NAME :- Yash kumar varshney

Course :- MCA 2C

Roll no:- 2484200220

Class roll:- 53

1. .NET Framework architecture — structured explanation

Top-level components

- **Common Language Runtime (CLR)**
  The execution engine. Handles JIT compilation, garbage collection (GC), security, threading, exception handling, and interop with unmanaged code.
- **Framework Class Library (FCL) / Base Class Library (BCL)**
  A large class library providing common functionality: collections, I/O, networking, XML, WinForms/WPF (Framework), ADO.NET, etc.
- **Languages & Compilers**
  Multiple languages (C#, VB.NET, F#) compile to the Common Intermediate Language (CIL or IL).
- **Metadata & Assemblies**
  Compiled code packaged as assemblies (.dll/.exe) containing IL + metadata (types, methods, versioning info).
- **Application Domains (AppDomain)**
  Lightweight isolation boundaries inside a process. Provide isolation of assemblies so you can load/unload sets of assemblies. (Note: AppDomains exist in .NET Framework; .NET Core/.NET 5+ use processes and other isolation mechanisms.)
- **Runtime Services**
  Type safety, security policy, code access security (CAS, in .NET Framework), interop, profiling/debugging APIs.

How it fits together

- Source code (C#) → compiler → IL + metadata inside an assembly.
- At runtime the CLR loads the assembly, JIT-compiles IL to native code, provides memory management (GC), and runtime services.
- The FCL/BCL is consumed by applications; developers rely on it to avoid reinventing common code.

2. Key runtime concepts (CLR, CTS, CLS) — how to present to a team

Common Language Runtime (CLR)

- The runtime engine that runs managed code: JIT, GC, exception handling, security, threading.
- Relevance: If we get an OutOfMemoryException or performance issue, the CLR is where GC and JIT behavior matter.

Common Type System (CTS)

- Defines all data types and how they behave across languages (value types, reference types, type inheritance).
- Relevance: Ensures types from one .NET language (e.g., VB) interoperate with C#. When we design public APIs, pick types that map well across languages.

Common Language Specification (CLS)

- A subset of CTS that defines rules for language interop (e.g., names should not differ only by case).
- Relevance: If our library advertises "CLS-compliant," other .NET languages can use it safely. In practice: avoid unsigned public API types if cross-language compatibility is needed.

Presentation tip: Use concrete examples: "If you expose `public unsigned int` in C# it may not be usable from some languages — CLS matters." Tie each concept to a common issue: performance (CLR/GC), cross-language API design (CTS/CLS).

3. Assemblies — explanation + scenario

Definition

- An **assembly** is the deployment unit for .NET: a `.dll` or `.exe` containing IL, metadata, and optional resources and manifest. It provides versioning and identity (name, version, culture, public key token).

Types

- **Private assemblies**: deployed with the application (local folder).
- **Shared (GAC) assemblies**: installed system-wide (Global Assembly Cache, .NET Framework only).
- **Satellite assemblies**: for localization resources.

Why use assemblies

- Modularization: split code into logical units (UI, business logic, data access).
- Independent versioning and deployment.

- Security boundaries and load/unload (AppDomains).

Example scenario
Large app divided into assemblies:

- `MyApp.UI.dll` — UI forms / web front-end.
- `MyApp.Business.dll` — business rules, services.
- `MyApp.Data.dll` — data access, repository implementations.
- `MyApp.Common.dll` — shared models, utils.

Deploy: `MyApp.UI.exe` references the other three DLLs. Teams can work separately; you can update the data layer assembly without changing UI if the contract (public API) remains stable.

4. Namespaces — explanation + example

Definition

- A **namespace** groups related classes and prevents naming conflicts. Syntax: `namespace Company.Project.Module { ... }`.

Why

- Avoid collisions (`Company.Project.Logging.Logger` vs `ThirdParty.Logging.Logger`).
- Organize code logically and make discoverability easier in IDEs (IntelliSense).

Example

```
// File: Company.Project.Logging.Logger.cs
namespace Company.Project.Logging
{
    public class Logger
    {
        public void Log(string msg) => Console.WriteLine($"[LOG] {msg}");
    }
}

// File: ThirdParty.Logging.Logger.cs
namespace ThirdParty.Logging
{
    public class Logger
    {
        public void LogInfo(string msg) => Console.WriteLine($"[3P] {msg}");
    }
}
```

```
// Usage
using Company.Project.Logging;
using ThirdParty.Logging; // possible conflict if types named same — use
aliases.

class Program
{
    static void Main()
    {
        var myLogger = new Company.Project.Logging.Logger();
        myLogger.Log("Hello");

        var thirdLogger = new ThirdParty.Logging.Logger();
        thirdLogger.LogInfo("Hi");
    }
}
```

Tip: Use `using alias = ThirdParty.Logging.Logger;` if you need to resolve same type names.

5. Primitive types vs reference types — explanation

Primitive types (commonly called value types)

- Built-in simple types like `int`, `double`, `bool`, `char`, structs. They store the actual value.
- Example: `int x = 5;` — `x` directly contains `5`.

Reference types

- Objects, arrays, `string`, delegates. Variables hold a reference (pointer) to an object on the heap.
- Example: `class Person { } Person p = new Person();` — `p` points to an object on the heap.

Key differences

- Copying value type copies the value; copying reference type copies the reference (both variables point to the same object).
- Value types often allocated on the stack (or inline inside objects) and are generally faster/short-lived; reference types live on the heap and are GC-managed.
- `string` is a reference type even though it behaves like a primitive in many ways (immutable).

6. Value types vs reference types in C# — memory behavior + examples

Value types

- Include primitives (`int`, `double`, `bool`) and `struct`.
- Stored inline: local variables live on the stack (or inside objects/arrays).
- Assignment copies the entire value.
- No GC overhead for small transient values.
- Example:
- `int a = 10;`
- `int b = a; // b = 10; independent`
- `b = 20;`
- `// a still 10`

Reference types

- Instances created with `new` on the heap (objects, arrays, class instances).
- Assignment copies the reference. Multiple variables can reference same object.
- GC reclaims memory when no references remain.
- Example:
- `class Person { public string Name; }`
- `var p1 = new Person { Name = "Alice" };`
- `var p2 = p1;`
- `p2.Name = "Bob";`
- `// p1.Name is now "Bob" because p1 and p2 reference same object.`

Boxing/Unboxing

- Converting a value type to `object` boxes it to the heap; unboxing extracts value.

7. Implicit and explicit conversions — program (C#)

Explanation

- *Implicit conversion*: safe, no data loss (e.g., `int → double`). Compiler inserts conversion automatically.
- *Explicit conversion (cast)*: possible data loss or narrowing (e.g., `double → int`), so you must cast.

Program

```
using System;

class ConversionsDemo
{
    static void Main()
    {
        int i = 42;
        // Implicit conversion: int -> double (no data loss)
        double d = i;
        Console.WriteLine($"Implicit: int {i} -> double {d}");

        // Explicit conversion: double -> int (possible fractional loss)
        double pi = 3.14159;
        int truncated = (int)pi; // explicit cast, fractional part lost
        Console.WriteLine($"Explicit: double {pi} -> int {truncated}");

        // Show potential overflow case
        double large = 1e20;
        try
        {
            int castLarge = (int)large; // compiles; result is
implementation-defined (overflow)
            Console.WriteLine($"Casting large double {large} -> int
{castLarge}");
        }
        catch (OverflowException ex)
        {
            Console.WriteLine($"Overflow on cast: {ex.Message}");
        }
    }
}
```

Example output

```
Implicit: int 42 -> double 42
Explicit: double 3.14159 -> int 3
Casting large double 1E+20 -> int 2147483647
```

(Last line may vary but demonstrates data loss/overflow.)

8. Determine whether a number is positive, negative, or zero — C# with if-else

Logic

- If number > 0 → positive

- Else if number < 0 → negative
- Else → zero

Program

```
using System;

class PosNegZero
{
    static void Main()
    {
        Console.Write("Enter an integer: ");
        if (!int.TryParse(Console.ReadLine(), out int num))
        {
            Console.WriteLine("Invalid input.");
            return;
        }

        if (num > 0)
            Console.WriteLine("Number is positive.");
        else if (num < 0)
            Console.WriteLine("Number is negative.");
        else
            Console.WriteLine("Number is zero.");
    }
}
```

Example
Input: `-5` → Output: `Number is negative.`

9. Switch-case construct — number (1-5) to weekday — C#

Program

```
using System;

class WeekdaySwitch
{
    static void Main()
    {
        Console.Write("Enter a number (1-5): ");
        if (!int.TryParse(Console.ReadLine(), out int n))
        {
            Console.WriteLine("Invalid input.");
            return;
        }

        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("Number out of range (1-5).");
                break;
        }
    }
}
```

Example
Input `3` → Output `Wednesday`.

10. Nested if-else and switch-case together — program

Requirement: Check number, print even/odd and which range it falls into (0–10, 11–20).

Program

```
using System;

class NestedDecision
{
    static void Main()
    {
        Console.Write("Enter an integer: ");
        if (!int.TryParse(Console.ReadLine(), out int num))
        {
            Console.WriteLine("Invalid input.");
            return;
        }

        // Even or odd
        string parity = (num % 2 == 0) ? "even" : "odd";
        Console.WriteLine($"{num} is {parity}.");
```

```
        // Range check using nested if-else
        if (num >= 0 && num <= 10)
            Console.WriteLine("Range: 0-10");
        else if (num >= 11 && num <= 20)
            Console.WriteLine("Range: 11-20");
        else
            Console.WriteLine("Range: outside 0-20");

        // Illustrate mixing switch for parity and range category
        int rangeCategory = (num >= 0 && num <= 10) ? 1 : (num >= 11 && num
<= 20) ? 2 : 3;
        switch (rangeCategory)
        {
            case 1:
                Console.WriteLine("Switch: in 0-10");
                break;
            case 2:
                Console.WriteLine("Switch: in 11-20");
                break;
            default:
                Console.WriteLine("Switch: outside 0-20");
                break;
        }
    }
}
```

Example
Input `7` →

```
7 is odd.
Range: 0-10
Switch: in 0-10
```

11. Fibonacci series using a for loop — code + explanation

Approach

- Use iterative approach: start with `0, 1`, then each next = sum of previous two.
- Use a for loop for `n` terms.

Program

```
using System;

class FibonacciFor
{
    static void Main()
    {
```

```
        Console.Write("How many terms? ");
        if (!int.TryParse(Console.ReadLine(), out int n) || n < 1)
        {
            Console.WriteLine("Enter a positive integer.");
            return;
        }

        long a = 0, b = 1;
        if (n >= 1) Console.Write(a);
        if (n >= 2) Console.Write(" " + b);

        for (int i = 3; i <= n; i++)
        {
            long c = a + b;
            Console.Write(" " + c);
            a = b;
            b = c;
        }

        Console.WriteLine();
    }
}
```

Explanation inside code: `a` and `b` track the last two numbers; loop computes next `c`, prints it, then shifts `a=b`, `b=c`.

Example
Input `6` → Output: `0 1 1 2 3 5`

12. while vs do-while loops — differences + examples

Key differences

- `while(condition) { }` — condition checked before entering loop. If false initially, body never runs.
- `do { } while(condition);` — body runs at least once; condition checked after body.

When to use

- `while`: when you may want zero iterations (e.g., read input only if available).
- `do-while`: when you need the body to execute at least once (e.g., show a menu and ask user to continue).

Examples

while:

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

do-while:

```
int choice;
do
{
    Console.WriteLine("Menu: 1=Do, 0=Exit");
    choice = int.Parse(Console.ReadLine());
} while (choice != 0);
```

13. Pyramid pattern using nested loops — program

Goal: Pyramid of stars; e.g., for `n=4`:

```
   *
  ***
 *****
*******
```

Program

```
using System;

class PyramidPattern
{
    static void Main()
    {
        Console.Write("Enter number of rows: ");
        if (!int.TryParse(Console.ReadLine(), out int n) || n <= 0)
        {
            Console.WriteLine("Enter a positive integer.");
            return;
        }

        for (int i = 1; i <= n; i++)
        {
            // print spaces
            for (int s = 0; s < n - i; s++)
                Console.Write(' ');

            // print stars: 2*i - 1
```

```
            for (int star = 0; star < 2 * i - 1; star++)
                Console.Write('*');

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

How loops work: Outer loop iterates rows; first inner loop prints leading spaces to center pyramid; second inner loop prints the required stars.

14. OOP concepts — definitions + real-world C# examples

Encapsulation

- Hiding internal state and exposing behavior via methods/properties.
- Example: `class BankAccount { private decimal balance; public void Deposit(decimal amt) { ... } }`

Inheritance

- Deriving a new type from an existing one to reuse and extend behavior.
- Example: `class Animal { } class Dog : Animal { }`

Polymorphism

- Ability to treat objects of different derived types through a common base type (method overriding or interfaces).
- Example: `virtual void Speak()` in `Animal` overridden by `Dog`/`Cat`. Or interfaces like `IShape.Draw()` implemented differently.

Abstraction

- Expose only essential features and hide complexity. Use abstract classes/interfaces.
- Example: `interface IRepository<T> { void Add(T t); T Get(int id); }` — hides storage details.

15. Constructors and destructors — program + explanation

Constructor: special method invoked when object is created — initialize state.

Destructor (finalizer): `~ClassName()` called by GC at some indeterminate time before object memory reclaimed. Use only for releasing unmanaged resources; prefer `IDisposable` and `Dispose()`.

Program demonstration

```
using System;

class Demo
{
    public Demo()
    {
        Console.WriteLine("Constructor: object created.");
    }

    ~Demo()
    {
        // Finalizer — called by GC in non-deterministic time
        Console.WriteLine("Finalizer: object is being collected (non-
deterministic).");
    }
}

class Program
{
    static void Main()
    {
        CreateAndRelease();
        Console.WriteLine("Main finished. Forcing GC (for demo only).");
        GC.Collect();
        GC.WaitForPendingFinalizers();
        Console.WriteLine("After GC.WaitForPendingFinalizers().");
    }

    static void CreateAndRelease()
    {
        Demo d = new Demo();
        // d goes out of scope at end of method
    }
}
```

Lifecycle: `new Demo()` → constructor runs immediately. When object becomes unreachable, GC may run and call finalizer at some later time. For deterministic cleanup of unmanaged resources, implement `IDisposable` and call `Dispose()` (or use `using`).

16. Access modifiers — explanation + example class

Modifiers

- `public`: accessible everywhere.
- `private`: accessible only within the containing class.
- `protected`: accessible in the containing class and derived classes.
- `internal`: accessible within the same assembly.
- `protected internal` / `private protected` combine semantics.

Example

```
public class Sample
{
    public int PublicValue = 1;
    private int PrivateValue = 2;
    protected int ProtectedValue = 3;
    internal int InternalValue = 4;

    public void Demo()
    {
        Console.WriteLine($"Inside class: {PublicValue}, {PrivateValue},
{ProtectedValue}, {InternalValue}");
    }
}

public class Derived : Sample
{
    public void Show()
    {
        Console.WriteLine($"In derived: {PublicValue}, {ProtectedValue},
{InternalValue}");
        // PrivateValue is not accessible here.
    }
}

class Program
{
    static void Main()
    {
        var s = new Sample();
        Console.WriteLine(s.PublicValue);    // OK
        // Console.WriteLine(s.PrivateValue); // Not accessible
        Console.WriteLine(s.InternalValue);  // OK within same assembly
    }
}
```

17. Inheritance example — Vehicle, Car, Bike

Program

```
using System;

class Vehicle
{
    public string Make { get; set; }
    public void Start() => Console.WriteLine("Vehicle started.");
}

class Car : Vehicle
{
    public void OpenTrunk() => Console.WriteLine("Car trunk opened.");
}

class Bike : Vehicle
{
    public void KickStart() => Console.WriteLine("Bike kick-started.");
}

class Program
{
    static void Main()
    {
        Car car = new Car { Make = "Toyota" };
        car.Start();
        car.OpenTrunk();

        Bike bike = new Bike { Make = "Honda" };
        bike.Start();
        bike.KickStart();
    }
}
```

Explanation: `Car` and `Bike` inherit `Start()` from `Vehicle` — code reuse. They add specific methods.

18. try-catch-finally — arithmetic exception example

Explanation

- `try` block: code that might throw exceptions.
- `catch` block(s): handle specific exception types.

- `finally` block: always executed, used for cleanup (runs even if exception thrown, except in rare cases like process termination).

Program

```
using System;

class TryCatchFinallyDemo
{
    static void Main()
    {
        try
        {
            Console.Write("Enter divisor (int): ");
            int divisor = int.Parse(Console.ReadLine());
            int result = 10 / divisor; // may throw DivideByZeroException
            Console.WriteLine($"Result: {result}");
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine("Cannot divide by zero: " + ex.Message);
        }
        catch (FormatException ex)
        {
            Console.WriteLine("Invalid number format: " + ex.Message);
        }
        catch (Exception ex)
        {
            Console.WriteLine("General error: " + ex.Message);
        }
        finally
        {
            Console.WriteLine("Finally block executed (cleanup if needed).");
        }
    }
}
```

Example
Input `0` → Output:

```
Cannot divide by zero: Attempted to divide by zero.
Finally block executed (cleanup if needed).
```

19. Custom exception — program + why useful

Why custom exceptions

- Provide domain-specific error types that make error handling clearer and allow callers to catch specific problems.

Program

```
using System;

public class InvalidAgeException : Exception
{
    public InvalidAgeException() { }
    public InvalidAgeException(string message) : base(message) { }
    public InvalidAgeException(string message, Exception inner) :
base(message, inner) { }
}

class Registration
{
    public static void RegisterUser(string name, int age)
    {
        if (age < 0 || age > 120)
            throw new InvalidAgeException($"Age {age} is not valid for
registration.");
        Console.WriteLine($"User {name} registered, age {age}.");
    }
}

class Program
{
    static void Main()
    {
        try
        {
            Registration.RegisterUser("Alice", -1);
        }
        catch (InvalidAgeException ex)
        {
            Console.WriteLine("Custom exception caught: " + ex.Message);
        }
    }
}
```

Output

```
Custom exception caught: Age -1 is not valid for registration.
```

20. Advantages of exception handling in C# — explanation

Key benefits

- **Improves robustness**: code can gracefully handle unexpected conditions instead of crashing.
- **Separation of concerns**: error-handling logic separated from normal code flow.
- **Precise handling**: catch specific exceptions to deal with different failure modes.
- **Resource safety**: `finally`/`using` ensures resources (files, DB connections) are cleaned up.
- **Debugging support**: exceptions carry stack traces and inner exceptions to aid diagnostics.

Best practices

- Don’t use exceptions for normal control flow.
- Catch specific exceptions, not `catch(Exception)` unless rethrowing or logging.
- Always clean up resources (`using` for `IDisposable`, `finally` for general cleanup).
- Wrap high-level entry points with top-level handlers to log/translate failures gracefully.

Lab Assignment 5

Q1 Handling Division by Zero

Read two numbers and perform division. Use try-catch-finally. Catch DivideByZeroException and display “Division by zero is not allowed.” In the finally block display “Execution completed.” Ensure finally executes regardless of exceptions

Ans:-

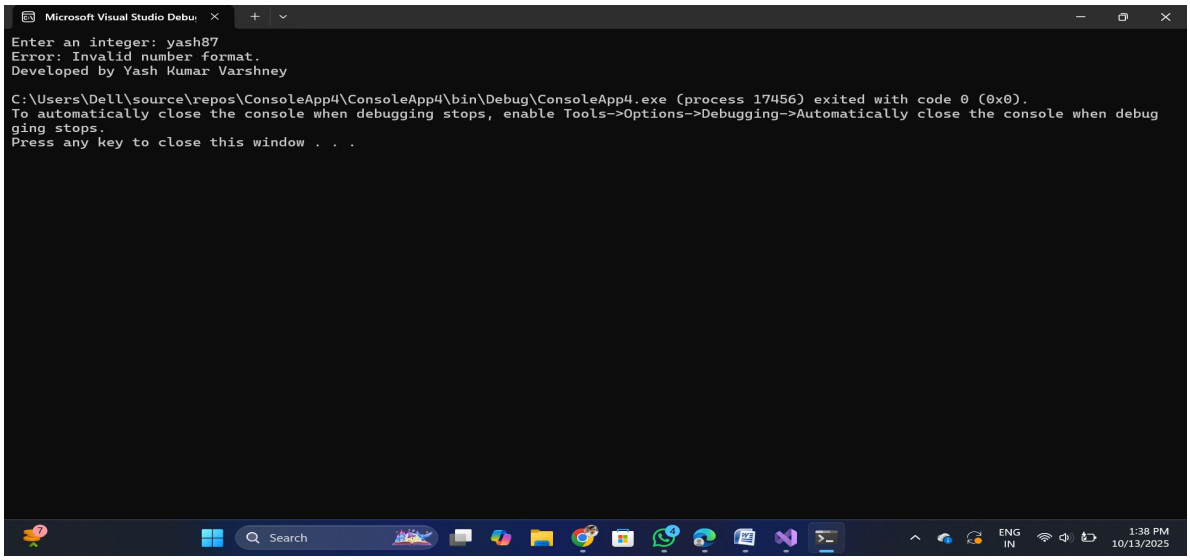
```
// Assignment 5 - Exception Handling
// Developed by Yash Kumar Varshney

using System;

class DivisionExample
{
   static void Main()
   {
      try
      {
         Console.Write("Enter first number: ");
         int num1 = Convert.ToInt32(Console.ReadLine());

         Console.Write("Enter second number: ");
         int num2 = Convert.ToInt32(Console.ReadLine());

         int result = num1 / num2;
         Console.WriteLine("Result: " + result);
      }
      catch (DivideByZeroException)
      {
         Console.WriteLine("Division by zero is not allowed.");
      }
      finally
      {
         Console.WriteLine("Execution completed.");
      }

      Console.WriteLine("Developed by Yash Kumar Varshney");
   }
}
```

Q2. Multiple Catch Blocks

Read console input and convert to int. Handle FormatException, OverflowException, and a generic Exception, with distinct messages.

Ans:-

```
using System;

class MultipleCatchDemo
{
   static void Main(String[] agrs)
   {
      try
      {
         Console.Write("Enter an integer: ");
         int num = Convert.ToInt32(Console.ReadLine());
         Console.WriteLine("Number entered: " + num);
      }
      catch (FormatException)
      {
         Console.WriteLine("Error: Invalid number format.");
      }
      catch (OverflowException)
      {
         Console.WriteLine("Error: Number is too large or too small.");
      }
      catch (Exception ex)
      {
         Console.WriteLine("General Error: " + ex.Message);
      }

            Console.WriteLine("Developed by Yash Kumar Varshney);

   }
}
```



Q3. Custom Exception —

NegativeSalaryException Define NegativeSalaryException : Exception. If entered salary < 0, throw it and handle with a clear error message.

Ans:-

```
using System;

class NegativeSalaryException : Exception
{
   public NegativeSalaryException(string message) : base(message) { }
}

class Employee
{
   public double Salary { get; set; }

   public void SetSalary(double salary)
   {
      if (salary < 0)
         throw new NegativeSalaryException("Salary cannot be negative!");
      else
         Salary = salary;
   }
}

class Program
{
   static void Main()
   {
      Employee emp = new Employee();

      try
      {
         Console.Write("Enter salary: ");
         double s = Convert.ToDouble(Console.ReadLine());
         emp.SetSalary(s);
         Console.WriteLine("Salary set successfully: " + emp.Salary);
      }
      catch (NegativeSalaryException ex)
      {
         Console.WriteLine("Error: " + ex.Message);
      }
      Console.WriteLine("Developed by Yash Kumar Varshney);

   }
}
```

```
Enter salary: -8000
Error: Salary cannot be negative!
Developed by Yash Kumar Varshney

C:\Users\Dell\source\repos\ConsoleApp4\ConsoleApp4\bin\Debug\ConsoleApp4.exe (process 18228) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

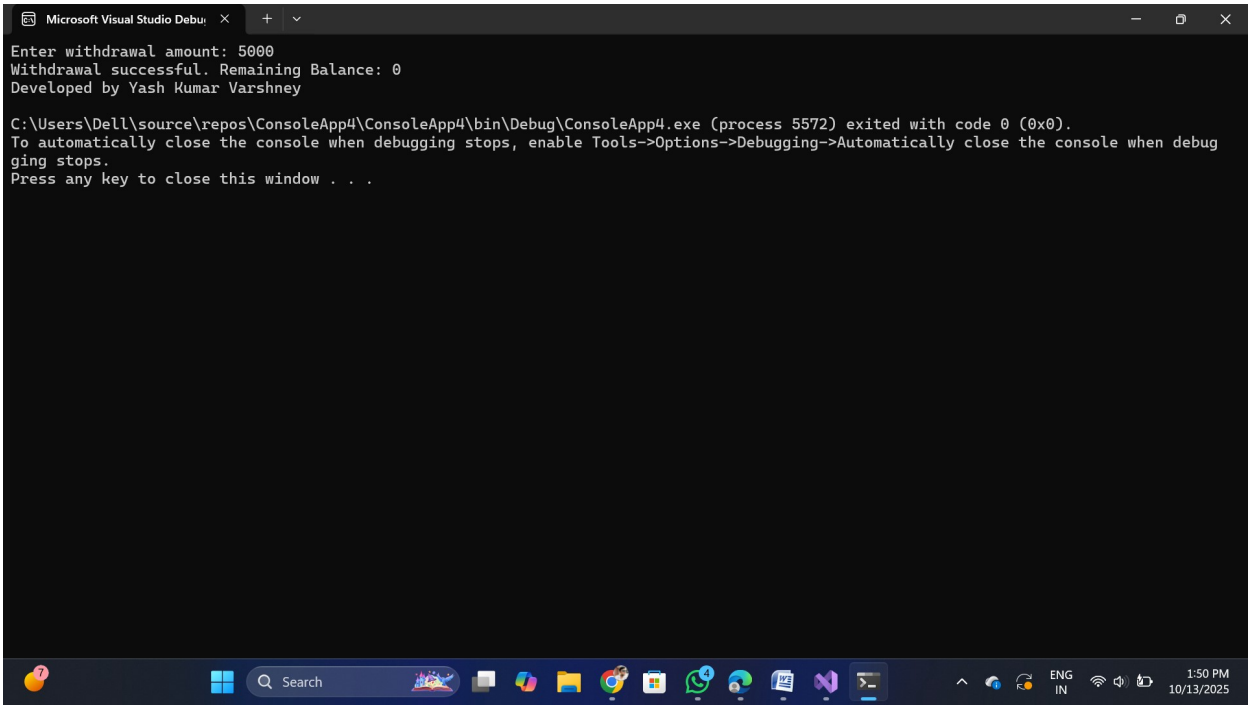
```
Enter salary: 60000
Salary set successfully: 60000
Developed by Yash Kumar Varshney

C:\Users\Dell\source\repos\ConsoleApp4\ConsoleApp4\bin\Debug\ConsoleApp4.exe (process 14856) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the conso
le when debugging stops.
Press any key to close this window . . .
```

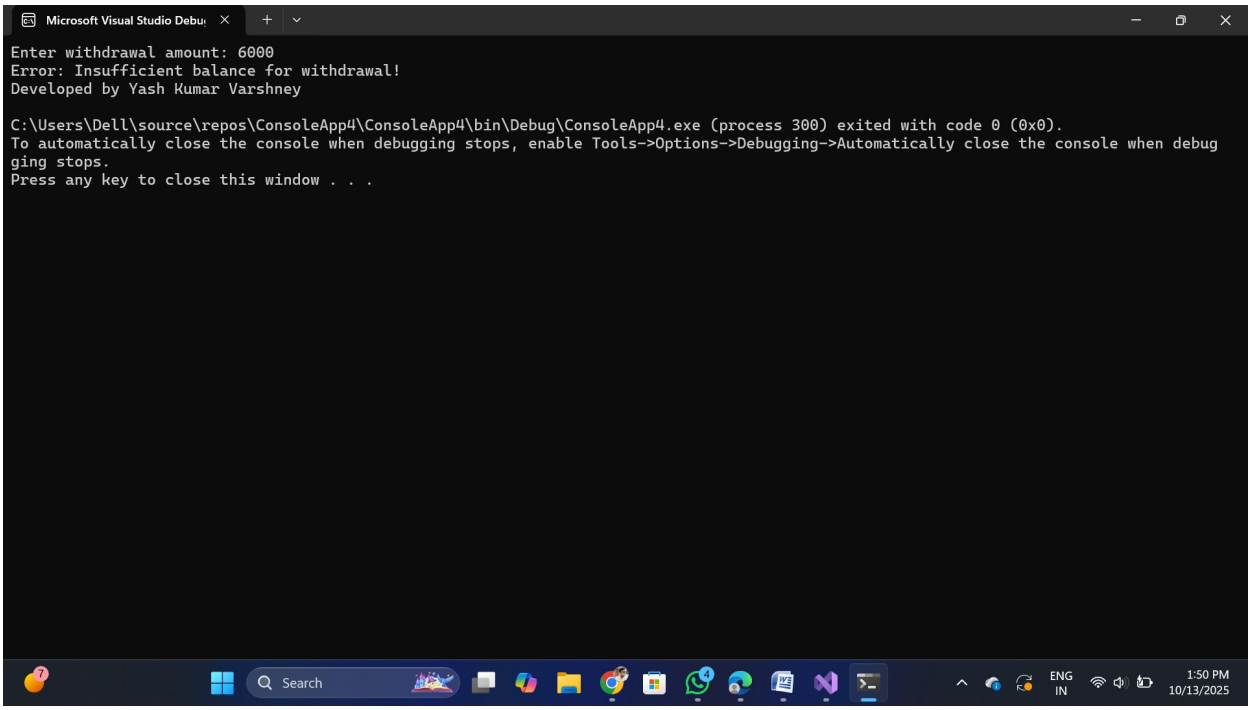
Q4. Banking Scenario —

InsufficientBalanceException Simulate withdrawal: if withdrawal > balance, throw custom InsufficientBalanceException; otherwise print remaining balance.

Ans:-

```
using System;

class InsufficientBalanceException : Exception
{
   public InsufficientBalanceException(string message) : base(message) { }
}

class BankAccount
{
   public double Balance { get; private set; }

   public BankAccount(double initialBalance)
   {
      Balance = initialBalance;
   }

   public void Withdraw(double amount)
   {
      if (amount > Balance)
         throw new InsufficientBalanceException("Insufficient balance for withdrawal!");
      else
      {
         Balance -= amount;
         Console.WriteLine("Withdrawal successful. Remaining Balance: " + Balance);
      }
   }
}

class Program
{
   static void Main()
   {
      BankAccount account = new BankAccount(5000);

      try
      {
         Console.Write("Enter withdrawal amount: ");
         double amount = Convert.ToDouble(Console.ReadLine());
         account.Withdraw(amount);
      }
      catch (InsufficientBalanceException ex)
      {
         Console.WriteLine("Error: " + ex.Message);
      }
      Console.WriteLine("Developed by Yash Kumar Varshney");
   }
}
```


Microsoft Visual Studio Debu
Enter withdrawal amount: 5000
Withdrawal successful. Remaining Balance: 0
Developed by Yash Kumar Varshney
C:\Users\Dell\source\repos\ConsoleApp4\ConsoleApp4\bin\Debug\ConsoleApp4.exe (process 5572) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debug
ging stops.
Press any key to close this window . . .
Search
ENG
IN
1:50 PM
10/13/2025


Microsoft Visual Studio Debu
Enter withdrawal amount: 6000
Error: Insufficient balance for withdrawal!
Developed by Yash Kumar Varshney
C:\Users\Dell\source\repos\ConsoleApp4\ConsoleApp4\bin\Debug\ConsoleApp4.exe (process 300) exited with code 0 (0x0).
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debug
ging stops.
Press any key to close this window . . .
Search
ENG
IN
1:50 PM
10/13/2025