**Class assignment of .NET Framework using C# (MCAE0402)**

**Name-Ritik Mittal**

**Roll No.- 33**

**MCA-2D**

**Q1. Imagine you are explaining the .NET framework architecture to a colleague unfamiliar with the framework. How would you break down the architecture and its components, such as the CLR, FCL, and the application domains? Provide a structured explanation.**

**Answer:**  
The **.NET Framework architecture** provides a managed environment for building and running applications.  
It consists of the following components:

1. **Common Language Runtime (CLR):**
   * The execution engine of .NET.
   * Handles memory management, garbage collection, thread management, and security.
   * Converts Intermediate Language (IL) into machine code using JIT (Just-In-Time) compilation.
2. **Framework Class Library (FCL):**
   * A collection of reusable classes, interfaces, and types.
   * Provides functionalities for file handling, database operations, XML, web services, etc.
3. **Application Domains:**
   * Logical boundaries for application execution.
   * Help isolate multiple applications running in a single process to prevent one crash from affecting others.
4. **Languages:**
   * .NET supports multiple languages like C#, VB.NET, and F#.
   * Source code is compiled into Intermediate Language (IL), which runs under the CLR.

**Q2. In a team meeting, you are asked to explain key .NET framework runtime concepts like the Common Language Runtime (CLR), Common Type System (CTS), and Common Language Specification (CLS). How would you present these to ensure clarity and relevance to the team's work?**

**Answer:**

1. **Common Language Runtime (CLR):**
   * Provides runtime services such as memory management, exception handling, and security.
   * Converts IL code into native machine code via JIT compiler.
2. **Common Type System (CTS):**
   * Defines how data types are declared and managed across all .NET languages.
   * Ensures type safety and language interoperability.
3. **Common Language Specification (CLS):**
   * A set of rules that all .NET languages must follow.
   * Ensures that code written in one .NET language can be used in another.

**Q3. You are developing a large-scale application and need to explain to a junior developer how assemblies are used in .NET framework to organize and deploy the application. Provide an explanation of assemblies and include an example scenario where multiple assemblies are used.**

**Answer:**

* An **Assembly** is the **building block** of a .NET application.
* It contains metadata, MSIL code, and resources.
* Assemblies are used for **code modularity, versioning, and deployment**.

**Types:**

1. **Private Assembly:** Used by one application only.
2. **Shared Assembly:** Can be shared among multiple applications (stored in the Global Assembly Cache).

**Example:**  
An E-commerce application might have:

* UI.dll (user interface)
* BusinessLogic.dll (logic)
* DataAccess.dll (database operations)  
  Each assembly can be updated or reused independently.

**Q4. In your project, you notice a developer struggling to organize classes and methods properly. How would you explain the concept of namespaces in .NET framework and demonstrate how they are used to avoid naming conflicts in large projects?**

**Answer:**

* A **namespace** organizes classes, methods, and other members logically.
* It avoids **naming conflicts** in large projects.

**Example:**

namespace Project.Module1

{

class Employee { }

}

namespace Project.Module2

{

class Employee { }

}

Both classes can coexist because they are under different namespaces.

**Q5. During a code review, a developer confuses primitive types with reference types in their application. How would you explain the difference between primitive types and reference types?**

**Answer:**

|  |  |
| --- | --- |
| **Primitive Types** | **Reference Types** |
| Store actual values directly. | Store reference (address) of data. |
| Stored in **stack** memory. | Stored in **heap** memory. |
| Faster access. | Slightly slower. |
| Examples: int, float, bool. | Examples: string, class, array. |

**Q6. While refactoring a piece of C# code, you notice both value types and reference types are being used incorrectly. Explain the difference between value types and reference types in C#, and provide examples to clarify their behaviour in memory.**

**Answer:**

* **Value Types:**  
  Store data directly on the stack.  
  Example:
* int x = 10;
* int y = x; // copy of value
* y = 20; // x remains 10
* **Reference Types:**  
  Store references to objects in heap memory.  
  Example:
* class Demo { public int num; }
* Demo a = new Demo();
* Demo b = a;
* b.num = 50; // affects both a and b

**Q7. You are tasked with creating a method that demonstrates both implicit and explicit type conversions. Write a program in C# that converts an int to a double implicitly and a double to an int explicitly, explaining each step in your code.**

**Answer:**

using System;

class Program

{

static void Main()

{

int num = 10;

double d = num; // Implicit conversion

Console.WriteLine("Implicit: " + d);

double val = 12.56;

int result = (int)val; // Explicit conversion

Console.WriteLine("Explicit: " + result);

}

}

**Explanation:**

* Implicit conversion happens automatically (no data loss).
* Explicit conversion (casting) is required when data might be lost.

**Q8. A junior developer asks for help writing a program to determine whether a number is positive, negative, or zero. Use if-else statements to write this program in C#, and explain the logic behind the code.**

**Answer:**

int num = Convert.ToInt32(Console.ReadLine());

if (num > 0)

Console.WriteLine("Positive");

else if (num < 0)

Console.WriteLine("Negative");

else

Console.WriteLine("Zero");

**Logic:**  
Sequentially checks conditions using if-else structure.

**Q9. 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.**

**Answer:**

int day = Convert.ToInt32(Console.ReadLine());

switch(day)

{

case 1: Console.WriteLine("Monday"); break;

case 2: Console.WriteLine("Tuesday"); break;

case 3: Console.WriteLine("Wednesday"); break;

case 4: Console.WriteLine("Thursday"); break;

case 5: Console.WriteLine("Friday"); break;

default: Console.WriteLine("Invalid Day"); break;

}

**Explanation:**  
Switch executes the block matching the case value and skips others.

**Q10. 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).**

**Answer:**

int num = 12;

if (num % 2 == 0)

{

Console.WriteLine("Even");

switch (num)

{

case <=10: Console.WriteLine("Range 0-10"); break;

case <=20: Console.WriteLine("Range 11-20"); break;

default: Console.WriteLine("Above 20"); break;

}

}

else

Console.WriteLine("Odd");

**Q11. 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.**

**Answer:**

int n = 10, a = 0, b = 1, c;

Console.Write(a + " " + b + " ");

for (int i = 2; i < n; i++)

{

c = a + b;

Console.Write(c + " ");

a = b;

b = c;

}

**Explanation:**  
The loop adds the previous two numbers to generate the next one.

**Q12. 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.**

**Answer:**

|  |  |
| --- | --- |
| **While Loop** | **Do-While Loop** |
| Checks condition **before** execution. | Checks condition **after** execution. |
| May not run even once. | Runs **at least once**. |

**Example:**

int i = 0;

while (i < 3)

{

Console.WriteLine(i);

i++;

}

int j = 0;

do

{

Console.WriteLine(j);

j++;

} while (j < 3);

**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.**

**Answer:**

for (int i = 1; i <= 5; i++)

{

for (int j = 5; j > i; j--)

Console.Write(" ");

for (int k = 1; k <= (2\*i-1); k++)

Console.Write("\*");

Console.WriteLine();

}

**Explanation:**  
Outer loop controls rows, and inner loops handle spaces and stars.

**Q14. 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.**

**Answer:**

1. **Encapsulation:**  
   Wrapping data and functions together.  
   Example: Private variables with public getters/setters.
2. **Inheritance:**  
   Reusing code from base class.  
   Example: class Car : Vehicle { }
3. **Polymorphism:**  
   One interface, multiple implementations.  
   Example: Method overriding.
4. **Abstraction:**  
   Hiding complex details, showing only essentials.  
   Example: Abstract class or Interface.

**Q15. 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.**

**Answer:**

class Demo

{

public Demo() { Console.WriteLine("Constructor called"); }

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

}

class Program

{

static void Main() { Demo obj = new Demo(); }

}

**Explanation:**  
Constructor initializes object, Destructor cleans up memory when object is destroyed.

**Q16. 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?**

**Answer:**

class AccessDemo

{

public void PublicMethod() { }

private void PrivateMethod() { }

protected void ProtectedMethod() { }

internal void InternalMethod() { }

}

|  |  |
| --- | --- |
| **Modifier** | **Access Level** |
| public | Anywhere |
| private | Within same class |
| protected | In derived classes |
| internal | Within same assembly |

**Q17. 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.**

**Answer:**

class Vehicle { public void Start() => Console.WriteLine("Vehicle starts"); }

class Car : Vehicle { public void Drive() => Console.WriteLine("Car drives"); }

class Bike : Vehicle { public void Ride() => Console.WriteLine("Bike rides"); }

class Program

{

static void Main()

{

Car c = new Car();

c.Start();

c.Drive();

}

}

**Explanation:**  
Derived classes reuse base class methods, reducing redundancy.

**Q18. 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.**

**Answer:**

try

{

int a = 10, b = 0;

int c = a / b;

}

catch (DivideByZeroException ex)

{

Console.WriteLine("Error: " + ex.Message);

}

finally

{

Console.WriteLine("Finally block executed");

}

**Explanation:**

* try: Code that might fail.
* catch: Handles exceptions.
* finally: Executes always, for cleanup.

**Q19. 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.**

**Answer:**

class NegativeNumberException : Exception

{

public NegativeNumberException(string msg) : base(msg) { }

}

class Program

{

static void Main()

{

try

{

int num = -5;

if (num < 0)

throw new NegativeNumberException("Number cannot be negative!");

}

catch (NegativeNumberException e)

{

Console.WriteLine(e.Message);

}

}

}

**Explanation:**  
Custom exceptions make error handling specific to business needs.

**Q20. 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.**

**Answer:**

1. Prevents abrupt application crashes.
2. Improves reliability and robustness.
3. Makes debugging easier.
4. Allows centralized error handling.
5. Ensures finally block executes for cleanup.