080 -5 marks].

Generic Collection	Non-Generic Collection
1. Stores a specific data type (type-safe).	1. Stores objects of any type (not type-safe).
2. Checked at compile-time, reducing errors.	2. Errors may occur at runtime due to type casting.
3. No boxing/unboxing for value types, better performance.	3. Boxing/unboxing required for value types, slower performance.
4. Examples: List<T>, Dictionary<TKey,TValue>, Stack<T>.	Examples: ArrayList, Hashtable, Stack.
5. Cleaner and more maintainable code.	Requires casting, code may be messy and error-prone.

Program:

```
using System;

// Generic class
class MyGenericClass<T>
{
    // Generic member variable
    private T data;

    // Generic constructor
    public MyGenericClass(T value)
    {
        data = value;
    }

    // Generic property
    public T Data
    {
        get { return data; }
        set { data = value; }
    }

    // Generic method
    public void Show<U>(U extra)
    {
        Console.WriteLine("Data: " + data);
        Console.WriteLine("Extra: " + extra);
    }
}

class Program
{
    static void Main()
    {
        // Creating generic object with int
        MyGenericClass<int> obj1 = new MyGenericClass<int>(100);
        obj1.Show<string>("Hello");

        // Creating generic object with string
        MyGenericClass<string> obj2 = new MyGenericClass<string>("Adarsha");
        obj2.Show<int>(25);
    }
}
```

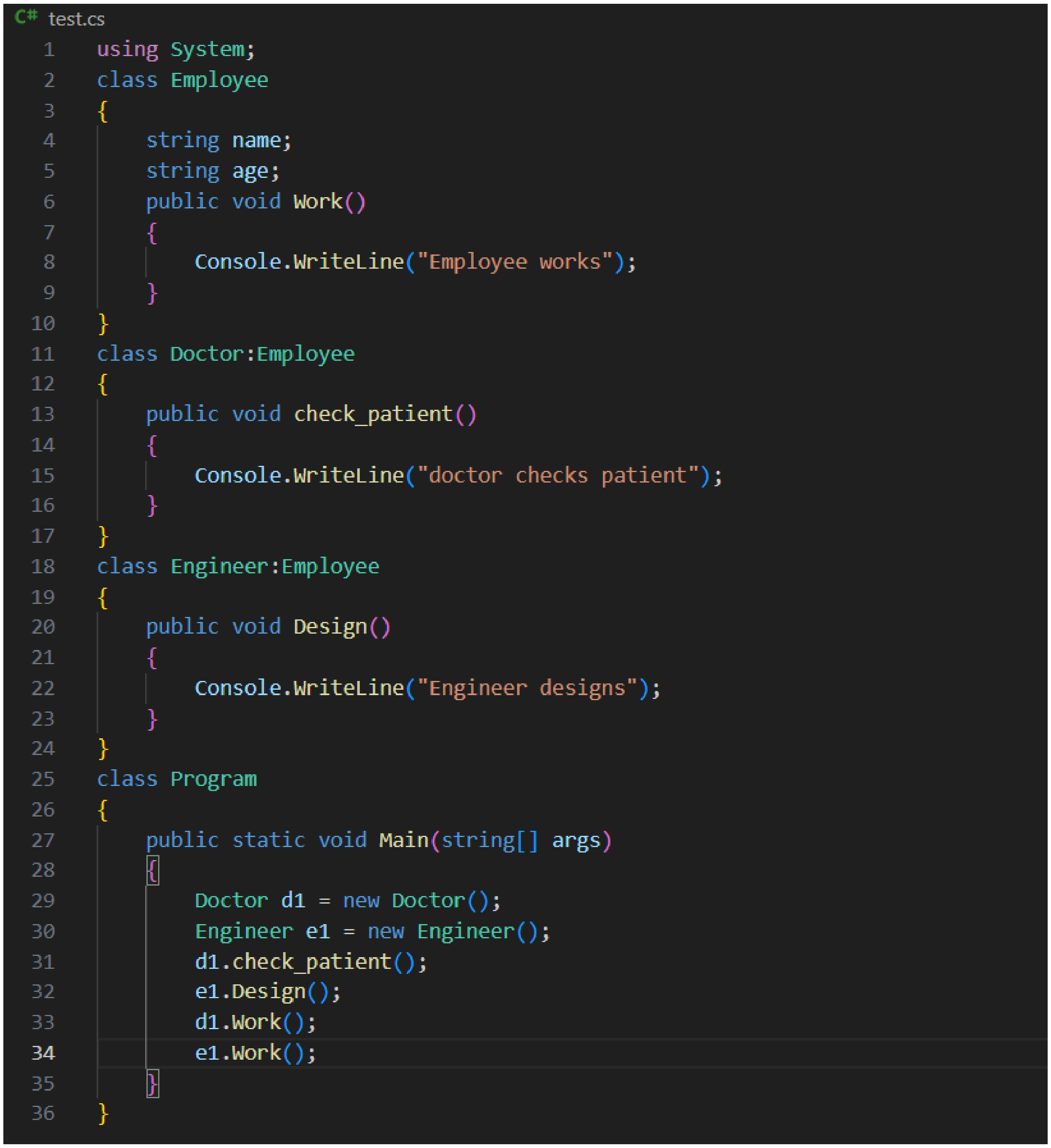
14. Distinguish between collections and generics. What are named and positional attribute parameters? Write a program to create your own exception when the user gives subject name other than "C#". [2081 (2077 batch) -10 marks] [Lonq question].

See above for first and second part:

Program:

```
using System;

// Custom Exception Class
class InvalidSubjectException : Exception
{
    public InvalidSubjectException(string message) : base(message)
    {
    }
}

class Program
{
    static void CheckSubject(string subject)
    {
        if (subject != "C#")
        {
            throw new InvalidSubjectException("Invalid subject! Only 'C#' is
allowed.");
        }
        else
        {
            Console.WriteLine("Valid subject: " + subject);
        }
    }

    static void Main()
    {
        Console.Write("Enter subject name: ");
        string input = Console.ReadLine();

        try
        {
            CheckSubject(input);
        }
        catch (InvalidSubjectException ex)
        {
            Console.WriteLine("Custom Exception: " + ex.Message);
        }
        finally
        {
            Console.WriteLine("Program ended.");
        }
    }
}
```

15. Create a class named EMPLOYEE as super class and ENGINEER and DOCTOR as sub class. Make your own assumptions as properties and methods. [2081 (2077 batch) -5 marks].



test.cs

```csharp
using System;
class Employee
{
    string name;
    string age;
    public void Work()
    {
        Console.WriteLine("Employee works");
    }
}
class Doctor:Employee
{
    public void check_patient()
    {
        Console.WriteLine("doctor checks patient");
    }
}
class Engineer:Employee
{
    public void Design()
    {
        Console.WriteLine("Engineer designs");
    }
}
class Program
{
    public static void Main(string[] args)
    {
        Doctor d1 = new Doctor();
        Engineer e1 = new Engineer();
        d1.check_patient();
        e1.Design();
        d1.Work();
        e1.Work();
    }
}
```

16. Polymorphism (short notes). [2081 (2077 batch) -2.5 marks]

Polymorphism in C# means “many forms” and allows a single entity (like a method or object) to behave differently in different situations. It is a core concept of object-oriented programming. There are two types of polymorphism: compile-time (method overloading) and run-time (method overriding). Compile-time polymorphism occurs when multiple methods have the same name but different parameters, while run-time polymorphism occurs when a subclass provides a specific implementation of a method defined in its superclass. Polymorphism improves code flexibility, reusability, and maintainability.

Example (Run-time Polymorphism):

```
using System;

class Employee
{
    public virtual void Work()
    {
        Console.WriteLine("Employee is working.");
    }
}

class Engineer : Employee
{
    public override void Work()
    {
        Console.WriteLine("Engineer is designing.");
    }
}

class Program
{
    static void Main()
    {
        Employee emp = new Engineer();
        emp.Work();  // Output: Engineer is designing.
    }
}
```

Here, Work() in Employee is overridden by Engineer. When we call emp.Work(), it executes the subclass version, demonstrating run-time polymorphism.

17. How do you compile and execute .NET application? [2081 (78 batch) -3 marks)] [Lonq question].

See above

18. Show the chain of constructor in inheritance with an example. [2081 (78 batch) -5 marks)].

Constructor chaining occurs when a subclass constructor calls the constructor of its superclass to initialize inherited members. In C#, this is done using the base keyword. Constructor chaining ensures that the base class is initialized before the derived class, maintaining proper object initialization. This is commonly used in inheritance to avoid code duplication and to initialize common attributes in the base class.

```
using System;

// Grandfather class
class Grandfather
{
    public Grandfather()
    {
        Console.WriteLine("Grandfather constructor called");
    }
}

// Father class
class Father : Grandfather
{
    public Father()
    {
        Console.WriteLine("Father constructor called");
    }
}

// Son class
class Son : Father
{
    public Son()
    {
        Console.WriteLine("Son constructor called");
    }
}

class Program
{
    static void Main()
    {
        Son s = new Son();
    }
}
```

Output:

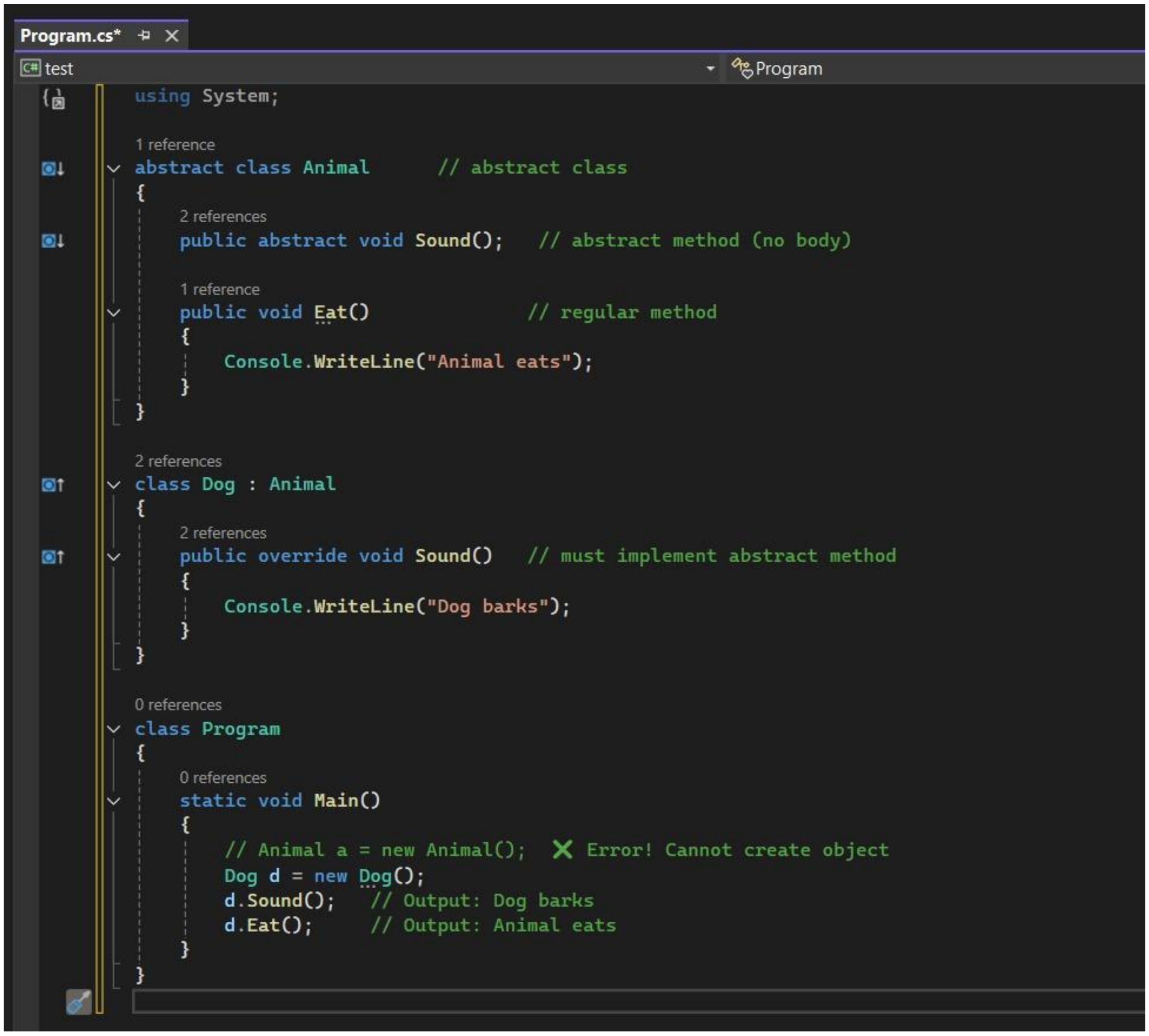
```
Grandfather constructor called
Father constructor called
Son constructor called
```

When we create a Son object, the constructors are called in hierarchical order: Grandfather → Father → Son, showing constructor chaining in inheritance.

19. Explain the needs of abstract class, sealed class and partial class. [2081 (78 batch) -4 marks)].

An abstract class is used when we want to define a base class that cannot be instantiated directly but provides a common structure for derived classes. It can contain both abstract methods (without body) and regular methods, allowing derived classes to implement their own version of the abstract methods. This is useful for code reusability and enforcing a standard structure in inheritance hierarchies.

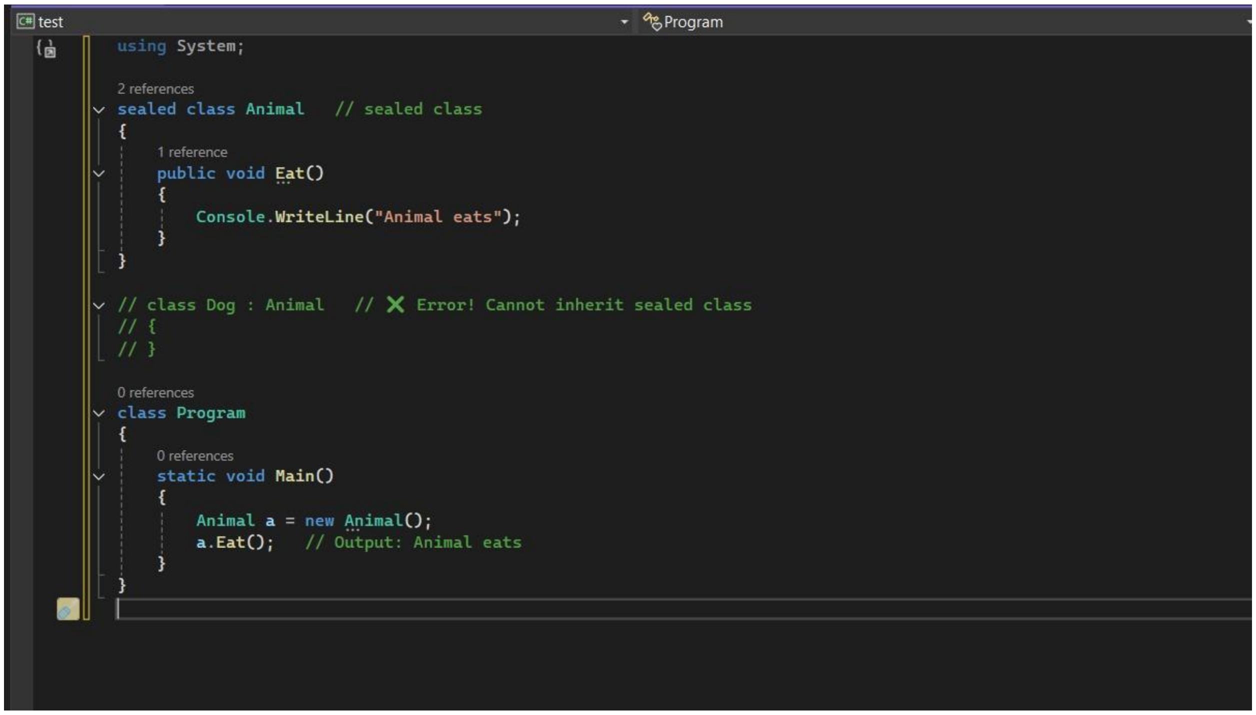
Example:



```
using System;

abstract class Animal      // abstract class
{
    public abstract void Sound();   // abstract method (no body)

    public void Eat()              // regular method
    {
        Console.WriteLine("Animal eats");
    }
}

class Dog : Animal
{
    public override void Sound()   // must implement abstract method
    {
        Console.WriteLine("Dog barks");
    }
}

class Program
{
    static void Main()
    {
        // Animal a = new Animal();  ✗ Error! Cannot create object
        Dog d = new Dog();
        d.Sound();    // Output: Dog barks
        d.Eat();      // Output: Animal eats
    }
}
```

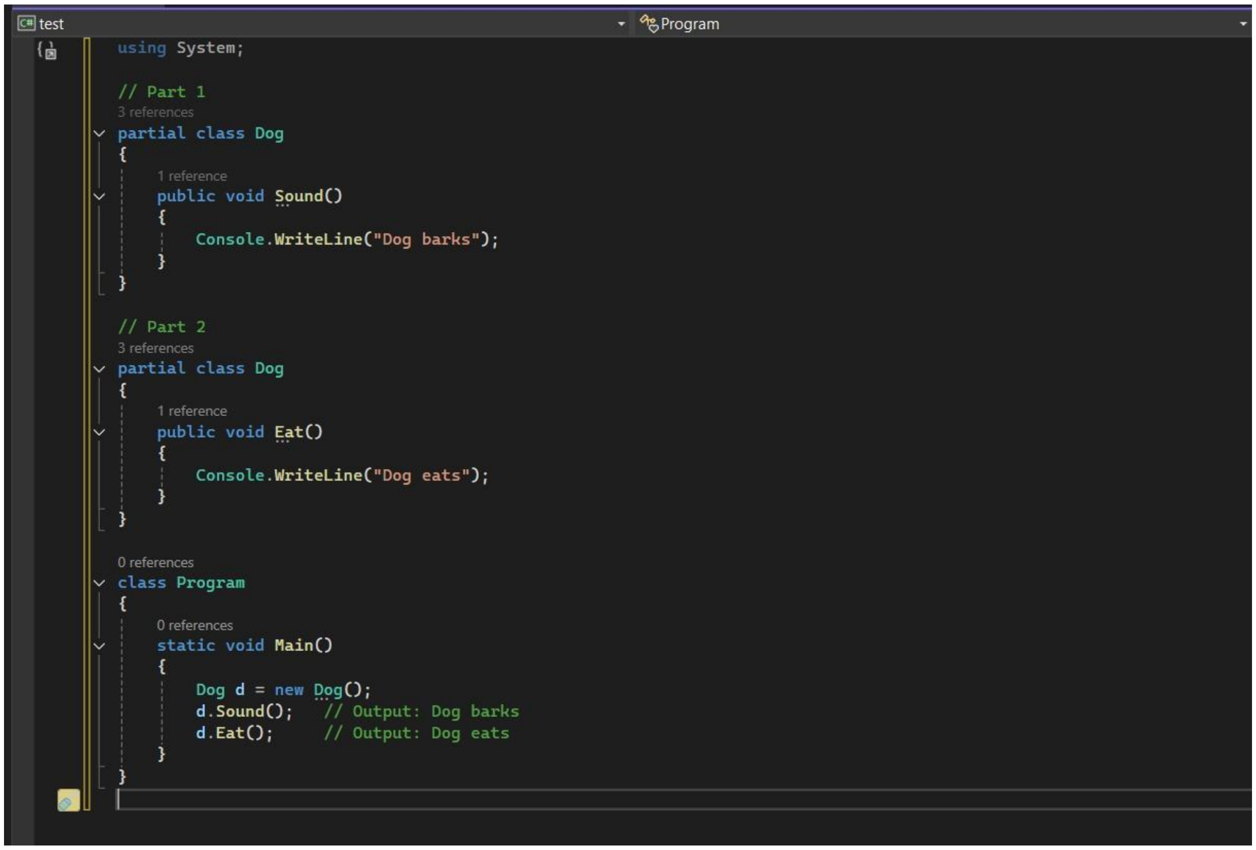
A sealed class is used when we want to prevent a class from being inherited. This ensures that the behavior of the class cannot be modified, which is important for security, stability, or design reasons. Sealed classes are commonly used when the class provides complete functionality and should not be extended.

Example:



```
using System;

2 references
sealed class Animal   // sealed class
{
    1 reference
    public void Eat()
    {
        Console.WriteLine("Animal eats");
    }
}

// class Dog : Animal   // ❌ Error! Cannot inherit sealed class
// {
// }

0 references
class Program
{
    0 references
    static void Main()
    {
        Animal a = new Animal();
        a.Eat();   // Output: Animal eats
    }
}
```

Using a partial class, we can write the code of a single class in parts, either within different files or in the same file at different places. All parts must belong to the same namespace and use the partial keyword. This helps in organizing large projects, allows multiple programmers to work on the same class simultaneously, and makes auto-generated code (like Windows Forms or ASP.NET) easier to manage. At compile time, all parts are combined into a single class by the compiler.



```
using System;

// Part 1
3 references
partial class Dog
{
    1 reference
    public void Sound()
    {
        Console.WriteLine("Dog barks");
    }
}

// Part 2
3 references
partial class Dog
{
    1 reference
    public void Eat()
    {
        Console.WriteLine("Dog eats");
    }
}

0 references
class Program
{
    0 references
    static void Main()
    {
        Dog d = new Dog();
        d.Sound();    // Output: Dog barks
        d.Eat();      // Output: Dog eats
    }
}
```

20. Write a program in C# to find the positive numbers from a list of numbers using multiple where conditions in LINQ Query.[2081 (78 batch) -5 marks].

```
using System;
using System.Linq;  // Required for LINQ
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        // List of numbers
        List<int> numbers = new List<int> { -5, 10, 0, 25, -3, 8, -1 };

        // LINQ query with multiple where conditions
        var positiveNumbers = from n in numbers
                              where n > 0
                              where n % 2 == 0   // example: also select even
positive numbers
                              select n;

        Console.WriteLine("Positive even numbers in the list:");
        foreach (var num in positiveNumbers)
        {
            Console.WriteLine(num);
        }
    }
}
```

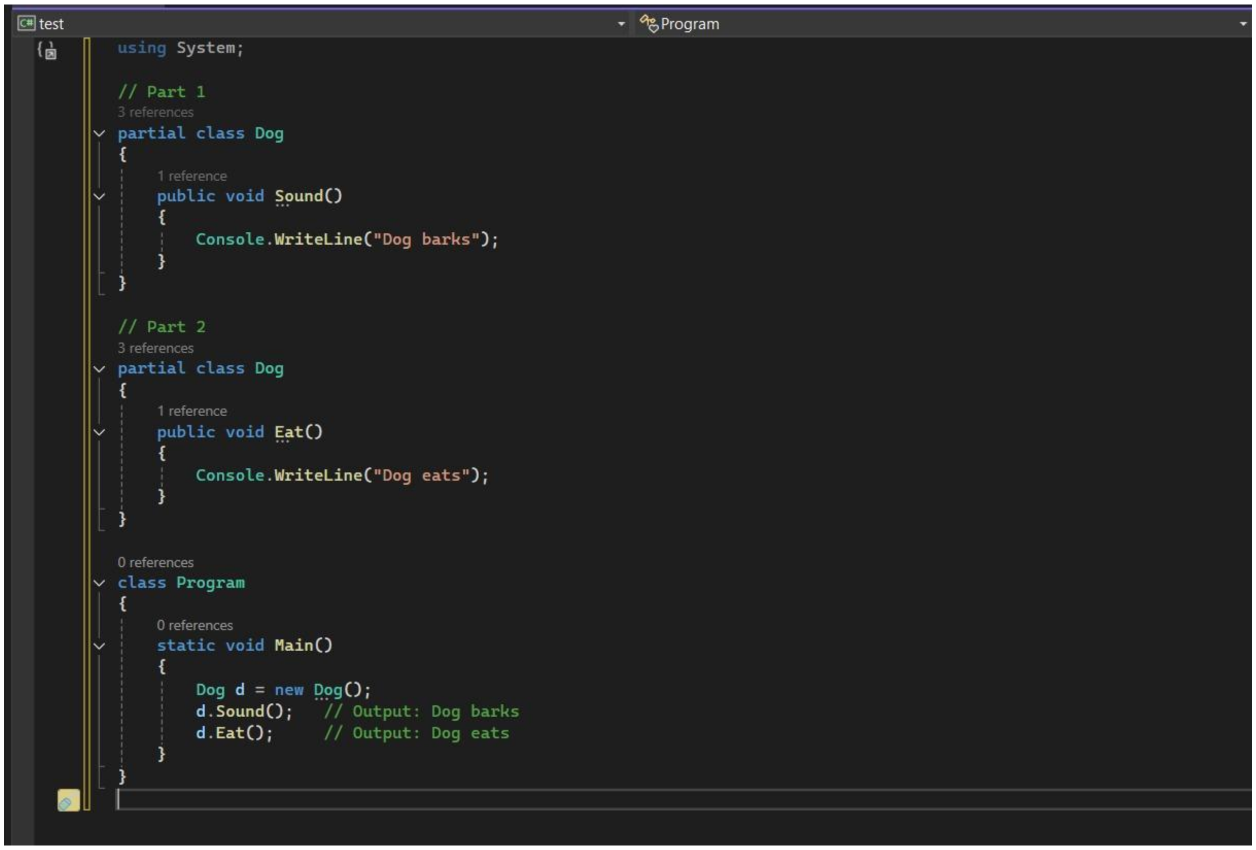
BIT

1. Describe the roles of delegates in event handling. Illustrate the use of partial class with an example. [model question -10 marks] [Lonq question].

Answer: see above solution for first part

Partial class:

Using a partial class, we can write the code of a single class in parts, either within different files or in the same file at different places. All parts must belong to the same namespace and use the partial keyword. This helps in organizing large projects, allows multiple programmers to work on the same class simultaneously, and makes auto-generated code (like Windows Forms or ASP.NET) easier to manage. At compile time, all parts are combined into a single class by the compiler.



```
using System;

// Part 1
3 references
partial class Dog
{
    1 reference
    public void Sound()
    {
        Console.WriteLine("Dog barks");
    }
}

// Part 2
3 references
partial class Dog
{
    1 reference
    public void Eat()
    {
        Console.WriteLine("Dog eats");
    }
}

0 references
class Program
{
    0 references
    static void Main()
    {
        Dog d = new Dog();
        d.Sound();   // Output: Dog barks
        d.Eat();     // Output: Dog eats
    }
}
```

This C# program demonstrates the concept of a **partial class**, which allows a single class to be split into multiple parts, either within the same file or across different files. In this example, the class Dog is divided into two parts using the partial keyword. The first part defines the method Sound() that prints "Dog barks", while the second part defines the method Eat() that prints "Dog eats". When the program is compiled, the C# compiler combines both parts into a single

Dog class containing both methods. In the Main() method of the Program class, an object d of type Dog is created, and both d.Sound() and d.Eat() methods are called, producing the output “Dog barks” and “Dog eats”. This example clearly shows how partial classes help organize large classes into smaller, more manageable sections without affecting functionality.

2. What is lambda expression? Describe the needs of asynchronous programming. [model question -5 marks]

Lambda expression: see above solution

Need of asynchronous programming:

Asynchronous programming in C# allows a program to perform multiple tasks at the same time without blocking the main thread. It is mainly used when one operation (like file access, API calls, or downloading data) takes a long time to complete. Instead of waiting for that operation to finish, the program continues running other tasks.

It uses async and await keywords, which make the code non-blocking, faster, and more responsive — especially useful in desktop apps, web servers, and network programming.

Need for Asynchronous Programming:

1. Non-blocking execution:
   The main thread (UI or main program) continues working while waiting for time-consuming tasks to complete.
2. Improved performance:
   It helps make programs faster by using system resources efficiently.
3. Better user experience:
   In UI applications, asynchronous code prevents the screen from freezing while background work (like downloading files) is in progress.
4. Efficient resource use:
   The CPU and memory can handle other tasks while waiting for I/O operations.

Example:

```
using System;
using System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        Console.WriteLine("Downloading started...");
        await DownloadFileAsync();  // Runs asynchronously
        Console.WriteLine("Download completed!");
    }

    static async Task DownloadFileAsync()
    {
        await Task.Delay(3000); // Simulates time-consuming task
        Console.WriteLine("File downloaded.");
    }
}
```

Here, the Main() method calls the DownloadFileAsync() method using the await keyword. The program does not freeze during the 3-second delay; it continues running smoothly. This demonstrates how asynchronous programming helps perform time-consuming tasks without blocking the main program.

3. Differentiate between collection and generics. [2080 -5 marks].

Answer: see above solution

4. Explain about MSIL and CLR. [2080 -5 marks].

Answer: see csit question no: 3 and here is the answer for MSIL add this before CLR IN CSIT question no 3

MSIL:

MSIL (Microsoft Intermediate Language) is an intermediate, platform-independent code generated by the C# compiler after compiling the source code. Instead of being converted directly into machine code, the program is first converted into MSIL, which contains CPU-independent instructions for tasks such as memory management, object creation, and exception handling. This MSIL code is then stored in an assembly (.exe or .dll file) and later converted into native machine code by the Common Language Runtime (CLR) using the Just-In-Time (JIT) compiler during program execution. The most important feature of MSIL is that it provides **language interoperability**, meaning programs written in different .NET languages such as **C#, VB.NET, or F#** can easily work together and share code, since they all compile down to the same intermediate MSIL before execution. This makes .NET a language-independent platform and allows developers to build mixed-language applications seamlessly.

5. Why do we need abstract class? Explain with an example. [2080 -5 marks].

Answer: see above solution from CSIT

6. Define indexers. What is the use of base keyword? How can you apply polymorphism in code extensibility? [2081 set 1 – 10 marks] [Lonq question].

Indexers:

Indexers in C# allow objects to be **indexed like arrays**.They enable you to access class objects using **index notation** instead of calling methods.Indexers are defined using the **this keyword** and help in **storing and retrieving values** inside classes or structures just like arrays.

Example:

```
class Sample
{
    private int[] data = new int[5];
    public int this[int index]
    {
        get
        {
            return data[index];
        }
        set
        {
            data[index] = value;
        }
    }
}

class Program
{
    static void Main()
    {
        Sample s = new Sample();
        s[0] = 10;    // Using indexer
        Console.WriteLine(s[0]);
    }
}
```

Use of base Keyword:

The **base** keyword is used in **inheritance** to **access members of the parent (base) class** from the child (derived) class.It is mainly used to call **base class constructors** or **access overridden methods or properties**.

```
class Parent
{
    public Parent() { Console.WriteLine("Parent constructor"); }
    public void Show() { Console.WriteLine("Hello from Parent"); }
}

class Child : Parent
{
    public Child() : base()   // Calls Parent constructor
    {
        Console.WriteLine("Child constructor");
    }

    public void Display()
    {
        base.Show();  // Access parent method
    }
}
```

Applying Polymorphism in Code Extensibility

Polymorphism means “one name, many forms.”It allows the same method name to perform **different tasks** based on context.It improves **code reusability** and **extensibility** — new classes can be added easily without changing existing code.

Polymorphism is of **two types**:

(a) Static Polymorphism (Compile-time)

It is achieved through **Method Overloading**, where multiple methods have the **same name but different parameters**.

Example:

```
class Calculator
{
    public void Add(int a, int b)
    {
        Console.WriteLine("Sum: " + (a + b));
    }

    public void Add(double a, double b)
    {
        Console.WriteLine("Sum: " + (a + b));
    }
}
```

The method to be called is decided **at compile-time**.

(b) Dynamic Polymorphism (Run-time)

It is achieved through **Method Overriding**, using **virtual** and **override** keywords.

Example:

```
class Animal
{
    public virtual void Speak()
    {
        Console.WriteLine("Animal speaks");
    }
}
```

```
class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Dog barks");
    }
}

class Program
{
    static void Main()
    {
        Animal a = new Dog();  // Base class reference, child object
        a.Speak();             // Calls Dog's method at runtime
    }
}
```

The method to be called is decided **at runtime**.

7. Differentiate between error and exception. Write a program to take the input for any five subjects and throw the exception if the given marks is negative or exceed 100. [2081 set 2 – 10 marks] [Lonq question].

Error	Exception
Errors are serious problems that occur during program execution or system failure.	Exceptions occur due to invalid user input or logical mistakes in code.
Errors are usually non-recoverable and cause program termination.	Exceptions are recoverable using `try`, `catch`, and `throw`.
Caused by system-level issues like memory overflow or hardware failure.	Caused by runtime conditions like divide by zero or invalid input.
Cannot be handled by the programmer.	Can be handled by the programmer using exception handling.
Examples: StackOverflowError, OutOfMemoryError	Examples: DivideByZeroException, IndexOutOfRangeException

Program:

```
using System;

class Program
{
    static void Main()
    {
        int[] marks = new int[5];

        try
        {
            for (int i = 0; i < 5; i++)
            {
                Console.Write("Enter marks of subject " + (i + 1) + ": ");
                marks[i] = int.Parse(Console.ReadLine());

                if (marks[i] < 0 || marks[i] > 100)
                {
                    throw new Exception("Marks must be between 0 and 100!");
                }
            }

            Console.WriteLine("\nAll marks are valid.");
            Console.WriteLine("Entered Marks:");
            foreach (int m in marks)
                Console.WriteLine(m);
        }
        catch (Exception e)
        {
            Console.WriteLine("Error: " + e.Message);
        }
    }
}
```

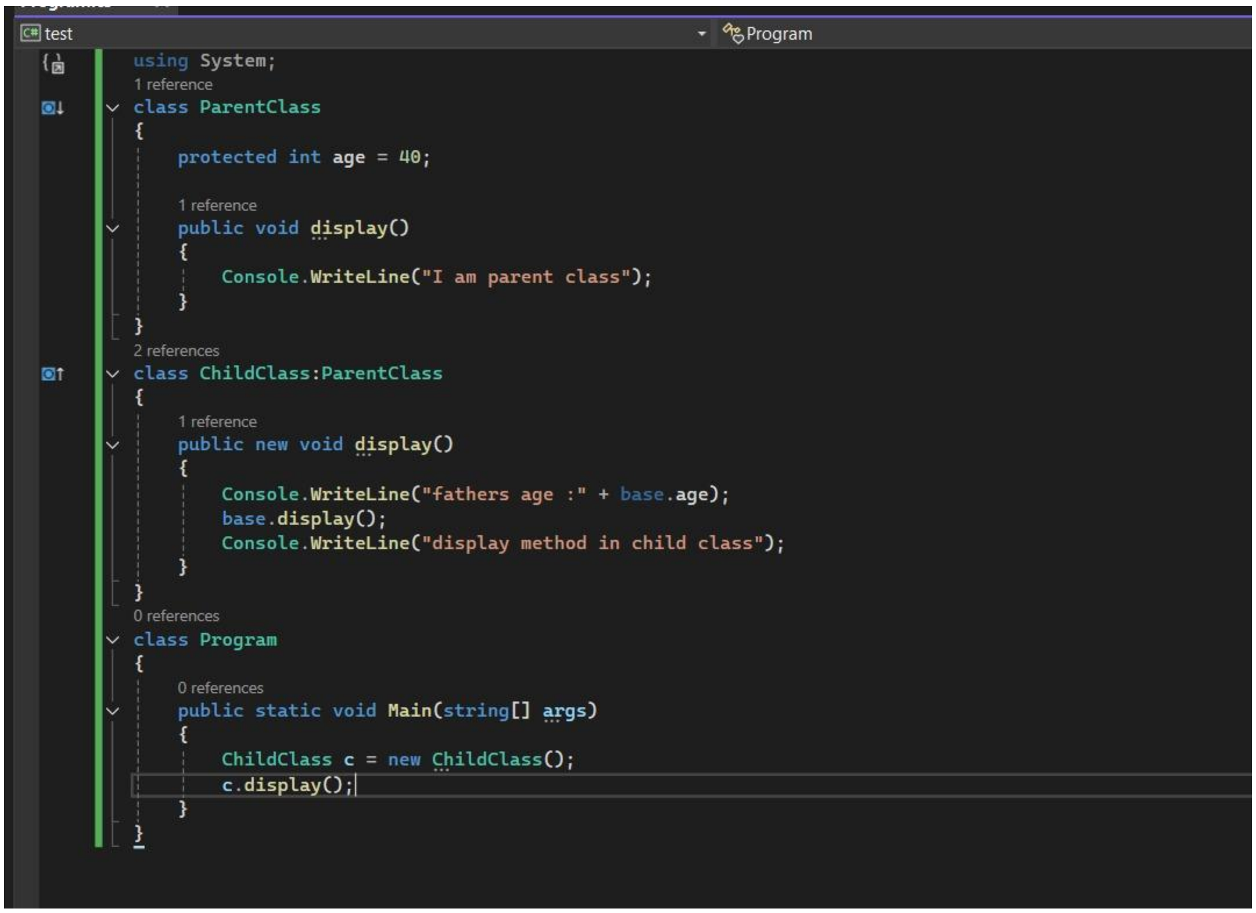
8. Explain about MSIL and CLR. [2081 set 2 – 5 marks]

Answer: see above solution

9. What is the use of base keyword? Give an example. [2081 set 2 -3marks]

The base keyword is used to access members of a base class from a derived class. It has three main uses:

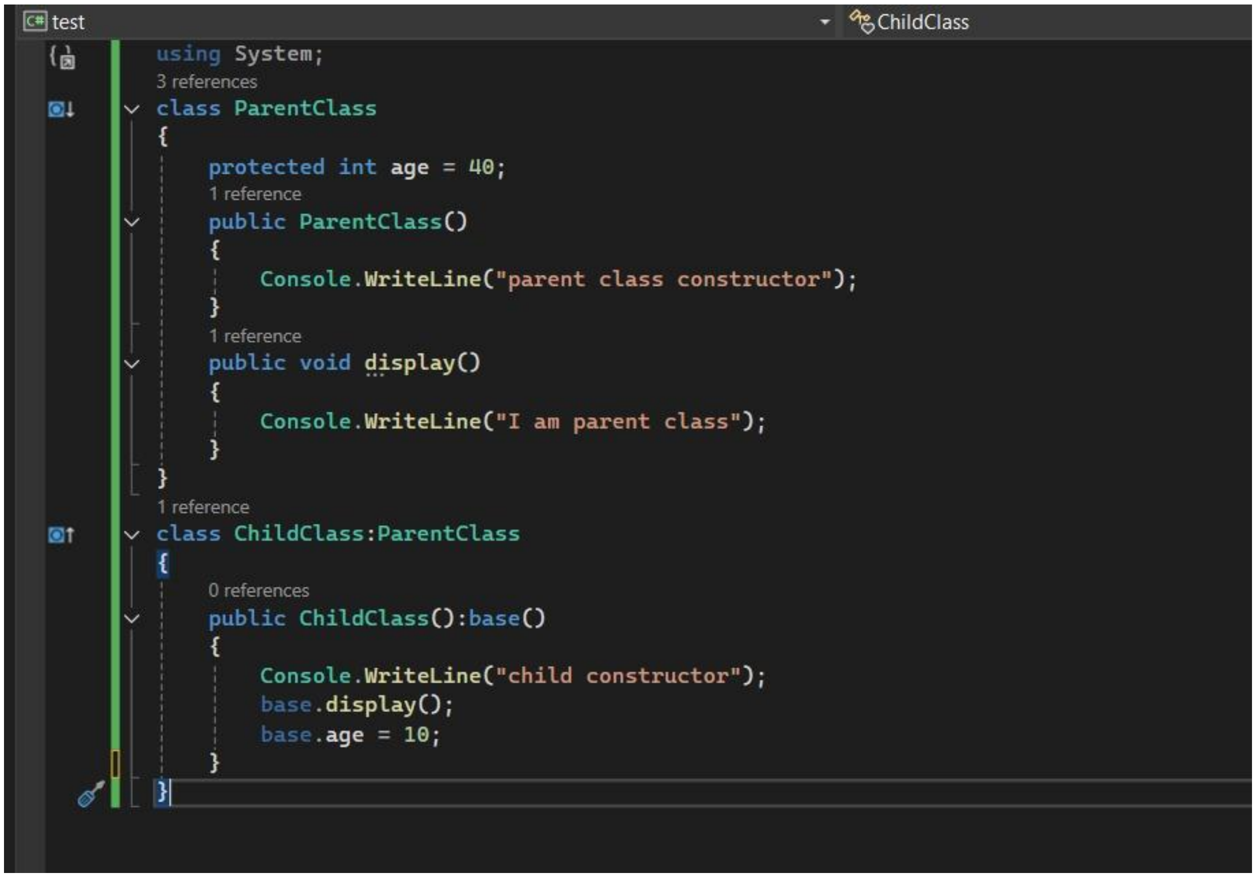
1. **Access base class methods:** We can call a method of the base class from the derived class using base.MethodName().
2. **Access base class properties:** We can access or set properties of the base class using base.PropertyName.



test Program

```
using System;
1 reference
class ParentClass
{
    protected int age = 40;

    1 reference
    public void display()
    {
        Console.WriteLine("I am parent class");
    }
}
2 references
class ChildClass:ParentClass
{
    1 reference
    public new void display()
    {
        Console.WriteLine("fathers age :" + base.age);
        base.display();
        Console.WriteLine("display method in child class");
    }
}
0 references
class Program
{
    0 references
    public static void Main(string[] args)
    {
        ChildClass c = new ChildClass();
        c.display();
    }
}
```

3. **Call base class constructor:** We can call a constructor of the base class from the derived class using base() in the derived class constructor.



```
using System;
3 references
class ParentClass
{
    protected int age = 40;
    1 reference
    public ParentClass()
    {
        Console.WriteLine("parent class constructor");
    }
    1 reference
    public void display()
    {
        Console.WriteLine("I am parent class");
    }
}
1 reference
class ChildClass:ParentClass
{
    0 references
    public ChildClass():base()
    {
        Console.WriteLine("child constructor");
        base.display();
        base.age = 10;
    }
}
```

This helps in reusing and extending functionality of the base class in the derived class.