TIMOTHY KARIUKI GITONGA

SCT221-0850/2021

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

C#

```
using System;

class Calculator

{

   static void Main()

   {

      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("Select an 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.");
            break;
        default:
            Console.WriteLine("Invalid operation.");
            break;
    }

    if (operation == "+" || operation == "-" || operation == "*" || (operation == "/" && num2 !=
0))
    {
        Console.WriteLine("The result is: " + result);
    }
}
```

}

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

c#

```
using System;


class Calculator
{
    static void Main()
    {
        int[] numbers = { 10, 20, 30, 40, 50 };
        double average = CalculateAverage(numbers);
        Console.WriteLine("The average is: " + average);
    }


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


        int sum = 0;
        foreach (int num in nums)
        {
```

```
            sum += num;
        }
        return (double)sum / nums.Length;
    }
}
```

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.*

Explanation:

Constructors are special methods in a class that are automatically called when an object of that class is created. They are used to initialize the object's state. Unlike other methods, constructors do not have a return type, not even void. Constructors can be overloaded to allow different ways of initializing objects.

Example:

csharp

```
using System;
class ObjectTracker
{
    public string ObjectName { get; set; }
    public int ObjectID { get; set; }
    // Default constructor
    public ObjectTracker()
    {
        ObjectName = "Unknown";
        ObjectID = 0;
    }
```

```
    // Overloaded constructor
    public ObjectTracker(string name, int id)
    {
        ObjectName = name;
        ObjectID = id;
    }
    static void Main()
    {
       // Using default constructor
        ObjectTracker obj1 = new ObjectTracker();
        Console.WriteLine($"Object1: Name = {obj1.ObjectName}, ID = {obj1.ObjectID}");
        // Using overloaded constructor
        ObjectTracker obj2 = new ObjectTracker("MyObject", 1234);
        Console.WriteLine($"Object2: Name = {obj2.ObjectName}, ID = {obj2.ObjectID}");
    }
}
```

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.*

c#

```
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;
        Department = "Unknown";
        Salary = 0.0;
    }
    // Overloaded constructor with optional parameters
    public Employee(string name, int id, string department = "Unknown", double salary = 0.0)
    {
        Name = name;
        ID = id;
        Department = department;
        Salary = salary;
    }
    static void Main()
    {
        // Creating an instance using the first constructor
        Employee emp1 = new Employee("John Doe", 1001);
        Console.WriteLine($"Employee1: Name = {emp1.Name}, ID = {emp1.ID}, Department = {emp1.Department}, Salary = {emp1.Salary}");
```

```
 // Creating an instance using the overloaded constructor

        Employee emp2 = new Employee("Jane Smith", 1002, "HR", 50000);

        Console.WriteLine($"Employee2: Name = {emp2.Name}, ID = {emp2.ID}, Department = {emp2.Department}, Salary = {emp2.Salary}");

    }

}
```

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

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));
```

= Operator:

- The == operator is used to compare the values of two operands.
- In C#, when used with reference types like strings, == checks for reference equality by default, meaning it compares whether both operands refer to the same object in memory.
- However, the == operator is overloaded in the System.String class to compare the contents of strings rather than their references.

Equals() Method:

- The Equals() method is a method defined in the System.Object class and is overridden in the System.String class to perform a content comparison of two strings.
- Equals() checks if the contents of the two strings are the same, regardless of whether they are stored in the same memory location or not.
- Use Equals() when you want to ensure that you are performing a value-based comparison rather than relying on any potential operator overloading.

When to Use Each:

- == operator: Use it for general comparisons, especially when dealing with primitive types or when you are certain you want to check value equality in strings.

- Equals() method: Use it when you specifically need to compare the contents of objects, especially when working with custom objects or ensuring content comparison without ambiguity.

Code Example and Explanation

C#

```
using System;


class Program
{
    static void Main()
    {
        string str1 = "Hello";
        string str2 = "Hello";
        string str3 = new string(new char[] { 'H', 'e', 'l', 'l', 'o' });


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

Output and Explanation

1. Console.WriteLine(str1 == str2);

- Output: True

- Explanation: Both str1 and str2 refer to the same string literal "Hello" in the string pool. Since the == operator is overloaded for strings to compare values, it returns True.

2. Console.WriteLine(str1 == str3);

- Output: True

- Explanation:* str3 is created using the new keyword, so it refers to a different object in memory. However, the == operator compares the values of the strings, not their references, so it returns True because the content of str1 and str3 is the same ("Hello").

3. Console.WriteLine(str1.Equals(str3));*

- Output: True

- Explanation: The Equals() method compares the content of str1 and str3. Since both contain the same string "Hello", it returns True.

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.

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.

1. Common Language Runtime (CLR):*

- Role: The CLR is the execution engine of the .NET Framework. It manages the execution of .NET programs, providing essential services like memory management, security, and exception handling. The CLR also handles Just-In-Time (JIT) compilation, converting the Intermediate Language (IL) code into native machine code that the operating system can execute.

- How it Works:

  - Memory Management:The CLR automatically handles memory allocation and deallocation for managed objects through a process known as garbage collection, which helps prevent memory leaks.

  - Type Safety and Security: The CLR enforces strict type safety and manages code access security, ensuring that the code behaves as expected and doesn't perform unauthorized operations.

  - Exception Handling: The CLR provides a structured way to handle exceptions and errors, making the code more robust and easier to debug.

2. Base Class Library (BCL):

- Role: The BCL is a comprehensive collection of reusable classes, interfaces, and value types that provide a wide range of functionalities. These include file I/O, data access, XML processing, network communications, and more. The BCL is part of the larger .NET Framework Class Library (FCL).

- How it Works:

  - Common Functionality: The BCL provides fundamental building blocks that developers can use to build applications, reducing the need to write common functionalities from scratch.

- Interoperability: The BCL provides a consistent set of APIs that work across different .NET languages (e.g., C#, VB.NET), making it easier for developers to switch between languages without having to learn different libraries.

How They Work Together:

- The CLR and BCL work in tandem to provide a powerful, yet simplified development experience. The CLR ensures that code runs safely and efficiently, while the BCL provides a rich set of tools to build applications. This combination allows developers to focus more on solving business problems rather than dealing with low-level system details.

Example Program: File Operations in a Library Management System

Here's an example program that demonstrates how to use the System.IO.File class in C# to create, read, and write to a file containing a list of books.

C#

```
using System;

using System.IO;

class LibraryManagementSystem

{

   static void Main()

   {

      string filePath = "books.txt";


      // 1. Create a file and write a list of books to it

      WriteToFile(filePath);


      // 2. Read the contents of the file

      ReadFromFile(filePath);

   }
```

```
static void WriteToFile(string filePath)
{
    string[] books = {
        "The Great Gatsby by F. Scott Fitzgerald",
        "To Kill a Mockingbird by Harper Lee",
        "1984 by George Orwell",
        "Pride and Prejudice by Jane Austen"
    };

    try
    {
        // Writing the array of books to the file
        File.WriteAllLines(filePath, books);
        Console.WriteLine("Books have been written to the file.");
    }
    catch (Exception ex)
    {
        Console.WriteLine("An error occurred while writing to the file: " + ex.Message);
    }
}

static void ReadFromFile(string filePath)
{
    try
```

```
        {
            if (File.Exists(filePath))
            {
                // Reading all lines from the file
                string[] books = File.ReadAllLines(filePath);
                Console.WriteLine("Reading from file:");
                foreach (string book in books)
                {
                    Console.WriteLine(book);
                }
            }
            else
            {
                Console.WriteLine("File does not exist.");
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("An error occurred while reading from the file: " + ex.Message);
        }
    }
}
```

- File Creation and Writing (WriteToFile):

  - The WriteToFile method creates a new text file (books.txt) and writes a list of book titles to it. The File.WriteAllLines method is used to write an array of strings to the file.

- File Reading (ReadFromFile):

  - The ReadFromFile method checks if the file exists using File.Exists. If it exists, it reads all the lines from the file using File.ReadAllLines and then prints each book title to the console.

- Error Handling:

  - The program includes basic exception handling to manage any errors that might occur during file operations, such as the file being inaccessible or non-existent.

4 Peter needs to track different data types in his application. Explain the difference between value types and reference types in C# and provide examples of each. Discuss scenarios where choosing one type over the other could impact performance or behavior.

a. Write a C# 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.

Value Types:

- Memory Allocation: Value types store their data directly in the memory allocated on the stack.

- Examples: Common value types include primitive types like int, float, bool, char, and structs.

- Behavior: When a value type is assigned to another variable, a copy of the value is made. This means changes to one variable do not affect the other.

Reference Types:

- Memory Allocation: Reference types store a reference (or pointer) to the data, which is stored on the heap. The reference itself is stored on the stack.

- Examples: Common reference types include string, arrays, class objects, and delegates.

- Behavior: When a reference type is assigned to another variable, both variables point to the same memory location. Changes made through one reference affect the object referenced by both.

1. Performance:

- Value Types: Since they are stored on the stack, value types can be faster for small, short-lived data because stack allocation and deallocation are quicker.

- Reference Types: For large or complex objects that need to be shared across different parts of an application, reference types are better. However, heap allocation and garbage collection can incur a performance cost.

2. Memory Usage:

- Value Types: Generally use less memory because the data is stored directly.

- Reference Types: Can use more memory due to the overhead of storing references and the data separately.

3. Behavioral Impact:

- Value Types: Suitable when you need independent copies of data.

- Reference Types: Necessary when you want changes in one variable to be reflected across all references.

C# Program Demonstrating Value Types and Reference Types

C#

```
using System;

class Program
{
    static void Main()
    {
        // Value Type Example
```

```
int a = 10;

int b = a;

b = 20;

Console.WriteLine("Value Type:");

Console.WriteLine($"a: {a}, b: {b}"); // a: 10, b: 20


// Reference Type Example

int[] array1 = { 1, 2, 3 };

int[] array2 = array1;

array2[0] = 99;

Console.WriteLine("\nReference Type:");

Console.WriteLine($"array1[0]: {array1[0]}, array2[0]: {array2[0]}"); // array1[0]: 99, array2[0]: 99


// Using Object.ReferenceEquals to compare memory addresses

Console.WriteLine("\nMemory Address Comparison:");

Console.WriteLine(Object.ReferenceEquals(a, b)); // False, because they are value types

Console.WriteLine(Object.ReferenceEquals(array1, array2)); // True, because they refer to the same object


// String Example (Reference Type)

string str1 = "hello";

string str2 = str1;

str2 = "world";

Console.WriteLine("\nString Reference Type:");

Console.WriteLine($"str1: {str1}, str2: {str2}"); // str1: hello, str2: world
```

```
        // Comparing strings using Object.ReferenceEquals
        Console.WriteLine("\nString Memory Address Comparison:");
        Console.WriteLine(Object.ReferenceEquals(str1, str2)); // False, because strings are immutable and a new object is created for "world"
    }
}
```

- Value Type Example: a and b are separate copies. Modifying b does not affect a.

- Reference Type Example: array1 and array2 reference the same array object. Changing one affects the other.

- Memory Address Comparison:

  - Value Types: Object.ReferenceEquals returns False because a and b are independent.

  - Reference Types: Returns True for arrays because both variables point to the same memory location. For strings, since they are immutable, str2 points to a different location after reassignment.

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

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.

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It involves bundling the data (fields) and methods that operate on the data into a single unit, or class, and restricting access to some of the object's components. This is done to prevent the outside world from accessing and modifying the internal state of an object in ways that are not intended.

In C#, encapsulation is typically implemented using access modifiers such as private, protected, and public. Fields are often made private to hide them from outside access, and public properties are used to provide controlled access to these fields. This allows for validation and logic to be applied when getting or setting a value.

Here’s how Maria could design a Person class with private fields and public properties that include validation:

C#

```
using System;

class Person
{
    // Private fields
    private string name;
    private int age;

    // Public property for Name with no validation (just an example)
```

```
public string Name
{
    get { return name; }
    set { name = value; }
}
// Public property for Age with validation
public int Age
{
    get { return age; }
    set
    {
        if (value >= 0) // Ensuring age is non-negative
        {
            age = value;
        }
        else
        {
            throw new ArgumentException("Age cannot be negative.");
        }
    }
}
// Constructor
public Person(string name, int age)
{
    Name = name; // Uses the property to allow future validation
```

```
        Age = age;   // Uses the property to enforce the validation
    }
}
class Program
{
    static void Main()
    {
        try
        {
            Person person = new Person("Maria", 25);
            Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");

            // Attempting to set a negative age
            person.Age = -5; // This will throw an exception
        }
        catch (ArgumentException ex)
        {
            Console.WriteLine(ex.Message);
        }
    }
}
```

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

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

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.

Single-Dimensional Array:

A single-dimensional array is a linear array where elements are accessed using a single index. It's essentially a list of elements in sequence.

Jagged Array:

A jagged array is an array of arrays, meaning that each element in the main array is itself an array. The sub-arrays can have different lengths, making jagged arrays more flexible for handling irregular data structures.

Use Cases:

- Single-Dimensional Array: A simple list of temperatures recorded over a week.

- Jagged Array: A list where each element is a collection of test scores for students, with each student having a different number of scores.

Example: Method to Sum Elements in a Two-Dimensional Array

Here's a method that sums up all elements in a two-dimensional array, handling irregular shapes:

C#

```
using System;


class ArraySumExample
{
    static void Main()
    {
        int[,] regularArray =
        {
            { 1, 2, 3 },
            { 4, 5, 6 },
            { 7, 8, 9 }
        };
        int sum = SumOfElements(regularArray);
        Console.WriteLine($"Sum of all elements: {sum}");
    }
    static int SumOfElements(int[,] array)
    {
        int sum = 0;


        for (int i = 0; i < array.GetLength(0); i++)
        {
            for (int j = 0; j < array.GetLength(1); j++)
            {
                sum += array[i, j];
```

```
            }
        }

        return sum;
    }
}
```

Jagged Array Example:

C#

```
using System;
class JaggedArrayExample
{
    static void Main()
    {
        int[][] jaggedArray = new int[][]
        {
            new int[] {1, 2, 3},
            new int[] {4, 5},
            new int[] {6, 7, 8, 9}
        };
        int sum = SumOfElements(jaggedArray);
        Console.WriteLine($"Sum of all elements in jagged array: {sum}");
    }
    static int SumOfElements(int[][] array)
    {
        int sum = 0;
```

```
        for each (var sub Array in array)
        {
            For each (var element in sub Array)
            {
                sum += element;
            }
        }

        return sum;
    }
}
```

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

a. For a list management 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.

Exception Handling* in C# is critical for managing errors and exceptional situations in a controlled manner. The try, catch, and finally blocks are the primary constructs used for handling exceptions.

Try Block:

Contains the code that might throw an exception.

Catch Block:

Handles the exception that was thrown in the try block. You can catch specific exceptions or use a general catch for any exception.

Finally Block:

Executes code regardless of whether an exception was thrown or not, typically used for cleaning up resources.

Best Practices:

- Only catch exceptions you can handle meaningfully.

- Avoid catching general exceptions unless you have a good reason to handle all types of exceptions.

- Use finally blocks to ensure that resources are always released, such as closing files or database connections.

C#

```
using System;

class Exception Handling Example

{

   static void Main()

   {

      int[] array = { 1, 2, 3 };


      try

      {

         // Nested try-catch blocks for different exception types

         try

         {

            Console.Write Line(array[3]); // This will throw an Index Out Of Range Exception

         }

         catch (Index Out Of Range Exception ex)

         {

            Console. WriteLine($"Index out of bounds: {ex .Message}");

         }


         int result = 10 / 0; // This will throw a Divide By Zero Exception

      }
```

```
        catch (Divide By Zero Exception ex)
        {
            Console. WriteLine($"Cannot divide by zero: {ex .Message}");
        }
        finally
        {
            Console. WriteLine("Cleanup code, if any, would go here.");
        }
    }
}
```

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

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

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.

c. 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.

exception Handling in C#:

- try Block: The try block contains the code that may potentially throw an exception. When an exception occurs, the program control is transferred to the appropriate catch block.

- catch Block: The catch block handles the exception thrown by the try block. You can have multiple catch blocks to handle different types of exceptions specifically. If an exception occurs that matches the type specified in a catch block, that block is executed.

- finally Block: The finally block is used for code that must be executed regardless of whether an exception was thrown or not. This is typically used for cleaning up resources like closing files, releasing memory, or closing database connections.

- Catch Specific Exceptions: Always try to catch specific exceptions rather than a general Exception class. This helps in identifying and fixing specific issues.

- Use Finally for Cleanup: Always include a finally block for cleanup activities, especially if you are working with resources that need to be released.

- Avoid Silent Catches: Avoid catching exceptions without handling them (e.g., empty catch blocks). Always log the exception or handle it appropriately.

- Throw New Exceptions Sparingly: Avoid throwing new exceptions within a catch block unless necessary, as it may obscure the original error.

- Use Exception Filters: You can use exception filters (when keyword) to handle exceptions more precisely.

challenges

- Overusing Exception Handling: Relying too much on exception handling can lead to code that is hard to understand and maintain. Exceptions should be used for exceptional situations, not for regular control flow.

- Catching General Exceptions: Catching the base Exception class without considering specific exception types can hide errors and make debugging difficult.

Part A: C# Program Demonstrating Exception Handling with Nested Try-Catch Blocks

C#

```
using System;


class List ManagementApp
{
    public static void Main()
    {
```

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


try
{
    try
    {
        // Attempt to access an element outside the bounds of the array
        Console. WriteLine(numbers[10]);
    }
    catch (Index Out Of Range Exception ex)
    {
        Console. WriteLine($"Index Out Of Range Exception caught: {ex. Message}");
    }
    catch (Exception ex)
    {
        Console. WriteLine($"General exception caught in nested block: {ex. Message}");
    }
}
catch (Exception ex)
{
    // Handling any other exceptions that were not caught in the inner try-catch
    Console. WriteLine($"General exception caught in outer block: {ex.Message}");
}
finally
{
```

```
            Console. WriteLine("Cleanup operations can be performed here.");
        }
    }
}
```

Explanation:

- The program attempts to access an array element at index 10, which is out of bounds for the numbers array (which only has 5 elements).

- The inner try-catch block catches the Index Out Of Range Exception and prints an appropriate message.

- The outer try-catch block is ready to catch any exceptions that were not handled by the inner block.

- The finally block will execute regardless of whether an exception occurred, ensuring any necessary cleanup is performed.

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

a. 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.

Threads in C# are used for concurrent execution of code, which allows multiple operations to run simultaneously. This can enhance application performance and responsiveness. Here's how they are used:

- Concurrency: Enables multiple tasks to make progress simultaneously.
- Parallelism: Utilizes multiple CPU cores to execute threads in parallel, which can improve performance for CPU-intensive operations.
- Responsiveness: Ensures applications remain responsive, especially in scenarios where tasks might take time to complete, such as file I/O operations or network requests.

Difference Between Thread Class and Task Class

Thread Class:

- *Low-Level Control:* Provides direct control over thread creation, management, and synchronization.
- *Manual Management:* Requires explicit management of thread lifecycle, synchronization, and exception handling.
- *Use Case:* Suitable for scenarios where you need precise control over threading and are managing a small number of threads.

Example Using Thread Class:

C#

```
using System;
using System. Threading;

class Program
{
    private static int  shared Resource = 0;
    private static read only object lock Object = new object();

    static void Main()
    {
        Thread thread = new Thread (IncrementSharedResource);
        thread. Start();

        // Wait for the new thread to complete
        thread. Join();

        // Output the result
        Console. WriteLine($"Final value of shared Resource: {shared Resource}");
    }

    static void Increment Shared Resource()
    {
        for (int i = 0; i < 1000; i++)
        {
```

```
            lock (lock Object) // Synchronize access to the shared resource
            {
                Shared Resource++;
            }
        }
    }
}
```

Task Class:

- Higher-Level Abstraction: Provides a higher-level abstraction for managing asynchronous operations.

- Automatic Management: Handles thread management internally, which simplifies code and reduces the risk of errors.

- Use Case: Ideal for concurrent programming where tasks need to be managed in an asynchronous manner, and provides built-in support for continuations and asynchronous operations.

Example Using Task Class:

C #

```
using System;
using System. Threading. Tasks;

class Program
{
    private static int shared Resource = 0;
    private static readonly object lock Object = new object();
```

```
    static async Task Main()
    {
        Task task = Task. Run(() => Increment Shared Resource());

        // Await the task to complete
        await task;

        // Output the result
        Console. WriteLine($"Final value of shared Resource: {shared Resource}");
    }

    static void Increment Shared Resource()
    {
        for (int i = 0; i < 1000; i++)
        {
            lock (lock Object) // Synchronize access to the shared resource
            {
                Shared Resource++;
            }
        }
    }
}
```

C# Program Using the Thread Class with Synchronization

Here’s a C# program that demonstrates how to use the Thread class to create and start a new thread. It includes synchronization using a lock to ensure thread safety:

C#

```
using System;
using System. Threading;
class Program
{
    private static int shared Resource = 0;
    private static read only object lock Object = new object();

    static void Main()
    {
        Thread thread1 = new Thread(Increment Shared Resource);
        Thread thread2 = new Thread(Increment Shared Resource);
        thread1.Start();
        thread2.Start();
        // Wait for both threads to complete
        thread1.Join();
        thread2.Join();
        // Output the result

        Console. WriteLine ($"Final value of shared Resource: {shared Resource}");
```

```
    }

    static void Increment Shared Resource()
    {
        for (int i = 0; i < 1000; i++)
        {
            lock (lock Object) // Synchronize access to the shared resource
            {
                Shared Resource++;
            }
        }
    }
}
```

Explanation:

- Threads Creation: Two threads (thread1 and thread2) are created to run the Increment Shared Resource method concurrently.

- Synchronization: The lock statement is used to synchronize access to sharedResource, ensuring that only one thread can modify it at a time.

- Joining Threads: The Join method is used to wait for both threads to complete before proceeding.

- Output: After both threads complete, the final value of sharedResource is printed.

10. For a news aggregation application, 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.

a. 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.

sharp

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq; // Install-Package Newtonsoft.Json

class Program
{
    Static  async Task Main()
    {
        string url = "https://jsonplaceholder.typicode.com/posts"; // Public API endpoint
        using Http Client client = new Http Client();

        try
        {
            Http Response Message response = await client .GetAsync(url);
            Response .Ensure Success StatusCode();
            string response Body = await response .Content .Read As String Async ();
```

```
        // Parse JSON data
        JArray  json Array = JArray.Parse(responseBody);

        // Display article titles and summaries
        Fo reach (var item in json Array)
        {
            string title = item["title"]?.ToString();
            string summary = item["body"]?.ToString();
            Console. WriteLine($"Title: {title}");
            Console. WriteLine($"Summary: {summary}");
            Console. WriteLine(new string('-', 40));
        }
    }
    catch (Http Request Exception e)
    {
        Console. WriteLine($"Request error: {e. Message}");
    }
  }
}
```

Explanation:

- HttpClient: Used to make an asynchronous GET request to the API.
- Parsing JSON: The response is parsed into a JArray using the Newtonsoft.Json library.
- Displaying Data: Extracts and displays the title and summary of each article.

- Error Handling: Catches and displays request errors

C#

```
using System;
using System.IO;
using System. Linq;

class Program
{
   static void Main()
   {
      string input FilePath = "input.txt";
      string output FilePath = "output.txt";
      string[] keywords = { "important", "urgent" }; // Keywords for filtering
      int min Length = 20; // Minimum length of lines to include

      try
      {
         // Read lines from the input file
         var lines = File. Read All Lines(inputFilePath);

         // Filter lines based on keywords and length
         var filtered Lines = lines
            .Where(line => line.Length >= minLength && keywords.Any(keyword =>
line.Contains(keyword, StringComparison.OrdinalIgnoreCase)))
```

```
            .To Array();


        // Write filtered lines to the output file
        File .WriteAllLines(outputFilePath, filteredLines);


        Console. WriteLine($"Filtered lines have been written to {outputFilePath}");
    }
    catch (IO Exception e)
    {
        Console.WriteLine ($"File error: {e.Message}");
    }
  }
}
```

Explanation:

- Reading Lines: Reads all lines from the input file.

- Filtering Lines: Filters lines based on a minimum length and whether they contain any of the specified keywords.

- Writing Lines: Writes the filtered lines to a new output file.

- Error Handling: Catches and displays file-related

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.

a. Write a C# program that uses a NuGet package to perform JSON serialization

and deserialization. Add a twist by handling complex nested JSON structures.

Packages* in C# are used to distribute reusable code components and libraries. They simplify development by providing pre-built functionality that developers can easily integrate into their projects. This approach promotes code reuse, helps in managing dependencies, and ensures consistency across different projects.

Benefits of Using Packages:

- Code Reusability: Packages allow you to reuse code across multiple projects without having to rewrite or copy it.

- Dependency Management: Packages handle external dependencies, so you don’t need to manually include and manage libraries.

- Version Control: NuGet packages include versioning, which helps in managing different versions of dependencies and ensures compatibility.

- Consistency: Using well-maintained packages ensures that you are using tested and reliable code, leading to more consistent and stable applications.

Installing and Using a NuGet Package

1. Install a NuGet Package:

- Open your project in Visual Studio.

- Go to Tools > NuGet Package Manager > Manage NuGet Packages for Solution*.

- Search for the package you want to install (e.g., Newtonsoft.Json).

- Select the package and click *Install*.

Alternatively, you can use the Package Manager Console:

bash

Install-Package Newtonsoft.Json

2. Use a NuGet Package in Your Code:

- Add using directives for the namespaces provided by the package.
- Use the package's classes and methods as needed.

Example: Using a NuGet Package for JSON Serialization and Deserialization

Let's use the Newtonsoft.Json package to handle JSON serialization and deserialization, including complex nested JSON structures. First, install the Newtonsoft.Json package as described above.

Here's a C# program demonstrating JSON serialization and deserialization:

C#

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

class Program
{
    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 Address Address { get; set; }
    public string[] PhoneNumbers { get; set; }
}

static void Main()
{
    // Sample JSON with nested structure
    string json = @"
    {
        'Name': 'John Doe',
        'Age': 30,
        'Address': {
            'Street': '123 Elm St',
            'City': 'Springfield'
        },
        'Phone Numbers': ['555-1234', '555-5678']
    }";
```

```
        // Deserialize JSON to Person object
        Person person = JsonConvert.DeserializeObject<Person>(json);
        Console.WriteLine($"Name: {person.Name}");
        Console.WriteLine($"Age: {person.Age}");
        Console.WriteLine($"Street: {person.Address.Street}");
        Console.WriteLine($"City: {person.Address.City}");
        Console.WriteLine($"Phone Numbers: {string.Join(", ", person.PhoneNumbers)}");

        // Serialize Person object to JSON
        string serializedJson = JsonConvert.SerializeObject(person, Formatting.Indented);
        Console.WriteLine("Serialized JSON:");
        Console.WriteLine(serializedJson);
    }
}
```

Explanation:

- Classes Definition: Address and Person classes represent the structure of the JSON data.

- Deserialization: Converts JSON string to a Person object using JsonConvert.DeserializeObject<Person>().

- Serialization: Converts the Person object back to a JSON string using JsonConvert.SerializeObject() with indented formatting for readability.

- Handling Nested Structures: The Person class contains an Address object and an array of phone numbers, demonstrating how to manage nested JSON.

Benefits of Using NuGet Packages:

- Simplifies Code: Packages like Newtonsoft.Json abstract complex JSON handling, allowing you to focus on your application's logic.

- Ensures Consistency: Standard packages ensure consistent behavior and reliability, reducing the likelihood of bugs in JSON processing.

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.

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.

- Description: A List<T> is a dynamic array that provides fast access to elements via an index. It allows for random access and modification.

- Use Cases:

  - General Collection: When you need a flexible array-like structure that allows for adding, removing, and accessing elements by index.

  - Example Scenario: Managing a list of employees where you need to frequently access specific employees by index.

2. Queue<T>:

- Description: A Queue<T> is a first-in, first-out (FIFO) data structure. Elements are added to the end and removed from the front.

- Use Cases:

  - Task Scheduling: When tasks or items need to be processed in the order they were added.

  - Example Scenario: Implementing a print job queue where print jobs are processed in the order they are received.

3. Stack<T>:

- Description: A Stack<T> is a last-in, first-out (LIFO) data structure. Elements are added to and removed from the top of the stack.

- Use Cases:

- Undo Operations: When you need to revert to previous states or operations.

- Example Scenario: Managing browser history where you can navigate back to the last visited page.

Example Program Using Queue<T> with Priority Handling

Here's a C# program that demonstrates the use of a Queue<T> to manage a line of people in a bank, with a twist of prioritizing VIP customers:

csharp

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

class Program
{
    public class Customer
    {
        public string Name { get; set; }
        public bool IsVIP { get; set; }

        public override string ToString()
        {
            return $"{Name} (VIP: {IsVIP})";
        }
    }
```

```
    static void Main()
    {
        Queue<Customer> regularQueue = new Queue<Customer>();
        Queue<Customer> vipQueue = new Queue<Customer>();

        // Add customers to the queues
        AddCustomer(vipQueue, regularQueue, new Customer { Name = "Alice", IsVIP = true });
        AddCustomer(vipQueue, regularQueue, new Customer { Name = "Bob", IsVIP = false });
        AddCustomer(vipQueue, regularQueue, new Customer { Name = "Charlie", IsVIP =
true });
        AddCustomer(vipQueue, regularQueue, new Customer { Name = "Diana", IsVIP = false });

        Console.WriteLine("Processing customers...");
        ProcessCustomers(vipQueue, regularQueue);
    }

    static void AddCustomer(Queue<Customer> vipQueue, Queue<Customer> regularQueue,
Customer customer)
    {
        if (customer.IsVIP)
        {
            vipQueue.Enqueue(customer);
        }
        else
        {
            regularQueue.Enqueue(customer);
```

```
        }
    }
    static void ProcessCustomers(Queue<Customer> vipQueue, Queue<Customer> regularQueue)
    {
        // Process VIP customers first
        while (vipQueue.Count > 0)
        {
            Customer customer = vipQueue.Dequeue();
            Console.WriteLine($"Serving VIP customer: {customer}");
        }
        // Then process regular customers
        while (regularQueue.Count > 0)
        {
            Customer customer = regularQueue.Dequeue();
            Console.WriteLine($"Serving regular customer: {customer}");
        }
    }
}
```

Explanation:

- Customer Class: Represents customers with a Name and a IsVIP flag.

- Queues Initialization: Two queues are used—vipQueue for VIP customers and regularQueue for regular customers.

- adding Customers: The AddCustomer method places VIP customers in the vipQueue and regular customers in the regularQueue.

- Processing Customers: The ProcessCustomers method serves all VIP customers first, followed by regular customers.

Use Cases:

- Queue<T>: Ideal for scenarios where items need to be processed in the order they arrive, such as managing a customer service line.

- Priority Handling: By using separate queues for VIP and regular customers, the program ensures that VIPs are prioritized, demonstrating a practical application of the FIFO principle with added complexity.

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

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.

Inheritance in C# allows a class (the derived class) to inherit members (fields, properties, methods) from another class (the base class). This enables code reuse and establishes a natural hierarchy between classes.

Key Concepts:

- Base Class: The class that provides common functionality to derived classes.
- Derived Class: The class that inherits from the base class and can extend or override its functionality.
- Access Modifiers: Control the visibility of members in a class. Common modifiers include:
  - public: Accessible from any code.
  - protected: Accessible within the same class and derived classes.
  - private: Accessible only within the same class.
  - internal: Accessible within the same assembly.
  - protected internal: Accessible within the same assembly or from derived classes.

Implementing Inheritance:

1. Define a Base Class: Create a base class with members to be inherited.
2. Create Derived Classes: Use the : symbol to specify that a class is inheriting from another class.
3. Override Methods: Use the virtual keyword in the base class for methods that can be overridden and the override keyword in derived classes to provide specific implementations.

Here’s a C# example demonstrating inheritance and polymorphism with Animal, Dog, Cat, and Bird classes:

C#

```
using System;

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


public class Animal
{
    // Method to be overridden by derived classes
    public virtual void Speak()
```

```
    {
        Console.WriteLine("The animal makes a sound.");
    }
}

public class Dog : Animal
{
    // Override the Speak method to provide specific implementation
    public override void Speak()
    {
        Console.WriteLine("The dog barks.");
    }
}
public class Cat : Animal
{
    // Override the Speak method to provide specific implementation
    public override void Speak()
    {
        Console.WriteLine("The cat meows.");
    }
}
public class Bird : Animal
{
    // Override the Speak method to provide specific implementation
    public override void Speak()
```

```
    {
        Console.WriteLine("The bird chirps.");
    }
}
```

Explanation:

- Base Class (Animal): Defines a Speak method marked with virtual, indicating it can be overridden by derived classes.

- Derived Classes (Dog, Cat, Bird): Each class overrides the Speak method to provide specific behavior.

- Polymorphism: Demonstrated in the Main method where an array of Animal objects (holding Dog, Cat, and Bird) calls the Speak method. The actual method invoked depends on the runtime type of the object, illustrating polymorphism.

Access Modifiers in Context:

- Public: The Speak method in the base class is public to allow derived classes to override it and to enable access from outside the class.

- Protected: If you want to allow derived classes to access members but not external code, use the protected modifier

14. Explain polymorphism in C# and how it can be achieved. Provide examples using

base and derived classes.

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.

Polymorphism in C# is a concept that allows objects of different types to be treated as objects of a common base type. It enables a single interface to be used for a general class of actions, and the specific action is determined by the exact type of object that is invoked. Polymorphism can be achieved through:

1. Method Overriding: Allows a derived class to provide a specific implementation of a method that is already defined in its base class. This requires the base class method to be marked as virtual and the derived class method to be marked as override.

2. interfaces: Define a contract that derived classes must implement. Each derived class can provide its own implementation of the interface methods.

Example: Polymorphism with Vehicle, Car, and Bike

Here's a C# program demonstrating polymorphism using a base class Vehicle, derived classes Car and Bike, and an interface IDrive:

C#

```
using System;

// Define the IDrive interface
```

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

// Base class Vehicle
public abstract class Vehicle : IDrive
{
    // Abstract method to be implemented by derived classes
    public abstract void Drive();

    // Method to be overridden in derived classes
    public virtual void Start()
    {
        Console.WriteLine("Vehicle is starting.");
    }
}

// Derived class Car
public class Car : Vehicle
{
    // Implement the Drive method from IDrive interface
    public override void Drive()
    {
        Console.WriteLine("The car is driving.");
```

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.

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.

Abstraction in C# is a fundamental concept in object-oriented programming that focuses on hiding the implementation details and showing only the necessary features of an object. It helps in simplifying complex systems by allowing you to interact with objects at a high level of understanding.

Abstraction can be achieved using:

- Abstract Classes: These are classes that cannot be instantiated on their own and are meant to be inherited by other classes. They can include abstract methods (without implementations) and concrete methods (with implementations).

- Interfaces: Define a contract with method signatures that derived classes must implement. Interfaces do not provide any implementation themselves.

Benefits of Abstraction

- Simplified Interaction: Provides a simplified interface for interacting with complex systems.

- Code Reuse: Allows common functionality to be defined in base classes or interfaces, promoting code reuse.

- Maintainability: Easier to update or extend the system because changes are made in one place (the base class or interface).

Example: Abstraction with Shape, Circle, and Square

Here's a C# program demonstrating abstraction with an abstract class Shape and derived classes Circle and Square:

C#

```
using System;

// Abstract base class Shape
public abstract class Shape
{
    // Abstract method to be implemented by derived classes
    public abstract void Draw();

    // Property common to all shapes
    public string Color { get; set; }

    // Concrete method with implementation
    public void PrintInfo()
    {
        Console.WriteLine($"Drawing a {Color} shape.");
    }
}
```

```
// Derived class Circle
public class Circle : Shape
{
    // Implement the Draw method
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle.");
    }

    // Additional property specific to Circle
    public double Radius { get; set; }

    // Additional method specific to Circle
    public void CalculateArea()
    {
        double area = Math.PI * Radius * Radius;
        Console.WriteLine($"Area of the circle: {area}");
    }
}

// Derived class Square
public class Square : Shape
{
    // Implement the Draw method
    public override void Draw()
```

```
    {
        Console.WriteLine("Drawing a square.");
    }

    // Additional property specific to Square
    public double SideLength { get; set; }

    // Additional method specific to Square
    public void CalculateArea()
    {
        double area = SideLength * SideLength;
        Console.WriteLine($"Area of the square: {area}");
    }
}

class Program
{
    static void Main()
    {
        Shape circle = new Circle
        {
            Color = "Red",
            Radius = 5
        };
        Shape square = new Square
```

```
    {
        Color = "Blue",
        SideLength = 4
    };

    // Use abstraction to interact with shapes
    DrawShape(circle);
    DrawShape(square);
}

static void DrawShape(Shape shape)
{
    shape.PrintInfo();
    shape.Draw();

    // Use type checking to access derived class methods
    if (shape is Circle circle)
    {
        circle.CalculateArea();
    }
    else if (shape is Square square)
    {
        square.CalculateArea();
    }
}
```

}

Explanation:

- Abstract Class Shape: Defines an abstract method Draw that must be implemented by derived classes. It also includes a concrete method PrintInfo and a property Color.

- Derived Class Circle: Implements the Draw method and adds specific properties (Radius) and methods (CalculateArea).

- Derived Class Square: Implements the Draw method and adds specific properties (SideLength) and methods (CalculateArea).

- Polymorphism: Demonstrated in the DrawShape method, which uses the abstract class Shape to interact with different types of shapes. It also shows how to use type checking (is keyword) to access specific methods from derived classes.

Scenarios Where Abstraction Simplifies Code

- Designing Frameworks: Provides a common interface for different implementations, making it easier to extend or modify functionality without affecting existing code.

- API Development: Hides implementation details and exposes only necessary methods to users, simplifying interaction with complex systems.

- Software Architecture: Helps in defining clear boundaries and responsibilities, leading to more maintainable and scalable code.

Abstraction helps in creating a more manageable and understandable codebase by focusing on essential features and hiding complexity.

}

```
// Override the Start method
public override void Start()
{
```

```
        Console.WriteLine("The car engine is starting.");
    }
}

// Derived class Bike
public class Bike : Vehicle
{
    // Implement the Drive method from IDrive interface
    public override void Drive()
    {
        Console.WriteLine("The bike is riding.");
    }

    // Override the Start method
    public override void Start()
    {
        Console.WriteLine("The bike is starting.");
    }
}

class Program
{
    static void Main()
    {
        // Create instances of Car and Bike
```

```
        Vehicle myCar = new Car();
        Vehicle myBike = new Bike();

        // Call Start and Drive methods
        StartVehicle(myCar);
        StartVehicle(myBike);

        // Demonstrate polymorphism
        Vehicle[] vehicles = new Vehicle[] { myCar, myBike };
        foreach (Vehicle vehicle in vehicles)
        {
            vehicle.Drive();
        }
    }

    static void StartVehicle(Vehicle vehicle)
    {
        vehicle.Start();
        vehicle.Drive();
    }
}
```

Explanation:

- Interface IDrive: Defines a contract with the Drive method.

- Base Class Vehicle: Implements the IDrive interface and defines an abstract method Drive and a virtual method Start.

- Derived Classes Car and Bike: Implement the Drive method and override the Start method from the Vehicle class.

- Polymorphism Demonstration:

  - StartVehicle Method: Accepts a Vehicle parameter and calls the Start and Drive methods. The actual method called depends on the runtime type of the object (Car or Bike).

  - Vehicle[] Array: Contains both Car and Bike objects. The Drive method demonstrates polymorphism by invoking the appropriate method based on the actual object type.

Key Points:

- Abstract Methods: Used in the base class to enforce derived classes to provide specific implementations.

- Virtual Methods: Allow derived classes to override base class implementations.

- Interfaces: Define a contract that multiple classes can implement differently, allowing for a common interface with varied behavior.

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])}
```

a Predict the output of the following code:

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

b Predict the output of the following code:

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

c Predict the output of the following code:

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

1. Array Output

csharp

```
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

Explanation: The loop iterates over each element in the array and prints it. The array.Length property gives the number of elements in the array (5), so the loop prints each element from 0 to 4.

2. String Comparison

csharp

```
string str1 = "Hello";

string str2 = "hello";

Console.WriteLine(str1.Equals(str2, StringComparison.OrdinalIgnoreCase));
```

Output:

True

Explanation: The Equals method with StringComparison.OrdinalIgnoreCase compares the two strings case-insensitively. Since "Hello" and "hello" are the same when case is ignored, the output is True.

3. Object Equality

C#

```
object obj1 = new object();

object obj2 = new object();

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

Output:

False

Explanation: The == operator checks for reference equality when used with objects. Since obj1 and obj2 are two different instances of object, they do not refer to the same memory location, so the result is False.

4. Integer Addition and Assignment

C#

```
int a = 5;

int b = 10;

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

Output:

15

Explanation: The += operator adds the value of b (10) to a (5) and assigns the result to a. Therefore, a becomes 15, and this value is printed.

17. Given a list of integers, write a C# method that returns the largest and smallest integers in the list. Add a twist by handling an empty list.

a. 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.

b. 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).

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

d. 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.

1. Largest and Smallest Integers in a List

csharp

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

class Program
{
    public static void FindLargestAndSmallest(List<int> numbers)
    {
        if (numbers.Count == 0)
```

```
        {
            Console.WriteLine("The list is empty.");
            return;
        }

        int largest = numbers[0];
        int smallest = numbers[0];

        foreach (int number in numbers)
        {
            if (number > largest)
                largest = number;
            if (number < smallest)
                smallest = number;
        }

        Console.WriteLine($"Largest: {largest}, Smallest: {smallest}");
    }

    static void Main()
    {
        List<int> numbers = new List<int> { 3, 5, 1, 9, -2, 4 }; // Example list
        FindLargestAndSmallest(numbers);
    }
}
```

2. Sum of Entered Integers Ignoring Duplicates

csharp

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

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

        Console.WriteLine("Enter integers (negative number to stop):");
        while (true)
        {
            int input = int.Parse(Console.ReadLine());
            if (input < 0)
                break;

            if (uniqueNumbers.Add(input))
                sum += input;
        }
```

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

3. Reverse a String

```
csharp
using System;

class Program
{
    static void Main()
    {
        Console.Write("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);
    }
}
```

4. Dictionary for Student Grades

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

class Program
{
    static void Main()
    {
        // Using string as key and double as value
        Dictionary<string, double> studentGrades = new Dictionary<string, double>
        {
            { "Alice", 92.5 },
            { "Bob", 85.0 },
            { "Charlie", 88.3 }
        };

        // Retrieve and display grades
        foreach (var student in studentGrades)
```

```
        {
            Console.WriteLine($"{student.Key}: {student.Value}");
        }

        // Add example with different key and value types
        Dictionary<int, string> studentNameById = new Dictionary<int, string>
        {
            { 101, "Alice" },
            { 102, "Bob" },
            { 103, "Charlie" }
        };

        // Retrieve and display names by ID
        foreach (var entry in studentNameById)
        {
            Console.WriteLine($"ID {entry.Key}: {entry.Value}");
        }
    }
}
```

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

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.

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

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.

Purpose:

- Contracts for Classes: Interfaces define a contract that classes must follow, specifying methods and properties that the implementing classes must provide.

- Separation of Concerns: They help separate the definition of methods from their implementation, allowing different classes to implement the same interface in various ways.

Benefits:

1. Loose Coupling: Interfaces help reduce dependencies between classes by decoupling the interface contract from the implementation. This makes it easier to change the implementation without affecting the code that uses the interface.

2. Code Reusability: By defining common behaviors through interfaces, you can reuse code across different classes without forcing them to inherit from a common base class.

3. Enhanced Flexibility: Classes can implement multiple interfaces, providing flexibility in how they are used and extended.

a. Interface Implementation and Polymorphism

Interface Definition and Implementation:

C#

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

// Define the IDrive interface
interface IDrive
{
    void Drive();
}

// Implement the IDrive interface in the Car class
class Car : IDrive
{
    public void Drive()
    {
        Console.WriteLine("Car is driving.");
    }
}

// Implement the IDrive interface in the Bike class
```

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

class Program
{
    static void Main()
    {
        // Create a list of IDrive objects
        List<IDrive> vehicles = new List<IDrive>
        {
            new Car(),
            new Bike()
        };

        // Demonstrate polymorphism by calling Drive() on each object
        foreach (IDrive vehicle in vehicles)
        {
            vehicle.Drive();
        }
    }
```

}

b. Role of Abstract Classes vs. Interfaces

Abstract Classes:

- Purpose: Provide a partial implementation of functionality and allow derived classes to build upon that base. They can contain both abstract methods (without implementation) and concrete methods (with implementation).

- When to Use:

  - When you want to share code among closely related classes.

  - When you have a base class with common functionality that derived classes can extend or override.

  - When you need to define some default behavior.

Interfaces:

- Purpose: Define a contract that classes must adhere to, without providing any implementation. They focus solely on what methods or properties a class should implement.

- When to Use:

  - When you need to define a common set of operations that can be implemented by unrelated classes.

  - When you want to enable a class to implement multiple sets of behaviors (multiple inheritance).

c. Abstract Class and Polymorphism

Abstract Class Definition and Implementation:

C#

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

// Define the abstract class Animal
abstract class Animal
{
    // Define an abstract method MakeSound
    public abstract void MakeSound();
}

// Derive the Dog class from Animal and implement MakeSound
class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("Dog barks.");
    }
}

// Derive the Cat class from Animal and implement MakeSound
class Cat : Animal
{
    public override void MakeSound()
```

```
    {
        Console.WriteLine("Cat meows.");
    }
}

class Program
{
    static void Main()
    {
        // Create a list of Animal objects
        List<Animal> animals = new List<Animal>
        {
            new Dog(),
            new Cat()
        };

        // Demonstrate polymorphism by calling MakeSound() on each object
        foreach (Animal animal in animals)
        {
            animal.MakeSound();
        }
    }
}
```

Key Points:

- Interfaces are used to define a contract without implementation and can be implemented by any class. They promote loose coupling and reusability.

- Abstract classes provide a partial implementation and are best used when you have a common base class with shared code. They support single inheritance and are suitable when derived classes need to share code or default behavior.

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 lowerlevel functions. Provide an example project where top-down might be the best

approach.

a. 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.

Top-Down Approach

Overview:

In a top-down approach, the project starts with defining the high-level structure and overall design of the application. The development proceeds by breaking down the major components or modules into smaller, more manageable parts until reaching the implementation of lower-level functions.

Benefits:

1. Clear Vision: Establishing a high-level architecture early on ensures that the project has a clear vision and direction. It helps in identifying the key components and how they interact with each other.

2. Improved Organization: By planning the overall structure, the project can be better organized, reducing the likelihood of conflicts and inconsistencies as the development progresses.

3. Early Problem Detection: Issues related to the integration of components can be identified and addressed early, leading to a more coherent final system.

4. Focused Development: Developers can focus on building and integrating high-level components first, which can help in identifying and addressing system-wide concerns before delving into lower-level details.

Example Project:

Enterprise Resource Planning (ERP) System

- Description: An ERP system manages various business processes such as finance, HR, and supply chain management. Due to its complexity, it's crucial to plan the system's overall architecture, module interactions, and data flow before implementing specific features.

- Top-Down Benefits: Planning the high-level structure allows for defining the interactions between modules (e.g., finance, HR, supply chain) and ensuring that each module integrates smoothly with others. It helps in maintaining consistency and addressing cross-functional requirements from the start.

Bottom-Up Approach

Overview:

In a bottom-up approach, the development begins with the implementation of small, independent functions or components. These are gradually integrated to form larger units and eventually the complete system.

Benefits:

1. Modular Development: Starting with small, independent functions allows for modular development, where each function can be developed and tested in isolation.

2. Flexibility: Changes and refinements can be made to individual components without affecting the entire system. This approach supports iterative development and continuous improvement.

3. Early Testing: Small functions can be tested individually, leading to more robust and reliable components. Early detection of bugs at the function level ensures a more stable integration process.

4. Incremental Integration: Gradual integration of functions into larger units helps in building the system incrementally, allowing for continuous feedback and adjustments.

Example Project:

Library Management System

- Description: A library management system involves various components such as book cataloging, user management, and transaction handling. Each component can be developed as a small, independent function (e.g., adding a book, managing user accounts).

- Bottom-Up Benefits: Starting with the development of core functions like book addition and user management allows for testing and refining these components before integrating them into the complete system. It provides flexibility to modify individual functions based on testing and feedback, leading to a more robust final system.

Summary:

- Top-Down Approach: Best suited for projects where a clear architectural plan is essential for ensuring proper integration and coherence, such as complex ERP systems.

- Bottom-Up Approach: Ideal for projects where developing and testing small, independent functions first allows for greater flexibility and modular development, such as library management systems.