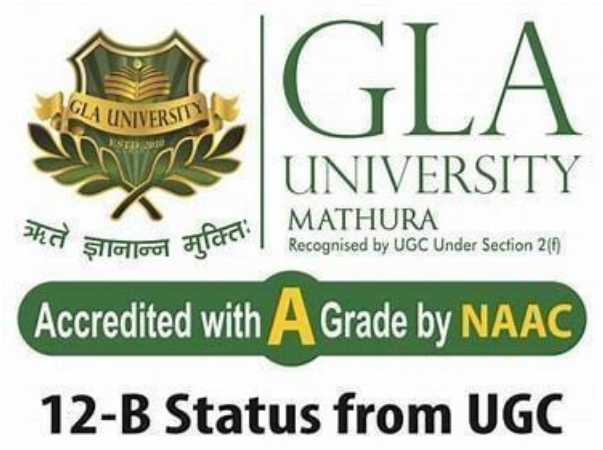
.Net Framework using C# LAB

MCAE0471

MCA 2nd Year



Session:-2025-26

Submitted to:-

Mr. Sachendra Singh Chauhan

Assistant Professor

Dept. CEA

Submitted by:-

Shivam Pal

Roll No. – 38

Uni. Roll No. – 2484200184

Module 1 Assignment 1 — .NET Framework using C#

1. .NET Framework Architecture

The .NET Framework architecture consists of several logical components that work together to run managed applications. Key components:

- Common Language Runtime (CLR): The execution engine that manages program execution, memory, garbage collection, JIT compilation, security, threading and exception handling.

- Framework Class Library (FCL): A large set of reusable types (classes, interfaces, value types) organized into namespaces such as System, System.IO, System.Net that provide common functionality.

- Application Domains: Logical containers within a process used to isolate executed applications. They allow unloading assemblies, provide security boundaries, and prevent faults in one app domain from affecting others.

- Metadata & Assemblies: Assemblies (EXE/DLL) contain metadata describing types and references; metadata enables language interoperability.

- Language Compilers & CTS/CLS: Compilers (C#, VB.NET) produce IL (Intermediate Language) which targets the CLR. The Common Type System (CTS) defines types, and the Common Language Specification (CLS) defines rules for language interoperability.

Structure: Source code -> Compiler -> MSIL + Metadata -> Assembly -> CLR (JIT) -> Native code at runtime.

2. Key runtime concepts: CLR, CTS, CLS

- CLR (Common Language Runtime): Responsible for executing managed code. It provides services like memory management (GC), security (code access security in older .NET), threading, JIT compilation, and exception handling.

- CTS (Common Type System): A specification that defines how data types are declared, used, and managed. It ensures that types from different languages can interoperate because they share a common representation (e.g., System.Int32).

- CLS (Common Language Specification): A subset of CTS; rules that languages must follow to interoperate. For example, CLS-compliant libraries avoid unsigned types in public signatures because some languages don’t support them.

How to present to the team: Use examples—show a C# method compiled to IL and explain how CLR executes it; show a VB.NET and C# type both mapped to System.String and explain interoperability via CTS/CLS.

3. Assemblies: organization and deployment

An assembly is the basic deployment unit in .NET — a physical file (DLL or EXE) containing compiled MSIL, metadata, and optional resources. Assemblies provide:

• Versioning and identity (name, culture, public key token).

• Encapsulation of types and resources.

• Security boundaries and deployment units.

Example scenario:

Project: Large application with layers:

• MyApp.Core.dll — domain entities and interfaces

• MyApp.Data.dll — data access layer (EF or ADO.NET)

• MyApp.Services.dll — business logic

• MyApp.Web.exe — presentation (ASP.NET) referencing above DLLs

Each assembly can be developed, tested, and versioned independently; deployment copies the required assemblies alongside the EXE or to the GAC (Global Assembly Cache in .NET Framework).

4. Namespaces: avoiding naming conflicts

Namespaces are logical groupings of related types. They prevent naming collisions by qualifying type names.

Example:

• System.IO.File and MyApp.Utilities.File can both exist because their fully-qualified names differ.

Demonstration (C#):

```
namespace MyApp.Utilities {     public class Logger { public void
Log(string m) { /*...*/ } }
} namespace
MyApp.Features {
```

```
    public class Logger { public void Info(string m) { /*...*/ } }
}
```

Using directive and aliasing:

```
using Utils = MyApp.Utilities;
// then: Utils.Logger log = new Utils.Logger();
```

5. Primitive types vs Reference types

Primitive types (in C# these are built-in simple types like int, bool, char, float) are value types that store the actual data. Reference types (classes, arrays, strings, delegates) store a reference to the memory location where the data is stored.

Key differences:

- Value types allocated on stack (or inline in objects); reference types on the heap with GCmanaged lifetime.
- Assignment of value types copies data; assignment of reference types copies the reference.
- Examples: int a = 5; int b = a; // b is copy; class Person p1 = new Person(); Person p2 = p1; // p2 references same object

6. Value types vs Reference types (memory behaviour)

Value types: structs, enums, primitive numeric types. Stored directly where declared. Passing a value type to a method copies it (unless passed by ref).

Reference types: class, interface, array, delegate. Variable holds a reference; multiple variables can point to same object. The object lives on the managed heap and is reclaimed by GC when unreachable.

Examples (C#):

```
struct Point { public int X, Y; }

Point p1 = new Point { X=1, Y=2 };
Point p2 = p1; // copy p2.X = 10;
// p1.X still 1

class Node { public int Value; } Node
n1 = new Node { Value = 5 }; Node n2
= n1; // reference n2.Value = 20; //
n1.Value becomes 20
```

7. Implicit and Explicit Type Conversions (C#)

Implicit conversion: safe conversions done automatically (e.g., int -> double). Explicit conversion (casting): may lose data (double -> int) and must be specified.

```
using System; class Program {
static void Main() {          int i =
42;         // integer
        double d = i;       // implicit conversion (int -> double)
        Console.WriteLine($"implicit: {d}");
         double x = 9.78;          int j = (int)x;     // explicit
conversion (double -> int) - fractional part truncated
        Console.WriteLine($"explicit: {j}");
    }
}
```

Explanation: The runtime safely widens int to double. Converting double to int requires cast and may truncate fractional part.

8. Program to check positive, negative or zero

```
using System; class
Program {     static
void Main() {
        Console.Write("Enter a number: ");
double n = double.Parse(Console.ReadLine());
if (n > 0) Console.WriteLine("Positive");          else
if  (n  <  0)  Console.WriteLine("Negative");     else
Console.WriteLine("Zero");
    }
}
```

Logic: Compare the number to 0 using if-else chain: first >0, else <0, else (==0).

9. Switch-case

```
using System; class
Program {
static void Main()
{
        Console.Write("Enter number (1-5): ");          int n =
int.Parse(Console.ReadLine());          switch(n) {
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 input"); break;
        }
    }
}
```

Explanation: switch compares the value against cases; matching case executes until break. Default handles unexpected values.

10. Nested if-else and switch combined

```
using System; class
Program {     static
void Main() {
        Console.Write("Enter number: ");
int n = int.Parse(Console.ReadLine());
        // even/odd
        if (n % 2 == 0) Console.WriteLine("Even"); else
Console.WriteLine("Odd");

        // range using switch on a small mapped category
int range = (n <= 10) ? 1 : (n <= 20) ? 2 : 3;
switch(range) {             case 1: Console.WriteLine("Range: 0-
10"); break;             case 2: Console.WriteLine("Range: 11-
20"); break;             default: Console.WriteLine("Range: >20 or
<0"); break;
        }
    }
}
```

Explanation: The program first uses if-else for parity, then maps number to a range and uses switch for descriptive output.

11. Fibonacci series using for loop

```
using System; class
Program {     static
void Main() {
        Console.Write("How many terms? ");
        int terms = int.Parse(Console.ReadLine());
int a = 0, b = 1;
        for (int i = 0; i < terms; i++) {
Console.Write(a + " ");             int
temp = a + b;             a = b;
b = temp;
```

```
        }
    }
}
```

Explanation: Initialize first two terms (0,1). On each loop print 'a', compute next term as a+b, then shift a=b, b=next.

12. while vs do-while loops

- while: checks condition before executing loop body. If condition false initially, body may never execute.

Example use: read input until user types 'exit' and you don't want to execute body when condition false initially.

- do-while: executes body at least once, checks condition after execution. Useful when you must run the body once before validation (e.g., menu shown once).

Examples:

```
// while example:
int i = 0; while (i < 3) {
Console.WriteLine(i); i++; }

// do-while example:
int j = 0; do { Console.WriteLine(j); j++; }
while (j < 3);
```

13. Pyramid pattern using nested loop

```
using System; class Program {     static
void Main() {          int rows = 5;
for (int i = 1; i <= rows; i++) {
            // spaces
            for (int s = 0; s < rows - i; s++) Console.Write(" ");
            // stars
            for (int k = 0; k < (2*i - 1); k++) Console.Write("*");
            Console.WriteLine();
        }
    }
}
```

Explanation: Outer loop controls rows; inner loops print leading spaces and stars to form centered pyramid.

14. OOP Principles: Encapsulation, Inheritance, Polymorphism, Abstraction

- Encapsulation: Bundling data and methods that operate on that data, and restricting access via access modifiers. Example: a class BankAccount with private balance and public methods Deposit/Withdraw.

- Inheritance: Deriving new classes from existing ones to reuse code. Example: class Vehicle -> class Car, Bike.

- Polymorphism: Ability to treat objects of different classes through a common interface or base class. Example: virtual method Drive() overridden by Car and Bike; you can call Drive() on Vehicle reference and get derived behavior.

- Abstraction: Exposing only necessary details while hiding implementation. Example: exposing an interface IRepository<T> while concrete classes handle DB details.

15. Constructors and Destructors in C#

```
using System; class Demo {     public Demo() {
Console.WriteLine("Constructor called"); }
    ~Demo() { Console.WriteLine("Destructor (finalizer) called"); }
} class Program {     static
void Main() {         Demo d
= new Demo();
        // destructor called by GC at some later time; to force:
GC.Collect(); GC.WaitForPendingFinalizers();
    }
}
```

Explanation: Constructor runs on object creation. Destructor (finalizer) runs before GC reclaims object; timing is non-deterministic. Use IDisposable and Dispose for deterministic cleanup.

16. Access modifiers: public, private, protected, internal

```
public class Sample {     public
void PublicMethod() { }
private void PrivateMethod() { }
protected void ProtectedMethod() {
}     internal void
InternalMethod() { }
}

// Explanation:
// public: accessible from any assembly.
// private: accessible only within containing class.
// protected: accessible within class and derived classes.
```

```
// internal: accessible within the same assembly (project).
```

17. Inheritance example: Vehicle -> Car, Bike

```
using System; class Vehicle {     public void Start() {
Console.WriteLine("Vehicle started"); }
} class Car : Vehicle {             public void OpenTrunk() {
Console.WriteLine("Trunk opened"); }
} class Bike : Vehicle {     public void KickStart() {
Console.WriteLine("Bike kickstarted"); }
} class Program {
static void Main() {
        Car c = new Car(); c.Start(); c.OpenTrunk();
Bike b = new Bike(); b.Start(); b.KickStart();
    }
}
```

Explanation: Car and Bike reuse Start() from Vehicle; they add their own behavior, demonstrating code reuse.

18. try-catch-finally and handling ArithmeticException

```
using System; class
Program {     static
void Main() {
try {
            int x = 10, y = 0;
            int z = x / y; // throws DivideByZeroException
            Console.WriteLine(z);
        }
        catch (DivideByZeroException ex) {
            Console.WriteLine("Cannot divide by zero: " + ex.Message);
        }
finally {
Console.WriteLine
("Finally block
executed
always.");
        }
    }
}
```

Explanation: Code in try may throw; catch handles specific exceptions; finally executes regardless, used for cleanup.

19. Custom exception example

```
using System; class InvalidAgeException : Exception {     public
InvalidAgeException(string message) : base(message) { }
} class Program {     static void ValidateAge(int age) {         if
(age < 0 || age > 150) throw new InvalidAgeException("Age is invalid: "
+ age);
    }     static void
Main() {         try {
            ValidateAge(-5);
        }
        catch (InvalidAgeException ex) {
            Console.WriteLine("Custom exception caught: " +
ex.Message);
        }
    }
}
```

Why custom exceptions: Provide clearer semantics, enable specific catch blocks, and carry domain-specific information.

20. Advantages of Exception Handling

- Improves robustness: prevents application crashes by catching and handling unexpected conditions.
- Separation of error-handling logic from regular logic makes code cleaner and easier to maintain.
- Allows centralized handling/logging of errors (e.g., top-level exception handlers, logging frameworks).
- Enables resource cleanup via finally or IDisposable/using patterns, preventing resource leaks.
- Supports fault-tolerant and resilient designs (retry logic, fallbacks).