GUYATU ROBA JIRMO

SCT221-0851/2022

APPLICATION PROGRAMMING II

Answers

1.<u>CALCULATOR APPLICATION</u>

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

      Console.WriteLine("Enter 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));
   }
}
```

a)<u>CALCULATE AVERAGE</u>

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

   double sum = 0;
```

```
    foreach (int score in scores)
    {
        sum += score;
    }

    return sum / scores.Length;
}
```

B) **Constructors** are special methods used to initialize objects. They are called when an object is created using the new keyword. Unlike other methods, constructors do not have a return type and have the same name as the class. They can be overloaded to provide different ways to initialize an object.

```
public class ObjectTracker
{
    public int ID;
public string Name;

    // Default constructor
public ObjectTracker()
    {
        ID = 0;
        Name = "Unknown";
    }

    // Overloaded constructor
    public ObjectTracker(int id, string name)
    {
        ID = id;
        Name = name;
    }
}
```

C)EMPLOYEE MANAGEMENT SYSTEM

```
public 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 optional parameters
   public Employee(string name, int id, string department, double salary)
   {
       Name = name;
       ID = id;
       Department = department;
       Salary = salary;
   }
}
```

2. **String Comparison in C#**

Difference between == and Equals():

- The == operator compares the reference (for objects) or the value (for primitive types).
- The Equals() method compares the values within the objects.

Console.WriteLine(str1 == str2); =true ,because str1 and str2 refer to the same string object

Console.WriteLine(str1 == str3); =false , because str1 and str3 are different objects

Console.WriteLine(str1.Equals(str3));=true, because the values within the strings are the same

3 **.NET Framework: CLR and BCL**

- **Common Language Runtime (CLR)**: Provides a managed execution environment for .NET programs, handling memory management, type safety, exception handling, and garbage collection.
- **Base Class Library (BCL)**: A library of classes, interfaces, and value types that provides access to system functionality. A)FILE OPERATIONS using System; using System.IO;

```
class LibraryManagement

{

   static void Main()

   {

      string path = "books.txt";


      // Create and write to file

      using (StreamWriter sw = new StreamWriter(path))

      {

         sw.WriteLine("Book1");

sw.WriteLine("Book2");            sw.WriteLine("Book3");

      }


      // Read from file

      using (StreamReader sr = new StreamReader(path))

      {

         string line;

         while ((line = sr.ReadLine()) != null)

         {

            Console.WriteLine(line);

         }

      }

   }

}
```

4 **Value Types vs Reference Types**

- **Value Types**: Stored in the stack and contain the data directly (e.g., int, float).
- **Reference Types**: Stored in the heap, and the stack holds a reference to the data (e.g., string, class).

A) Value Types and Reference Types using

```
System;


class Program

{

   static void Main()

   {

      int x = 10;

int y = x;


      string[] array1 = { "a", "b", "c" };

string[] array2 = array1;


      Console.WriteLine(Object.ReferenceEquals(array1, array2)); // True

Console.WriteLine(Object.ReferenceEquals(x, y)); // False

   }

}
```

5. **Encapsulation in C#**

Encapsulation: A principle of OOP that restricts access to certain components of an object. In C#, it is achieved using access modifiers like private, public, protected, and internal.

a. Person Class with Encapsulation public

```
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;
            }
        }
    }
```

6. **Arrays and Enums**

- **Single-Dimensional Array**: A linear array of elements.
- **Jagged Array**: An array of arrays, where each array can be of different lengths.

   a. Sum of Two-Dimensional Array

```
public static int SumElements(int[,] array)
{
    int sum = 0;     foreach
(int item in array)
    {
        sum += item;
    }
    return sum;
}
```

   b. Color Enum and Shape Class public enum Color

```
{
    Red,
    Green,
    Blue
}

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

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

7. **Exception Handling in C#**

Exception Handling: Using try, catch, and finally blocks to handle errors. The try block contains code that may throw an exception, the catch block handles the exception, and finally executes code regardless of an exception.

a. Exception Handling try

```
{
    int[] array = { 1, 2, 3 };
    Console.WriteLine(array[3]); // IndexOutOfRangeException
}
catch (IndexOutOfRangeException ex)
{
    Console.WriteLine("Error: " + ex.Message);
}
```

8 **Control Structures in C# Loops**:

- **While Loop**: Executes as long as a condition is true.
- **Do-While Loop**: Executes at least once, then repeats if the condition is true.
- **For Loop**: Executes a block of code a specified number of times.

a. Even, Odd, Positive, Negative, or Zero

```
Console.WriteLine("Enter a number:"); int number
= Convert.ToInt32(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"); b
Factorial Calculation public static
int Factorial(int num)
{
   int result = 1;     for (int i =
1; i <= num; i++)
   {
       result *= i;
   }
   return result;
}
c. Right-Angled Triangle int
n = 5;
```

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

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

9 **. Threads in C#**

Threads: Threads allow multiple operations to run concurrently. The Thread class is used for lower-level thread control, while the Task class is used for higher-level asynchronous operations. a. Thread Class Example using System;

```
using System.Threading;

class Program
{
    static void Main()
    {
        Thread thread = new Thread(new ThreadStart(Work));
        thread.Start();
        thread.Join(); // Main thread waits for the worker thread to finish
    }

    static void Work()
    {
        Console.WriteLine("Thread work");
    }
}
```

10. HTTP Request in C#

HTTP Request: Use HttpClient to make HTTP requests

a. Making HTTP Request

```
using System; using
System.Net.Http; using
System.Threading.Tasks;

class Program
{
    static async Task Main()
    {
        HttpClient client = new HttpClient();
        HttpResponseMessage response = await client.GetAsync("https://example.com");
string content = await response.Content.ReadAsStringAsync(); Console.WriteLine(content);

    }
}
```

11. **Purpose of Packages in C# and How to Install and Use a NuGet Package**

Purpose of Packages in C#:

- Packages in C# provide reusable code libraries that can be easily integrated into your project. They simplify development by providing pre-built functionalities, reducing the need to write code from scratch.

- NuGet is the official package manager for .NET, which allows developers to find, install, and use libraries and tools directly within the IDE.

Installing and Using a NuGet Package:

- **Installation:**
    1. Open your project in Visual Studio.
    2. Right-click on the project in the Solution Explorer.
    3. Select "Manage NuGet Packages."
    4. Browse or search for the package you need (e.g., Newtonsoft.Json).
    5. Click "Install" to add the package to your project.
- **Using the Package:** After installation, you can use the package by importing the relevant namespace and invoking its classes and methods in your code.

How Packages Simplify Development and Ensure Code Consistency:

- **Simplification:** Packages encapsulate complex logic into easy-to-use components, allowing developers to focus on higher-level problems.
- **Code Consistency:** By using standardized packages, you ensure consistency across different parts of your application and across different projects, reducing errors and maintenance overhead.

a. C# Program for JSON Serialization and Deserialization Using a NuGet Package

```
using System; using Newtonsoft.Json; using System.Collections.Generic;


public class Address

{

   public string Street { get; set; }

   public string City { get; set; }

}


public class Person

{

   public string Name { get; set; }     public int

Age { get; set; }     public List<Address>

Addresses { get; set; }

}
```

```
class Program
{
  static void Main()
  {
    // Creating a complex object
var person = new Person
    {
      Name = "John Doe",
      Age = 30,
      Addresses = new List<Address>
      {
        new Address { Street = "123 Main St", City = "Springfield" },
new Address { Street = "456 Elm St", City = "Shelbyville" }
      }
    };

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

    // Deserializing from JSON     var deserializedPerson =
JsonConvert.DeserializeObject<Person>(json);
    Console.WriteLine("\nDeserialized Object:");
    Console.WriteLine($"Name: {deserializedPerson.Name}, Age: {deserializedPerson.Age}");
foreach (var address in deserializedPerson.Addresses)
    {
      Console.WriteLine($"Address: {address.Street}, {address.City}");
    }
  }
```

}

12. Differences Between List<T>, Queue<T>, and Stack<T>

- **List<T>**:
  - **Purpose**: A dynamic array that allows random access to elements. ○ **Use Case**:

    When you need to access elements by index or iterate over them in order.
  - **Example**: Storing a collection of items where order matters, like a list of students in a class.
- **Queue<T>**: ○ **Purpose**: A first-in, first-out (FIFO)

  collection.
  - **Use Case**: When you need to process elements in the order they were added, like task scheduling or managing a line.
  - **Example**: Managing customer service requests where the first request should be handled first.
- **Stack<T>**: ○ **Purpose**: A last-in, first-out (LIFO)

  collection.
  - **Use Case**: When you need to reverse order or backtrack, like undo functionality in text editors. ○ **Example**: Evaluating mathematical expressions using the stack to hold operands and operators.

Example: Queue in a Bank with VIP Priority: using

```
System;

using System.Collections.Generic;


class Program

{

   static void Main()

   {

      Queue<string> queue = new Queue<string>();

      Queue<string> vipQueue = new Queue<string>();
```

```
        // Add regular customers
queue.Enqueue("Customer 1");
queue.Enqueue("Customer 2");

        // Add VIP customers
vipQueue.Enqueue("VIP Customer 1");

        // Process VIPs first
while (vipQueue.Count > 0)
        {
            Console.WriteLine(vipQueue.Dequeue() + " is being served.");
        }

        // Process regular customers
while (queue.Count > 0)
        {
            Console.WriteLine(queue.Dequeue() + " is being served.");
        }
    }
}
```

13. Inheritance in C# Inheritance:

- **Purpose**: Inheritance allows a class (derived class) to inherit fields, methods, and properties from another class (base class).
- **Implementation**: Use the : symbol to indicate inheritance in C#. Access modifiers like public, protected, and private determine the accessibility of the members in the derived classes.

Example: Animal Hierarchy with Speak() Method:

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

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

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

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

class Program
{
    static void Main()
    {
        Animal[] animals = { new Dog(), new Cat(), new Bird() };

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

14. Polymorphism in C# Polymorphism:

- **Purpose**: Polymorphism allows methods to do different things based on the object it is acting upon, even if they share the same method name.
- **Implementation**: Achieved through method overriding in derived classes and interfaces.

```
public interface IDrive
{
    void Drive();
}

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

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

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

class Program
{
    static void Main()
    {
        IDrive[] vehicles = { new Car(), new Bike() };

        foreach (IDrive vehicle in vehicles)
        {
            vehicle.Drive();  // Demonstrates polymorphism
        }
    }
}
```

15. Abstraction in C# Abstraction:

- **Purpose**: Abstraction hides the implementation details and only exposes the necessary features of an object.
- **Implementation**: Achieved using abstract classes and interfaces.

Example: Shape with Abstract Draw() Method:

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

public 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);
    }
}

public 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()
    {
        Shape[] shapes = { new Circle(5), new Square(4) };

        foreach (Shape shape in shapes)
        {
            shape.Draw();  // Demonstrates abstraction and polymorphism
        }
    }
}
```

16.predicting code output

Output=1,2,3,4,5

b. Predict the Output of the Following Code:

true

c. Predict the Output of the Following Code: false

d. Predict the Output of the Following Code:

15

17. Handling Collections and Data Types

a. Method to Return the Largest and Smallest Integers in a List**:**

```
public (int largest, int smallest) FindLargestAndSmallest(List<int> numbers)
{
    if (numbers == null || numbers.Count == 0)        throw
new ArgumentException("List cannot be empty");
```

```
    int largest = numbers.Max();
int smallest = numbers.Min();

    return (largest, smallest);
}
```

b. Program to Read Integers Until a Negative Number is Entered:

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        HashSet<int> uniqueNumbers = new HashSet<int>();
        int sum = 0;

        while (true)
        {
            int input = int.Parse(Console.ReadLine());

            if (input < 0)
break;

            if (uniqueNumbers.Add(input))
            {
                sum += input;
            }
        }
        Console.WriteLine("Sum of unique numbers: " + sum);
    }
}
```

c. Program to Reverse a String

Here's a simple 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();

        char[] charArray = input.ToCharArray();
        Array.Reverse(charArray);

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

d. Using Dictionary<TKey, TValue> to Store and Retrieve Student Grades

This program demonstrates how to use a Dictionary<TKey, TValue> to store and retrieve student grades, handling different data types for keys and values:

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        Dictionary<int, Dictionary<string, double>> studentGrades = new Dictionary<int, Dictionary<string,
double>>();

        // Adding student grades
        studentGrades[1] = new Dictionary<string, double>
        {
            { "Math", 85.5 },
            { "Science", 92.3 }
        };

        studentGrades[2] = new Dictionary<string, double>
        {
            { "Math", 78.9 },
            { "Science", 88.5 }
        };

        // Retrieving and displaying student grades
foreach (var student in studentGrades)
        {
            Console.WriteLine("Student ID: " + student.Key);

            foreach (var grade in student.Value)
            {
                Console.WriteLine("Subject: " + grade.Key + ", Grade: " + grade.Value);
            }
        }
    }
}
```

18. Purpose and Benefits of Using Interfaces in C# Purpose

of Interfaces:

- Contractual Obligations: Interfaces define a contract that classes must adhere to, ensuring consistency in implementation.
- Loose Coupling: Interfaces promote loose coupling by separating the definition of methods from their implementation. This allows for more flexible and interchangeable components.
- Code Reusability: Since different classes can implement the same interface, it allows for code reuse across different parts of an application.

Example: IDrive Interface and Polymorphism:

```
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 driving");
   }
}


class Program
```

```
{
   static void Main()
   {
      List<IDrive> vehicles = new List<IDrive> { new Car(), new Bike() };


      foreach (IDrive vehicle in vehicles)
      {
         vehicle.Drive();  // Demonstrates polymorphism
      }
   }
}
```

b. Role of Abstract Classes in C# and Comparison with Interfaces Abstract

Classes:

- Purpose: Abstract classes provide a base class with some implementation details while leaving other methods abstract (to be implemented by derived classes).
- Comparison with Interfaces:
    - Abstract classes can have method implementations, fields, and constructors, whereas interfaces can only have method signatures (prior to C# 8.0).
    - Use abstract classes when you need to share code among several closely related classes.
    - Use interfaces when you want to define a contract for a broader range of unrelated classes.

Scenarios:

- Abstract Class: If you have a base class Animal that shares common code among animals (e.g., Eat() method) but needs specific MakeSound() implementations for each type of animal.
- Interface: If you have various classes like Car, Bicycle, and Boat that need to implement a Move() method, but they are otherwise unrelated.

Example: Animal Abstract Class:

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

```
    public void Eat()
    {
        Console.WriteLine("Animal is eating");
    }
}

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

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

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

        foreach (Animal animal in animals)
        {
            animal.MakeSound();  // Demonstrates polymorphism
animal.Eat();  // Common method from abstract class
        }
    }
}
```

19. Top-Down Approach in Project Development Top-Down

Approach:

- Benefits:
  - Structured Planning: By designing the high-level structure first, it ensures that the application is well-organized and that all components fit together. o Clear Vision: It provides a clear roadmap for development, helping teams to understand how individual components contribute to the overall system.
  - Integration Ease: Ensures that lower-level functions are built to fit into the larger system, reducing integration issues later.

Example Project:

- Enterprise Resource Planning (ERP) System: In an ERP system, the top-down approach can be beneficial. You start by designing the overall architecture, including modules like finance, HR, and inventory, before diving into the implementation of specific functionalities like payroll processing or inventory tracking.

Bottom-Up Approach:

- Benefits:
  - Flexibility: By focusing on small, independent functions first, developers can experiment with different approaches and refactor them as needed.
  - Testability: Smaller functions are easier to test in isolation, leading to more reliable components.
  - Incremental Development: It allows for gradual integration and testing, making it easier to identify and fix issues early.

Example Project:

- Library Management System: A bottom-up approach might be suitable. Start by developing small, independent functions like book search, check-in/check-out, and user registration. These can be tested individually and then integrated into a larger system.

This approach ensures that the system remains flexible and can be adapted as new requirements emerge.