



# 75+ C# Coding Round

Interview Questions and Answers

Sai Reddy 5/23/25 Interview Tips

# 75+ C# Coding Round Interview Q&A

**C# 0-2 years exp**

**1. Reverse a string without using built-in methods**

**Understanding**: Reverse a string without using built-in methods like Reverse() or Array.Reverse().

```
public string ReverseString(string input)
{
   char[] chars = input.ToCharArray();
   string reversed = "";
   for (int i = chars.Length - 1; i >= 0; i--)
   {
      reversed += chars[i];
   }
   return reversed;
}
```

**Explanation**: We convert the string into a character array, then iterate backward and build a new string character by character.

**2. Check if a string is a palindrome**

**Understanding**: Check if the given string is a palindrome (same forwards and backwards).

```
public bool IsPalindrome(string input)
{
   int start = 0, end = input.Length - 1;
   while (start < end)
   {
      if (input[start] != input[end])
         return false;
      start++;
      end--;
   }
   return true;
}
```

**Explanation**: We use two pointers, one at the beginning and one at the end, and compare characters while moving inward.

### 3. Count the number of vowels in a string

**Understanding**: Count the total number of vowels (a, e, i, o, u) in a string.

```
public int CountVowels(string input)
{
   int count = 0;
   for (int i = 0; i < input.Length; i++)
   {
      char ch = input[i];
      if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u' ||
         ch == 'A' || ch == 'E' || ch == 'I' || ch == 'O' || ch == 'U')
      {
         count++;
      }
   }
   return count;
}
```

**Explanation**: We iterate through each character and increment a counter if it's a vowel.

### 4. Swap two numbers without using a third variable

**Understanding**: Swap the values of two integers without using a third variable.

```
public void SwapWithoutTemp(ref int a, ref int b)
{
   a = a + b;
   b = a - b;
   a = a - b;
}
```

**Explanation**: We use arithmetic operations to exchange values without needing a temporary variable.

### 5. Print Fibonacci series up to N terms

**Understanding**: Print the first N terms of the Fibonacci series.

```
public void PrintFibonacci(int n)
{
   int a = 0, b = 1;
   for (int i = 0; i < n; i++)
   {
      Console.Write(a + " ");
      int temp = a + b;
      a = b;
```

```
        b = temp;
    }
}
```

**Explanation**: We use a loop to generate and print Fibonacci numbers by summing the last two numbers.

### 6. Find factorial using recursion

**Understanding**: Write a recursive function to calculate the factorial of a number.

```
public int Factorial(int n)
{
    if (n <= 1) return 1;
    return n * Factorial(n - 1);
}
```

**Explanation**: We use recursion where each function call multiplies the number with the factorial of the previous number.

### 7. Print prime numbers from 1 to N

**Understanding**: Print all prime numbers between 1 and N.

```
public void PrintPrimes(int n)
{
    for (int i = 2; i <= n; i++)
    {
        bool isPrime = true;
        for (int j = 2; j * j <= i; j++)
        {
            if (i % j == 0)
            {
                isPrime = false;
                break;
            }
        }
        if (isPrime)
            Console.Write(i + " ");
    }
}
```

**Explanation**: For each number from 2 to N, we check divisibility up to square root to determine if it's prime.

## 8. Check Armstrong number

**Understanding**: Check whether a number is an Armstrong number (e.g., 153 = 1^3 + 5^3 + 3^3).

```
public bool IsArmstrong(int num)
{
   int temp = num, sum = 0;
   int digits = 0;
   int n = num;
   while (n > 0) { digits++; n /= 10; }
   while (temp > 0)
   {
      int rem = temp % 10;
      int pow = 1;
      for (int i = 0; i < digits; i++) pow *= rem;
      sum += pow;
      temp /= 10;
   }
   return sum == num;
}
```

**Explanation**: We extract each digit and raise it to the power equal to number of digits, summing the results to check equality.

## 9. Check for leap year

**Understanding**: Determine if a given year is a leap year.

```
public bool IsLeapYear(int year)
{
   return (year % 4 == 0 && year % 100 != 0) || (year % 400 == 0);
}
```

**Explanation**: We apply the standard rules to check for a leap year.

## 10. Basic calculator using switch-case

**Understanding**: Implement a basic calculator supporting +, -, *, /

```
public double Calculator(double a, double b, char op)
{
   switch (op)
   {
      case '+': return a + b;
      case '-': return a - b;
```

```
        case '*': return a * b;
        case '/': return b != 0 ? a / b : 0;
        default: return 0;
    }
}
```

**Explanation**: We use a switch-case statement to perform arithmetic operations based on the operator.

### 11. Reverse a number

```
public int ReverseNumber(int num)
{
    int reversed = 0;
    while (num > 0)
    {
        int digit = num % 10;
        reversed = reversed * 10 + digit;
        num /= 10;
    }
    return reversed;
}
```

**Explanation**: We extract digits from the number using modulus and rebuild it in reverse using multiplication.

### 12. Find the largest of three numbers

```
public int LargestOfThree(int a, int b, int c)
{
    if (a >= b && a >= c) return a;
    else if (b >= a && b >= c) return b;
    else return c;
}
```

**Explanation**: We use conditional statements to compare values and determine the largest one.

### 13. Sort an array in ascending and descending order

```
public void SortArray(int[] arr, bool ascending = true)
{
    for (int i = 0; i < arr.Length - 1; i++)
    {
```

```
        for (int j = i + 1; j < arr.Length; j++)
        {
            if ((ascending && arr[i] > arr[j]) || (!ascending && arr[i] < arr[j]))
            {
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

**Explanation**: We use nested loops to sort by swapping elements based on ascending or descending logic.

### 14. Remove duplicates from an array

```
public int[] RemoveDuplicates(int[] arr)
{
    int[] temp = new int[arr.Length];
    int index = 0;
    for (int i = 0; i < arr.Length; i++)
    {
        bool exists = false;
        for (int j = 0; j < index; j++)
        {
            if (arr[i] == temp[j])
            {
                exists = true;
                break;
            }
        }
        if (!exists)
        {
            temp[index++] = arr[i];
        }
    }
    int[] result = new int[index];
    for (int i = 0; i < index; i++) result[i] = temp[i];
    return result;
}
```

**Explanation**: We build a new array with only unique values using nested loops.

### 15. Find the second highest number in an array

```
public int SecondHighest(int[] arr)
{
   int highest = int.MinValue, second = int.MinValue;
   for (int i = 0; i < arr.Length; i++)
   {
      if (arr[i] > highest)
      {
         second = highest;
         highest = arr[i];
      }
      else if (arr[i] > second && arr[i] != highest)
      {
         second = arr[i];
      }
   }
   return second;
}
```

**Explanation**: We track two variables while scanning the array to get the second highest distinct value.

### 16. Check if two strings are anagrams

**Understanding**: Check whether two strings are anagrams of each other.

```
public bool AreAnagrams(string str1, string str2)
{
   if (str1.Length != str2.Length) return false;

   int[] count = new int[256];
   for (int i = 0; i < str1.Length; i++)
   {
      count[str1[i]]++;
      count[str2[i]]--;
   }
   for (int i = 0; i < 256; i++)
   {
      if (count[i] != 0) return false;
   }
```

```
    return true;
}
```

**Explanation**: We count character frequencies for both strings and compare the frequency arrays.

## 17. Count the frequency of characters in a string

Understanding: Count how many times each character appears in a string.

```
public void CharacterFrequency(string input)
{
    int[] freq = new int[256];
    for (int i = 0; i < input.Length; i++)
    {
        freq[input[i]]++;
    }
    for (int i = 0; i < 256; i++)
    {
        if (freq[i] > 0)
        {
            Console.WriteLine((char)i + ": " + freq[i]);
        }
    }
}
```

**Explanation**: We use an integer array to track frequency of each character.

## 18. Implement bubble sort manually

**Understanding**: Sort an array using the bubble sort algorithm.

```
public void BubbleSort(int[] arr)
{
    for (int i = 0; i < arr.Length - 1; i++)
    {
        for (int j = 0; j < arr.Length - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
```

```
    }
}
```

**Explanation**: We repeatedly swap adjacent elements if they are in the wrong order.

## 19. Use a Dictionary to store and display key-value pairs

**Understanding**: Demonstrate how to store and display key-value pairs using Dictionary.

```
public void DisplayDictionary()
{
    Dictionary<string, int> dict = new Dictionary<string, int>();
    dict["Apple"] = 10;
    dict["Banana"] = 20;
    dict["Cherry"] = 15;

    foreach (var item in dict)
    {
        Console.WriteLine(item.Key + ": " + item.Value);
    }
}
```

**Explanation**: We use Dictionary to map keys to values and loop through it using foreach.

## 20. Demonstrate simple class and object usage in C#

Understanding: Create a class with a method and demonstrate object instantiation.

```
public class Person
{
    public string Name;
    public void Greet()
    {
        Console.WriteLine("Hello, my name is " + Name);
    }
}

public void UsePersonClass()
{
    Person p = new Person();
    p.Name = "John";
    p.Greet();
}
```

**Explanation**: We define a class with a method, instantiate an object, and call the method.

## 21. Create a method that accepts parameters and returns a value

**Understanding**: Write a method that takes two integers and returns their sum.

```
public int AddNumbers(int a, int b)
{
   return a + b;
}
```

**Explanation**: This method simply returns the result of adding two parameters.

## 22. Explain and use `ref` and `out` keywords

**Understanding**: Show difference between ref and out using examples.

```
public void RefExample(ref int a)
{
   a += 10;
}

public void OutExample(out int a)
{
   a = 10;
   a += 20;
}
```

**Explanation**: `ref` requires the variable to be initialized before passing; `out` does not.

## 23. Demonstrate exception handling using try-catch-finally

**Understanding**: Show how to handle exceptions in C#.

```
public void HandleException()
{
   try
   {
      int x = 10;
      int y = 0;
      int z = x / y;
   }
   catch (DivideByZeroException ex)
   {
      Console.WriteLine("Cannot divide by zero: " + ex.Message);
   }
   finally
   {
      Console.WriteLine("Finally block executed.");
```

```
    }
}
```

**Explanation**: We catch divide-by-zero exception and use finally to ensure cleanup.

### 24. Implement a basic class with properties and methods

**Understanding**: Create a class with properties and a method that displays data.

```
public class Car
{
    public string Make { get; set; }
    public int Year { get; set; }

    public void DisplayInfo()
    {
        Console.WriteLine($"Car Make: {Make}, Year: {Year}");
    }
}
```

**Explanation**: Properties are auto-implemented and used to store and display values.

### 25. Use loops and conditionals to draw patterns (e.g., star pyramid)

**Understanding**: Draw a pyramid pattern of stars using nested loops.

```
public void DrawPyramid(int rows)
{
    for (int i = 1; i <= rows; i++)
    {
        for (int j = 1; j <= rows - i; j++)
            Console.Write(" ");
        for (int k = 1; k <= 2 * i - 1; k++)
            Console.Write("*");
        Console.WriteLine();
    }
}
```

**Explanation**: We use nested loops to print spaces and stars to form a pyramid.

## 3-6 Years exp QA

### 1. Implement custom sorting using IComparer

**Understanding**: Sort a list of custom objects using IComparer

```
class Person
{
```

```
    public string Name;
    public int Age;
}
class AgeComparer : IComparer<Person>
{
    public int Compare(Person x, Person y) => x.Age.CompareTo(y.Age);
}
```

**Explanation**: IComparer allows custom logic for sorting by defining Compare().

### 2. Reverse words in a sentence

```
public string ReverseWords(string sentence)
{
    string[] words = sentence.Split(' ');
    string reversed = "";
    for (int i = words.Length - 1; i >= 0; i--)
        reversed += words[i] + " ";
    return reversed.Trim();
}
```

**Explanation**: Split the sentence and loop in reverse to rebuild it.

### 3. Serialize and deserialize an object using JSON

```
class Car { public string Model; public int Year; }
public string Serialize(Car car) => JsonSerializer.Serialize(car);
public Car Deserialize(string json) => JsonSerializer.Deserialize<Car>(json);
```

**Explanation**: JsonSerializer handles conversion to and from JSON.

### 4. Create a custom exception class

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

**Explanation**: Inherit from Exception and pass message to base class.

### 5. Basic Stack and Queue

```
Stack<int> stack = new Stack<int>(); stack.Push(1); int val = stack.Pop();
Queue<int> queue = new Queue<int>(); queue.Enqueue(1); int qVal = queue.Dequeue();
```

**Explanation**: Stack follows LIFO, Queue follows FIFO.

### 6. Use LINQ to filter and group

```
var data = new[] { new { Name = "A", Group = "X" }, new { Name = "B", Group = "Y" }, new
{ Name = "C", Group = "X" } };
var grouped = data.GroupBy(d => d.Group);
```

**Explanation**: GroupBy groups data based on a key.

### 7. Flatten nested list using recursion

```
List<object> Flatten(List<object> nested)
{
   List<object> flat = new();
   foreach (var item in nested)
   {
      if (item is List<object> sublist)
         flat.AddRange(Flatten(sublist));
      else
         flat.Add(item);
   }
   return flat;
}
```

**Explanation**: Recursively extract items from nested lists.

### 8. Singleton pattern

```
class Singleton
{
   private static Singleton instance;
   private Singleton() { }
   public static Singleton Instance => instance ??= new Singleton();
}
```

**Explanation**: Only one instance exists and is lazily created.

### 9. Extension method for string

```
public static class StringExtensions
{
   public static bool IsCapitalized(this string str) => str.Length > 0 && char.IsUpper(str[0]);
}
```

**Explanation**: Extends string with custom method using 'this'.

### 10. Intersection of arrays

```
public int[] IntersectArrays(int[] a, int[] b)
{
   List<int> result = new();
   foreach (var item in a)
      if (b.Contains(item) && !result.Contains(item))
         result.Add(item);
   return result.ToArray();
}
```

**Explanation**: Check presence in both arrays manually.

### 11. Implement simple CRUD using a List of objects

```
class Student
{
   public int Id;
   public string Name;
}
class StudentRepo
{
   private List<Student> students = new List<Student>();

    Create
   public void Add(Student s) => students.Add(s);

    Read
   public Student GetById(int id)
   {
      foreach (var s in students)
         if (s.Id == id)
            return s;
      return null;
   }

    Update
   public bool Update(int id, string newName)
   {
      foreach (var s in students)
```

```
        {
            if (s.Id == id)
            {
                s.Name = newName;
                return true;
            }
        }
        return false;
    }

     Delete
    public bool Delete(int id)
    {
        for (int i = 0; i < students.Count; i++)
        {
            if (students[i].Id == id)
            {
                students.RemoveAt(i);
                return true;
            }
        }
        return false;
    }
}
```

**Explanation**: We use List<T> and implement CRUD by searching and modifying elements manually.

## 12. Calculate occurrences of each word in a paragraph

```
public Dictionary<string, int> WordCount(string paragraph)
{
    Dictionary<string, int> counts = new Dictionary<string, int>();
    string[] words = paragraph.ToLower().Split(new char[] { ' ', '.', ',', ';', '!' },
StringSplitOptions.RemoveEmptyEntries);
    foreach (string w in words)
    {
        if (counts.ContainsKey(w)) counts[w]++;
        else counts[w] = 1;
    }
    return counts;
```

```
}
```

**Explanation**: Split text into words and count frequency using a dictionary.

### 13. Boxing and unboxing examples

```
public void BoxingUnboxing()
{
   int i = 123;          value type
   object o = i;          boxing: value copied to heap
   int j = (int)o;        unboxing: cast back to value type
}
```

**Explanation**: Boxing wraps a value type into an object; unboxing extracts it back.

### 14. Use async-await for I/O bound operation

```
public async Task<string> ReadFileAsync(string path)
{
   using StreamReader reader = new StreamReader(path);
   string content = await reader.ReadToEndAsync();
   return content;
}
```

**Explanation**: async-await allows asynchronous non-blocking file reading.

### 15. Demonstrate Dependency Injection with a sample program

```
interface IMessageService
{
   void Send(string message);
}
class EmailService : IMessageService
{
   public void Send(string message) => Console.WriteLine("Email sent: " + message);
}
class Notification
{
   private IMessageService _service;
   public Notification(IMessageService service) => _service = service;
   public void Notify(string msg) => _service.Send(msg);
}
```

**Explanation**: We inject dependency via constructor for loose coupling.

## 16. Implement caching in console application

```
class Cache<T>
{
   private Dictionary<string, T> cache = new Dictionary<string, T>();

   public void Add(string key, T value) => cache[key] = value;

   public bool TryGetValue(string key, out T value) => cache.TryGetValue(key, out value);
}
```

**Explanation**: Simple in-memory cache using Dictionary.

## 17. Use Task.Run() and show how to cancel tasks

```
CancellationTokenSource cts = new CancellationTokenSource();
Task task = Task.Run(() =>
{
   for (int i = 0; i < 1000; i++)
   {
      if (cts.Token.IsCancellationRequested) break;
      Console.WriteLine(i);
      Thread.Sleep(10);
   }
}, cts.Token);
```

To cancel:

```
cts.Cancel();
```

**Explanation**: We pass CancellationToken to Task.Run to allow cancelling.

## 18. Create a class with indexers

```
class SampleCollection
{
   private string[] data = new string[10];
   public string this[int index]
   {
      get => data[index];
      set => data[index] = value;
   }
}
```

**Explanation**: Indexers provide array-like access to objects.

### 19. Convert decimal to binary

```
public string DecimalToBinary(int num)
{
   if (num == 0) return "0";
   string binary = "";
   while (num > 0)
   {
      binary = (num % 2) + binary;
      num /= 2;
   }
   return binary;
}
```

**Explanation**: We repeatedly divide by 2 and prepend remainders.

### 20. Detect if a string contains only digits

```
public bool IsDigitsOnly(string str)
{
   foreach (char c in str)
      if (c < '0' || c > '9') return false;
   return true;
}
```

**Explanation**: Check every character falls within digit range.

### 21. Find duplicate elements in an array using HashSet

```
public List<int> FindDuplicates(int[] arr)
{
   HashSet<int> seen = new HashSet<int>();
   List<int> duplicates = new List<int>();
   foreach (int num in arr)
   {
      if (!seen.Add(num))  returns false if num already in set
         duplicates.Add(num);
   }
   return duplicates;
}
```

**Explanation**: HashSet.Add returns false if item is duplicate.

## 22. Build simple in-memory database with search

```
class SimpleDB
{
   private List<Student> data = new List<Student>();
   public void Add(Student s) => data.Add(s);
   public List<Student> SearchByName(string name)
   {
      List<Student> result = new List<Student>();
      foreach (var s in data)
         if (s.Name.Contains(name, StringComparison.OrdinalIgnoreCase))
            result.Add(s);
      return result;
   }
}
```

**Explanation**: Store and search in-memory list with manual string matching.

## 23. Create and use custom attributes

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
class DeveloperAttribute : Attribute
{
   public string Name;
   public DeveloperAttribute(string name) => Name = name;
}

[Developer("John Doe")]
class SomeClass { }
```

**Explanation**: Custom attributes add metadata and can be retrieved via reflection.

## 24. Difference between abstract class and interface

Abstract class: can have implemented methods, fields; supports inheritance with shared code.

Interface: only method/property signatures; supports multiple inheritance.

Code example:

```
abstract class Animal
{
   public abstract void MakeSound();
   public void Sleep() { Console.WriteLine(\"Sleeping\"); }
```

```
}
interface IFlyable
{
   void Fly();
}
```

**Explanation**: Abstract class is base with partial implementation; interface is contract only.

**25. Build basic REST API using ASP.NET Core Web API**

Example Controller:

```
[ApiController]
[Route(\"api/[controller]\")]
public class ProductsController : ControllerBase
{
   private static List<string> products = new List<string> { \"Apple\", \"Banana\" };

   [HttpGet]
   public IEnumerable<string> Get() => products;

   [HttpPost]
   public IActionResult Post([FromBody] string product)
   {
      products.Add(product);
      return Ok();
   }
}
```

**Explanation**: Defines API endpoints for GET and POST using attributes and routing.

**7-15 years Exp Architect Level**

**1. Design and implement a thread-safe singleton**

```
public sealed class ThreadSafeSingleton
{
   private static readonly Lazy<ThreadSafeSingleton> _instance =
      new Lazy<ThreadSafeSingleton>(() => new ThreadSafeSingleton());

   private ThreadSafeSingleton() { }

   public static ThreadSafeSingleton Instance => _instance.Value;
}
```

**Explanation:**

- We use Lazy<T> to ensure thread-safe lazy initialization without explicit locks.
- sealed prevents subclassing, ensuring a single instance.
- Access via Instance property guarantees only one instance in the app domain.

**2. Implement a producer-consumer pattern using BlockingCollection**

```
BlockingCollection<int> buffer = new BlockingCollection<int>(boundedCapacity: 5);

void Producer()
{
   for (int i = 0; i < 20; i++)
   {
      buffer.Add(i);
      Console.WriteLine($"Produced: {i}");
      Thread.Sleep(100);
   }
   buffer.CompleteAdding();
}

void Consumer()
{
   foreach (var item in buffer.GetConsumingEnumerable())
   {
      Console.WriteLine($"Consumed: {item}");
      Thread.Sleep(150);
   }
}
```

**Explanation:**

- BlockingCollection<T> is a thread-safe collection for producer-consumer scenarios.
- Producers add items, blocking if full; consumers take items, blocking if empty.
- CompleteAdding() signals no more items.

**3. Create a custom LINQ operator**

```
public static class LinqExtensions
{
   public static IEnumerable<T> TakeEvery<T>(this IEnumerable<T> source, int step)
   {
      int index = 0;
      foreach (var item in source)
      {
         if (index % step == 0)
```

```
            yield return item;
        index++;
      }
   }
}
```

**Explanation:**

- This extension method returns every stepth element from the source sequence.
- Uses yield return to lazily produce results.

## 4. Build a plug-in architecture using reflection

```
public interface IPlugin
{
   void Execute();
}

public class PluginLoader
{
   public List<IPlugin> LoadPlugins(string folderPath)
   {
      List<IPlugin> plugins = new();
      foreach (var file in Directory.GetFiles(folderPath, "*.dll"))
      {
         var assembly = Assembly.LoadFrom(file);
         var types = assembly.GetTypes()
            .Where(t => typeof(IPlugin).IsAssignableFrom(t) && !t.IsInterface);

         foreach (var type in types)
            plugins.Add((IPlugin)Activator.CreateInstance(type));
      }
      return plugins;
   }
}
```

**Explanation:**

- Load assemblies dynamically and find classes implementing IPlugin.
- Use reflection to instantiate and run plugins without recompilation.

## 5. Explain and implement Dependency Injection using a custom container

```
public class SimpleContainer
{
```

```
    private readonly Dictionary<Type, Func<object>> _registrations = new();

    public void Register<TService, TImplementation>() where TImplementation : TService,
new()
    {
        _registrations[typeof(TService)] = () => new TImplementation();
    }

    public TService Resolve<TService>()
    {
        return (TService)_registrations[typeof(TService)]();
    }
}
```

**Explanation:**

- Register maps interface to implementation factory function.
- Resolve creates an instance when requested.
- Simple manual DI without external frameworks.

**6. Design a scalable logging framework using interfaces and abstraction**

```
public interface ILogger
{
    void Log(string message);
}

public class ConsoleLogger : ILogger
{
    public void Log(string message) => Console.WriteLine(message);
}

public class FileLogger : ILogger
{
    private string path;
    public FileLogger(string path) => this.path = path;
    public void Log(string message) => File.AppendAllText(path, message +
Environment.NewLine);
}

public class LoggerManager
{
```

```
    private readonly List<ILogger> loggers;
    public LoggerManager(params ILogger[] loggers) => this.loggers = loggers.ToList();

    public void Log(string msg)
    {
        foreach (var logger in loggers)
            logger.Log(msg);
    }
}
```

**Explanation:**

- Abstract logging via ILogger.
- Support multiple destinations by aggregating loggers.
- Easy to extend for different log targets.

**7. Write code to demonstrate async programming pitfalls (e.g., deadlocks)**

```
public string GetData()
{
    Blocking call inside async context causes deadlock
    var task = GetDataAsync();
    return task.Result;   .Result blocks main thread, causing deadlock in UI apps
}

public async Task<string> GetDataAsync()
{
    await Task.Delay(1000);
    return "Hello";
}
```

**Explanation:**

- Calling .Result or .Wait() inside an async context blocks the thread.
- This can cause deadlocks, especially in UI or ASP.NET synchronization contexts.

**8. Use Span<T> and Memory<T> for performance-sensitive operations**

```
public void ProcessSpan()
{
    Span<int> numbers = stackalloc int[5] { 1, 2, 3, 4, 5 };
    for (int i = 0; i < numbers.Length; i++)
        numbers[i] *= 2;
}
```

**Explanation:**

- Span<T> allows stack-allocated, memory-safe slices without heap allocations.

- Great for high-performance, low-overhead memory access.

**9. Implement a caching layer with LRU eviction strategy**

```
public class LRUCache<K, V>
{
   private readonly int capacity;
   private readonly Dictionary<K, LinkedListNode<(K key, V val)>> cacheMap;
   private readonly LinkedList<(K key, V val)> lruList;

   public LRUCache(int capacity)
   {
      this.capacity = capacity;
      cacheMap = new Dictionary<K, LinkedListNode<(K, V)>>();
      lruList = new LinkedList<(K, V)>();
   }

   public V Get(K key)
   {
      if (!cacheMap.ContainsKey(key)) throw new KeyNotFoundException();
      var node = cacheMap[key];
      lruList.Remove(node);
      lruList.AddFirst(node);
      return node.Value.val;
   }

   public void Put(K key, V val)
   {
      if (cacheMap.ContainsKey(key))
      {
         var node = cacheMap[key];
         lruList.Remove(node);
      }
      else if (cacheMap.Count == capacity)
      {
         var last = lruList.Last;
         lruList.RemoveLast();
         cacheMap.Remove(last.Value.key);
      }
      var newNode = new LinkedListNode<(K, V)>((key, val));
      lruList.AddFirst(newNode);
```

```
            cacheMap[key] = newNode;
        }
    }
```

**Explanation:**

- Use a LinkedList to track usage order and Dictionary for O(1) access.
- On access, move node to front (most recently used).
- When full, evict the least recently used (tail node).

**10. Write a multi-threaded program to simulate bank transactions**

```
class BankAccount
{
    private int balance;
    private readonly object balanceLock = new();

    public BankAccount(int initialBalance) => balance = initialBalance;

    public void Deposit(int amount)
    {
        lock (balanceLock)
        {
            balance += amount;
            Console.WriteLine($"Deposited {amount}, Balance: {balance}");
        }
    }

    public void Withdraw(int amount)
    {
        lock (balanceLock)
        {
            if (balance >= amount)
            {
                balance -= amount;
                Console.WriteLine($"Withdrew {amount}, Balance: {balance}");
            }
            else
                Console.WriteLine("Insufficient funds");
        }
    }
}
```

**Explanation:**

- Use locks to synchronize access to shared balance variable.
- Prevent race conditions during concurrent deposits/withdrawals.

Absolutely! Here are answers for questions **11 to 25** with example code and explanations:

**11. Design a rate limiter using SemaphoreSlim**

```
public class RateLimiter
{
   private readonly SemaphoreSlim _semaphore;
   private readonly TimeSpan _timeWindow;
   private readonly Queue<DateTime> _requestTimes = new();

   public RateLimiter(int maxRequests, TimeSpan timeWindow)
   {
      _semaphore = new SemaphoreSlim(maxRequests, maxRequests);
      _timeWindow = timeWindow;
   }

   public async Task<bool> WaitToProceedAsync()
   {
      lock (_requestTimes)
      {
         while (_requestTimes.Count > 0 && DateTime.UtcNow - _requestTimes.Peek() >
_timeWindow)
            _requestTimes.Dequeue();
         if (_requestTimes.Count >= _semaphore.CurrentCount)
            return false;  Rate limit exceeded
         _requestTimes.Enqueue(DateTime.UtcNow);
      }
      await _semaphore.WaitAsync();
      return true;
   }

   public void Release() => _semaphore.Release();
}
```

**Explanation:**

- SemaphoreSlim controls concurrency (max requests allowed).
- A queue tracks timestamps to enforce a time window.
- Reject or allow requests based on count and time.

**12. Build a custom middleware in ASP.NET Core**

```
public class RequestLoggingMiddleware
```

```
{
    private readonly RequestDelegate _next;
    public RequestLoggingMiddleware(RequestDelegate next) => _next = next;

    public async Task InvokeAsync(HttpContext context)
    {
        Console.WriteLine($"Request: {context.Request.Method} {context.Request.Path}");
        await _next(context);
        Console.WriteLine($"Response: {context.Response.StatusCode}");
    }
}
```

In Startup.cs or Program.cs:

```
app.UseMiddleware<RequestLoggingMiddleware>();
```

**Explanation:**

- Middleware intercepts HTTP requests and responses.
- Calls _next(context) to pass control downstream.
- Useful for cross-cutting concerns like logging.

## 13. Implement circuit breaker and retry pattern manually

```
public class CircuitBreaker
{
    private int failureCount = 0;
    private readonly int failureThreshold = 3;
    private bool isOpen = false;
    private DateTime openTime;
    private readonly TimeSpan resetTimeout = TimeSpan.FromSeconds(10);

    public bool CanExecute()
    {
        if (isOpen && DateTime.UtcNow - openTime > resetTimeout)
        {
            isOpen = false;
            failureCount = 0;
        }
        return !isOpen;
    }

    public void Success() => failureCount = 0;

    public void Failure()
```

```
    {
        failureCount++;
        if (failureCount >= failureThreshold)
        {
            isOpen = true;
            openTime = DateTime.UtcNow;
        }
    }
}

public async Task<T> ExecuteWithRetry<T>(Func<Task<T>> action, int retryCount = 3)
{
    int attempts = 0;
    while (true)
    {
        try
        {
            return await action();
        }
        catch
        {
            if (++attempts >= retryCount) throw;
        }
    }
}
```

**Explanation:**

- Circuit breaker trips after failure threshold, preventing calls temporarily.
- Retry logic retries a failed call up to N times.
- Helps improve resilience in distributed systems.

**14. Solve a real-world problem using strategy pattern**

```
public interface ICompressionStrategy
{
    void Compress(string fileName);
}

public class ZipCompression : ICompressionStrategy
{
    public void Compress(string fileName) => Console.WriteLine($"Compressing {fileName}
using ZIP");
```

```
}

public class RarCompression : ICompressionStrategy
{
   public void Compress(string fileName) => Console.WriteLine($"Compressing {fileName}
using RAR");
}

public class CompressionContext
{
   private ICompressionStrategy _strategy;
   public void SetStrategy(ICompressionStrategy strategy) => _strategy = strategy;
   public void CreateArchive(string file) => _strategy.Compress(file);
}
```

**Explanation:**

- Strategy pattern encapsulates algorithms (compression types).
- Context selects strategy dynamically at runtime.

**15. Refactor legacy code using SOLID principles**

**Example:** Refactor a class violating Single Responsibility Principle into smaller classes.

```
 Before (violates SRP)
public class OrderProcessor
{
   public void Process(Order order)
   {
      Process payment
      Send email notification
   }
}

 After
public class PaymentProcessor
{
   public void ProcessPayment(Order order) { /*...*/ }
}

public class NotificationService
{
   public void SendOrderConfirmation(Order order) { /*...*/ }
}
```

```
public class OrderProcessor
{
   private readonly PaymentProcessor _paymentProcessor = new();
   private readonly NotificationService _notificationService = new();

   public void Process(Order order)
   {
      _paymentProcessor.ProcessPayment(order);
      _notificationService.SendOrderConfirmation(order);
   }
}
```

**Explanation:**

- Split responsibilities to separate classes to improve maintainability and testability.

**16. Create a multi-tenant architecture in ASP.NET Core**

```
public class Tenant
{
   public string Id { get; set; }
   public string Name { get; set; }
}

public class TenantMiddleware
{
   private readonly RequestDelegate _next;
   public TenantMiddleware(RequestDelegate next) => _next = next;

   public async Task InvokeAsync(HttpContext context)
   {
      var tenantId = context.Request.Headers["X-Tenant-ID"];
      if (!string.IsNullOrEmpty(tenantId))
         context.Items["Tenant"] = new Tenant { Id = tenantId, Name = $"Tenant {tenantId}"
};
      await _next(context);
   }
}
```

```
Usage: app.UseMiddleware<TenantMiddleware>();
```

**Explanation:**

- Middleware extracts tenant info from request (e.g., header).
- Tenant info stored in HttpContext.Items accessible downstream.

- Supports tenant-specific logic/data.

**17. Use TPL Dataflow for concurrent data processing**

```
var block = new TransformBlock<int, int>(n => n * 2,
   new ExecutionDataflowBlockOptions { MaxDegreeOfParallelism = 4 });

var printBlock = new ActionBlock<int>(n => Console.WriteLine(n));

block.LinkTo(printBlock, new DataflowLinkOptions { PropagateCompletion = true });

for (int i = 0; i < 10; i++) block.Post(i);
block.Complete();

await printBlock.Completion;
```

**Explanation:**

- TPL Dataflow allows building async pipelines with concurrency.
- TransformBlock transforms data; ActionBlock processes it.
- Supports data buffering, parallelism, and backpressure.

**18. Demonstrate unit testing with mocking (e.g., Moq + xUnit)**

```
public interface IService
{
   int GetValue();
}

public class Consumer
{
   private readonly IService _service;
   public Consumer(IService service) => _service = service;
   public int DoubleValue() => _service.GetValue() * 2;
}

 Unit Test using Moq and xUnit
public class ConsumerTests
{
   [Fact]
   public void DoubleValue_ReturnsDouble()
   {
      var mock = new Mock<IService>();
      mock.Setup(s => s.GetValue()).Returns(5);
```

```
        var consumer = new Consumer(mock.Object);
        Assert.Equal(10, consumer.DoubleValue());
    }
}
```

**Explanation:**

- Moq creates mocks of dependencies.
- xUnit provides test framework.
- Test verifies logic isolated from real service.

## 19. Write a high-performance file parser for large CSV files

```
public IEnumerable<string[]> ParseCsv(string filePath)
{
    using var stream = new FileStream(filePath, FileMode.Open);
    using var reader = new StreamReader(stream);

    while (!reader.EndOfStream)
    {
        var line = reader.ReadLine();
        if (line == null) yield break;
        var values = line.Split(',');
        yield return values;
    }
}
```

**Explanation:**

- Uses streaming with StreamReader to process files line-by-line.
- Avoids loading entire file into memory.
- Suitable for large files.

## 20. Implement CQRS and MediatR pattern in a mini-project

```
 Query
public record GetUserQuery(int Id) : IRequest<User>;

public class GetUserHandler : IRequestHandler<GetUserQuery, User>
{
    public Task<User> Handle(GetUserQuery request, CancellationToken cancellationToken)
    {
        Simulate DB call
        return Task.FromResult(new User { Id = request.Id, Name = "John" });
    }
}
```

Usage

```
var mediator = new Mediator(...);
var user = await mediator.Send(new GetUserQuery(1));
```

**Explanation:**

- CQRS separates commands and queries for better scalability.
- MediatR is a mediator pattern library that handles request dispatching.
- Handlers encapsulate business logic.

**21. Build a rule engine using expressions and reflection**

```
public class Rule<T>
{
   public Func<T, bool> Predicate { get; }
   public string Message { get; }
   public Rule(Func<T, bool> predicate, string message)
   {
      Predicate = predicate; Message = message;
   }
}

public class RuleEngine<T>
{
   private readonly List<Rule<T>> _rules = new();
   public void AddRule(Rule<T> rule) => _rules.Add(rule);

   public IEnumerable<string> Validate(T entity)
   {
      return _rules.Where(r => !r.Predicate(entity)).Select(r => r
.Message);
}
}
```

**Explanation**:

- Define rules as lambda expressions.
- Run validations dynamically at runtime.
- Easily extensible and configurable.

**22. Demonstrate cross-cutting concerns with AOP (Aspect-Oriented Programming)**

Using a simple proxy-based interceptor (e.g., Castle DynamicProxy)

```
public class LoggingInterceptor : IInterceptor
```

```
{
    public void Intercept(IInvocation invocation)
    {
        Console.WriteLine($"Before executing {invocation.Method.Name}");
        invocation.Proceed();
        Console.WriteLine($"After executing {invocation.Method.Name}");
    }
}
```

**Explanation:**

- AOP intercepts method calls to add behavior (logging, security).
- Popular with DI containers or dynamic proxies.
- Keeps business logic clean.

**23. Write a console app simulating DI lifecycle scopes (Transient, Scoped, Singleton)**

```
public interface IService { Guid Id { get; } }
public class TransientService : IService { public Guid Id { get; } = Guid.NewGuid(); }
public class ScopedService : IService { public Guid Id { get; } = Guid.NewGuid(); }
public class SingletonService : IService { public Guid Id { get; } = Guid.NewGuid(); }

var services = new ServiceCollection();
services.AddTransient<IService, TransientService>();
services.AddScoped<IService, ScopedService>();
services.AddSingleton<IService, SingletonService>();

var provider = services.BuildServiceProvider();

using (var scope1 = provider.CreateScope())
{
    var s1a = scope1.ServiceProvider.GetService<IService>();
    var s1b = scope1.ServiceProvider.GetService<IService>();
    Console.WriteLine($"Scope1: {s1a.Id} {s1b.Id}");
}

using (var scope2 = provider.CreateScope())
{
    var s2 = scope2.ServiceProvider.GetService<IService>();
    Console.WriteLine($"Scope2: {s2.Id}");
}
```

**Explanation:**

- Transient: new instance every request.
- Scoped: one instance per scope (e.g., web request).
- Singleton: single instance application-wide.

**24. Integrate Polly for transient fault handling**

```
var retryPolicy = Policy
   .Handle<HttpRequestException>()
   .WaitAndRetryAsync(3, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
retryAttempt)));

await retryPolicy.ExecuteAsync(async () =>
{
   var response = await httpClient.GetAsync("https:api.example.com/data");
   response.EnsureSuccessStatusCode();
});
```

**Explanation:**

- Polly provides robust resilience policies like retries, circuit breakers.
- Retry with exponential backoff for transient errors.

**25. Build a modular app using MEF (Managed Extensibility Framework)**

Define contract

```
public interface IModule
{
   void Run();
}
```

Export module

```
[Export(typeof(IModule))]
public class SampleModule : IModule
{
   public void Run() => Console.WriteLine("Module running");
}
```

Compose parts

```
var catalog = new AssemblyCatalog(Assembly.GetExecutingAssembly());
var container = new CompositionContainer(catalog);

var modules = container.GetExports<IModule>();
foreach (var module in modules)
{
```

```
    module.Value.Run();
}
```

**Explanation:**

- MEF discovers and composes modules dynamically.
- Supports extensibility without recompilation.
- Useful in plugin and modular apps.