Q-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.

The .NET Framework is a software development platform by Microsoft that provides a unified environment to build and run applications across various programming languages like C#, VB.NET, and F#. To help your colleague understand it clearly, I will break down its key components and how they work together:

1. Common Language Runtime (CLR)

- The CLR is the core execution engine of the .NET Framework.
- It manages running programs, handling:
    - Memory Management (automatic allocation and garbage collection)
    - Security (enforcing type safety and permissions)
    - Exception Handling (managing runtime errors)
    - Thread Management (controlling parallel operations)
    - Just-In-Time (JIT) Compilation converts intermediate code into machine code when needed.
- CLR enables cross-language interoperability, allowing components made in different supported languages to work together seamlessly.

2. Framework Class Library (FCL)

- The FCL is a comprehensive collection of reusable classes, interfaces, and APIs.
- It simplifies and accelerates application development by providing ready-made functionality such as:
    - File input/output
    - Database connectivity
    - Network communication
    - Graphical User Interface (GUI) construction
    - Web services support

3. Application Domains

- Application Domains provide isolation boundaries within the CLR.
- They allow multiple applications to run securely and independently within the same process.
- This helps in resource management and fault isolation.

Q-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?

1. Common Language Runtime (CLR):
   The CLR is the core runtime engine of the .NET Framework. It is responsible for executing .NET programs by converting the intermediate language (IL) code into machine code using Just-In-Time (JIT) compilation. CLR manages important services like automatic memory management (garbage collection), exception handling, type safety, security, and thread management. It also enables cross-language integration, allowing different .NET languages to work together seamlessly.

2. Common Type System (CTS):
   CTS defines a standardized set of data types and rules that all .NET languages must follow. This ensures that types declared in one language are compatible and understood in another, enabling smooth interaction and data exchange between components written in different .NET languages. For example, an int in C# is the same as an Integer in VB.NET as per CTS specifications.

3. Common Language Specification (CLS):
   CLS is a set of rules or guidelines that ensures language interoperability within the .NET ecosystem. It defines a subset of features that all .NET languages must support to ensure that code written in one language can be used by another. Writing CLS-compliant code promotes component reuse and seamless integration across different .NET languages.

Example of Integration in a .NET Application

- When code is written in C#, it is first compiled into an intermediate language (IL) following CTS rules, ensuring type compatibility.
- At runtime, the CLR compiles this IL into native machine code via the JIT compiler and manages memory and exceptions during execution.
- If the code is CLS-compliant, it can be used by any other .NET language without compatibility issues, enhancing teamwork and code reuse.

Q-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.

An assembly in the .NET Framework is the fundamental unit used to organize, deploy, and version an application. It is a compiled code library that contains a collection of types (classes, interfaces, etc.) and resources that logically work together to provide functionality.

Assemblies can be in the form of executable files (.exe) or libraries (.dll). Each assembly contains an assembly manifest, which includes metadata such as version information, security permissions, and references to other assemblies. This metadata helps the Common Language Runtime (CLR) manage execution, loading, and security.

Assemblies enable:

- Reusability: Code can be packaged and shared across multiple applications.

- Version control: Different versions of assemblies can coexist and be managed via the Global Assembly Cache (GAC).
- Deployment: Assemblies are the smallest deployable unit in .NET.
- Security and type safety: CLR uses assembly information to enforce security and proper type usage.

Example Scenario Using Multiple Assemblies

Consider a large-scale application divided into:

1. Core Library Assembly (CoreLib.dll): Contains common business logic and utility classes.
2. Data Access Assembly (DataAccess.dll): Manages database operations.
3. User Interface Assembly (UI.exe): The executable that handles user interactions.

Each assembly is separately developed and compiled. The UI assembly references the CoreLib and DataAccess assemblies to use their functionalities. These assemblies can be updated or replaced independently without requiring the entire application to be rebuilt. Shared assemblies can also be installed in the Global Assembly Cache if used by multiple applications.

This modular approach helps in organizing code, improving maintenance, facilitating team collaboration, and optimizing deployment.

Q-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?

Namespaces are containers used in .NET to organize and group related classes, interfaces, structs, and other types into logical units. They help avoid naming conflicts in large projects where many developers might create classes with identical names. By defining a namespace, you create a unique scope for identifiers to prevent collisions and improve code manageability.

Each class or type has a fully qualified name which includes its namespace and the class name, separated by a dot (.). For example, the class Console belongs to the namespace System, so its fully qualified name is System.Console.

You can simplify references in code by including the using directive (in C#) or Imports statement (in VB.NET), which lets you use class names without their full namespace every time.

How Namespaces Prevent Naming Conflicts

- Two different classes can have the same name as long as they belong to different namespaces.
- For example, two classes named ListBox can exist peacefully if one is in the System.Windows.Forms namespace and the other is in your project's custom namespace.
- When referencing such classes, the fully qualified namespace name disambiguates which ListBox is meant.

Example Usage in a Large Project

Suppose your project has multiple modules like:

1. Project.UI namespace for user interface related classes.
2. Project.Data namespace for data access and database classes.
3. Project.Services namespace for business logic services.

Each namespace contains classes with potentially overlapping names like Manager:

csharp

```
namespace Project.UI {
   class Manager {
      // UI-related manager code
   }
}
namespace Project.Data {
   class Manager {
      // Data-related manager code
   }
}
// Usage in code:
Project.UI.Manager uiManager = new Project.UI.Manager();
Project.Data.Manager dataManager = new Project.Data.Manager();
```

Here, both Manager classes coexist without conflict due to distinct namespaces. This makes the code modular, easier to understand, and maintain, especially when multiple developers work on different project parts.

Q-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?

Difference Between Primitive Types (Value Types) and Reference Types in .NET

1. Primitive Types (Value Types):

- These types directly hold their data in the memory allocated to the variable.
- Common examples include int, bool, float, char, and struct types.
- When you assign a value type variable to another, a copy of the actual data is made. Changing one variable does not affect the other.
- Value types are usually stored on the stack, which makes access faster.

2. Reference Types:

- Reference types store a reference (or pointer) to the actual data, which is stored in the heap memory.
- Examples include classes, arrays, strings, interfaces, and delegates.
- When you assign a reference type variable to another, both variables point to the same object in memory. Changing the object via one reference affects all references.
- Reference types support null as a value to indicate the absence of an instance.

Example to Illustrate the Difference

```
int a = 10;
int b = a;     // b gets a copy of a's value
b = 20;
// a is still 10, b is 20
class Person {
public string Name;
}
Person p1 = new Person();
p1.Name = "Alice";
Person p2 = p1; // p2 references same object as p1
p2.Name = "Bob";
// p1.Name is now "Bob" too
```

Q-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.

Difference Between Value Types and Reference Types in C#

1. Value Types:

- Value types directly store their data in memory.
- They are usually stored on the stack, which allows quick access and automatic memory management.
- When assigning one value type variable to another, a copy of the data is made. Changes to one variable do not affect the other.
- Examples include primitive types such as int, float, bool, char, and struct.

2. Reference Types:

- Reference types store a reference (address) to the actual data, which is located in the heap memory.

- When assigning a reference type variable to another, only the reference is copied, meaning both variables point to the same object. Changes through one reference affect the object seen by the other.
- Examples include classes, arrays, strings, interfaces, and delegates.

Memory Behavior

- Value types are stored on the stack, leading to faster access but limited scope and lifetime (tied to the method or block).
- Reference types are stored on the heap, allowing shared usage but requiring garbage collection for memory management.

Example:

```
// Value type example
int x = 10;
int y = x;  // y gets a copy of x
y = 20;
// x is still 10, y is 20
// Reference type example
class Person {
   public string Name;
}
Person p1 = new Person();
p1.Name = "Alice";
Person p2 = p1;  // p2 references same object as p1
p2.Name = "Bob";
// Now p1.Name is also "Bob"
```

Q-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.

Type conversion is the process of converting a value from one data type to another. In C#, there are two types of conversions:

1. Implicit Conversion:

- This happens automatically when converting a smaller data type to a larger compatible type.
- It is safe and does not result in loss of data.
- Example: converting int to double.

2. Explicit Conversion (Casting):

- Required when converting a larger or incompatible type to a smaller type.
- Must be done manually using a cast operator (type) because data loss or overflow might occur.
- Example: converting double to int.

Example Program Demonstrating Implicit and Explicit Conversio

```
using System;

class Program
{
    static void Main()
    {
        // Implicit Conversion: int to double
        int intValue = 100;
        double doubleValue = intValue;  // Automatic conversion
        Console.WriteLine("Implicit Conversion: int to double");
        Console.WriteLine($"intValue = {intValue}");
        Console.WriteLine($"doubleValue = {doubleValue}");
        // Explicit Conversion: double to int
        double originalDouble = 123.45;
        int convertedInt = (int)originalDouble;  // Manual cast required
        Console.WriteLine("Explicit Conversion: double to int");
        Console.WriteLine($"originalDouble = {originalDouble}");
        Console.WriteLine($"convertedInt = {convertedInt}");  // Decimal part lost
    }
}
```

Explanation:

- The implicit conversion from int to double happens automatically because double can represent all int values safely.
- The explicit conversion from double to int requires a cast (int) because converting from a larger to smaller type can lose data — here, the decimal part .45 is truncated.

Q-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.

```
using System;

class Program {
   static void Main() {
      Console.WriteLine("Enter a number: ");
      int number = Convert.ToInt32(Console.ReadLine());
      if (number > 0) {
         Console.WriteLine("Number is positive");
      } else if (number == 0) {
         Console.WriteLine("Number is zero");
      } else {
         Console.WriteLine("Number is negative");
      }
   }
}
```

Explanation of the Logic:

- The program first prompts the user to enter a number and reads it as an integer.
- The if statement checks whether the number is greater than zero. If yes, it prints "Number is positive".
- If the if condition is false, the else if checks whether the number is exactly zero. If yes, it prints "Number is zero".
- If neither of the above conditions is true, the else block executes, printing "Number is negative".

Q-9, You are explaining control flow constructs to a new hire. 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.

a switch-case statement is a control flow construct that allows you to execute different blocks of code based on the value of a single expression. It's often cleaner than using multiple if-else statements when checking for specific known values.

How it works:

1. Evaluate the switch expression – The expression inside the switch is computed once.

2. Match against case labels – Execution jumps to the first case whose value equals the switch expression.
3. Break statement – Terminates the current case to prevent “fall-through” (executing subsequent cases unintentionally).
4. Default case – Executes if none of the case labels match.

Example**:**

```
using System;

class Program

{

   static void Main()

   {

      Console.Write("Enter a number (1-5): ");

      int dayNumber = int.Parse(Console.ReadLine());

      switch (dayNumber)

      {

         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. Please enter a number between 1 and 5.");
```

```
            break;
        }
    }
}
```

Q-10, You are mentoring a developer on decision constructs in C#. 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).

```
using System;
class Program
{
    static void Main()
    {
        Console.Write("Enter a number: ");
        int number = int.Parse(Console.ReadLine());
        // Step 1: Check even/odd
        if (number % 2 == 0)
        {
            Console.WriteLine($"{number} is Even.");
        }
        else
        {
            Console.WriteLine($"{number} is Odd.");
        }
        // Step 2: Check range via switch-case
        int rangeID = 0;
        if (number >= 0 && number <= 10) rangeID = 1;
        else if (number >= 11 && number <= 20) rangeID = 2;
        switch (rangeID)
        {
            case 1:
                Console.WriteLine($"{number} falls in the range 0–10.");
                break;
```

```
            case 2:
                Console.WriteLine($"{number} falls in the range 11–20.");
                break;
            default:
                Console.WriteLine($"{number} is outside the 0–20 range.");
                break;
        }
    }
}
```

Q-11, During a live coding session, you are asked to write a program that prints the Fibonacci series using a for loop in C#. Provide a detailed explanation of your approach, and explain how the loop is used to generate the series.

Approach Explanation:

1. Initialize two variables to hold the starting values of the series: 0 (number1) and 1 (number2).
2. Print the first two numbers since they start the series.
3. Use a for loop starting from index 2, up to the desired length.
4. Inside the loop:
    - Calculate nextNumber as the sum of number1 and number2.
    - Print nextNumber.
    - Update number1 and number2 for the next iteration.

This loop iteratively builds the sequence by always adding the last two numbers.

Code:

```
using System;
class Program
{
    static void Main()
    {
        int count = 10; // Number of Fibonacci terms to print
        int number1 = 0, number2 = 1, nextNumber;
        Console.Write("Fibonacci Series: {0} {1} ", number1, number2);
        for (int i = 2; i < count; i++)
```

```
    {
        nextNumber = number1 + number2;
        Console.Write(nextNumber + " ");
        number1 = number2;
        number2 = nextNumber;
    }
  }
}
```

Q-12, You are leading a training session on loops in C#. Explain the key differences between while and do-while loops, and provide examples of each where one might be more appropriate than the other.

A while loop and a do-while loop are both used to execute a block of code repeatedly as long as a specified condition holds true. However, they differ in the point at which the condition is evaluated, which affects how many times the loop body executes in different scenarios.

While Loop

- It is an entry-controlled loop because the condition is evaluated *before* the loop body is executed.
- If the condition is false at the start, the loop body may not execute even once.
- Syntax does not require a semicolon after the closing brace.
- Suitable when the number of iterations is not known beforehand and the loop may need to be skipped if the condition is initially false.

General syntax:

```
while (condition)
{
  // Loop body statements
}
```

Do-While Loop

- It is an exit-controlled loop because the loop body is executed *once first*, and then the condition is tested.
- The loop body always executes at least once, regardless of whether the condition is true or false initially.
- Syntax requires a semicolon after the closing while parenthesis.
- Useful when the loop body must be executed at least once, such as in menu-driven programs or input validation.

General syntax:

```
do
{
   // Loop body statements
} while (condition);
```

Example of while loop:

```
int count = 0;
while (count < 5)
{
   Console.WriteLine(count);
   count++;
}
```

Example of do-while loop:

```
int count = 0;
do
{
   Console.WriteLine(count);
   count++;
} while (count < 5);
```

Q-13, You are developing a pattern generation tool for a project. 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.

```
using System;
class PyramidPattern
{
   static void Main()
   {
      Console.Write("Enter the number of rows for the pyramid: ");
      int n = int.Parse(Console.ReadLine());
      for (int i = 1; i <= n; i++)
      {
```

```
        // Print spaces
        for (int space = 1; space <= n - i; space++)
        {
            Console.Write(" ");
        }
        // Print stars
        for (int star = 1; star <= 2 * i - 1; star++)
        {
            Console.Write("*");
        }
        // Move to next line
        Console.WriteLine();
    }
  }
}
```

How the loops work:

- The outer loop iterates over each row (i from 1 to n), controlling the total number of lines.
- The first inner loop prints spaces before the stars. The count of spaces decreases each row to center the stars, typically printing (n - i) spaces.
- The second inner loop prints stars (*). The number of stars increases in each row and follows the formula (2 * i - 1) to ensure odd count, which creates a symmetrical pyramid.
- After the inner loops finish for a row, a newline is printed to move to the next line.

Each iteration builds one row of the pyramid by first printing spaces (to shift stars to the right place) and then printing stars.

Q-14, You are giving a presentation on object-oriented programming. Define Encapsulation, Inheritance, Polymorphism, and Abstraction, and provide real world examples of each in the context of C# development.

Here are the four core OOP concepts with definitions and real-world C# examples:

1. Encapsulation

Definition: Encapsulation is the practice of hiding an object's internal state and requiring all interaction to be performed through an object's methods. This protects data integrity by preventing external code from directly accessing or modifying it.

C# Example: Using private fields and public properties to control access to data.

```
public class BankAccount
{
    private double balance; // hidden from outside
    public double Balance
    {
        get { return balance; }
        private set { balance = value; } // only changed internally
    }
    public void Deposit(double amount)
    {
        if (amount > 0)
            balance += amount;
    }
    public void Withdraw(double amount)
    {
        if (amount > 0 && amount <= balance)
            balance -= amount;
    }
}
```

The internal balance is protected, and access happens only through controlled methods.

2. Inheritance

Definition: Inheritance allows a new class (child/derived) to acquire properties and behaviors (methods) from an existing class (parent/base), enabling code reuse and hierarchical class organization.

C# Example: A SavingsAccount class inherits from BankAccount, extending functionality.

```
public class SavingsAccount : BankAccount
{
    public double InterestRate { get; set; }

    public void AddInterest()
    {
        Deposit(Balance * InterestRate); // uses inherited Balance and Deposit
```

```
    }
}
```

SavingsAccount reuses and builds upon BankAccount.

3. Polymorphism

Definition: Polymorphism means the ability for different classes to be treated as instances of the same base class while invoking different implementations of the same method.

C# Example: Method overriding to process payments differently.

```
public abstract class Payment
{
    public abstract void ProcessPayment(double amount);
}
public class CreditCardPayment : Payment
{
    public override void ProcessPayment(double amount)
    {
        Console.WriteLine($"Processing credit card payment of {amount}.");
    }
}
public class PayPalPayment : Payment
{
    public override void ProcessPayment(double amount)
    {
        Console.WriteLine($"Processing PayPal payment of {amount}.");
    }
}
// Usage:
Payment payment = new CreditCardPayment();
payment.ProcessPayment(100);
payment = new PayPalPayment();
payment.ProcessPayment(200);
```

Each derived class provides its own specific version of ProcessPayment, demonstrating polymorphism.

4. Abstraction

Definition: Abstraction focuses on exposing only relevant details while hiding the implementation complexity. It models essential features of an object ignoring internal details.

Real-World Analogy: A mobile phone abstracts the functionality "call a number" without exposing the internal circuitry.

C# Example: Using abstract classes or interfaces to define simple contracts.

csharp

```
public abstract class Animal
{
   public abstract void MakeSound();
}
public class Dog : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Bark");
   }
}
public class Cat : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Meow");
   }
}
```

Users interact with Animal abstraction without worrying about the specific sound logic in each subclass.

Q-15, In a team discussion, you are asked to demonstrate the use of constructors and destructors in C#. Write a C# program that includes both, explaining the lifecycle of an object from creation to destruction.

In C#, constructors and destructors manage the lifecycle of an object:

- A constructor initializes an object at the moment it is created.

- A destructor performs cleanup just before the object is destroyed by the garbage collector.

Key Points:

- The constructor has the same name as the class and no return type.
- The destructor has the same name as the class preceded by a tilde ~ and cannot take parameters or have access modifiers.
- Constructors are called explicitly when new objects are created.
- Destructors are called automatically, non-deterministically by the .NET Garbage Collector when the object is no longer in use.

Example C# Program Demonstrating Both:

```
using System;

class Sample
{
    // Constructor
    public Sample()
    {
        Console.WriteLine("Constructor called: Object created.");
    }
    // Destructor
    ~Sample()
    {
        Console.WriteLine("Destructor called: Object is being destroyed.");
    }
}
class Program
{
    static void Main()
    {
        Sample obj = new Sample();
        // Use the object here
        Console.WriteLine("Inside Main method.");
        // Object goes out of scope here
    }
```

}

What happens when this runs:

1. new Sample() calls the constructor, printing "Constructor called: Object created."
2. The program prints "Inside Main method."
3. When obj is no longer referenced (after Main ends), the Garbage Collector eventually calls the destructor, printing "Destructor called: Object is being destroyed."

Q-16, A team member is confused about access modifiers in C#. How would you explain public, private, protected, and internal modifiers, and demonstrate their use by writing a small C# class with methods using different access levels?

Access modifiers control the visibility and accessibility of classes, methods, and members in C#. Here are the four basic ones:

- public: Accessible from anywhere, both inside and outside the class or assembly.
- private: Accessible only within the class where declared.
- protected: Accessible within the class and by derived (child) classes.
- internal: Accessible only within the same assembly/project but not outside it.

These modifiers help enforce encapsulation and define clear boundaries for how parts of your code can be used or extended.

C# Class Example Demonstrating Access Modifiers

```
using System;

class AccessDemo

{

   public void PublicMethod()

   {

      Console.WriteLine("Public method - accessible from anywhere.");

   }

   private void PrivateMethod()

   {

      Console.WriteLine("Private method - accessible only within this class.");

   }

   protected void ProtectedMethod()

   {

      Console.WriteLine("Protected method - accessible in this class and derived classes.");
```

```
    }
    internal void InternalMethod()
    {
        Console.WriteLine("Internal method - accessible within the same assembly.");
    }
    // Method to demonstrate calling the private method internally
    public void CallPrivateMethod()
    {
        PrivateMethod();
    }
}
// Derived class to demonstrate protected access
class DerivedDemo : AccessDemo
{
    public void CallProtectedMethod()
    {
        ProtectedMethod();  // Allowed because it is protected
    }
}

class Program
{
    static void Main()
    {
        AccessDemo demo = new AccessDemo();
        demo.PublicMethod();           // Works
        demo.InternalMethod();         // Works within same assembly
        demo.CallPrivateMethod();      // Private method accessed via a public method

        DerivedDemo derived = new DerivedDemo();
        derived.CallProtectedMethod(); // Accesses protected method via derived class method
```

```
        // The following are NOT allowed and will cause compile errors:
        // demo.PrivateMethod();       // Error: inaccessible due to protection level
        // demo.ProtectedMethod();     // Error: inaccessible from outside class or derived class
    }
}
```

Q-17, You are tasked with illustrating the concept of inheritance in C#. 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.

Inheritance allows a child class to reuse fields and methods from a parent class, promoting code reuse and easier maintenance.

In this example, Vehicle is the base class (parent), with common properties and methods for all vehicles. Car and Bike inherit from Vehicle and add their specific methods.

```
using System;
// Base class
class Vehicle
{
    public string Brand = "Generic Vehicle";

    public void Honk()
    {
        Console.WriteLine("Beep beep!");
    }
}
// Derived class Car inherits from Vehicle
class Car : Vehicle
{
    public void DisplayCarInfo()
    {
        Console.WriteLine($"Car Brand: {Brand}");
        Console.WriteLine("Car specific method: Opening sunroof.");
    }
```

```
// Derived class Bike inherits from Vehicle
class Bike : Vehicle
{
    public void DisplayBikeInfo()
    {
        Console.WriteLine($"Bike Brand: {Brand}");
        Console.WriteLine("Bike specific method: Kickstand deployed.");
    }
}
class Program
{
    static void Main()
    {
        Car myCar = new Car();
        myCar.Brand = "Toyota"; // inherited field
        myCar.Honk();           // inherited method
        myCar.DisplayCarInfo();  // own method
        Console.WriteLine();
        Bike myBike = new Bike();
        myBike.Brand = "Harley-Davidson"; // inherited field
        myBike.Honk();                  // inherited method
        myBike.DisplayBikeInfo();       // own method
    }
}
```

Q-18, In a bug-fixing scenario, your team needs to handle unexpected runtime errors. 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.

In C#, the try-catch-finally blocks are used to handle runtime errors (exceptions) gracefully.

- try block: Contains code that might throw an exception.
- catch block: Handles specific exceptions thrown in the try block.

- finally block: Contains code that always runs, regardless of whether an exception occurred or was caught, commonly used for cleanup.

How it works

1. The program enters the try block and tries to execute the code.
2. If an exception occurs, control jumps to the matching catch block.
3. After executing try or catch, the finally block always runs, even if no exception happened or a return statement was used.

Example: Handling an Arithmetic Exception (Divide by Zero)

```
using System;

class Program
{
   static void Main()
   {
      try
      {
         int numerator = 10;
         int denominator = 0; // will cause divide by zero exception
         int result = numerator / denominator;
         Console.WriteLine("Result: " + result);
      }
      catch (DivideByZeroException ex)
      {
         Console.WriteLine("Error: Cannot divide by zero.");
         Console.WriteLine("Exception message: " + ex.Message);
      }
      finally
      {
         Console.WriteLine("Finally block executed: Cleanup or final steps happen here.");
      }
   }
}
```

Q-19, You are implementing a custom exception for a specific error scenario in your application. Write a C# program that demonstrates exception handling by throwing and catching a custom exception, explaining why custom exceptions are beneficial.

Creating custom exceptions allows you to define specific error types tailored to your application's needs, improving clarity, maintainability, and error handling precision. Custom exceptions inherit from the built-in Exception class and can carry meaningful messages and additional properties.

Why use custom exceptions?

- They make error semantics clearer for developers and users.
- Allow more precise exception handling by differentiating error types.
- Support adding domain-specific data to exception objects.

Example: Defining and Using a Custom Exception

This example defines a NegativeValueException for cases where a negative value input is invalid.

```
using System;

// Define a custom exception inheriting from Exception
public class NegativeValueException : Exception
{
    public NegativeValueException() { }

    public NegativeValueException(string message) : base(message) { }

    public NegativeValueException(string message, Exception inner) : base(message, inner) { }
}
class Calculator
{
    public int SquareRoot(int number)
    {
        if (number < 0)
            throw new NegativeValueException("Input cannot be negative.");
        // Simple example, returns integer part
        return (int)Math.Sqrt(number);
    }
}
```

```
class Program
{
   static void Main()
   {
      Calculator calc = new Calculator();
      try
      {
         int result = calc.SquareRoot(-10);
         Console.WriteLine("Result: " + result);
      }
      catch (NegativeValueException ex)
      {
         Console.WriteLine("Custom Exception Caught: " + ex.Message);
      }
   }
}
```

Q-20, During a code quality meeting, you are asked to highlight the advantages of using exception handling in C#. Explain how proper exception handling improves application's robustness.

Proper exception handling improves the robustness and reliability of applications in several ways:

1. Prevents Unexpected Crashes

By catching runtime errors using try-catch blocks, applications avoid abrupt termination, maintaining service availability and a smooth user experience.

2. Improves Code Readability and Maintenance

Exception handling separates error-handling code from regular logic, making the program structure clearer and reducing clutter caused by repetitive error checks.

3. Supports Error Propagation and Centralized Handling

Exceptions automatically propagate up the call stack to methods that know how to handle them, eliminating the need for manual error code checks at every level, centralizing error management.

4. Enables Graceful Recovery

Applications can attempt recovery or provide fallback behaviour after an error occurs—for example, retrying operations, releasing resources, or showing user-friendly messages.

5. Ensures Cleanup with finally

The finally block guarantees the execution of cleanup code, such as closing files or database connections, regardless of success or failure.

6. Facilitates Debugging and Diagnostics

C# exceptions provide rich information, including stack traces and messages, which aid developers in diagnosing and fixing problems effectively.