RAMADHAN SAGE

SCT221-0816/2021

APPLICATION PROGRAMMING II

BIT2323

Question 1:

James is working on a calculator application. Write a program that performs addition, subtraction, multiplication, and division on two numbers provided by the user.

```
using System;

class Calculator
{
   static void Main(string[] args)
   {
      Console.Write("Enter the first number: ");
      double num1 = Convert.ToDouble(Console.ReadLine());

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

      Console.WriteLine("Addition: " + (num1 + num2));
      Console.`WriteLine("Subtraction: " + (num1  num2));
      Console.WriteLine("Multiplication: " + (num1  num2));
      Console.WriteLine("Division: " + (num1 / num2));
   }
}
```

Question 1(a):

James wants to add a feature to calculate the average of a list of integers. Write a method in that takes an array of integers and returns the average of the scores. Handle edge cases such as an empty array by returning 0.

Answer:

```
using System;

class Calculator
```

```
{
   static void Main(string[] args)
   {
      int[] numbers = { 10, 20, 30, 40, 50 };
      double average = CalculateAverage(numbers);
      Console.WriteLine("Average: " + average);
   }

   static double CalculateAverage(int[] numbers)
   {
      if (numbers.Length == 0)
         return 0;

      int sum = 0;
      foreach (int number in numbers)
      {
         sum += number;
      }
      return (double)sum / numbers.Length;
   }
}
```

Question 1(b):

Mary is developing a system to track object creation. 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 automatically called when an object of the class is created. They are used to initialize the object's state. Constructors differ from other methods as they do not have a return type, not even void. A class can have multiple constructors with different parameters, known as overloaded constructors.

```
using System;

class Product
{
   public string Name;
   public double Price;

   public Product()
   {
```

```
        Name = "Unknown";
        Price = 0.0;
    }
    public Product(string name, double price)
    {
        Name = name;
        Price = price;
    }

    static void Main(string[] args)
    {
        Product product1 = new Product();
        Console.WriteLine(product1.Name + "  " + product1.Price);

        Product product2 = new Product("Laptop", 1500);
        Console.WriteLine(product2.Name + "  " + product2.Price);
    }
}
```

Question 1(c):

Sam is creating an employee management system. 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.

```
using System;

class Employee
{
    public string Name;
    public int ID;
    public string Department;
    public double Salary;

    public Employee(string name, int id)
    {
        Name = name;
        ID = id;
        Department = "Unknown";
        Salary = 0.0;
    }

    public Employee(string name, int id, string department, double salary)
```

```
    {
        Name = name;
        ID = id;
        Department = department;
        Salary = salary;
    }

    static void Main(string[] args)
    {
        Employee emp1 = new Employee("Alice", 101);
        Console.WriteLine($"{emp1.Name}, ID: {emp1.ID}, Dept: {emp1.Department},
Salary: {emp1.Salary}");

        Employee emp2 = new Employee("Bob", 102, "HR", 50000);
        Console.WriteLine($"{emp2.Name}, ID: {emp2.ID}, Dept: {emp2.Department},
Salary: {emp2.Salary}");
    }
}
```

Question 2:

Lucy is developing a program to compare string inputs from users. Explain the difference between the `==` operator and the `Equals()` method in . When should each be used?

The `==` operator is used for reference comparison, meaning it checks if two references point to the same object in memory. The `Equals()` method, on the other hand, is used for value comparison, meaning it checks if two objects have the same value.

Question 2(a):

Predict the output of the following code for a system that compares string values. Explain why each comparison evaluates to either true or false:

```
string str1 = "Hello";
string str2 = "Hello";
string str3 = new string(new char[] { 'H', 'e', 'l', 'l', 'o' });

Console.WriteLine(str1 == str2);
Console.WriteLine(str1 == str3);
Console.WriteLine(str1.Equals(str3));
```

Question 3:

George wants to understand the main components of the .NET Framework for a development project. Explain the role of the Common Language Runtime (CLR) and the Base Class Library (BCL) in the .NET Framework and how they work together to provide a smooth development experience.

The Common Language Runtime (CLR) is the execution engine for .NET applications, providing services like memory management, security, and exception handling. The Base Class Library (BCL) is a collection of reusable types that provide a foundation for building applications. Together, they allow developers to write and execute managed code efficiently.

Question 3(a):

In a library management system, a function needs to handle file operations. 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 Library
{
   static void Main(string[] args)
   {
      string filePath = "books.txt";

      File.WriteAllText(filePath, "Book1\nBook2\nBook3");
      string content = File.ReadAllText(filePath);
      Console.WriteLine("Books List:\n" + content);

       File.AppendAllText(filePath, "Book4\n");
      string updatedContent = File.ReadAllText(filePath);
      Console.WriteLine("Updated Books List:\n" + updatedContent);
   }
}
```

Question 4:

Peter needs to track different data types in his application. Explain the difference between value types and reference types in and provide examples of each. Discuss

scenarios where choosing one type over the other could impact performance or behavior.

Value types (e.g., `int`, `float`, `struct`) are stored in the stack and contain the actual data. Reference types (e.g., `class`, `string`, `object`) are stored in the heap and contain a reference to the data's memory address. Value types are generally faster to access, while reference types offer more flexibility in handling data.

Question 4(a):

Write a program that demonstrates the concept of value types and reference types using primitive data types and objects. Include comparisons between `int` and `string` arrays and their memory addresses using `Object.ReferenceEquals`.

```
using System;

class DataTypeDemo
{
   static void Main(string[] args)
   {
      int a = 10;
      int b = a;
      Console.WriteLine($"Value Type: a={a}, b={b} (b is a copy of a)");

      string[] array1 = { "Hello", "World" };
      string[] array2 = array1;
      Console.WriteLine($"Reference Type: array1==array2:
{Object.ReferenceEquals(array1, array2)}");

      array2[0] = "Hi";
      Console.WriteLine($"Updated array1[0]: {array1[0]}"); // Reflects change in
array2 because they reference the same memory
   }
}
```

Question 5:

Maria wants to design a class in with encapsulation principles. Describe how encapsulation applies to classes and objects in and how it can help control access to fields and methods.

Encapsulation is the concept of wrapping data (fields) and methods into a single unit, the class. It helps control access to the class members by restricting direct access to private fields and exposing only necessary properties or methods. This ens that the

internal representation of the object is hidden from the outside, promoting data integrity.

Question 5(a):

Maria is creating a system for managing people's data. 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 nonnegative.

```
using System;

class Person
{
    private string name;
    private int age;

    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0)
                age = value;
            else
                Console.WriteLine("Age cannot be negative.");
        }
    }

    static void Main(string[] args)
    {
        Person person = new Person();
        person.Name = "John";
        person.Age = 25;
        Console.WriteLine($"{person.Name}, Age: {person.Age}");

        person.Age = 5;
    }
}
```

Question 6:

John is developing an application that uses arrays and enums. Explain the difference between a singledimensional array and a jagged array and provide a use case for each.

A singledimensional array is a linear array, where elements are accessed with a single index. A jagged array is an array of arrays, where each "row" can have a different number of elements. Singledimensional arrays are useful for storing data in a straightforward list, while jagged arrays are suitable for data structures where the size of each "row" varies, such as a list of varyinglength strings.

Question 6(a):

Create a method in that takes a twodimensional array of integers and returns the sum of all its elements. Include support for arrays with irregular shapes or missing values.

```
using System;

class ArraySum
{
   static int SumJaggedArray(int[][] jaggedArray)
   {
      int sum = 0;
      foreach (int[] array in jaggedArray)
      {
         foreach (int value in array)
         {
            sum += value;
         }
      }
      return sum;
   }

   static void Main(string[] args)
   {
      int[][] jaggedArray = new int[][]
      {
         new int[] { 1, 2, 3 },
         new int[] { 4, 5 },
         new int[] { 6, 7, 8, 9 }
      };
```

```
        int sum = SumJaggedArray(jaggedArray);
        Console.WriteLine("Sum of all elements: " + sum);
    }
}
```

Question 6(b):

Emma is designing a color picker for an art application. 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;

enum Color
{
    Red,
    Green,
    Blue
}

class Shape
{
    public class Circle
    {
        public Color CircleColor { get; set; }

        public Circle(Color color)
        {
            CircleColor = color;
        }

        public void DisplayColor()
        {
            Console.WriteLine("Circle color: " + CircleColor);
        }
    }

    static void Main(string[] args)
    {
        Circle circle = new Circle(Color.Green);
        circle.DisplayColor();
    }
}
```

Question 7:

Michael is working on a program that needs to handle various exceptions. Describe how exceptions are handled in using `try`, `catch`, and `finally` blocks. Discuss best practices and potential pitfalls.

Exceptions in are handled using `try`, `catch`, and `finally` blocks. The `try` block contains code that might throw an exception. The `catch` block handles specific exceptions, allowing the program to recover or log the error. The `finally` block is optional and is used to execute code that should run regardless of whether an exception was thrown. Best practices include catching only specific exceptions you can handle and avoiding empty catch blocks that hide errors.

Question 7(a):

For a list management application, write a program that demonstrates handling an exception when trying to access an element outside of the bounds of an array. Introduce nested `trycatch` blocks for different error types.

```
using System;

class ListManager
{
   static void Main(string[] args)
   {
      int[] numbers = { 1, 2, 3 };

      try
      {
         try
         {
            Console.WriteLine(numbers[5]);
         }
         catch (IndexOutOfRangeException ex)
         {
            Console.WriteLine("Index out of range: " + ex.Message);
         }

         int number = int.Parse("invalid");
      }
      catch (FormatException ex)
      {
         Console.WriteLine("Format error: " + ex.Message);
      }
      finally
```

```
        {
            Console.WriteLine("Operation completed.");
        }
    }
}
```

Question 8:

Chloe wants to determine if a number is even, odd, positive, negative, or zero. Write a program that takes an integer input from the user and uses `ifelse` conditions to print the appropriate message.

```
using System;

class NumberCheck
{
    static void Main(string[] args)
    {
        Console.Write("Enter an integer: ");
        int number = int.Parse(Console.ReadLine());

        if (number > 0)
            Console.WriteLine("Positive");
        else if (number < 0)
            Console.WriteLine("Negative");
        else
            Console.WriteLine("Zero");

        if (number % 2 == 0)
            Console.WriteLine("Even");
        else
            Console.WriteLine("Odd");
    }
}
```

Question 8(a):

Explain the differences between `while`, `dowhile`, and `for` loops, and provide examples of each. Discuss scenarios where each loop type would be appropriate.

`while` loop: Repeats a block of code as long as the condition is true. Suitable for situations where the number of iterations isn't known beforehand.

`dowhile` loop: Similar to `while`, but guarantees at least one iteration since the condition is checked after the loop's body. Useful when the loop needs to execute at least once.

`for` loop: Used when the number of iterations is known. It provides initialization, condition, and increment/decrement in one line, making it ideal for counting loops.

Question 8(b):

A sequence generator needs to calculate the factorial of a given number. Write a program using a loop to compute the factorial. Add a twist by calculating the factorial for odd numbers.

```
using System;

class FactorialGenerator
{
   static void Main(string[] args)
   {
      Console.Write("Enter a number: ");
      int number = int.Parse(Console.ReadLine());

      int factorial = 1;
      for (int i = 1; i <= number; i++)
      {
         if (i % 2 != 0)
            factorial = i;
      }

      Console.WriteLine($"Factorial of {number} (odd numbers only): {factorial}");
   }
}
```

Question 8(c):

Write a program that uses nested loops to print a pattern of asterisks in the shape of a rightangled triangle. Add complexity by adjusting the program to print an inverted triangle.

```
using System;
```

```
class TrianglePattern
{
   static void Main(string[] args)
   {
      int n = 5;


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

      Console.WriteLine(); // Separator


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

Question 9:

David is designing a program that uses threads for concurrent execution. Explain the role of threads in . Discuss the main difference between using the `Thread` class and the `Task` class, and provide an example where each would be useful.

Threads in allow multiple operations to run concurrently, improving performance, especially in applications requiring multitasking. The `Thread` class provides lowlevel control over threads, making it suitable for simple scenarios. The `Task` class, part of the Task Parallel Library (TPL), is more powerful and is used for managing asynchronous operations, handling exceptions, and supporting more complex parallelism.

Question 9(a):

Write a 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
{
   static int counter = 0;
   static object lockObject = new object();

   static void IncrementCounter()
   {
      for (int i = 0; i < 10; i++)
      {
         lock (lockObject)
         {
            counter++;
            Console.WriteLine($"Counter: {counter}");
         }
         Thread.Sleep(100);
      }
   }

   static void Main(string[] args)
   {
      Thread thread = new Thread(IncrementCounter);
      thread.Start();

      thread.Join();
      Console.WriteLine("Thread has finished execution.");
   }
}
```

Question 10:

For a news aggregation application, write a 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 Newtonsoft.Json.Linq;

class NewsAggregator
{
   static async Task FetchNewsAsync()
   {
      using (HttpClient client = new HttpClient())
      {
         HttpResponseMessage response = await
client.GetAsync("https://jsonplaceholder.typicode.com/posts");
         response.EnSuccessStatusCode();
         string responseBody = await response.Content.ReadAsStringAsync();

         JArray articles = JArray.Parse(responseBody);
         foreach (var article in articles)
         {
            Console.WriteLine("Title: " + article["title"]);
            Console.WriteLine("Summary: " + article["body"]);
            Console.WriteLine();
         }
      }
   }

   static void Main(string[] args)
   {
      FetchNewsAsync().Wait();
   }
}
```

Question 10(a):

Write a 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 FileProcessor
{
   static void Main(string[] args)
   {
      string inputFile = "input.txt";
      string outputFile = "output.txt";
      string keyword = "filter";

      using (StreamReader reader = new StreamReader(inputFile))
      using (StreamWriter writer = new StreamWriter(outputFile))
      {
         string line;
         while ((line = reader.ReadLine()) != null)
         {
            if (line.Contains(keyword) && line.Length > 20)
            {
               writer.WriteLine(line);
            }
         }
      }

      Console.WriteLine("File processing complete.");
   }
}
```

Question 11:

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

Packages in are collections of reusable code that can be shared across projects. NuGet is a package manager that allows developers to install and manage these packages, making development faster and more consistent. Packages can be installed via Visual Studio or the command line, and they help developers avoid "reinventing the wheel" by providing prebuilt functionality.

Question 11(a):

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

```
using System;
using Newtonsoft.Json;
```

```
class Program
{
   class Address
   {
      public string Street { get; set; }
      public string City { get; set; }
   }

   class Person
   {
      public string Name { get; set; }
      public int Age { get; set; }
      public Address Address { get; set; }
   }

   static void Main(string[] args)
   {

      Person person = new Person
      {
         Name = "John",
         Age = 30,
         Address = new Address { Street = "123 Main St", City = "Anytown" }
      };

      string json = JsonConvert.SerializeObject(person, Formatting.Indented);
      Console.WriteLine("Serialized JSON:\n" + json);


      Person deserializedPerson = JsonConvert.DeserializeObject<Person>(json);
      Console.WriteLine($"\nDeserialized Person:\nName: {deserializedPerson.Name},
Age: {deserializedPerson.Age}, Street: {deserializedPerson.Address.Street}, City:
{deserializedPerson.Address.City}");
   }
}
```

Question 12:

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

`List<T>`: A dynamic array that allows random access and efficient insertions/removals at the end. Suitable for generalpurpose lists.

`Queue<T>`: A firstin, firstout (FIFO) collection. Ideal for scenarios like task scheduling.

`Stack<T>`: A lastin, firstout (LIFO) collection. Useful for undo operations or evaluating expressions.

Question 12(a):

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(string[] args)
   {
      Queue<string> regularQueue = new Queue<string>();
      Queue<string> vipQueue = new Queue<string>();

      regularQueue.Enqueue("Customer 1");
      regularQueue.Enqueue("Customer 2");
      vipQueue.Enqueue("VIP 1");
      regularQueue.Enqueue("Customer 3");
      vipQueue.Enqueue("VIP 2");

      Console.WriteLine("Serving customers:");

      while (vipQueue.Count > 0 || regularQueue.Count > 0)
      {
         if (vipQueue.Count > 0)
            Console.WriteLine(vipQueue.Dequeue() + " (VIP)");
         else if (regularQueue.Count > 0)
            Console.WriteLine(regularQueue.Dequeue());
      }
   }
}
```

Question 13:

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

Inheritance in allows a class to inherit members (fields, methods, properties) from another class. The base class provides the common functionality, and the derived class extends or overrides this functionality. Access modifiers (`public`, `protected`, `private`) control the visibility of members to derived classes. `public` members are accessible from anywhere, `protected` members are accessible within the base and derived classes, and `private` members are only accessible within the base class.

Question 13(a):

Create a base class `Animal` with a method `Speak()`. Create a derived class `Dog` that overrides the `Speak()` method. Add complexity by including additional derived classes like `Cat` and `Bird` and demonstrate polymorphism.

```
using System;

class Animal
{
   public virtual void Speak()
   {
      Console.WriteLine("Animal speaks.");
   }
}

class Dog : Animal
{
   public override void Speak()
   {
      Console.WriteLine("Dog barks.");
   }
}

class Cat : Animal
{
   public override void Speak()
   {
      Console.WriteLine("Cat meows.");
   }
}

class Bird : Animal
{
   public override void Speak()
   {
      Console.WriteLine("Bird chirps.");
   }
```

```
}

class Program
{
   static void Main(string[] args)
   {
      Animal myDog = new Dog();
      Animal myCat = new Cat();
      Animal myBird = new Bird();

      myDog.Speak();
      myCat.Speak();
      myBird.Speak();
   }
}
```

Question 14:

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

Polymorphism in allows objects of different types to be treated as objects of a common base type. This is typically achieved through method overriding, where a derived class provides its own implementation of a method defined in the base class. Polymorphism enables code to be more flexible and reusable, as different objects can be processed in a uniform way.

Question 14(a):

Write a program that demonstrates polymorphism using a base class `Vehicle` and derived classes `Car` and `Bike`. Add complexity by including an interface for `Drive()` and implementing it differently in each derived class.

```
using System;

interface IDrive
{
   void Drive();
}

class Vehicle : IDrive
{
```

```
    public virtual void Drive()
    {
        Console.WriteLine("Vehicle is driving.");
    }
}

class Car : Vehicle
{
    public override void Drive()
    {
        Console.WriteLine("Car is driving.");
    }
}

class Bike : Vehicle
{
    public override void Drive()
    {
        Console.WriteLine("Bike is driving.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        IDrive myCar = new Car();
        IDrive myBike = new Bike();

        myCar.Drive();
        myBike.Drive();
    }
}
```

Question 15:

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

Abstraction in is the concept of hiding the implementation details and showing only the essential features of an object. Abstract classes and interfaces are used to implement abstraction. An abstract class can provide a base for other classes to build upon, while interfaces define a contract that implementing classes must follow.

Abstraction is useful for simplifying complex systems and promoting maintainable, reusable code.

Question 15(a):

Create an abstract base class `Shape` with an abstract method `Draw()`. Create derived classes `Circle` and `Square` that implement the `Draw()` method. Add complexity by introducing additional properties and methods in derived classes.

```
using System;

abstract class Shape
{
    public abstract void Draw();
}

class Circle : Shape
{
    public double Radius { get; set; }

    public Circle(double radius)
    {
        Radius = radius;
    }

    public override void Draw()
    {
        Console.WriteLine($"Drawing a Circle with Radius: {Radius}");
    }
}

class Square : Shape
{
    public double SideLength { get; set; }

    public Square(double sideLength)
    {
        SideLength = sideLength;
    }

    public override void Draw()
    {
        Console.WriteLine($"Drawing a Square with Side Length: {SideLength}");
    }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      Shape myCircle = new Circle(5.5);
      Shape mySquare = new Square(4.0);

      myCircle.Draw();
      mySquare.Draw();
   }
}
```

Question 16:

Predict the output of the following code:

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

Question 16(a):

Predict the output of the following code:

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

The code will print `True` because `StringComparison.OrdinalIgnoreCase` performs a caseinsensitive comparison.

Question 16(b):

Predict the output of the following code:

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

The code will print `False` because `obj1` and `obj2` are different instances of the `object` class, and `==` compares their references, not their content.

Question 16(c):

Predict the output of the following code:

```
int a = 5;
int b = 10;
Console.WriteLine(a += b);
```

The code will print `15` because `a += b` is equivalent to `a = a + b`, so `a` becomes `15`.

Question 17:

Given a list of integers, write a 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 (int min, int max) FindMinMax(List<int> numbers)
   {
      if (numbers == null || numbers.Count == 0)
      {
         return (0, 0);
      }

      int min = numbers.Min();
      int max = numbers.Max();
      return (min, max);
   }

   static void Main(string[] args)
```

```
    {
        List<int> numbers = new List<int> { 3, 7, 1, 9, 4 };
        var result = FindMinMax(numbers);
        Console.WriteLine($"Min: {result.min}, Max: {result.max}");
    }
}
```

Question 17(a):

Write a 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(string[] args)
    {
        HashSet<int> uniqueNumbers = new HashSet<int>();
        int sum = 0;

        while (true)
        {
            Console.Write("Enter an integer (negative to stop): ");
            int number = int.Parse(Console.ReadLine());

            if (number < 0) break;

            if (uniqueNumbers.Add(number))
            {
                sum += number;
            }
        }

        Console.WriteLine($"Sum of unique numbers: {sum}");
    }
}
```

Question 17(b):

Write a 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).

```
using System;

enum DaysOfWeek
{
   Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday
}

class Program
{
   static void Main(string[] args)
   {
      DaysOfWeek today = DaysOfWeek.Saturday;

      switch (today)
      {
         case DaysOfWeek.Saturday:
         case DaysOfWeek.Sunday:
            Console.WriteLine("It's the weekend!");
            break;
         default:
            Console.WriteLine("It's a weekday.");
            break;
      }
   }
}
```

Question 17(c):

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

```
using System;

class Program
{
   static void Main(string[] args)
   {
      Console.Write("Enter a string: ");
      string input = Console.ReadLine();

      char[] charArray = input.ToCharArray();
```

```
        Array.Reverse(charArray);

        Console.WriteLine("Reversed string: " + new string(charArray));
    }
}
```

Question 17(d):

Write a 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(string[] args)
    {
        Dictionary<int, string> studentGrades = new Dictionary<int, string>();

        studentGrades.Add(101, "A");
        studentGrades.Add(102, "B");
        studentGrades.Add(103, "C");

        foreach (var student in studentGrades)
        {
            Console.WriteLine($"Student ID: {student.Key}, Grade: {student.Value}");
        }
    }
}
```

Question 18:

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

Interfaces in define a contract that classes must adhere to, without dictating how the methods are implemented. This allows for flexibility, as different classes can implement the same interface in different ways. Interfaces promote loose coupling, meaning that classes can interact with each other without being tightly dependent on one another's implementations. This makes code more modular, easier to maintain, and more reusable.

Question 18(a):

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;

interface IDrive
{
    void Drive();
}

class Car : IDrive
{
    public void Drive()
    {
        Console.WriteLine("Car is driving.");
    }
}

class Bike : IDrive
{
    public void Drive()
    {
        Console.WriteLine("Bike is driving.");
    }
}

class Program
{
    static void Main(string[] args)
    {
        List<IDrive> vehicles = new List<IDrive>
        {
            new Car(),
            new Bike()
        };

        foreach (var vehicle in vehicles)
        {
            vehicle.Drive();
        }
    }
```

```
}
```

Question 18(b):

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

Abstract classes in provide a base class that cannot be instantiated on its own but can be inherited by other classes. Unlike interfaces, abstract classes can contain both abstract methods (which must be implemented by derived classes) and nonabstract methods (with implementations). Abstract classes are appropriate when there is shared behavior or data that multiple derived classes should inherit.

Question 18(c):

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;

abstract class Animal
{
   public abstract void MakeSound();
}

class Dog : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Dog barks.");
   }
}

class Cat : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Cat meows.");
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      List<Animal> animals = new List<Animal>
      {
         new Dog(),
         new Cat()
      };

      foreach (var animal in animals)
      {
         animal.MakeSound();
      }
   }
}
```

Question 19:

Explain how planning the overall structure and modules of a large-scale application using a top-down approach can be advantageous before developing the detailed components. Provide an example of a project where this method would be ideal.

A top-down approach focuses on outlining the system's high-level architecture first, then breaking it down into smaller, more specific parts. This is particularly advantageous for large-scale projects as it offers a comprehensive view of the system, ensuring that all components work together seamlessly. Additionally, it helps in identifying potential design flaws early, reducing the chances of costly revisions later on.

Example:

Developing an enterprise resource planning (ERP) system, where a top-down approach allows you to define major modules like finance, human resources, and inventory management before detailing each sub-module, ensuring all elements align with the overall business objectives.

Question 19(a):

Describe how beginning with the development of small, independent functions and gradually integrating them into larger units can create a more adaptable and testable

application using a bottom-up approach. Provide an example project where this method would be most suitable.

A bottom-up approach starts by creating and testing smaller, independent modules, which are then combined to form the complete system. This method is beneficial when individual components are complex or when the focus is on flexibility and ease of testing. It enables thorough testing of each module before integration, minimizing errors and enhancing the system's adaptability to changes.

Example:

Building a microservices-based architecture where each service (like user authentication, payment processing, etc.) is developed and tested independently before integrating them into the larger application ensures robustness and flexibility.