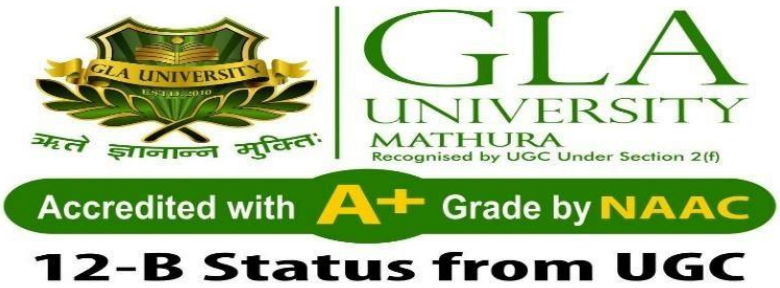
MASTER OF COMPUTER APPLICATION

Department of Computer Engineering & Applications



Session- 2025 – 26

Practical File

Subject Name & Code: **.NET Framework using C# (MCAE 0402)**
.
Course: MCA
Semester: 3rd
Section: A
Class Roll No: 35
University Roll No: 2484200144

Submitted To

Dr. Sachendra Singh Chauhan
(Assistant Professor)

Submitted By

Priyanshi Agrawal

Question 1:

Imagine you are explaining the .NET Framework architecture to a colleague unfamiliar with the framework. How would you break down the architecture and its components, such as the CLR, FCL, and the application domains? Provide a structured explanation.

Answer:

The **.NET Framework architecture** is a powerful software platform developed by Microsoft. It provides a consistent environment for developing and running applications written in different programming languages such as C#, VB.NET, or F#.

It mainly consists of three core components:

1. Common Language Runtime (CLR):

- The **CLR** is like the *engine* of the .NET Framework.
- It provides crucial services such as:
  - **Memory management** (allocating and freeing memory automatically) ○ **Garbage collection** ○ **Exception handling** ○ **Security management**
- It also converts the **Intermediate Language (IL)** code (produced by the compiler) into **native machine code** using the *Just-In-Time (JIT)* compiler.

Think of CLR as the heart of .NET — it runs your program efficiently and safely.

---

2. Framework Class Library (FCL):

- The **FCL** is a large collection of pre-built, reusable classes and functions.
- It contains classes for: ○ File handling (System.IO) ○ Database operations (System.Data) ○ Collections (System.Collections) ○ Networking (System.Net)
  - User interfaces (System.Windows.Forms)

It saves time — you don't have to write common functionalities from scratch.

---

3. Application Domains:

- An **Application Domain (AppDomain)** is an isolated environment where .NET applications run.
- It provides **isolation** between applications.
- If one application crashes, others are not affected.
- CLR manages these application domains internally.

Overall Architecture Flow:

C# Code → Compiler → Intermediate Language (IL) → CLR (JIT) → Executes using FCL

Question 2:

In a team meeting, you are asked to explain key .NET Framework runtime concepts like the Common Language Runtime (CLR), Common Type System (CTS), and Common Language Specification (CLS). How would you present these to ensure clarity and relevance to the team's work?

Answer:

These three concepts — **CLR, CTS, and CLS** — form the foundation of how .NET runs code consistently across multiple languages.

1. Common Language Runtime (CLR):

- Acts as the **runtime environment** of .NET.
- Manages the execution of all .NET programs.
- Responsibilities include:
  - Memory management
  - Type safety ○ Security ○ Garbage collection ○ Thread management

Example: When you run a C# program, CLR takes care of converting IL code to native code and running it on the system.

2. Common Type System (CTS):

- Defines how data types are declared and used in the .NET environment.
- Ensures all .NET languages (C#, VB.NET, F#) use the same type definitions. Example:
- int in C# and Integer in VB.NET are treated as the same type (System.Int32) because of CTS.
- This allows seamless data sharing between different .NET languages.

3. Common Language Specification (CLS):

- A **set of rules** that all .NET languages must follow for interoperability.
- Ensures that code written in one .NET language can be used in another.

Example:

If your C# code uses an unsigned integer (uint), it might not be CLS-compliant since VB.NET doesn't support unsigned types.

Summary Table:

Concept	Meaning	Purpose
CLR	Common Language Runtime	Executes code, manages memory & errors
CTS	Common Type System	Ensures consistent data types
CLS	Common Language Specification	Enables multi-language compatibility

Question 3:

You are developing a large-scale application and need to explain to a junior developer how assemblies are used in .NET Framework to organize and deploy the application. Provide an explanation of assemblies and include an example scenario where multiple assemblies are used.

Answer:

An **Assembly** is the **basic building block** of a .NET application. It is a compiled code library used for deployment, versioning, and security.

Key Points:

- An assembly can be: ○ **.exe** file → executable program ○ **.dll** file → reusable library
- Each assembly contains:
  - **IL code** (compiled code)

- **Metadata** (information about the code)
- **Manifest** (details like version, culture, references)
- **Resources** (images, text files, etc.)

Types of Assemblies:

1. **Private Assembly:**
   - Used by a single application.
   - Stored in the same folder as the executable.
2. **Shared Assembly:**
   - Can be used by multiple applications.
   - Stored in the **Global Assembly Cache (GAC)**.

Example Scenario:

Suppose you build an e-commerce application. You can organize like this:

- UI.dll → Contains user interface classes.
- BusinessLogic.dll → Contains calculations and validation.
- DataAccess.dll → Handles database connectivity.

These assemblies can be independently developed and maintained, making the system modular and easy to update.

Question 4:

In your project, you notice a developer struggling to organize classes and methods properly. How would you explain the concept of namespaces in .NET Framework and demonstrate how they are used to avoid naming conflicts in large projects?

Answer:

A **Namespace** in .NET is a logical grouping of related classes, interfaces, and methods. It helps organize code and prevents **naming conflicts**.

Why Use Namespaces:

- Avoids confusion when different parts of a project use the same class names.
- Makes large projects easier to manage.
- Provides structure and clarity.

Example Without Namespace:

```
class Employee
{
  public void Display() => Console.WriteLine("From HR Department");

}
  class Employee
{
   public void Display() => Console.WriteLine("From Sales Department");  }
```

This will cause a **naming conflict** because both classes have the same name.

Example With Namespace:

```
namespace HR
{
   class Employee
   {
      public void Display() => Console.WriteLine("HR Employee");
   }
}
namespace Sales
{
   class Employee
   {
      public void Display() => Console.WriteLine("Sales Employee");
   }
} class
Program
{
   static void Main()
   {
      HR.Employee e1 = new HR.Employee();
Sales.Employee e2 = new Sales.Employee();
e1.Display();        e2.Display();
   }
}
```

Output:
HR Employee
Sales Employee **Explanation:**
Using namespaces keeps similarly named classes separate and avoids conflicts.

Question 5:
During a code review, a developer confuses primitive types with reference types in their application. How would you explain the difference between primitive types and reference types?
Answer:
In C#, data types are divided into two categories:
1. Primitive (Value) Types:

- Store **actual data** directly in the memory.
- Stored in **stack** memory.
- When assigned to another variable, a *copy* of the value is made.

Examples:
int, float, char, bool, struct
Example:

```
int a = 5;
int b = a; b
= 10;
Console.WriteLine(a);  // Output: 5
```

Each variable holds its own value.

2. Reference Types:

- Store a **reference (address)** to the actual data in memory.
- Stored in **heap** memory.
- When assigned to another variable, both point to the same object.

Examples:
class, array, string, object
Example:

```
class Test { public int x; }

    Test t1 = new Test();
    t1.x = 5;

Test t2 = t1; // t2 points to same object
t2.x = 10;

Console.WriteLine(t1.x); // Output: 10
```

Explanation:
For value types, each variable is independent. For reference types, variables share the same memory reference.

Question 6:
While refactoring a piece of C# code, you notice both value types and reference types are being used incorrectly. Explain the difference between value types and reference types in C#, and provide examples to clarify their behaviour in memory.
Answer:
In C#, **value types** and **reference types** differ mainly in *how and where* they are stored in memory.

Value Types

- Store **actual data** in the **stack memory**.
- When assigned to another variable, a *copy* of the data is created.
- Changing one variable doesn't affect the other.

Examples: int, float, double, bool, struct **Example**
Code:

```
int x = 10;

int y = x; // copy of x

y = 20;

Console.WriteLine($"x = {x}, y = {y}");
```

Output: x =
10, y = 20

Reference Types

- Store the **address (reference)** of the data in **heap memory**.
- When assigned, both variables refer to the *same* memory location.
- Changing one affects the other.

Examples: class, array, string, object **Example**
Code:

```
class Demo { public int num; }

Demo d1 = new Demo();

d1.num = 5;

    Demo d2 = d1;  // d2 refers to same object
    d2.num = 10;
```

Summary:

Type	Stored In	Holds	Behavior
Value Type	Stack	Actual data	Independent copy
Reference Type	Heap	Address of object	Shared reference

Question 7:
You are tasked with creating a method that demonstrates both implicit and explicit type conversions. Write a program in C# that converts an int to a double implicitly and a double to an int explicitly, explaining each step in your code.
Answer:

```
using System;
class ConversionExample
{    static void
Main()
   {
      int num = 10;
      double implicitDouble = num;   // Implicit conversion (int → double)
Console.WriteLine("Implicit Conversion: int to double = " + implicitDouble);

      double val = 9.78;
      int explicitInt = (int)val;    // Explicit conversion (double → int)
      Console.WriteLine("Explicit Conversion: double to int = " + explicitInt);
   }
}
```

Explanation:

- **Implicit Conversion:** Happens automatically when no data loss occurs (e.g., smaller → larger type).
- **Explicit Conversion:** Requires casting because data might be lost (e.g., 9.78 becomes 9).

Question 8:

A junior developer asks for help writing a program to determine whether a number is positive, negative, or zero. Use if-else statements to write this program in C#, and explain the logic behind the code. Answer:

```
using System;  class CheckNumber
{
   static void Main()
   {
      Console.Write("Enter a number: ");
      int num = Convert.ToInt32(Console.ReadLine());

      if (num > 0)
         Console.WriteLine("The number is Positive");          else if (num < 0)
         Console.WriteLine("The number is Negative");           else
         Console.WriteLine("The number is Zero");
   }
}
```

Explanation:

- The if checks if number > 0. • else if checks if number < 0.
- If neither, the number must be 0.

Question 9:

Use a switch-case construct to explain how it works in C#. Illustrate the use of this construct by writing a program that takes a number (1–5) and prints the corresponding weekday. Answer:

```
using System;
class Weekdays
{
   static void Main()
   {
      Console.Write("Enter a number (1–5): ");
int day = Convert.ToInt32(Console.ReadLine());

      switch (day)
      {
         case 1: Console.WriteLine("Monday"); break;
case 2: Console.WriteLine("Tuesday"); break;          case
3: Console.WriteLine("Wednesday"); break;          case
4: Console.WriteLine("Thursday"); break;          case 5:
Console.WriteLine("Friday"); break;          default:
Console.WriteLine("Invalid day!"); break;
      }
   }
}
```

Explanation:

- The switch checks the value of day and jumps to the matching case.
- break stops further execution.
- default handles invalid input.

Question 10:

Demonstrate how to use nested if-else and switch-case statements together by writing a program that checks a number and prints whether it is even/odd and whether it falls into specific ranges (e.g., 0–10, 11–20). Answer:

```
using System;
class EvenOddRange
{
   static void Main()
   {
      Console.Write("Enter a number: ");
      int num = Convert.ToInt32(Console.ReadLine());

      if (num % 2 == 0)
         Console.WriteLine("Even Number");
else
         Console.WriteLine("Odd Number");

      switch (num)
      {
         case <= 10: Console.WriteLine("Range: 0–10"); break;
case <= 20: Console.WriteLine("Range: 11–20"); break;
default: Console.WriteLine("Range: Above 20"); break;
      }
   }
}
```

Explanation:

- The if-else checks even/odd using modulus %.
- The switch checks number range.

Question 11:
Write a program that prints the Fibonacci series using a for loop in C#. Provide a detailed explanation of your approach.
Answer:

```
using System; class
FibonacciSeries
{
   static void Main()
   {
      int n = 7;
      int a = 0, b = 1, c;

      Console.Write(a + " " + b + " ");
      for (int i = 2; i < n; i++)
      {
         c = a + b;
         Console.Write(c + " ");
a = b;          b = c;
      }
   }
}
```

Explanation:

- Starts with 0 and 1.
- Next number = sum of previous two.
- for loop repeats the calculation until n terms are printed.

Question 12:
Explain the key differences between while and do-while loops, and provide examples of each where one might be more appropriate than the other. Answer:

Loop	Condition Checked	Executes at least once?	Example Use
while	Before the body	No	When unsure if loop should run
do-while	After the body	Yes	When loop must run at least once

Example:

```
int i = 5; while
(i < 5)
   Console.WriteLine("While Loop"); // won't run

do {
   Console.WriteLine("Do-While Loop"); // runs once }
while (i < 5);
```

Question 13:

Write a program in C# that uses nested loops to generate a pyramid pattern of stars (*). Explain how the loops work together to produce the pattern.

Answer:

```
using System; class
StarPattern
{
   static void Main()
   {
      int rows = 5;

      for (int i = 1; i <= rows; i++)
      {
         for (int j = i; j < rows; j++)
            Console.Write(" ");

         for (int k = 1; k <= (2 * i - 1); k++)
            Console.Write("*");

         Console.WriteLine();
      }
   }
}
```

Explanation:

- The outer loop controls **rows**.
- The first inner loop prints **spaces**.
- The second inner loop prints **stars**.

  Together, they create a pyramid shape.

Question 14:
Define Encapsulation, Inheritance, Polymorphism, and Abstraction, and provide real-world examples of each in C#.
Answer:

1. **Encapsulation:**
    - Hiding internal data using access modifiers.
    - Example:
    -
```
using System;

class Account
{    private double balance; // data
hidden    public void Deposit(double
amount)
   {
      balance += amount;
   }
   public double GetBalance()
   {
      return balance;
   }
}
```

Real-world: ATM hides internal balance calculations.

2. **Inheritance:**
    - Allows one class to derive from another.
    - Example:

- ```
  using System;

  class Vehicle
  {
     public void Start()
     {
        Console.WriteLine("Vehicle started");
     }
  }

  class Car : Vehicle
  {    public void
  Drive()
     {
        Console.WriteLine("Car is driving");
     }
  }

  class Program
  {
     static void Main()
     {
        Car c = new Car();
        c.Start();  // from Vehicle
        c.Drive();  // from Car
     }
  }
  ```

Real-world: Car inherits general features of a vehicle.

3. **Polymorphism:** ○ Same method behaves differently in derived classes.
   - Example:
  ```

- ```
  using System;

  class Shape
  {
     public virtual void Draw() { }
  }

  class Circle : Shape
  {
     public override void Draw()
     {
        Console.WriteLine("Drawing Circle");
     }
  }

  class Program
  {
     static void Main()
     {
        Shape s = new Circle();
        s.Draw();  // Output: Drawing Circle
     }
  }
  ```

4. **Abstraction:**
   - Shows only essential features, hides complex details.
  ```

- Example:

```
using System; abstract
class Animal
{
   public abstract void Sound();
}
class Dog : Animal
{
   public override void Sound()
   {
      Console.WriteLine("Bark");
   }
}
class Program
{
   static void Main()
   {
      Animal a = new Dog();
      a.Sound(); // Output: Bark
   }
}
```

Question 15:
Write a C# program that includes both a constructor and a destructor, explaining the lifecycle of an object from creation to destruction.
Answer:

```
    using System;
    class Demo
{
   public Demo()
   {
      Console.WriteLine("Constructor called: Object created");
   }

   ~Demo()
   {
      Console.WriteLine("Destructor called: Object destroyed");
   }
}

class Program
{
   static void Main()
```

```
    {
        Demo d = new Demo();
    }
}
```

Explanation:

- The **constructor** initializes the object.
- The **destructor** runs automatically when the object is destroyed (usually by garbage collector).

Question 16:
Explain public, private, protected, and internal access modifiers, and demonstrate their use in a small C# class.
Answer:

```
using System;

class AccessExample
{
    public void PublicMethod() { Console.WriteLine("Public"); }
private void PrivateMethod() { Console.WriteLine("Private"); }
protected void ProtectedMethod() { Console.WriteLine("Protected"); }
internal void InternalMethod() { Console.WriteLine("Internal"); }  }
  class
Program
{
    static void Main()
      {
         AccessExample obj = new AccessExample();
         obj.PublicMethod();   // Accessible
  obj.InternalMethod(); // Accessible within same assembly
      }
 }
```

Explanation:

Modifier	Accessible From	Example Use
public	Anywhere	Utility methods
private	Same class	Encapsulated logic
protected	Same + derived class	Base class features
internal	Same assembly	Shared internal code

Question 17:
Write a program where a Vehicle class is inherited by a Car class and a Bike class, each with their own unique methods. Demonstrate how inheritance allows code reuse. Answer:

Explanation:

```
using System;

// Base class  class
Vehicle
{
    public void Start()
    {
        Console.WriteLine("Vehicle Starting...");
}
}

// Derived class Car  class
Car : Vehicle
{
    public void Drive()
    {
        Console.WriteLine("Car Driving...");
    }
}

// Derived class Bike  class
Bike : Vehicle
{
    public void Ride()
    {
        Console.WriteLine("Bike Riding...");
    }
}

// Main Program  class
Program
{
    static void Main()
    {
```

Explanation:

```
        // Car object
        Car myCar = new Car();
myCar.Start();     // Inherited from Vehicle
myCar.Drive();  // Car-specific

         // Bike object
         Bike myBike = new Bike();
         myBike.Start(); // Inherited from Vehicle
  myBike.Ride();  // Bike-specific
     }
  }
```

- Car and Bike reuse the Start() method from Vehicle.
- Demonstrates **code reusability** and **inheritance**.

Explanation:

Question 18:
Explain how the try-catch-finally blocks work in C# with an example of catching and handling an arithmetic exception, and how finally is always executed.
Answer:

```
try {     int x = 10,
y = 0;     int result
= x / y;
}
catch (DivideByZeroException ex)
{
   Console.WriteLine("Error: Cannot divide by zero.");
} finally
{
   Console.WriteLine("Finally block executed."); }
```

Explanation:

- Code in try may cause an error.
- catch handles that error gracefully.
- finally always runs, whether an error occurs or not — often used to release resources.

Question 19:
Write a C# program that demonstrates exception handling by throwing and catching a custom exception, explaining why custom exceptions are beneficial.
Answer:

Explanation:

```
using System; class
InvalidMarksException : Exception
{    public InvalidMarksException(string message) : base(message) {
} }

class Program
{    static void
Main()
   {
      int marks = -10;

try
      {
        if (marks < 0 || marks > 100)                    throw new
InvalidMarksException("Marks must be between 0 and 100!");
      }
      catch (InvalidMarksException e)
      {
        Console.WriteLine(e.Message);
      }
   }
}
```

- Custom exceptions make errors **specific and meaningful**.
- Instead of a generic error, the program explains what went wrong (e.g., invalid marks).

Question 20:

Explain how proper exception handling improves an application's robustness.

Answer:

Proper exception handling ensures that a program can **gracefully respond to unexpected errors** instead of crashing. It separates **normal code** from **error-handling code**, making the program easier to read and maintain.

Benefits:

1. **Prevents crashes:** The program can catch errors and continue running safely.
2. **Improves user experience:** Users see clear error messages instead of confusing crashes.
3. **Easier debugging:** Developers can log detailed error information.
4. **Resource management:** Using finally ensures resources like files or database connections are always released.
5. **Reliable software:** Programs become more stable and robust under different scenarios.

Example in C#:

Explanation:

```
try
{
   int x = 10, y = 0;
   int result = x / y; // Will cause DivideByZeroException
}
catch (DivideByZeroException ex)
{
   Console.WriteLine("Error: Cannot divide by zero!");
} finally
{
   Console.WriteLine("This block always executes.");
}
```

Explanation:

- The program catches the error instead of crashing.
- finally ensures cleanup or final messages.
- Users see a friendly message and the program continues safely.

Summary: Well-handled exceptions turn potential crashes into controlled, recoverable situations, making the application more stable and reliable.