.Net Framework Class Assignment

Session:2025-2026

SUBMITTED TO

Mr. Sachendra Singh Chauhan

SUBMITTED BY

Anita Kumari

MCA-II(C)

2484200026

C# Class Assignment

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.

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

- It's the core part that runs the program.
- Converts Intermediate Language (IL) code into machine code using JIT compiler.
- Handles memory management, garbage collection, security, and error handling.

2. FCL (Framework Class Library)

- A large collection of reusable classes and functions.
- Provides built-in support for file handling, data access, web services, and more.
- Saves time because developers don't need to write common code from scratch.

3. Application Domains

- Acts as an isolated environment where applications run inside the CLR.
- Prevents one app from affecting another.
- Improves security and stability when multiple apps run together.

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?

Ans: **Key .NET Runtime Concepts**

1. CLR (Common Language Runtime)

- The CLR is the execution engine of the .NET Framework.
- It manages how the program runs — converting Intermediate Language (IL) code into machine code using the JIT compiler.
- It also takes care of memory management, garbage collection, exception handling, and security.
- In short, CLR makes sure the application runs smoothly and safely.

2. CTS (Common Type System)

- The **CTS** defines how data types are declared and used in .NET.
- It ensures that all .NET languages (like C#, VB.NET, F#) share the same data types.
- For example, an int in C# and an Integer in VB.NET mean the same thing because of CTS.
- This helps different languages work together without data-type conflicts.

3. CLS (Common Language Specification)

- The **CLS** is a set of rules and guidelines that all .NET languages must follow.
- It ensures language interoperability — meaning code written in one .NET language can be used in another.
- Example: If you write a class in C#, it can be accessed in VB.NET as long as it follows CLS rules.

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.

Ans. **Assemblies in .NET Framework**

An assembly is the basic building block of a .NET application.
It's a compiled unit that contains code, resources, and metadata needed to run a program.
In simple words, an assembly is what you actually deploy and share — it's usually a .dll (Dynamic Link Library) or .exe file.

Key Points

- Each assembly has a manifest (contains information like version, culture, and referenced assemblies).
- Assemblies help in code organization, security, and version control.
- They allow you to reuse code easily by referencing DLLs in other projects.

Example Scenario

Suppose your team is building a large e-commerce application.
You can split it into multiple assemblies like this:

- UI.dll → handles the user interface
- BusinessLogic.dll → contains business rules and calculations
- DataAccess.dll → manages database operations
- ECommerce.exe → the main executable that connects all the above.

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?

Ans. **Namespaces in .NET Framework**

A namespace in .NET is used to organize classes, methods, and other code elements in a logical way.
Think of a namespace as a folder that groups related classes together so your code stays neat and easy to manage.

Why Namespaces Are Important

- They help avoid naming conflicts when different parts of a large project have classes with the same name.
- They make code more readable and maintainable.

- You can easily locate and reuse related code.

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?

Ans: **1. Primitive (Value) Types**

- These types store the actual value directly in memory.
- When you assign one variable to another, a copy of the value is made.
- Stored in the stack memory, which is faster.
- Examples: int, float, bool, char, double, struct

Example:

```
int a = 10;

int b = a;

b = 20;

Console.WriteLine(a);
```

2. Reference Types

- These types store a reference (address) to the actual data, not the value itself.
- When assigned to another variable, both variables refer to the same object in memory.
- Stored in the heap memory.
- Examples: class, array, string, interface, object

Example:

```
class Person { public string Name; }

Person p1 = new Person();

p1.Name = "John";

Person p2 = p1;

p2.Name = "David";
```

Console.WriteLine(p1.Name);

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.

Ans: 1. Value Types

- Store the actual value directly in memory.
- Stored in the stack, which makes them faster to access.
- When you assign one variable to another, a copy of the value is made.
- Changing one variable does not affect the other.
- Examples: int, float, bool, char, struct, enum.

Example:

int a = 10;

int b = a;

b = 20;

Console.WriteLine(a);

2. Reference Types

- Store a reference (memory address) of the actual data.
- The data itself is stored in the heap, while the reference is on the stack.
- When one variable is assigned to another, both point to the same object.
- Changing one variable's data affects the other.
- Examples: class, array, string, object, interface

Example:

class Person { public string Name; }

Person p1 = new Person();

p1.Name = "John";

Person p2 = p1;

```
p2.Name = "David";

Console.WriteLine(p1.Name);
```

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.

Ans: I**mplicit conversion** from int -> double (no data loss, no cast needed), and

Explicit conversion from double -> int

Code:

```
using System;

class ConversionDemo  {

   static void Main()  {

      int intValue = 42;

      double doubleFromInt = intValue; // implicit conversion, no cast required

      Console.WriteLine($"intValue (int): {intValue}");

      Console.WriteLine($"doubleFromInt (double, implicit): {doubleFromInt}");

      Console.WriteLine();

      double doubleValue = 42.79;

      int intFromDouble = (int)doubleValue; // explicit cast: fractional part truncated

      Console.WriteLine($"doubleValue (double): {doubleValue}");

      Console.WriteLine($"intFromDouble (int, explicit cast): {intFromDouble}");

      Console.WriteLine();

      double negative = -3.99;
```

```
        int truncatedNegative = (int)negative; // becomes -3 (fraction truncated
toward zero)

        Console.WriteLine($"negative (double): {negative} -> truncatedNegative
(int): {truncatedNegative}");    }

}
```

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.

Ans: using System;

```
class NumberCheck

{

    static void Main()

    {

        Console.Write("Enter a number: ");

        double number = Convert.ToDouble(Console.ReadLine());

        if (number > 0)

        {

            Console.WriteLine("The number is positive.");

        }

        else if (number < 0)

        {

            Console.WriteLine("The number is negative.");

        }

        else

        {

            Console.WriteLine("The number is zero.");

        }
    }
}
```

Logic Explanation

1. Read the input:
   - We ask the user to enter a number and convert it to double to handle both integers and decimals.
2. Check conditions using if-else:
   - if (number > 0) → If the number is greater than zero, it is positive.
   - else if (number < 0) → If the number is less than zero, it is negative.
   - else → If neither of the above, the number must be zero.
3. Output result:
   - The program prints the result based on which condition is true.

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.

Ans: The switch statement in C# is used to execute different blocks of code based on the value of a variable.

- It's an alternative to multiple if-else if statements.
- Each possible value is called a case.
- The default case runs if none of the other cases match.

Example:

```
using System;

class WeekdaySwitch

{

   static void Main()

   {      Console.Write("Enter a number (1-5) to get the weekday: ");
```

```
        int dayNumber = Convert.ToInt32(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 number. Please enter 1-5.");
                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).

Ans:

```
using System;
class NumberAnalysis
{
   static void Main()
   {
      Console.Write("Enter a number: ");
      int number = Convert.ToInt32(Console.ReadLine());
      if (number % 2 == 0)
      {
         Console.WriteLine("The number is even.");
      }
      else
      {
         Console.WriteLine("The number is odd.");
      }
      switch (number)
      {
         case int n when (n >= 0 && n <= 10):
            Console.WriteLine("The number is in the range 0-10.");
            break;
         case int n when (n >= 11 && n <= 20):
            Console.WriteLine("The number is in the range 11-20.");
            break;
         case int n when (n > 20):
            Console.WriteLine("The number is greater than 20.");
            break;
          default:
            Console.WriteLine("The number is negative.");
            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.

Ans: **Code:**

```
using System;
class FibonacciSeries
{
    static void Main()
    {
        Console.Write("Enter the number of terms: ");
        int n = Convert.ToInt32(Console.ReadLine());
        int first = 0, second = 1;
        Console.WriteLine("Fibonacci Series:");
        for (int i = 1; i <= n; i++)
        {
            Console.Write(first + " "); // print current term
            int next = first + second;
            first = second;    // move forward in the series
            second = next;
        }
    }
}
```

Step-by-Step Explanation

1. **Initialize first two numbers**
    - first = 0 and second = 1 → the starting numbers of the Fibonacci sequence.
2. **Use a for loop to generate terms**
    - Loop runs n times (the number of terms the user wants).
    - In each iteration:
        - Print the current number (first).
        - Calculate the next number as next = first + second.
        - Move forward:
            - first = second
            - second = next
3. **Result**
    - The loop continues to calculate and print the sequence until the required number of terms is generated.

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.

Ans: **While Loop**

- Checks condition before executing the loop body.
- May execute 0 times if condition is false.

```
int i = 1;

while (i <= 5)
```

```
{
   Console.WriteLine(i);
   i++;
}
```

Do-While Loop

- Executes the loop body first, then checks condition.
- Always executes at least once.

```
int number;
do
{
   Console.Write("Enter a positive number: ");
   number = Convert.ToInt32(Console.ReadLine());
} while (number <= 0);
Console.WriteLine("You entered: " + number);
```

Q13. 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.

Ans: using System;

```
class PyramidPattern
{
   static void Main()
   {
      Console.Write("Enter the number of rows for the pyramid: ");
      int rows = Convert.ToInt32(Console.ReadLine());

      for (int i = 1; i <= rows; i++)  // Outer loop: controls number of rows
      {
            for (int j = 1; j <= rows - i; j++)
            {
                Console.Write(" ");
            }
            for (int k = 1; k <= (2 * i - 1); k++)
            {
                Console.Write("*");
            }
            Console.WriteLine();  // Move to next line after each row
        }
    }
}
```

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.

Ans: **1. Encapsulation**

- **Definition:** Encapsulation is the practice of hiding the internal details of a class and exposing only what is necessary through methods or properties.
- **Purpose:** Protects data from unintended access or modification.

Example:

```
class BankAccount
{
    private double balance; // hidden from outside
    public void Deposit(double amount)
    {
        if(amount > 0)
        balance += amount;
    }
    public double GetBalance()
    {
        return balance;
    }
}
```

Real-world analogy: A **bank account** — you can deposit or withdraw money using provided methods, but you cannot directly change the balance from outside.

2. Inheritance

- **Definition:** Inheritance allows a class to **derive properties and methods from another class**, promoting code reuse.

Example:

```
class Vehicle
{
    public void Start() => Console.WriteLine("Vehicle started");
}
class Car : Vehicle // Car inherits Vehicle
{
    public void OpenTrunk() => Console.WriteLine("Trunk opened");
}
```

Real-world analogy: A **Car is a Vehicle**. It inherits general behavior like starting the engine, but also has specific features like opening the trunk.

3. Polymorphism

- **Definition:** Polymorphism allows objects to **take many forms**; methods can behave differently based on context.

Example:

```
class Animal

{

   public virtual void Speak() => Console.WriteLine("Animal speaks");

}

class Dog : Animal

{

   public override void Speak() => Console.WriteLine("Dog barks");

}

class Cat : Animal

{

   public override void Speak() => Console.WriteLine("Cat meows");

}
```

Real-world analogy: Animals speak differently — even though we call Speak() on all animals, the behavior depends on the actual type.

4. Abstraction

- **Definition:** Abstraction is the concept of **hiding complex implementation details** and exposing only essential functionality.

Example:

```
abstract class Shape

{

   public abstract double Area(); // abstract method

}

class Circle : Shape

{

   public double Radius { get; set; }

   public override double Area() => 3.14 * Radius * Radius;

}
```

Real-world analogy: Driving a car — you know how to accelerate or brake, but you don't need to know how the engine works internally.

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.

Ans: **Constructor:** A special method called automatically when an object is created. Used to initialize the object.

Destructor: A special method called automatically by the garbage collector when the object is no longer in use. Used for cleanup, such as releasing resources.

Example:

```
using System;
class Person
{
    public string Name;
    public Person(string name)
    {
        Name = name;
        Console.WriteLine($"Constructor called: {Name} object created.");
    }   ~Person()
    {
        Console.WriteLine($"Destructor called: {Name} object destroyed.");
    }
}
class Program
{
    static void Main()
    {
        Console.WriteLine("Program started.");
        Person p1 = new Person("Alice");
        Person p2 = new Person("Bob");
        Console.WriteLine("Objects are in use.");
        p1 = null;
        Console.WriteLine("Program ended.");
    }
}
```

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?

Ans: public → Accessible from anywhere (any class, any project).

private → Accessible only within the class it is defined.

protected → Accessible within the class and its derived classes.

internal → Accessible only inside the same project/assembly.

Example:

```
using System;
class Person
{
    public string Name;
    private int Age;
    protected string Gender;
    internal string City;
    public Person(string name, int age, string gender, string city)
    {
        Name = name;
        Age = age;
        Gender = gender;
        City = city;
    }
    private void ShowAge()
    {
        Console.WriteLine($"Age: {Age}");
    }
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, City: {City}");
        ShowAge(); // private method can be called inside class
    }
}
class Employee : Person
{
    public Employee(string name, int age, string gender, string city)
        : base(name, age, gender, city) { }

    public void ShowGender()
    {
        Console.WriteLine($"Gender: {Gender}"); // protected accessible
here
    }
}
class Program
{
    static void Main()
    {
        Person p = new Person("Alice", 25, "Female", "Mumbai");
        p.DisplayInfo();
        p.Name = "Bob";
        Employee e = new Employee("John", 30, "Male", "Delhi");
        e.ShowGender
    }
}
```

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.

```
using System;
class Vehicle
{
    public string Brand;
    public void Start() => Console.WriteLine($"{Brand} is starting.");
    public void Stop() => Console.WriteLine($"{Brand} is stopping.");
}
class Car : Vehicle
{
    public void OpenTrunk() => Console.WriteLine($"{Brand}'s trunk is 
opened.");
}
class Bike : Vehicle
{
    public void DoWheelie() => Console.WriteLine($"{Brand} is doing a
wheelie!");
}
class Program
{
    static void Main()
    {
        Car myCar = new Car();
        myCar.Brand = "Toyota";
        myCar.Start();
        myCar.OpenTrunk();
        myCar.Stop();
        Console.WriteLine();
        Bike myBike = new Bike();
        myBike.Brand = "Yamaha";
        myBike.Start();
        myBike.DoWheelie();
        myBike.Stop();
    }
}
```

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.

Ans: **try block** → Contains the code that might throw an exception.

catch block → Handles the exception if it occurs. You can catch specific exceptions like DivideByZeroException or use a general Exception.

finally block → Always executes regardless of whether an exception occurs. Useful for cleanup tasks.

Example:

```
using System;
class Program
{
    static void Main()
    {
        int numerator = 10;
        int denominator = 0;
        try
        {
            int result = numerator / denominator; // May throw DivideByZeroException
            Console.WriteLine("Result: " + result);
        }
        catch (DivideByZeroException ex)
        {
            Console.WriteLine("Error: Cannot divide by zero.");
            Console.WriteLine("Exception message: " + ex.Message);
        }
        finally
        {
            Console.WriteLine("This block always executes, cleaning up resources if needed.");
        }
        Console.WriteLine("Program continues after try-catch-finally.");
    }
}
```

Explanation

1. **try block**
   - Code that might throw an exception is placed here (numerator / denominator).
2. **catch block**
   - Catches the specific DivideByZeroException.
   - Allows you to handle the error gracefully without crashing the program.
3. **finally block**
   - Executes no matter what, even if no exception occurs or if a different exception is thrown.
   - Ideal for resource cleanup.

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.

Ans: **Why Use Custom Exceptions?**

- Provide more meaningful error messages specific to your application.
- Make error handling easier and more readable.
- Allow different types of errors to be handled separately.

Example:

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

```
class Program
{
    static void CheckPositive(int number)
    {
        if (number < 0)
        {
            throw new NegativeNumberException("Negative numbers are not
allowed: " + number);
        }
        else
        {
            Console.WriteLine("Number is positive: " + number);
        }
    }
    static void Main()
    {
        int[] numbers = { 10, -5, 20 };
        foreach (int num in numbers)
        {
            try
            {
                CheckPositive(num);
            }
            catch (NegativeNumberException ex)
            {
                Console.WriteLine("Custom Exception Caught: " + ex.Message);
            }
            finally
            {
                Console.WriteLine("Checked number: " + num);
                Console.WriteLine();
            }
        }
    }
}
```

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.

Ans: **Advantages of Exception Handling in C#**

1. **Prevents Application Crashes**
   - Without exception handling, unhandled errors terminate the program abruptly.
   - try-catch blocks allow the program to handle errors gracefully and continue running.
2. **Provides Meaningful Error Information**
   - Exceptions provide detailed error messages and stack traces, which help developers identify and fix problems quickly.
   - Example: DivideByZeroException clearly indicates a division by zero occurred.
3. **Centralizes Error Handling**
   - Exceptions can be caught in a centralized manner, avoiding repetitive checks throughout the code.
   - Example: Using global exception handlers in web or desktop applications.
4. **Improves Code Readability**

   - Using try-catch-finally separates normal program logic from error handling logic, making code easier to read and maintain.

5. **Ensures Cleanup with Finally**
   - finally blocks run regardless of whether an exception occurs, allowing resources like files, database connections, or network sockets to be safely released.

6. **Supports Custom Exceptions**
   - Custom exceptions allow domain-specific errors to be handled distinctly.
   - Example: NegativeNumberException or InvalidTransactionException.

7. **Enhances Application Robustness**
   - Proper exception handling ensures the application can recover from unexpected conditions instead of failing abruptly.
   - Increases user trust and prevents data corruption or loss.