SCT221-0809/2022
MISATI VALERIAN
BIT2323
APPLICATION PROGRAMMING

1a) 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. 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

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

      Console.Write("Enter 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.");
               return;
```

```
            }
            break;
        default:
            Console.WriteLine("Invalid operation.");
            return;
    }

    Console.WriteLine($"Result: {result}");
  }

  // Method to calculate the average of a list of integers
  public static double CalculateAverage(int[] numbers)
  {
    if (numbers.Length == 0)
    {
      return 0;
    }

    int sum = 0;
    foreach (int number in numbers)
    {
      sum += number;
    }

    return (double)sum / numbers.Length;
  }
}
```

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.

```
using System;
class MyClass
{
  public int Value;

  // Default constructor
  public MyClass()
  {
    Value = 0; // Default value
    Console.WriteLine("Default constructor called.");
  }

  // Overloaded constructor
  public MyClass(int initialValue)
  {
    Value = initialValue;
```

```
        Console.WriteLine("Overloaded constructor called.");
    }
}

class Program
{
    static void Main()
    {
        MyClass obj1 = new MyClass();      // Calls default constructor
        MyClass obj2 = new MyClass(100);   // Calls overloaded constructor
    }
}
```

c) Sam is creating an employee management system. Create a class Employee with a constructor that takes an employee's name and ID. Demonstrate how to create an instance of the class and include a secondary constructor that accepts optional parameters like department and salary.

```
using System;
class Employee
{
    public string Name { get; private set; }
    public int ID { get; private set; }
    public string Department { get; private set; }
    public double Salary { get; private set; }

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

    // Overloaded constructor with optional department and salary
    public Employee(string name, int id, string department = "Unknown", double salary = 0)
    {
        Name = name;
        ID = id;
        Department = department;
        Salary = salary;
    }

    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {Name}, ID: {ID}, Department: {Department}, Salary: {Salary}");
    }
}

class Program
```

```
{
   static void Main()
   {
      Employee emp1 = new Employee("Alice", 101);
      Employee emp2 = new Employee("Bob", 102, "HR", 50000);

      emp1.DisplayInfo();
      emp2.DisplayInfo();
   }
}
```

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?

== Operator:

The == operator is used to compare the **references** of two objects for equality by default.

Equals () Method:

Use == when you want a quick and easy comparison of strings, and when you are sure that the comparison will always involve strings or primitive types.

Use Equals () when you need to be explicit about value comparison, especially in cases where you might be comparing objects of different types or when overriding equality checks in custom classes.

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));
```

Line 1: True
Line 2: True
Line 3: 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.

Common Language Runtime (CLR)

The CLR is the execution engine for .NET applications. It provides a managed execution environment where your code runs. The key responsibilities of the CLR include:

Memory Management: The CLR automatically handles memory allocation and deallocation for your applications through garbage collection, preventing memory leaks.

Type Safety: The CLR ensures that code only accesses memory that it is authorized to access, enforcing type safety.

Exception Handling: The CLR provides a structured way to handle exceptions, making it easier to write robust and error-free code.

Security: The CLR offers various security models to ensure that code runs with appropriate permissions and that unauthorized operations are prevented.

Just-In-Time Compilation (JIT): The CLR compiles Intermediate Language (IL) code into native machine code just before execution, optimizing the performance of the application.

Base Class Library (BCL)

The BCL is a vast collection of reusable classes, interfaces, and value types that provide essential functionalities such as file I/O, data manipulation, collections, and more. The BCL simplifies development by providing pre-built, tested, and optimized code for common programming tasks, allowing developers to focus on writing business logic rather than reinventing the wheel.

How the CLR and BCL Work Together

Development: When you write a .NET application, you rely heavily on the BCL for common tasks like string manipulation, file handling, and collections. This allows for faster development and fewer errors.

Execution: When the application runs, the CLR manages the execution, handling memory management, type safety, and exception handling. The CLR relies on the BCL to provide the necessary functionalities during execution.

Smooth Development Experience: Together, the CLR and BCL create a smooth development experience by providing a robust environment that takes care of low-level details, allowing developers to write efficient, high-level code.

a. In a library management system, a function needs to handle file operations. Write a program that demonstrates the use of System.IO.File to create, read, and write to a file containing a list of books.

```
using System;
using System.IO;

class LibraryManagementSystem
{
   static void Main()
   {
      string filePath = "books.txt";
```

```
        // Create a file and write a list of books to it
        WriteBooksToFile(filePath);

        // Read the list of books from the file
        ReadBooksFromFile(filePath);
    }

    static void WriteBooksToFile(string filePath)
    {
        // List of books to write to the file
        string[] books = {
            "The Great Gatsby",
            "1984",
            "To Kill a Mockingbird",
            "Pride and Prejudice",
            "Moby-Dick"
        };

        // Write the array of books to the file
        File.WriteAllLines(filePath, books);

        Console.WriteLine("Books written to file successfully.");
    }

    static void ReadBooksFromFile(string filePath)
    {
        if (File.Exists(filePath))
        {
            // Read all lines from the file
            string[] books = File.ReadAllLines(filePath);

            Console.WriteLine("Books in the file:");
            foreach (string book in books)
            {
                Console.WriteLine(book);
            }
        }
        else
        {
            Console.WriteLine("File not found.");
        }
    }
}
```

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.

Value Types

Definition: Value types directly contain their data. When you assign a value type to a variable, the actual data is stored in the variable, not a reference to the data.

Memory Allocation: Value types are usually stored in the stack, which is a region of memory that is fast to allocate and deallocate.

Copying Behavior: When you assign one value type to another, a copy of the data is made. Changing the value in one variable does not affect the other.

Examples:

Primitive data types: int, float, char, bool

Structs: Custom value types defined using the struct keyword, like DateTime and Point.

Reference Types

Definition: Reference types store a reference (or address) to their data rather than the actual data itself. The reference points to an object stored in the heap.

Memory Allocation: Reference types are stored in the heap, which is a region of memory managed by the garbage collector. The heap allows for dynamic memory allocation but is slower compared to the stack.

Copying Behavior: When you assign one reference type to another, you copy the reference, not the data. Both variables will point to the same object, so changes made through one reference will affect the other.

Examples:

Classes: string, Array, List<T>

Objects: Instances of any class, including custom classes.

Scenarios Affecting Performance and Behavior

Performance:

Value types are typically faster to allocate and access because they are stored on the stack. However, large value types (like large structs) can be costly to copy, which might impact performance in scenarios involving frequent copying.

Reference types are better suited for large data objects because only the reference is passed around rather than the entire object. However, accessing data in the heap is slightly slower, and the garbage collector needs to manage memory, which can introduce overhead.

Behavior:

If you need a type that behaves independently when copied, use a value type. For example, using `int` for arithmetic ensures that each variable holds its own value.

If you need multiple references to the same data, use a reference type. For example, sharing a large data structure like a `List<T>` among multiple parts of your application is more efficient as they all reference the same object.

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.**

```
using System;

class Program
{
   static void Main()
   {
      // Value type example with int
      int a = 10;
      int b = a;
      b = 20;

      Console.WriteLine($"Value type - int:");
      Console.WriteLine($"a: {a}, b: {b}");  // a remains 10, b changes to 20

      // Reference type example with string array
      string[] arr1 = { "apple", "banana", "cherry" };
      string[] arr2 = arr1;
      arr2[0] = "grape";

      Console.WriteLine($"\nReference type - string array:");
      Console.WriteLine($"arr1[0]: {arr1[0]}, arr2[0]: {arr2[0]}");  // Both point to "grape"

      // Comparing memory addresses using Object.ReferenceEquals
      Console.WriteLine($"\nMemory comparison:");
      Console.WriteLine($"Object.ReferenceEquals(arr1, arr2): {Object.ReferenceEquals(arr1, arr2)}");
// True, both reference the same array

      // Value type with structs
      Point p1 = new Point(1, 2);
      Point p2 = p1;
      p2.X = 10;

      Console.WriteLine($"\nValue type - struct:");
```

```
        Console.WriteLine($"p1: {p1}, p2: {p2}");  // p1 remains (1, 2), p2 changes to (10, 2)
    }
}

struct Point
{
    public int X { get; set; }
    public int Y { get; set; }

    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    public override string ToString()
    {
        return $"({X}, {Y})";
    }
}
```

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.

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, known as a class. Encapsulation helps in controlling the access to the data by restricting direct access to some of the class's components. This is typically achieved using **access modifiers** like `private`, `public`, `protected`, and `internal`.

How Encapsulation Works:

Private Fields: Fields in a class are often marked as `private`, meaning they cannot be accessed directly from outside the class. This prevents unauthorized or unintended modification of the data.

Public Properties/Methods: Instead of providing direct access to the fields, encapsulation uses **public properties** or **methods** to allow controlled access to the fields. Properties can include validation logic to ensure that the data remains in a valid state.

Data Protection: By controlling access to the internal data, encapsulation protects the integrity of the data and the internal state of the object, reducing the risk of bugs or unintended behavior.

Benefits of Encapsulation:

Control: You can control how the data is accessed or modified. For example, you can make sure that a field like `age` cannot be set to a negative value.

Flexibility: Encapsulation allows you to change the internal implementation without affecting the external code that uses the class. For instance, you can modify the logic within a property while keeping the property name the same.

Maintenance: By hiding the complexity and exposing only what is necessary, encapsulation makes it easier to understand and maintain the code.

a)Maria is creating a system for managing people's data. Create a Person class with private fields for name and age and public properties to encapsulate these fields. Add validation in the properties, such as ensuring age is nonnegative.

```
using System;

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

    // Public property for name
    public string Name
    {
        get { return name; }
        set
        {
            if (!string.IsNullOrWhiteSpace(value))
            {
                name = value;
            }
            else
            {
                throw new ArgumentException("Name cannot be null or empty.");
            }
        }
    }

    // Public property for age with validation
    public int Age
    {
        get { return age; }
        set
        {
            if (value >= 0)
            {
                age = value;
            }
            else
            {
```

```
            throw new ArgumentOutOfRangeException("Age cannot be negative.");
        }
    }
  }

  // Constructor
  public Person(string name, int age)
  {
    Name = name;  // Uses the property to set the name, ensuring validation is applied
    Age = age;    // Uses the property to set the age, ensuring validation is applied
  }

  // Method to display person details
  public void DisplayInfo()
  {
    Console.WriteLine($"Name: {Name}, Age: {Age}");
  }
}

class Program
{
  static void Main()
  {
    try
    {
      // Creating a new Person object
      Person person = new Person("Maria", 30);
      person.DisplayInfo();

      // Attempting to set invalid age
      person.Age = -5;  // This will throw an exception
    }
    catch (Exception ex)
    {
      Console.WriteLine($"Error: {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.

Difference Between Single-Dimensional and Jagged Arrays

Single-Dimensional Array:

A single-dimensional array, also known as a "one-dimensional array," is a linear collection of elements of the same type. Each element in the array is accessed using a single index.

```
int[] numbers = { 1, 2, 3, 4, 5 };
```
Use Case: A single-dimensional array is ideal for storing a simple list of items, such as a list of temperatures recorded over a week or a collection of student grades in a course.

Jagged Array:

A jagged array is an array of arrays, where each "inner" array can have a different length. It's called a jagged array because the rows can be of varying lengths, creating a "jagged" structure.

```
int[][] jaggedArray = new int[][]
{
   new int[] { 1, 2, 3 },
   new int[] { 4, 5 },
   new int[] { 6, 7, 8, 9 }
};
```
Use Case: A jagged array is useful when you need to represent data with varying lengths, such as storing student test scores where each student has taken a different number of tests, or when representing a triangle where each row has a different number of elements.

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.**

```
using System;
class ArrayUtils
{
   public static int SumTwoDimensionalArray(int[,] array)
   {
      int sum = 0;

      for (int i = 0; i < array.GetLength(0); i++)
      {
         for (int j = 0; j < array.GetLength(1); j++)
         {
            // Check for any irregularities, like uninitialized or missing values
            sum += array[i, j];
         }
      }

      return sum;
   }

   static void Main()
   {
      int[,] array = {
         { 1, 2, 3 },
         { 4, 5, 6 },
         { 7, 8, 9 }
      };

      Console.WriteLine($"The sum of the array elements is: {SumTwoDimensionalArray(array)}");
   }
```

```
}
```

b) Emma is designing a color picker for an art application. Define an enum called Color with values Red, Green, and Blue. Also, define a class Shape with a nested class Circle that uses the enum to determine its color.

```
using System;
enum Color
{
   Red,
   Green,
   Blue
}

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

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

      public void DisplayColor()
      {
         Console.WriteLine($"The color of the circle is {CircleColor}");
      }
   }
}

class Program
{
   static void Main()
   {
      // Creating a Circle object with the color Red
      Shape.Circle circle = new Shape.Circle(Color.Red);
      circle.DisplayColor();

      // Creating another Circle object with the color Blue
      Shape.Circle anotherCircle = new Shape.Circle(Color.Blue);
      anotherCircle.DisplayColor();
   }
}
```

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.

Exception Handling in C#

Exception handling in C# is a mechanism that allows you to manage runtime errors in a controlled way, ensuring that your program can handle unexpected situations without crashing. The primary keywords used for exception handling in C# are `try`, `catch`, and `finally`.

How Exception Handling Works:

try Block:

The `try` block contains the code that might throw an exception. If an exception occurs within the `try` block, the control is passed to the appropriate `catch` block.

catch Block:

The `catch` block is used to handle the exception. You can have multiple `catch` blocks to handle different types of exceptions. Each `catch` block can handle a specific exception type (e.g., `IndexOutOfRangeException`, `NullReferenceException`).

If an exception is thrown, the `catch` block that matches the exception type is executed.

finally Block:

The `finally` block contains code that is always executed after the `try` and `catch` blocks, regardless of whether an exception was thrown or not. This is useful for cleaning up resources, such as closing files or releasing database connections.

Best Practices and Potential Pitfalls:

Catch Specific Exceptions: Always catch specific exceptions rather than using a general `catch` block, which could hide bugs or make debugging difficult.

Avoid Swallowing Exceptions: Don't just catch exceptions without handling them appropriately. If you catch an exception, make sure to either handle it or rethrow it.

Use finally for Cleanup: The `finally` block is ideal for resource cleanup, ensuring that resources are released even if an exception occurs.

Don't Overuse Exceptions: Exceptions should be used for exceptional cases, not for regular control flow. Overusing exceptions can lead to performance issues.

Rethrowing Exceptions: If you catch an exception and need to rethrow it, use `throw;` without specifying the exception to preserve the original stack trace.

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.

```
using System;

class ListManagement
{
   static void Main()
   {
      int[] numbers = { 1, 2, 3, 4, 5 };

      try
      {
         Console.WriteLine("Enter an index to access the array element:");
         int index = int.Parse(Console.ReadLine());

         try
         {
            // Accessing the element at the given index
            Console.WriteLine($"Element at index {index}: {numbers[index]}");
         }
         catch (IndexOutOfRangeException ex)
         {
            Console.WriteLine("Error: The index you entered is out of bounds.");
            Console.WriteLine($"Exception Message: {ex.Message}");
         }
      }
      catch (FormatException ex)
      {
         Console.WriteLine("Error: Please enter a valid integer.");
         Console.WriteLine($"Exception Message: {ex.Message}");
      }
      finally
      {
         Console.WriteLine("End of program execution.");
      }
   }
}
```

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.

```
using System;
class NumberAnalysis
{
   static void Main()
   {
```

```
        Console.WriteLine("Enter an integer:");
        int number = int.Parse(Console.ReadLine());

        if (number > 0)
        {
            Console.WriteLine($"{number} is positive.");
        }
        else if (number < 0)
        {
            Console.WriteLine($"{number} is negative.");
        }
        else
        {
            Console.WriteLine("The number is zero.");
        }

        if (number % 2 == 0)
        {
            Console.WriteLine($"{number} is even.");
        }
        else
        {
            Console.WriteLine($"{number} is odd.");
        }
    }
}
```

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.

while Loop:

Description: The while loop continues to execute as long as its condition remains true. The condition is checked before the loop's body is executed.

```
int i = 0;
while (i < 5)
{
 Console.WriteLine(i);
   i++;
}
```

Use Case: Use a while loop when the number of iterations is not known in advance and depends on a condition being met (e.g., reading input until the user provides a specific value).

do-while Loop:

Description: The do-while loop is similar to the while loop, but it guarantees that the loop's body is executed at least once because the condition is checked after the body is executed.

```
int i = 0;
do
{
   Console.WriteLine(i);
   i++;
} while (i < 5);
```
Use Case: Use a do-while loop when you need to execute the loop body at least once before checking the condition (e.g., a menu-driven program where the menu must be shown at least once).

for Loop:

Description: The for loop is a control flow statement that allows code to be executed repeatedly based on a boolean condition. The for loop is often used when the number of iterations is known in advance.

```
for (int i = 0; i < 5; i++)
{
   Console.WriteLine(i);
}
```
Use Case: Use a for loop when you know the exact number of iterations ahead of time (e.g., iterating through a list of elements).

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

```
using System;
class FactorialCalculator
{
   static void Main()
   {
      Console.WriteLine("Enter an integer to calculate its factorial:");
      int number = int.Parse(Console.ReadLine());

      long factorial = 1;

      for (int i = 1; i <= number; i++)
      {
         factorial *= i;
      }

      Console.WriteLine($"The factorial of {number} is {factorial}.");

      // Calculate factorial only if the number is odd
      if (number % 2 != 0)
      {
         long oddFactorial = 1;
         for (int i = 1; i <= number; i += 2)
         {
            oddFactorial *= i;
         }
```

```
            Console.WriteLine($"The factorial of odd numbers up to {number} is {oddFactorial}.");
        }
    }
}
```

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.

```
using System;
class AsteriskPatterns
{
    static void Main()
    {
        int rows = 5;

        // Right-angled triangle
        Console.WriteLine("Right-angled triangle:");
        for (int i = 1; i <= rows; i++)
        {
            for (int j = 1; j <= i; j++)
            {
                Console.Write("*");
            }
            Console.WriteLine();
        }

        // Inverted right-angled triangle
        Console.WriteLine("Inverted right-angled triangle:");
        for (int i = rows; i >= 1; i--)
        {
            for (int j = 1; j <= i; j++)
            {
                Console.Write("*");
            }
            Console.WriteLine();
        }
    }
}
```

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.

Threads in C#

In C#, threads are the fundamental units of execution within a process. They allow multiple operations to run concurrently within a program, making it possible to perform tasks like handling user inputs, processing data, and updating the UI simultaneously. Threads are

particularly useful for improving performance in applications that require parallel execution or need to stay responsive while performing long-running operations.

Thread Class vs. Task Class

Thread Class:

The Thread class represents a thread of execution. It is a lower-level, more explicit way to create and manage threads in C#.

You have direct control over the thread's lifecycle, such as starting, pausing, resuming, and stopping the thread.

Useful when you need fine-grained control over thread execution, such as in real-time systems or when working with legacy code that relies on explicit threading.

Task Class:

The Task class is part of the Task Parallel Library (TPL) and represents an asynchronous operation. It is higher-level compared to the Thread class and abstracts many of the complexities involved in thread management.

Task is more suitable for scenarios involving parallel processing, where you don't need to manage the thread lifecycle directly. It is often used with the async and await keywords for asynchronous programming.

It automatically handles thread pooling, making it more efficient for handling a large number of short-lived operations.

Example Use Cases

Thread Class Example:

Useful in scenarios where you need explicit control over threads, such as in real-time systems, custom threading models, or when interacting with low-level APIs.

Task Class Example:

Ideal for CPU-bound and I/O-bound operations, where you can take advantage of parallel processing without worrying about thread management. Commonly used in modern C# applications for tasks like file processing, web requests, and database operations.

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

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

```
class Program
{
   // Shared resource that needs thread safety
   private static int counter = 0;
   private static readonly object lockObject = new object();

   static void Main(string[] args)
   {
      // Create a new thread
      Thread newThread = new Thread(IncrementCounter);
      // Start the new thread
      newThread.Start();

      // Main thread also increments the counter
      IncrementCounter();

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

      Console.WriteLine("Final Counter Value: " + counter);
   }

   static void IncrementCounter()
   {
      for (int i = 0; i < 10; i++)
      {
         // Lock to ensure thread safety
         lock (lockObject)
         {
            counter++;
            Console.WriteLine($"Thread {Thread.CurrentThread.ManagedThreadId}: Counter =
{counter}");
         }
         // Simulate some work with a sleep
         Thread.Sleep(100);
      }
   }
}
```

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.

```
using System;
using System.Net.Http;
using System.Threading.Tasks;
using Newtonsoft.Json.Linq;
```

```
class Program
{
   static async Task Main(string[] args)
   {
      // Create an HttpClient instance
      using (HttpClient client = new HttpClient())
      {
         // Define the API endpoint (replace with a real API endpoint)
         string apiUrl = "https://newsapi.org/v2/top-headlines?country=us&apiKey=your_api_key";

         try
         {
            // Make a GET request
            HttpResponseMessage response = await client.GetAsync(apiUrl);

            // Ensure the response is successful
            response.EnsureSuccessStatusCode();

            // Read the response content as a string
            string responseBody = await response.Content.ReadAsStringAsync();

            // Parse the JSON data
            JObject jsonData = JObject.Parse(responseBody);

            // Extract and display article titles and summaries
            foreach (var article in jsonData["articles"])
            {
               Console.WriteLine("Title: " + article["title"]);
               Console.WriteLine("Summary: " + article["description"]);
               Console.WriteLine();
            }
         }
         catch (HttpRequestException e)
         {
            Console.WriteLine("Request error: " + e.Message);
         }
      }
   }
}
```

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.

```
using System;
using System.IO;

class Program
{
   static void Main(string[] args)
```

```
    {
        string inputFilePath = "input.txt";  // Replace with your input file path
        string outputFilePath = "output.txt"; // Replace with your output file path
        string keyword = "important";        // Keyword to filter lines
        int minLength = 10;                  // Minimum length of lines to include

        try
        {
            using (StreamReader reader = new StreamReader(inputFilePath))
            using (StreamWriter writer = new StreamWriter(outputFilePath))
            {
                string line;
                while ((line = reader.ReadLine()) != null)
                {
                    // Filter lines based on keyword and length
                    if (line.Contains(keyword) || line.Length >= minLength)
                    {
                        writer.WriteLine(line);
                    }
                }
            }

            Console.WriteLine("File processing completed successfully.");
        }
        catch (Exception e)
        {
            Console.WriteLine("An error occurred: " + e.Message);
        }
    }
}
```

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.

Purpose of Packages in C#

Packages in C# are collections of reusable code that can be shared and distributed across different projects. They often include libraries, tools, or frameworks that provide specific functionalities, making it easier to build and maintain applications.

Key Purposes of Packages:

Code Reusability: Packages allow developers to reuse code across multiple projects, reducing the need to rewrite common functionalities.

Modularity: By encapsulating related functionalities into packages, code becomes more modular, making it easier to manage and maintain.

Simplified Development: Packages can drastically reduce development time by providing pre-built solutions to common problems, such as data serialization, logging, or data access.

Consistency: Packages help ensure that the same version of a library is used across multiple projects, reducing the risk of compatibility issues and bugs.

Installing and Using a NuGet Package

NuGet is the package manager for .NET, and it provides a vast library of packages that can be easily integrated into C# projects. To install a NuGet package:

Open the NuGet Package Manager:

In Visual Studio, right-click on the project in the Solution Explorer and select "Manage NuGet Packages."

Alternatively, you can use the NuGet Package Manager Console.

Search for a Package:

Use the search bar to find the package you need (e.g., Newtonsoft.Json for JSON handling).

Install the Package:

Select the package and click "Install." This will download and add the package to your project.

Using the Package:

Once installed, you can use the package in your code by importing the relevant namespace (e.g., using Newtonsoft.Json;).

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.

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

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

   public class Person
   {
      public string Name { get; set; }
```

```
    public int Age { get; set; }
    public List<Address> Addresses { get; set; }
}

static void Main(string[] args)
{
    // Create a complex nested object
    var person = new Person
    {
        Name = "John Doe",
        Age = 30,
        Addresses = new List<Address>
        {
            new Address { Street = "123 Main St", City = "Springfield", State = "IL" },
            new Address { Street = "456 Elm St", City = "Metropolis", State = "NY" }
        }
    };

    // Serialize the object to JSON
    string jsonString = JsonConvert.SerializeObject(person, Formatting.Indented);
    Console.WriteLine("Serialized JSON:");
    Console.WriteLine(jsonString);

    // Deserialize the JSON back to an object
    var deserializedPerson = JsonConvert.DeserializeObject<Person>(jsonString);
    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}, {address.State}");
    }
}
}
```

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

In C#, List<T>, Queue<T>, and Stack<T> are data structures that are used to store and manage collections of elements. Each has different characteristics and is suited for different use cases.

List<T>

Definition: A List<T> is a dynamic array that allows random access to elements via an index. It can grow and shrink in size as elements are added or removed.

Characteristics:

Elements are stored sequentially.

Supports indexing, making it easy to access elements by position.

Allows insertion and removal of elements at any position.

Use Cases:

General Purpose Storage: Use when you need to store a collection of items that you want to access by index.

Search Operations: When you need to frequently search for elements within the collection.

Example: Storing a list of students in a class where you may want to access, insert, or remove students at specific positions.

Queue<T>

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

Characteristics:

Enforces order: the first element added is the first one removed.

Ideal for scenarios where you need to process items in the order they were added.

Use Cases:

Task Scheduling: Managing tasks in a system where tasks are processed in the order they arrive.

Event Handling: Handling events that occur in a sequence, such as print jobs in a printer queue.

Example: Managing a line of customers in a bank where customers are served in the order they arrive.

Stack<T>

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

Characteristics:

Enforces order: the last element added is the first one removed.

Used in scenarios where you need to reverse the order of items or keep track of history.

Use Cases:

Undo Mechanisms: Implementing undo functionality where the most recent action is undone first.

Depth-First Search: Useful in algorithms like depth-first search (DFS) where you need to explore elements in reverse order.

Example: Browser history where the most recent page visited is the first one to go back to.

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

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

class Program
{
   static void Main(string[] args)
   {
      Queue<string> regularQueue = new Queue<string>();
      Queue<string> vipQueue = new Queue<string>();

      // Enqueue regular customers
      regularQueue.Enqueue("Alice");
      regularQueue.Enqueue("Bob");
      regularQueue.Enqueue("Charlie");

      // Enqueue VIP customers
      vipQueue.Enqueue("David (VIP)");
      vipQueue.Enqueue("Eve (VIP)");

      Console.WriteLine("Serving customers in the bank:");

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

      // Then serve regular customers
      while (regularQueue.Count > 0)
      {
         Console.WriteLine(regularQueue.Dequeue() + " is being served.");
      }

      Console.WriteLine("All customers have been served.");
   }
}
```

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

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class to inherit properties and behaviors (methods) from another class. In C#, inheritance promotes code reuse and establishes a natural hierarchy between classes.

Key Concepts of Inheritance:

Base Class (Parent Class): The class whose properties and methods are inherited by another class. It is often referred to as the "superclass."

Derived Class (Child Class): The class that inherits from the base class. It can access and use the properties and methods of the base class and can also override them to provide specific implementations.

Access Modifiers: These control the visibility and accessibility of class members (fields, methods, properties) to other classes. In the context of inheritance:

public: Members are accessible from any other class.

protected: Members are accessible within the base class and derived classes but not outside.

private: Members are accessible only within the class itself, not by derived classes.

internal: Members are accessible within the same assembly but not outside of it.

protected internal: Members are accessible within the same assembly and also from derived classes.

When implementing inheritance in C#, you define a base class and then create derived classes that inherit from the base class. Derived classes can override base class methods to provide specific implementations.

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

```
using System;

class Animal
{
    // A protected field that can be accessed by derived classes
    protected string Name;

    // Constructor for the Animal class
    public Animal(string name)
    {
        Name = name;
    }

    // A virtual method that can be overridden in derived classes
    public virtual void Speak()
```

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

class Dog : Animal
{
    // Constructor for the Dog class
    public Dog(string name) : base(name) { }

    // Overriding the Speak method for Dog
    public override void Speak()
    {
        Console.WriteLine(Name + " says: Woof!");
    }
}

class Cat : Animal
{
    // Constructor for the Cat class
    public Cat(string name) : base(name) { }

    // Overriding the Speak method for Cat
    public override void Speak()
    {
        Console.WriteLine(Name + " says: Meow!");
    }
}

class Bird : Animal
{
    // Constructor for the Bird class
    public Bird(string name) : base(name) { }

    // Overriding the Speak method for Bird
    public override void Speak()
    {
        Console.WriteLine(Name + " says: Tweet!");
    }
}

class Program
{
    static void Main()
    {
        // Creating instances of derived classes
```

```
        Animal myDog = new Dog("Rex");
        Animal myCat = new Cat("Whiskers");
        Animal myBird = new Bird("Chirpy");

        // Demonstrating polymorphism
        myDog.Speak();  // Outputs: Rex says: Woof!
        myCat.Speak();  // Outputs: Whiskers says: Meow!
        myBird.Speak(); // Outputs: Chirpy says: Tweet!

        // Using a list to demonstrate polymorphism with different animals
        Animal[] animals = { myDog, myCat, myBird };

        Console.WriteLine("\nAll animals speak:");
        foreach (var animal in animals)
        {
            animal.Speak(); // Polymorphic behavior
        }
    }
}
```

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

Polymorphism in C#

Polymorphism is a core concept in object-oriented programming (OOP) that allows objects of different types to be treated as objects of a common super type. In C#, polymorphism enables the ability to invoke methods on an object without knowing the exact type of the object. This can be achieved through method overriding or interfaces, allowing for flexibility and extensibility in code.

Types of Polymorphism in C#

Compile-time Polymorphism (Method Overloading):

Achieved through method overloading and operator overloading.

The method to be invoked is determined at compile-time.

Run-time Polymorphism (Method Overriding):

Achieved through inheritance and interfaces.

The method to be invoked is determined at runtime based on the object's actual type.

How to Achieve Polymorphism in C#

Method Overriding:

A derived class can override a base class method using the override keyword.

The base class method must be marked as virtual, abstract, or override.

Interfaces:

Polymorphism can also be achieved through interfaces where different classes implement the same interface, but each provides a different implementation of the interface methods.

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

```
using System;

// Define an interface
interface IDrivable
{
    void Drive();
}

// Base class
class Vehicle : IDrivable
{
    public string Make { get; set; }

    public Vehicle(string make)
    {
        Make = make;
    }

    // Virtual method to be overridden
    public virtual void Drive()
    {
        Console.WriteLine($"Driving a vehicle of make: {Make}");
    }
}

// Derived class Car
class Car : Vehicle
{
    public int NumberOfDoors { get; set; }

    public Car(string make, int numberOfDoors) : base(make)
    {
        NumberOfDoors = numberOfDoors;
```

```
    }

    // Overriding the Drive method
    public override void Drive()
    {
        Console.WriteLine($"Driving a car of make: {Make} with {NumberOfDoors} doors.");
    }
}

// Derived class Bike
class Bike : Vehicle
{
    public bool HasSidecar { get; set; }

    public Bike(string make, bool hasSidecar) : base(make)
    {
        HasSidecar = hasSidecar;
    }

    // Overriding the Drive method
    public override void Drive()
    {
        string sidecarStatus = HasSidecar ? "with" : "without";
        Console.WriteLine($"Riding a bike of make: {Make} {sidecarStatus} a sidecar.");
    }
}

// Program demonstrating polymorphism
class Program
{
    static void Main()
    {
        // Create instances of derived classes
        Vehicle myCar = new Car("Toyota", 4);
        Vehicle myBike = new Bike("Harley-Davidson", false);

        // Demonstrating polymorphism
        myCar.Drive();  // Output: Driving a car of make: Toyota with 4 doors.
        myBike.Drive(); // Output: Riding a bike of make: Harley-Davidson without a sidecar.

        // Polymorphic behavior using a list of IDrivable objects
        IDrivable[] vehicles = { myCar, myBike };

        Console.WriteLine("\nAll vehicles driving:");
        foreach (var vehicle in vehicles)
        {
            vehicle.Drive(); // Polymorphic call to Drive method
```

```
    }
  }
}
```

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

Abstraction in C#

Abstraction is one of the four fundamental principles of object-oriented programming (OOP). It involves hiding the complex implementation details of a system and exposing only the necessary features. In C#, abstraction can be implemented using **abstract classes** and **interfaces**.

Abstract Classes: An abstract class cannot be instantiated directly and can contain abstract methods (without implementation) as well as concrete methods (with implementation). Abstract classes provide a common base for derived classes, ensuring that certain methods must be implemented by any subclass.

Interfaces: An interface is a contract that defines a set of methods and properties that a class must implement. Unlike abstract classes, interfaces cannot contain any implementation. They allow for a form of multiple inheritance, as a class can implement multiple interfaces.

Benefits of Abstraction

Simplified Code: By hiding complex logic, abstraction helps in writing code that is easier to understand and maintain.

Enhanced Maintainability: Changes to the implementation can be made without affecting the code that uses the abstracted functionality.

Flexibility and Extensibility: Abstract classes and interfaces allow for more flexible and extensible designs, making it easier to adapt to changes in requirements.

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

```
using System;

// Abstract base class
abstract class Shape
{
   public string Color { get; set; }

   public Shape(string color)
   {
      Color = color;
   }

   // Abstract method to be implemented by derived classes
```

```
    public abstract void Draw();

    // Concrete method that can be used by derived classes
    public void Describe()
    {
        Console.WriteLine($"This is a {Color} shape.");
    }
}

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

    public Circle(string color, double radius) : base(color)
    {
        Radius = radius;
    }

    // Implementing the abstract method
    public override void Draw()
    {
        Console.WriteLine($"Drawing a {Color} circle with a radius of {Radius}.");
    }

    // Additional method specific to Circle
    public double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}

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

    public Square(string color, double sideLength) : base(color)
    {
        SideLength = sideLength;
    }

    // Implementing the abstract method
    public override void Draw()
    {
        Console.WriteLine($"Drawing a {Color} square with a side length of {SideLength}.");
    }
```

```
    // Additional method specific to Square
    public double CalculateArea()
    {
        return SideLength * SideLength;
    }
}

// Program demonstrating abstraction
class Program
{
    static void Main()
    {
        Shape myCircle = new Circle("Red", 5.0);
        Shape mySquare = new Square("Blue", 4.0);

        // Using the abstract class methods
        myCircle.Describe(); // Output: This is a Red shape.
        myCircle.Draw();     // Output: Drawing a Red circle with a radius of 5.
        Console.WriteLine($"Area: {((Circle)myCircle).CalculateArea()}"); // Output: Area:
78.5398163397448

        Console.WriteLine();

        mySquare.Describe(); // Output: This is a Blue shape.
        mySquare.Draw();     // Output: Drawing a Blue square with a side length of 4.
        Console.WriteLine($"Area: {((Square)mySquare).CalculateArea()}"); // Output: Area: 16
    }
}
```

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
));
```

1
2
3
4
5

b. Predict the output of the following code:

object obj1 = new object(); object obj2 = new object();
Console.WriteLine(obj1 == obj2);
True
c.Predict the output of the following code:
int a = 5; int b = 10;
Console.WriteLine(a += b);
int a = 5;
int b = 10;
Console.WriteLine(a += b);
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.

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

class Program
{
   static (int? largest, int? smallest) FindLargestAndSmallest(List<int> numbers)
   {
      if (numbers == null || numbers.Count == 0)
      {
         return (null, null);
      }

      int largest = int.MinValue;
      int smallest = int.MaxValue;

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

      return (largest, smallest);
   }

   static void Main()
   {
      List<int> numbers = new List<int> { 10, 5, 3, 9, 2 };
      var result = FindLargestAndSmallest(numbers);
```

```
        if (result.largest.HasValue && result.smallest.HasValue)
        {
            Console.WriteLine($"Largest: {result.largest}, Smallest: {result.smallest}");
        }
        else
        {
            Console.WriteLine("The list is empty.");
        }
    }
}
```

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.

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

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

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

            if (number < 0)
            {
                break;
            }

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

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

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).

```
using System;
```

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

    static void Main()
    {
        DaysOfWeek today = DaysOfWeek.Saturday;

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

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

```
using System;

class Program
{
    static void Main()
    {
        Console.Write("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.Write a C# program that demonstrates how to use the Dictionary<TKey, TValue> class to store and retrieve student grades. Add complexity by handling a variety of data types as keys and values.

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

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

        // Adding student grades
        studentGrades.Add("Alice", 85.5);
        studentGrades.Add("Bob", 92.0);
        studentGrades.Add("Charlie", 78.0);

        // Retrieving and displaying student grades
        foreach (var student in studentGrades)
        {
            Console.WriteLine($"Student: {student.Key}, Grade: {student.Value}");
        }

        // Demonstrating variety in keys and values
        Dictionary<int, string> idToName = new Dictionary<int, string>();
        idToName.Add(1, "Alice");
        idToName.Add(2, "Bob");

        Console.WriteLine("\nStudent IDs and Names:");
        foreach (var entry in idToName)
        {
            Console.WriteLine($"ID: {entry.Key}, Name: {entry.Value}");
        }
    }
}
```

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

Definition: An interface in C# defines a contract that classes can implement. It specifies methods, properties, or events that the implementing class must provide.

Loose Coupling: Interfaces promote loose coupling by allowing code to interact with objects through their interfaces rather than concrete implementations. This decouples the code that uses the interface from the code that implements it, making it easier to modify or extend implementations without affecting dependent code.

Code Reusability: Interfaces enable code reusability by defining common operations that multiple classes can implement in their own way. This allows for the creation of generic algorithms and code that works with any class implementing the interface.

Flexibility: Interfaces support multiple inheritance, allowing a class to implement multiple interfaces, which is beneficial for creating versatile and modular code.

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

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

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

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

// Implement the interface in the Bike class
public class Bike : IDrive
{
    public void Drive()
    {
        Console.WriteLine("Riding a bike.");
    }
}

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

        foreach (var vehicle in vehicles)
        {
```

```
            vehicle.Drive(); // Demonstrates polymorphism
        }
    }
}
```

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.

Definition: An abstract class can provide some method implementations while leaving others as abstract (unimplemented) for derived classes to complete. Abstract classes can also include fields, constructors, and other methods.

When to Use: Abstract classes are appropriate when you need to provide a common base class with shared code or state, but also require derived classes to implement specific methods. Use abstract classes when there is a clear "is-a" relationship and you want to share code among related classes.

Differences from Interfaces:

Code Sharing: Abstract classes allow code sharing through non-abstract methods and fields, while interfaces only define method signatures without implementation.

Multiple Inheritance: C# does not support multiple inheritance of classes, but a class can implement multiple interfaces. This is useful when a class needs to conform to several different contracts.

When Abstract Classes Are More Appropriate:

When you need to provide a default implementation of methods.

When you have a common base class with shared functionality that should not be duplicated.

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

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

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

// Implement derived classes
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 (var animal in animals)
        {
            animal.MakeSound(); // Demonstrates polymorphism
        }
    }
}
```

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.

Top-Down Approach

Description:

Definition: The top-down approach involves starting with the highest-level design and breaking it down into smaller, more manageable modules or components. This approach focuses on defining the overall structure of the application first, before diving into the details of individual components.

Benefits:

Clear Structure: Provides a clear overall view of the system, ensuring that the project's goals and architecture are well-defined from the beginning.

Better Planning: Helps in identifying and addressing potential issues with the system's architecture early in the process.

Modular Design: Encourages modular design by defining high-level components and their interactions, which can simplify integration and maintenance.

Focused Development: Allows developers to understand the bigger picture and the relationships between different modules before working on the specifics.

Example Project:

Large Enterprise Resource Planning (ERP) System:

Reason: An ERP system integrates various business processes like finance, HR, supply chain, and sales. Using a top-down approach helps in designing a comprehensive architecture that aligns with business requirements and ensures that all modules integrate seamlessly. Planning the high-level structure first allows for defining clear interactions between modules, such as how the HR module interfaces with the payroll and finance modules.

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.

Bottom-Up Approach

Description:

Definition: The bottom-up approach involves starting with the development of small, independent functions or components and gradually combining them into larger, more complex units. This approach focuses on building and testing individual components before integrating them into a complete system.

Benefits:

Flexibility: Allows for incremental development and testing of individual components, making it easier to adjust or replace parts of the system as requirements evolve.

Ease of Testing: Small, independent functions or modules can be tested in isolation, which helps in identifying and fixing issues early.

Component Reusability: Promotes the creation of reusable components that can be combined in various ways to build the final system.

Gradual Integration: Facilitates a more gradual integration process, which can be less risky and easier to manage.

Example Project:

Library Management System:

Reason: In a library management system, functions like book check-in/check-out, user management, and inventory tracking can be developed as independent modules. Starting with these small, specific functionalities allows for detailed testing and refinement. Once these components are stable, they can be integrated to build the complete system. This approach is effective for projects where individual components need to be reliable and well-tested before integration.