KITUR CHELIMO MERCY
SCT 221-0840/2022
APPLICATION PROGRAMMING 2 ASSIGNMENT

Question 1

i. Write a C# program that performs addition, subtraction, multiplication, and division on two numbers

```
using System;
class Calculator
{
   static void Main()
   {
      Console.WriteLine("Enter the first number:");
      double num1 = Convert.ToDouble(Console.ReadLine());

      Console.WriteLine("Enter the second number:");
      double num2 = Convert.ToDouble(Console.ReadLine());

      Console.WriteLine("Select the operation: +, -, *, /");
      string operation = Console.ReadLine();

      double result = 0;
      switch (operation)
      {
         case "+":
            result = num1 + num2;
            break;
         case "-":
            result = num1 - num2;
            break;
         case "*":
            result = num1 * num2;
            break;
         case "/":
            if (num2 != 0)
            {
               result = num1 / num2;
            }
```

```
            else
            {
                Console.WriteLine("Cannot divide by zero!");
                return;
            }
            break;
        default:
            Console.WriteLine("Invalid operation");
            return;
    }
    Console.WriteLine($"Result: {result}");
  }
}
```

ii. Write a method in C# that takes an array of integers and returns the average of the scores.

```
using System;
class AverageCalculator
{
    public static double CalculateAverage(int[] numbers)
    {
        if (numbers == null || numbers.Length == 0)
        {
            return 0;
        }
        int sum = 0;
        foreach (int number in numbers)
        {
            sum += number;
        }
        return (double)sum / numbers.Length;
    }
    static void Main()
    {
        int[] scores = { 85, 90, 78, 92, 88 };
        double average = CalculateAverage(scores);
        Console.WriteLine($"Average: {average}");
    }
}
```

iii. Describe the role of constructors in class instantiation and how they differ from other methods. Illustrate the explanation with a class that includes a default constructor and an overloaded constructor.

Constructors are special methods in a class that are called when an object is created. They are used to initialize the object's state, setting up any necessary fields or properties.

- **Default Constructor**: A constructor with no parameters. It is automatically provided by C# if no other constructors are defined, or it can be explicitly defined.
- **Overloaded Constructor**: A constructor with parameters that allows for different ways to initialize an object.

```
using System;
class Person
{
   public string Name { get; set; }
   public int Age { get; set; }
   // Default constructor
   public Person()
   {
      Name = "Unknown";
      Age = 0;
      Console.WriteLine("Default constructor called.");
   }
   // Overloaded constructor
   public Person(string name, int age)
   {
      Name = name;
      Age = age;
      Console.WriteLine("Overloaded constructor called.");
   }
   public void DisplayInfo()
   {
      Console.WriteLine($"Name: {Name}, Age: {Age}");
   }
   static void Main()
   {
      Person person1 = new Person(); // Calls default constructor
      person1.DisplayInfo();

      Person person2 = new Person("John Doe", 30); // Calls overloaded constructor
      person2.DisplayInfo();
   }
}
```

iv. <u>Create a class Employee with a constructor that takes an employee's name and ID. Demonstrate how to create an instance of the class and include a secondary constructor that accepts optional parameters like department and salary.</u>

```
using System;
class Employee
{
   public string Name { get; set; }
```

```
    public int ID { get; set; }
    public string Department { get; set; }
    public double Salary { get; set; }

    // Constructor with name and ID
    public Employee(string name, int id)
    {
        Name = name;
        ID = id;
    }

    // Overloaded constructor with additional optional parameters
    public Employee(string name, int id, string department = "Unknown", double salary = 0)
    {
        Name = name;
        ID = id;
        Department = department;
        Salary = salary;
    }
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, ID: {ID}, Department: {Department}, Salary:
{Salary}");
    }
    static void Main()
    {
        Employee emp1 = new Employee("Alice Johnson", 101);
        emp1.DisplayInfo();

        Employee emp2 = new Employee("Bob Smith", 102, "IT", 50000);
        emp2.DisplayInfo();
    }
}
```

Question 2

Difference between the == operator and the Equals() method in C#.

The == operator checks for **reference equality** when used with objects. This means it checks whether the two operands refer to the same memory location.

The Equals() method is used to check for **value equality**. It compares the actual content of the objects.

When to Use Each

- Use == when you want to compare primitive types or when you want to compare strings for content in a simple, straightforward manner. For strings, it is often used due to its clarity and ease of use.
- Use **Equals()** when you want to ensure that the comparison is based on content rather than reference, especially when dealing with custom objects or more complex scenarios where == may be ambiguous or overridden.

Predicting the Output and Explanation

All the comparisons return True because both the == operator and Equals() method compare the content of the strings, which are the same in this case.

Question 3

Role of the Common Language Runtime (CLR) and Base Class Library (BCL) in the .NET Framework

Common Language Runtime (CLR) is the execution engine of the .NET Framework. Roles:

a. **Security**: CLR enforces security policies and permissions, protecting the system from untrusted code.
b. **Memory Management**: Automatically handles memory allocation and deallocation preventing memory leaks and other memory-related issues.
c. **Exception Handling**: Provides a structured exception handling model, allowing developers to handle errors gracefully.

Base Class Library (BCL) is a comprehensive collection of reusable types, classes, and functions provided by the .NET Framework. Roles:

a. **Security**: Classes for encryption, authentication, and access control
b. **System I/O**: Classes like *System.IO* for handling file operations, streams, and directories.
c. **Networking**: Classes for working with network protocols, sockets, and web services.

How CLR and BCL Work Together

The CLR and BCL work together to provide a smooth development experience:

- **Managed Environment**: The CLR provides a managed environment where applications are executed and BCL provides the foundational building blocks that developers use to create their applications within this environment.
- **Code Execution**: CLR uses JIT compilation to convert intermediate language into native code, which is executed on the machine and BCL provides the necessary classes and methods that your code interacts with during execution.
- **Resource Management**: The CLR's memory management, along with the rich set of utilities provided by the BCL, ensures efficient use of resources, better performance, and fewer bugs.

Write a program that demonstrates the use of System.IO.File to create, read, and write to a file containing a list of books.

```
using System;
using System.IO;
class LibraryManagement
{
   static void Main()
   {
      string filePath = "books.txt";
      // Create or overwrite the file and write some initial data
      WriteToFile(filePath, new string[]
      {
         "The Catcher in the Rye by J.D. Salinger",
         "To Kill a Mockingbird by Harper Lee",
         "1984 by George Orwell",
         "The Great Gatsby by F. Scott Fitzgerald"
      });

      // Read and display the file contents
      Console.WriteLine("Books in the Library:");
      ReadFromFile(filePath);
      // Append a new book to the file
      AppendToFile(filePath, "Moby Dick by Herman Melville");
      // Display updated file contents
      Console.WriteLine("\nUpdated Books in the Library:");
      ReadFromFile(filePath);
   }
   static void WriteToFile(string path, string[] contents)
   {
      File.WriteAllLines(path, contents);
   }
   static void ReadFromFile(string path)
   {
      if (File.Exists(path))
      {
         string[] lines = File.ReadAllLines(path);
         foreach (string line in lines)
         {
            Console.WriteLine(line);
         }
      }
      else
      {
         Console.WriteLine("File not found.");
      }
   }
   static void AppendToFile(string path, string content)
   {
      File.AppendAllText(path, content + Environment.NewLine);
   }
```

```
}
```

Question 4

Difference between value types and reference types in C# and provide examples of each.

Value types hold their data directly in memory. When you assign a value type to a variable, the actual data is stored in the variable, typically on the stack.

```
int x = 10;

int y = x;

y = 20; // x is still 10, y is now 20
```

Reference types store a reference (or pointer) to the actual data. The reference itself is stored on the stack, but the data it points to is stored on the heap.

```
string str1 = "Hello";

string str2 = str1;

str2 = "World"; // Both str1 and str2 now point to the new "World" string
```

Scenarios Where the Choice Matters

1. **Performance**:
   - Value types are generally faster to access because they are stored on the stack, which is faster
   - Reference types can be slower because they involve a level of indirection (a reference pointing to the data), and the heap is generally slower to allocate from and manage.
2. **Memory Usage**:
   - Value types can lead to more memory usage if used in large quantities because each instance occupies its own space on the stack.
   - Reference types can be more memory-efficient when the same data is used across multiple variables because only the reference is copied, not the entire object.
3. **Behavior**:
   - Use value types when you need independent copies of data and don't want changes in one instance to affect another.
   - Use reference types when you need multiple references to the same data, allowing changes in one place to be reflected across all references.

Write a C# program that demonstrates the concept of value types and reference types using primitive data types and objects.

```
using System;
class ValueTypeReferenceTypeDemo
{
   static void Main()
   {
      // Value Type Example
      int a = 10;
      int b = a;
      b = 20;
      Console.WriteLine("Value Types:");
      Console.WriteLine($"a = {a}, b = {b}"); // a = 10, b = 20
      // Reference Type Example
      string[] arr1 = { "Hello", "World" };
      string[] arr2 = arr1;
      arr2[0] = "Hi";
      Console.WriteLine("\nReference Types:");
      Console.WriteLine($"arr1[0] = {arr1[0]}, arr2[0] = {arr2[0]}"); // Both arr1[0] and arr2[0] = "Hi"
      // Checking memory addresses
      Console.WriteLine("\nMemory Address Comparison:");
      Console.WriteLine($"arr1 and arr2 refer to the same object: {Object.ReferenceEquals(arr1, arr2)}");
      // Comparing Value Types using ReferenceEquals (this will always return false)
      int c = 30;
      int d = c;
      Console.WriteLine($"c and d refer to the same object: {Object.ReferenceEquals(c, d)}"); // False
   }
}
```

Question 5

Describe how encapsulation applies to classes and objects in C# and how it can help control access to fields and methods.

Encapsulation refers to the practice of bundling the data (fields) and the methods (functions) that operate on the data into a single unit called a class, while restricting direct access to some of the object's components

Private Fields: In C#, fields (variables) within a class can be marked as private, meaning they cannot be accessed directly from outside the class. This prevents external code from making arbitrary changes to the object's state.

Public Properties: To allow controlled access to these private fields, you expose them through public properties. Properties are special methods (getters and setters) that provide a way to read and write the values of private fields. You can also include validation logic within these properties to ensure that only valid data is assigned to the fields.

Access Modifiers: C# uses access modifiers like private, protected, internal, and public to control the visibility of fields, methods, and properties. By setting appropriate access levels, you ensure that an object's internal state can only be modified in predefined ways.

Create a Person class with private fields for name and age and public properties to encapsulate these fields. Add validation in the properties, such as ensuring age is non negative.

```
using System;
class Person
{
   // Private fields
   private string name;
   private int age;
   // Public property for Name with basic validation
   public string Name
   {
      get { return name; }
      set
      {
         if (!string.IsNullOrWhiteSpace(value))
         {
            name = value;
         }
         else
         {
            throw new ArgumentException("Name cannot be empty.");
         }
      }
```

```
    }
    // Public property for Age with validation
    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0)
            {
                age = value;
            }
            else
            {
                throw new ArgumentException("Age cannot be negative.");
            }
        }
    }
    // Constructor
    public Person(string name, int age)
    {
        Name = name;  // Using the property to leverage validation
        Age = age;    // Using the property to leverage validation
    }
    // Method to display person info
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, Age: {Age}");
    }
    static void Main()
    {
        try
        {
            // Creating an instance of Person with valid data
            Person person1 = new Person("Maria", 30);
            person1.DisplayInfo();
            // Attempt to set invalid age (will throw an exception)
            person1.Age = -5;
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

Question 6

Explain the difference between a single-dimensional array and a jagged array and provide a use case for each.

A **single-dimensional array** (or one-dimensional array) is a linear collection of elements. All elements are stored in a single row or column, and each element is accessed by a single index.

```
int[] numbers = { 1, 2, 3, 4, 5 };
```

A **jagged array** is an array of arrays, meaning each element of the main array is itself an array. Jagged arrays can have arrays of different lengths, making them suitable for representing data structures like a collection of rows where each row can have a different number of columns.

```
int[][] jaggedArray = new int[][]
{
   new int[] { 1, 2 },
   new int[] { 3, 4, 5 },
   new int[] { 6 }
};
```

Create a method in C# that takes a two-dimensional array of integers and returns the sum of all its elements.

```
using System;
class ArraySumCalculator
{
   public static int SumTwoDimensionalArray(int[][] array)
   {
      int sum = 0;
      if (array == null)
      {
         return sum; // Return 0 if the array is null
      }
      foreach (int[] row in array)
      {
         if (row != null)
         {
            foreach (int element in row)
            {
               sum += element;
            }
         }
      }

      return sum;
   }
   static void Main()
   {
```

```
        // Example of a jagged array
        int[][] array = new int[][]
        {
            new int[] { 1, 2, 3 },
            new int[] { 4, 5 },
            null, // Simulate a missing row
            new int[] { 6, 7, 8, 9 }
        };
        int sum = SumTwoDimensionalArray(array);
        Console.WriteLine($"The sum of all elements in the array is: {sum}");
    }
}
```

Define an enum called Color with values Red, Green, and Blue. Also, define a class Shape with a nested class Circle that uses the enum to determine its color.

```
using System;
public enum Color
{
    Red,
    Green,
    Blue
}
public class Shape
{
    public class Circle
    {
        public Color CircleColor { get; set; }

        public Circle(Color color)
        {
            CircleColor = color;
        }
        public void DisplayColor()
        {
            Console.WriteLine($"The circle's color is {CircleColor}");
        }
    }
}
class Program
{
    static void Main()
    {
        // Create a Circle object with the color Blue
        Shape.Circle myCircle = new Shape.Circle(Color.Blue);
        // Display the color of the circle
        myCircle.DisplayColor();
    }
}
```

Question 7

Describe how exceptions are handled in C# using try, catch, and finally blocks.

Try Block:

- The code that might throw an exception is placed inside a `try` block. This block is used to "try" running a section of code while anticipating that something might go wrong.

Catch Block:

- If an exception occurs in the try block, control is passed to the `catch` block. This block is used to handle the exception. Multiple catch blocks can be used to handle different types of exceptions.

Finally Block:

- The finally block contains code that is always executed, regardless of whether an exception was thrown or not. This is typically used for cleanup activities, like closing files or releasing resources.

Best Practices and Potential Pitfalls

- **Specific Catch Blocks**: Always catch specific exceptions rather than a general `Exception` type. This helps in identifying and handling different types of errors more effectively.
- **Avoid Empty Catch Blocks**: Catch blocks should not be left empty. If you catch an exception, there should be a clear reason for it, like logging the error or taking corrective action.
- **Use Finally for Cleanup**: Use the `finally` block to clean up resources like closing files, streams, or database connections, ensuring that they are closed even if an exception occurs.
- **Don't Use Exceptions for Control Flow**: Exceptions should not be used to manage normal control flow logic, as this can lead to performance issues and unclear code.
- **Log Exceptions**: Always log exceptions, especially when they are caught in a catch block, to help with debugging and maintaining the application.

Write a C# program that demonstrates handling an exception when trying to access an element outside of the bounds of an array. Introduce nested try-catch blocks for different error types.

```
using System;
class ListManagement
{
   static void Main()
   {
      int[] numbers = { 1, 2, 3, 4, 5 };
      try
      {
         // Outer try block
         try
         {
            // Attempting to access an element outside the bounds of the array
            Console.WriteLine("Accessing the 10th element: " + numbers[9]);
         }
         catch (IndexOutOfRangeException ex)
         {
            Console.WriteLine("IndexOutOfRangeException caught: " + ex.Message);
            // Inner try block to demonstrate nested exception handling
            try
            {
               // Trigger a divide by zero exception
               int result = numbers[1] / 0;
            }
            catch (DivideByZeroException ex2)
            {
               Console.WriteLine("DivideByZeroException caught: " + ex2.Message);
            }
         }
      }
      catch (Exception ex)
      {
         Console.WriteLine("A general exception caught: " + ex.Message);
      }
      finally
      {
         Console.WriteLine("Cleanup code executed in the finally block.");
      }
   }
}
```

Question 8

Write a C# program that takes an integer input from the user and uses if-else conditions to print the appropriate message.

```
using System;
class NumberAnalysis
{
```

```
    static void Main()
    {
        Console.Write("Enter an integer: ");
        int number = int.Parse(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.");
        }
        if (number % 2 == 0)
        {
            Console.WriteLine("The number is even.");
        }
        else
        {
            Console.WriteLine("The number is odd.");
        }
    }
}
```

Explain the differences between while, do-while, and for loops, and provide examples of each

While loop continues to execute a block of code as long as the specified condition remains true. The condition is checked before the code block is executed.

Use Case: Use a while loop when the number of iterations is not known in advance, and you want the loop to run as long as a condition remains true.

Do-while loop is similar to a while loop, but the condition is checked after the code block is executed. This guarantees that the loop executes at least once.

Use Case: Use a do-while loop when you need the loop to execute at least once, regardless of whether the condition is initially true or false.

For loop is typically used for iterating over a range of values. It consists of three parts: initialization, condition, and iteration expression. The loop continues as long as the condition remains true.

Use Case:Use a for loop when the number of iterations is known beforehand, such as iterating over an array or a list.

Write a program using a loop to compute the factorial. Add a twist by calculating the factorial for odd numbers.

```
using System;
class FactorialCalculator
{
   static void Main()
   {
      Console.Write("Enter a number: ");
      int number = int.Parse(Console.ReadLine());
      if (number % 2 == 0)
      {
         Console.WriteLine("The number is even, so no factorial will be calculated.");
      }
      else
      {
         long factorial = 1;
         for (int i = 1; i <= number; i++)
         {
            factorial *= i;
         }
         Console.WriteLine($"The factorial of {number} is {factorial}");
      }
   }
}
```

Write a C# program that uses nested loops to print a pattern of asterisks in the shape of a right-angled triangle. Add complexity by adjusting the program to print an inverted triangle.

```
using System;
class PatternPrinter
{
   static void Main()
   {
      int n = 5; // Height of the triangle
      // Right-angled triangle
      Console.WriteLine("Right-angled triangle:");
      for (int i = 1; i <= n; i++)
      {
         for (int j = 1; j <= i; j++)
         {
            Console.Write("*");
         }
         Console.WriteLine();
      }
      // Inverted right-angled triangle
      Console.WriteLine("\nInverted right-angled triangle:");
```

```
        for (int i = n; i >= 1; i--)
        {
            for (int j = 1; j <= i; j++)
            {
                Console.Write("*");
            }
            Console.WriteLine();
        }
    }
}
```

Question 9

Explain the role of threads in C#.

Threads in C# allow a program to perform multiple operations concurrently.

Discuss the main difference between using the Thread class and the Task class, and provide an example where each would be useful.

Threads represents a low-level, operating system-managed unit of execution. You have direct control over thread creation, start, and management while Task represents a higher-level abstraction over threading, designed to simplify asynchronous programming.

Threads require manual synchronization using locks, mutexes, etc., to avoid race conditions while the .NET runtime manages task scheduling and execution, optimizing resource use and simplifying code.

Threads are useful when you need fine-grained control over the execution flow, such as managing long-running background tasks or interacting with hardware while asks are ideal for asynchronous operations, like I/O-bound or short-running operations, where the focus is on the result rather than the execution thread.

 Threads: Provide more control but require manual management, suitable for complex scenarios where thread management is critical.
 Tasks: Simplify concurrent and asynchronous programming, with automatic management and support for a sync/await, making them more suitable for most modern applications.

Write a C# program that demonstrates how to use the Thread class to create and start a new thread. Add complexity by having the main thread synchronize with the new thread and handle thread safety.

```
using System;
using System.Threading;
class ThreadExample
{
    private static readonly object _lock = new object();
```

```
    private static int _sharedResource = 0;
    static void Main()
    {
        // Create a new thread
        Thread newThread = new Thread(WorkerThread);
        // Start the new thread
        newThread.Start();
        // Synchronize with the new thread
        for (int i = 0; i < 5; i++)
        {
            lock (_lock)
            {
                _sharedResource++;
                Console.WriteLine($"Main Thread: {_sharedResource}");
            }
            // Pause for a short time to simulate work
            Thread.Sleep(100);
        }
        // Wait for the new thread to finish
        newThread.Join();
        Console.WriteLine("Main Thread: New thread has finished.");
    }
    // Method to be executed by the new thread
    static void WorkerThread()
    {
        for (int i = 0; i < 5; i++)
        {
            lock (_lock)
            {
                _sharedResource++;
                Console.WriteLine($"Worker Thread: {_sharedResource}");
            }

            // Pause for a short time to simulate work
            Thread.Sleep(150);
        }
    }
}
```

Question 10

Write a C# program that uses the HttpClient class to make a GET request to a public API and display the response. Parse JSON data from the response to display article titles and summaries.

```
using System;
using System.Net.Http;
```

```
using System.Threading.Tasks;
using System.Text.Json;
class NewsAggregator
{
   static async Task Main(string[] args)
   {
      string apiUrl = "https://newsapi.org/v2/top-headlines?country=us&apiKey=YOUR_API_KEY";
      using HttpClient client = new HttpClient();
      try
      {
         // Make a GET request
         HttpResponseMessage response = await client.GetAsync(apiUrl);
         response.EnsureSuccessStatusCode();
         // Read and parse JSON response
         string jsonResponse = await response.Content.ReadAsStringAsync();
         NewsResponse newsData = JsonSerializer.Deserialize<NewsResponse>(jsonResponse);
         // Display article titles and summaries
         if (newsData != null && newsData.Articles != null)
         {
            foreach (var article in newsData.Articles)
            {
               Console.WriteLine($"Title: {article.Title}");
               Console.WriteLine($"Summary: {article.Description}\n");
            }
         }
      }
      catch (Exception ex)
      {
         Console.WriteLine($"Error fetching news: {ex.Message}");
      }
   }
}
// Model classes for JSON deserialization
public class NewsResponse
{
   public Article[] Articles { get; set; }
}
public class Article
{
   public string Title { get; set; }
   public string Description { get; set; }
}
```

Write a C# program that opens a file, reads its contents line by line, and then writes each line to a new file. Add a twist by filtering lines based on certain keywords or length.

```
using System;
```

```
using System.IO;
class FileFilter
{
   static void Main()
   {
      string inputFilePath = "input.txt";
      string outputFilePath = "output.txt";
      string keyword = "important";  // Example keyword for filtering
      int minLength = 10;           // Minimum line length for filtering
      try
      {
         using StreamReader reader = new StreamReader(inputFilePath);
         using StreamWriter writer = new StreamWriter(outputFilePath);
         string line;
         while ((line = reader.ReadLine()) != null)
         {
            // Filter lines based on keyword and length
            if (line.Contains(keyword) && line.Length > minLength)
            {
               writer.WriteLine(line);
            }
         }
         Console.WriteLine("File filtering completed successfully.");
      }
      catch (Exception ex)
      {
         Console.WriteLine($"An error occurred: {ex.Message}");
      }
   }
}
```

Question 11

Discuss the purpose of packages in C# and how to install and use a NuGet package. Explain how packages can simplify development and ensure code consistency.

Packages in C# are collections of pre-written code that you can use to add functionality to your applications without having to write everything from scratch. They often include libraries, frameworks, and tools that can simplify development tasks and improve code consistency.

- **NuGet Packages**: NuGet is the package manager for .NET. It simplifies the process of adding, updating, and managing packages within your .NET projects. NuGet packages can contain libraries, tools, or frameworks that are distributed in a way that's easy to integrate into your project.

Benefits of Using Packages

1. **Code Reuse**: Packages allow you to reuse code written by others, reducing the need to reinvent the wheel.
2. **Consistency**: Using well-maintained packages ensures that you have consistent behavior and functionality across different projects.
3. **Efficiency**: Packages can save time and effort, as they provide tested and optimized solutions for common tasks.

Write a C# program that uses a NuGet package to perform JSON serialization and deserialization. Add a twist by handling complex nested JSON structures.

Step 1: Install the NuGet Package

1. **Via Package Manager Console**:

   Install-Package Newtonsoft.Json

2. **Via .NET CLI**:

   dotnet add package Newtonsoft.Json

```
using System;
using System.Collections.Generic;
using Newtonsoft.Json;
class Program
{
   static void Main()
   {
      // Create a sample object with nested structures
      var company = new Company
      {
         Name = "Tech Innovations",
         Employees = new List<Employee>
         {
            new Employee { Name = "Alice", Position = "Developer", Skills = new List<string> { "C#", "SQL" }
},
            new Employee { Name = "Bob", Position = "Manager", Skills = new List<string> { "Leadership",
"Project Management" } }
         }
      };
      // Serialize the object to JSON
      string jsonString = JsonConvert.SerializeObject(company, Formatting.Indented);
      Console.WriteLine("Serialized JSON:");
      Console.WriteLine(jsonString);
     // Deserialize the JSON back to object
```

```
        var deserializedCompany = JsonConvert.DeserializeObject<Company>(jsonString);
        Console.WriteLine("\nDeserialized Object:");
        foreach (var employee in deserializedCompany.Employees)
        {
            Console.WriteLine($"Name: {employee.Name}, Position: {employee.Position}");
            Console.WriteLine("Skills: " + string.Join(", ", employee.Skills));
        }
    }
}
public class Company
{
    public string Name { get; set; }
    public List<Employee> Employees { get; set; }
}
public class Employee
{
    public string Name { get; set; }
    public string Position { get; set; }
    public List<string> Skills { get; set; }  }
```

Question 12

Differences between the List, Queue, and Stack data structures. Provide examples of use cases for each, including scenarios where one data structure may be more appropriate than another.

List<T>:

- **Purpose**: Represents a dynamic array that allows random access and resizing.
- **Use Case**: When you need a collection that can be dynamically resized and where you need to access elements by index.

Queue<T>:

- **Purpose**: Represents a first-in, first-out (FIFO) collection. Elements are added at the end and removed from the beginning.
- **Use Case**: When managing items that need to be processed in the order they were added, such as task scheduling.

Stack<T>:

- **Purpose**: Represents a last-in, first-out (LIFO) collection. Elements are added and removed from the top.
- **Use Case**: When you need to process items in the reverse order of their addition, such as undo operations.

Write a program that demonstrates the use of a queue to manage a line of people in a bank. Add a twist by prioritizing certain customers (e.g., VIP) to jump the queue.

```
using System;
using System.Collections.Generic;
class BankQueue
{
   static void Main()
   {
      var queue = new Queue<Customer>();
      var vipQueue = new Queue<Customer>(); // VIP queue
      // Add customers
      queue.Enqueue(new Customer("Alice"));
      queue.Enqueue(new Customer("Bob"));
      vipQueue.Enqueue(new Customer("VIP John"));
      // Process queue with priority
      ProcessQueue(vipQueue, queue);
   }
   static void ProcessQueue(Queue<Customer> vipQueue, Queue<Customer> regularQueue)
   {
      Console.WriteLine("Processing VIP customers:");
      while (vipQueue.Count > 0)
      {
         var customer = vipQueue.Dequeue();
         Console.WriteLine($"Serving VIP customer: {customer.Name}");
      }
      Console.WriteLine("\nProcessing regular customers:");
      while (regularQueue.Count > 0)
      {
         var customer = regularQueue.Dequeue();
         Console.WriteLine($"Serving customer: {customer.Name}");
      }
   }
}
class Customer
{
   public string Name { get; }
   public Customer(string name)
   {
      Name = name;
   }
}
```

Question 13

Discuss inheritance in C#. Describe how to implement it and include access modifiers in the context of inheritance.

Inheritance allows you to create a new class based on an existing class, inheriting fields, properties, and methods. It promotes code reuse and establishes an "is-a" relationship between the base and derived classes.

Base Class: Defines common attributes and methods.
Derived Class: Inherits from the base class and can override or extend its behavior.

```
using System;
public class Animal
{
   public virtual void Speak()
   {
      Console.WriteLine("Animal speaks");
   }
}
public class Dog : Animal
{
   public override void Speak()
   {
      Console.WriteLine("Dog barks");
   }
}
public class Cat : Animal
{
   public override void Speak()
   {
      Console.WriteLine("Cat meows");
   }
}
public class Bird : Animal
{
   public override void Speak()
   {
      Console.WriteLine("Bird chirps");
   }
}
class Program
{
   static void Main()
   {
      Animal[] animals = { new Dog(), new Cat(), new Bird() };
```

```
        foreach (var animal in animals)
        {
            animal.Speak(); // Demonstrates polymorphism
        }
    }
}
```

Question 14

Explain polymorphism in C# and how it can be achieved. Provide examples using base and derived classes.

Polymorphism allows objects of different classes to be treated as objects of a common base class. It is achieved through inheritance and interface implementation.

```
using System;
public interface IDrive
{
    void Drive();
}
public class Vehicle
{
    public virtual void Start()
    {
        Console.WriteLine("Vehicle started");
    }
}
public class Car : Vehicle, IDrive
{
    public override void Start()
    {
        Console.WriteLine("Car started");
    }
    public void Drive()
    {
        Console.WriteLine("Car is driving");
    }
}
public class Bike : Vehicle, IDrive
{
    public override void Start()
    {
        Console.WriteLine("Bike started");
    }
    public void Drive()
    {
        Console.WriteLine("Bike is driving");
```

```
    }
}
class Program
{
    static void Main()
    {
        Vehicle[] vehicles = { new Car(), new Bike() };
        IDrive[] drivers = { new Car(), new Bike() };
        foreach (var vehicle in vehicles)
        {
            vehicle.Start();
        }
        foreach (var driver in drivers)
        {
            driver.Drive();
        }
    }
}
```

Question 15

Explain abstraction in C# and how it can be implemented using abstract classes and interfaces. Describe scenarios where abstraction can simplify code and enhance maintainability.

Abstraction hides the implementation details and exposes only the necessary parts of an object. It is implemented using abstract classes and interfaces.

Abstract Class: Can contain abstract methods (without implementation) and concrete methods (with implementation). Derived classes must implement abstract methods.
Interface: Defines a contract that implementing classes must follow. Interfaces cannot contain implementation.

```
using System;
public abstract class Shape
{
    public abstract void Draw();
    public void ShowType()
    {
        Console.WriteLine($"This is a {GetType().Name}");
    }
}
public class Circle : Shape
{
    public override void Draw()
    {
        Console.WriteLine("Drawing a Circle");
    }
```

```
}
public class Square : Shape
{
   public override void Draw()
   {
      Console.WriteLine("Drawing a Square");
   }
}
class Program
{
   static void Main()
   {
      Shape[] shapes = { new Circle(), new Square() };
      foreach (var shape in shapes)
      {
         shape.ShowType();
         shape.Draw();
      }
   }
}
```

Create an abstract base class Shape with an abstract method Draw(). Create derived classes Circle and Square that implement the Draw() method.

```
using System;
public abstract class Shape
{
   public abstract void Draw();
   public void ShowType()
   {
      Console.WriteLine($"This is a {GetType().Name}");
   }
}
public class Circle : Shape
{
   public override void Draw()
   {
      Console.WriteLine("Drawing a Circle");
   }
}
public class Square : Shape
{
   public override void Draw()
   {
      Console.WriteLine("Drawing a Square");
   }
}
```

```
class Program
{
   static void Main()
   {
      Shape[] shapes = { new Circle(), new Square() };
      foreach (var shape in shapes)
      {
         shape.ShowType();
         shape.Draw();
      }
   }
}
```

Question 16

Predict the output of the following code

1)
```
int[] array = {1, 2, 3, 4, 5};
 for (int i = 0; i < array.Length; i++) {
Console.WriteLine(array[i]); }
```

OUTPUT:

1

2

3

4

5 // The for loop iterates through each element of the array and prints each value.

2)
```
string str1 = "Hello";
string str2 = "hello";
Console.WriteLine(str1.Equals(str2, StringComparison.OrdinalIgnoreCase));
```

OUTPUT:

True // StringComparison.OrdinalIgnoreCase performs a case-insensitive comparison, so "Hello" and "hello" are considered equal.

3)
```
object obj1 = new object();
object obj2 = new object();
Console.WriteLine(obj1 == obj2);
```

OUTPUT:

False // obj1 and obj2 are two different instances of object, so == checks if they refer to the same instance, which they do not.

4)
```
int a = 5;
```

```
int b = 10;
Console.WriteLine(a += b);
```

OUTPUT:

15 //The += operator adds b to a and then assigns the result to a. The result is 5 + 10 = 15.

Question 17

Write a C# method that returns the largest and smallest integers in the list. Add a twist by handling an empty list.

```
using System;
using System.Collections.Generic;
using System.Linq;
class Program
{
   static void Main()
   {
      List<int> numbers = new List<int> { 3, 5, 1, 9, 7, 2 };
      var result = FindLargestAndSmallest(numbers);
      Console.WriteLine($"Largest: {result.Largest}");
      Console.WriteLine($"Smallest: {result.Smallest}");
   }
   static (int Largest, int Smallest) FindLargestAndSmallest(List<int> numbers)
   {
      if (numbers == null || numbers.Count == 0)
      {
         throw new ArgumentException("The list cannot be null or empty.");
      }
      int largest = numbers.Max();
      int smallest = numbers.Min();
      return (largest, smallest);
   }
}
```

Write a C# program that reads integers from the console until a negative number is entered. Calculate and display the sum of all entered integers. Add a twist by ignoring duplicate integers in the sum.

```
using System;
using System.Collections.Generic;
class Program
{
   static void Main()
   {
      HashSet<int> uniqueNumbers = new HashSet<int>();
```

```
        int number;
        int sum = 0;
        Console.WriteLine("Enter integers (negative number to stop):");
        while (true)
        {
            number = int.Parse(Console.ReadLine());
            if (number < 0) break;
            if (uniqueNumbers.Add(number)) // Adds the number only if it's not already in the set
            {
                sum += number;
            }
        }
        Console.WriteLine($"Sum of unique integers: {sum}");
    }
}
```

Write a C# program that demonstrates the use of an enum for the days of the week. Add a twist by performing operations based on the enum value (e.g., identifying weekend days).

Write a C# program that takes a string input from the user and prints the string in reverse order.

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Enter a string:");
        string input = Console.ReadLine();
        string reversed = ReverseString(input);
        Console.WriteLine($"Reversed string: {reversed}");
    }
    static string ReverseString(string str)
    {
        char[] array = str.ToCharArray();
        Array.Reverse(array);
        return new string(array);
    }
}
```

Write a C# program that demonstrates how to use the Dictionary <Tkey,Tvalue> class to store and retrieve student grades. Add complexity by handling a variety of data types as keys and values.

```
using System;
using System.Collections.Generic;
class Program
```

```
{
   static void Main()
   {
      // Create a dictionary to store student grades
      var studentGrades = new Dictionary<string, int>
      {
         { "Alice", 90 },
         { "Bob", 85 },
         { "Charlie", 88 }
      };
      // Add a new student with a different key type
      studentGrades.Add("Daisy", 92);
      // Retrieve and display grades
      Console.WriteLine("Student Grades:");
      foreach (var entry in studentGrades)
      {
         Console.WriteLine($"{entry.Key}: {entry.Value}");
      }
      // Access a grade with a key
      string studentName = "Bob";
      if (studentGrades.TryGetValue(studentName, out int grade))
      {
         Console.WriteLine($"{studentName}'s grade: {grade}");
      }
      else
      {
         Console.WriteLine($"{studentName} not found.");
      }
   }
}
```

Question 18

Explain the purpose and benefits of using interfaces in C#. Discuss how interfaces can promote loose coupling and code reusability.

Interfaces in C# define a contract that classes can implement. They specify methods, properties, events, or indexers that a class must implement but do not provide the actual implementation.

L**oose Coupling**: Interfaces promote loose coupling by allowing different classes to interact through common interfaces, making it easier to replace or modify implementations without affecting other parts of the system.

Code Reusability: Interfaces allow different classes to share the same set of methods, enabling code that operates on the interface type to work with any implementing class.

Create an interface IDrive with a method Drive(). Implement this interface in classes Car and Bike. Demonstrate polymorphism by using a list of IDrive objects and calling the Drive() method on each object.

```
using System;
using System.Collections.Generic;
public interface IDrive
{
   void Drive();
}
public class Car : IDrive
{
   public void Drive()
   {
      Console.WriteLine("Car is driving");
   }
}
public class Bike : IDrive
{
   public void Drive()
   {
      Console.WriteLine("Bike is riding");
   }
}
class Program
{
   static void Main()
   {
      List<IDrive> vehicles = new List<IDrive> { new Car(), new Bike() };
      foreach (var vehicle in vehicles)
      {
         vehicle.Drive();
      }
   }
}
```

Explain the role of abstract classes in C# and how they differ from interfaces. Describe scenarios where abstract classes may be more appropriate than interfaces.

Purpose: Provide a base class with some implementation that can be shared among derived classes. They can contain abstract methods (which must be implemented by derived classes) and concrete methods (which have an implementation).
Use Cases: Suitable when you want to provide a common base class with shared code and define a template for derived classes. They can also maintain state through fields and properties.

Difference: Abstract Class Can provide default behavior and maintain state while **Interface** Only defines a contract with no implementation or state.

When to Use Abstract Classes:

- **Shared Implementation**: When you need a common implementation to be shared by derived classes.
- **Partial Implementation**: When you want to provide default behavior but still require derived classes to provide specific details.

Create an abstract class Animal with an abstract method MakeSound(). Create derived classes Dog and Cat that implement the MakeSound() method. Demonstrate polymorphism by creating a list of Animal objects and calling MakeSound() on each object.

```
using System;
using System.Collections.Generic;
public abstract class Animal
{
   public abstract void MakeSound();
}
public class Dog : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Dog barks");
   }
}
public class Cat : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Cat meows");
   }
}

class Program
{
   static void Main()
   {
      List<Animal> animals = new List<Animal> { new Dog(), new Cat() };
      foreach (var animal in animals)
      {
         animal.MakeSound();
      }
   }
```

}

Question 19

Describe how a project with a top-down approach can benefit from planning the structure and modules of a large-scale application before implementing the lower level functions. Provide an example project where top-down might be the best approach.

The top-down approach involves planning the high-level structure and modules of an application before diving into the details. It starts with defining the system's overall architecture and then breaks it down into smaller, more manageable components.

Benefits:

- **Clear Structure**: Ensures a clear and well-defined structure before implementation begins.
- **Early Identification**: Helps in identifying potential issues and dependencies early in the design phase.
- **Alignment**: Ensures that all lower-level components align with the overall project goals and architecture.

Example

An ERP system involves various modules like inventory management, finance, human resources, and more. Planning the overall architecture and defining how these modules interact is crucial for creating a cohesive system.

Approach: A top-down approach allows for a clear understanding of the system's functionality and interactions before starting the detailed implementation of each module.

In a bottom-up approach, describe how starting with the implementation of small, independent functions and gradually combining them into larger units can lead to a more flexible and testable application. Provide an example project where bottom-up might be the best approach.

Benefits:

- **Flexibility**: Allows for iterative development and testing of individual components.
- **Modularity**: Encourages modular design, making it easier to test and maintain individual components.
- **Early Functionality**: Provides functional components early in the development process, which can be integrated and expanded upon.

Example Project: **Library Management System**

- **Description**: A library management system involves functionalities like book checkouts, returns, and catalog management. Implementing and testing these features individually allows for a more flexible development process.
- **Approach**: A bottom-up approach is beneficial here as it allows developers to focus on individual functionalities, such as user management or book inventory, before integrating them into the overall system.