

EDWIN KIMATHI
SCT221-0762/2021
BIT 2323
APPLICATION PROGRAMMING II
ASSIGNMENT

Question 1 (a) James calculator

```
using System;
using System.Linq; // Required for LINQ methods like Where
class Calculator
{
    static void Main(string[] args)
    {
        Console.WriteLine("Enter the first number:");
        double num1 = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Enter the second number:");
        double num2 = Convert.ToDouble(Console.ReadLine());
        Console.WriteLine("Choose an operation (+, -, *, /):");
        string operation = Console.ReadLine();
        double result = 0;
        switch (operation)
        {
            case "+":
                result = Add(num1, num2);
                break;
            case "-":
                result = Subtract(num1, num2);
                break;
            case "*":
                result = Multiply(num1, num2);
```

```

        break;
    case "/":
        result = Divide(num1, num2);
        break;
    default:
        Console.WriteLine("Invalid operation");
        return;
    }
    Console.WriteLine($"The result is: {result}");
    Console.WriteLine("Do you want to calculate the average of a list of integers? (yes/no)");
    string response = Console.ReadLine();
    if (response.ToLower() == "yes")
    {
        Console.WriteLine("Enter the integers separated by spaces:");
        string[] input = Console.ReadLine().Split(' ');
        // Filter out any empty strings and convert the remaining strings to integers
        int[] numbers = input
            .Where(s => !string.IsNullOrEmpty(s))
            .Select(int.Parse)
            .ToArray();

        double average = CalculateAverage(numbers);
        Console.WriteLine($"The average is: {average}");
    }
}

static double Add(double a, double b)
{
    return a + b;
}

static double Subtract(double a, double b)
{

```

```
        return a - b;
    }
    static double Multiply(double a, double b)
    {
        return a * b;
    }
    static double Divide(double a, double b)
    {
        if (b == 0)
        {
            Console.WriteLine("Error: Division by zero");
            return 0;
        }
        return a / b;
    }
    static double CalculateAverage(int[] numbers)
    {
        if (numbers.Length == 0)
        {
            return 0;
        }
        int sum = 0;
        foreach (int num in numbers)
        {
            sum += num;
        }

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

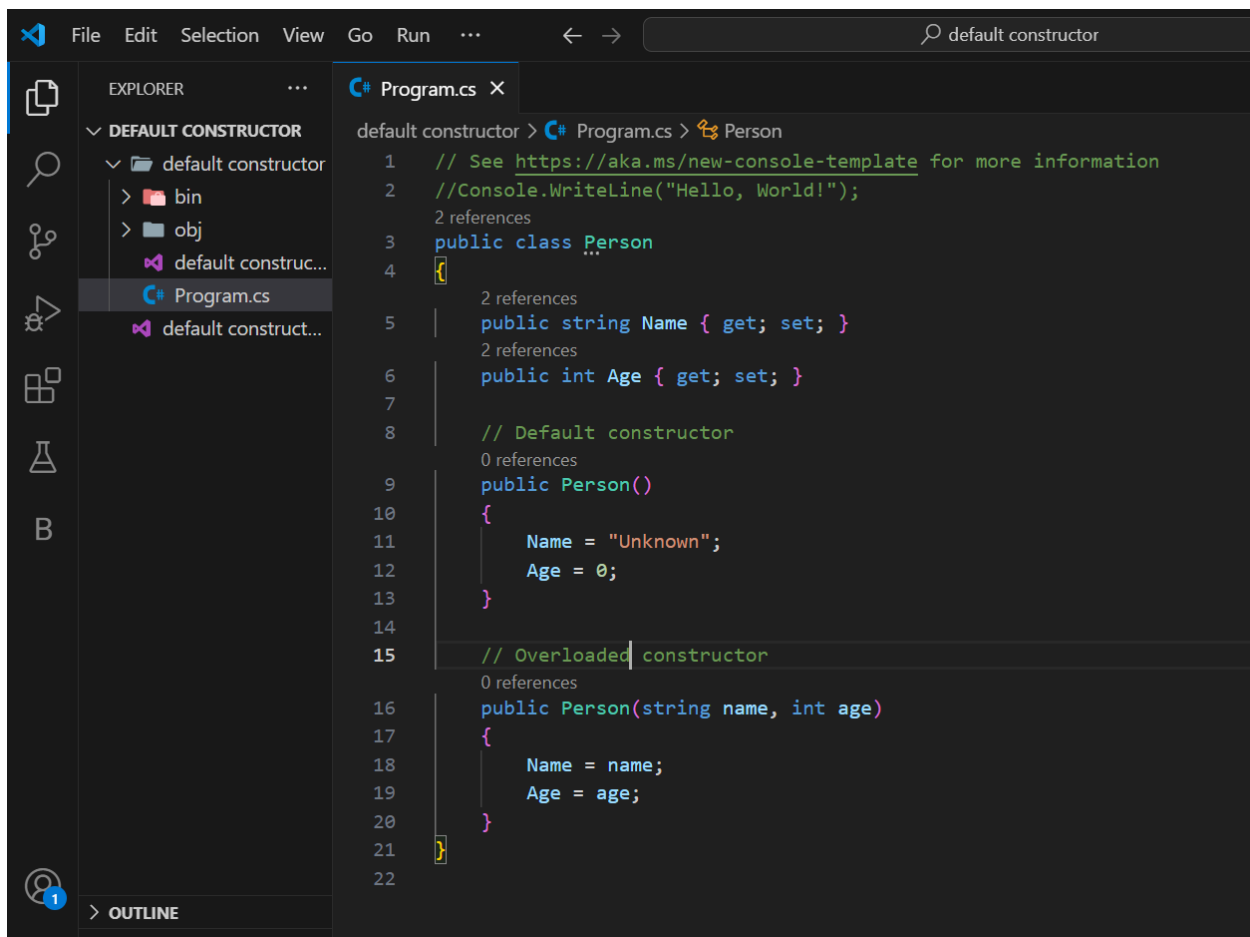
(b) Mary is developing a system to track object creation

Role of Constructors

A constructor is a special method within a class that is invoked automatically when an object of that class is created. Its primary function is to initialize the object's state, assigning initial values to its properties or fields. This ensures that objects are created in a consistent and valid state.

Key differences between constructors and other methods:

- **No return type:** Constructors do not have a return type, unlike other methods that can return values.
- **Same name as the class:** A constructor's name must match the class name.
- **Automatic invocation:** Constructors are called implicitly when an object is instantiated using the new keyword. Other methods must be called explicitly.



The screenshot shows the Visual Studio Code editor with a C# file named Program.cs. The Explorer sidebar on the left shows a project structure with a 'default constructor' folder containing 'bin', 'obj', and 'Program.cs'. The main editor area displays the following code:

```
1 // See https://aka.ms/new-console-template for more information
2 //Console.WriteLine("Hello, World!");
3 public class Person
4 {
5     public string Name { get; set; }
6     public int Age { get; set; }
7
8     // Default constructor
9     public Person()
10    {
11        Name = "Unknown";
12        Age = 0;
13    }
14
15    // Overloaded constructor
16    public Person(string name, int age)
17    {
18        Name = name;
19        Age = age;
20    }
21 }
```

(c) Sam is creating an employee management system

using System;

public class Employee

{

public string Name { get; set; }

public int ID { get; set; }

public string Department { get; set; }

public double Salary { get; set; }

// Primary constructor

public Employee(string name, int id)

{

 Name = name;

 ID = id;

}

// Secondary constructor

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

{

 Department = department;

 Salary = salary;

}

}

class Program

{

static void Main(string[] args)

{

 // Creating an instance with the primary constructor

 Employee employee1 = new Employee("John Doe", 12345);

 // Creating an instance with the secondary constructor

 Employee employee2 = new Employee("Jane Smith", 54321, "IT", 50000);

```

        Console.WriteLine($"Employee 1: Name: {employee1.Name}, ID: {employee1.ID}");

        Console.WriteLine($"Employee 2: Name: {employee2.Name}, ID: {employee2.ID},
Department: {employee2.Department}, Salary: {employee2.Salary}");
    }
}

```

2. Lucy is developing a program to compare string inputs from users.

☐ == operator:

- Compares references of objects, not the actual content.
- For value types (like int, double, bool), it compares the values.

☐ Equals() method:

- Compares the contents of objects.
- For value types, it behaves similarly to ==.

☐ Use ==: When you need a straightforward, case-sensitive comparison of string contents.

☐ Use Equals(): When you need more control over the comparison, such as case-insensitivity, or when comparing objects of other types

b) Predict the output

❖ **str1 == str2:** This will output True.

❖ Both str1 and str2 are assigned the same literal string "Hello". In C#, string literals are interned, meaning they reference the same object in memory for performance optimization. Thus, str1 and str2 point to the same string object, making the comparison true.

❖ **str1 == str3:** This will output False.

❖ While str1 and str3 have the same content, they are different objects in memory. The new string() syntax creates a new string object, even if the content is identical. The == operator compares references, not content in this case, so it returns false.

❖ **str1.Equals(str3):** This will output True.

❖ The Equals() method compares the actual content of the strings, regardless of whether they are different objects. Since both str1 and str3 contain the same characters, the Equals() method returns true.

Question 3: George wants to understand the main components of the .NET Framework for a development project. Explain the role of the Common Language Runtime (CLR) and the Base Class Library (BCL) in the .NET Framework

The Common Language Runtime (CLR)

The CLR is the execution engine of the .NET Framework. It manages the execution of .NET applications.

- **Manages code execution:** The CLR translates managed code (code written in a .NET language) into native code that the computer can understand. This process is called Just-In-Time (JIT) compilation.
- **Memory management:** The CLR handles memory allocation and garbage collection, freeing developers from manual memory management.

The Base Class Library (BCL)

The BCL is a collection of reusable types and functions that provide core functionality for .NET applications. It's like a toolbox that developers can use to build various applications.

- **Provides core functionality:** The BCL includes classes for input/output operations, string manipulation, data structures, networking, and much more.
- **Simplifies development:** By offering pre-built components, the BCL significantly reduces development time and effort.

How CLR and BCL Work Together

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

- **Execution environment:** The CLR provides the runtime environment for .NET applications, while the BCL offers the building blocks.
- **Language independence:** Developers can use any .NET language (C#, VB.NET, F#) to create applications that run on the CLR and utilize the BCL.

- 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.Collections.Generic;  
using System.IO;
```

```
public class Book  
{  
    public string Title { get; set; }  
    public string Author { get; set; }  
}
```

```

public class Library
{
    private const string filePath = "books.txt";

    public List<Book> ReadBooks()
    {
        List<Book> books = new List<Book>();
        if (!File.Exists(filePath))
        {
            return books; // Empty list if file doesn't exist
        }

        try
        {
            using (StreamReader reader = new StreamReader(filePath))
            {
                string line;
                while ((line = reader.ReadLine()) != null)
                {
                    string[] bookData = line.Split(',');
                    books.Add(new Book { Title = bookData[0], Author = bookData[1] });
                }
            }
        }
        catch (Exception ex)
        {
            Console.WriteLine("Error reading books: " + ex.Message);
        }

        return books;
    }

    public void WriteBooks(List<Book> books)
    {
        try
        {
            using (StreamWriter writer = new StreamWriter(filePath))
            {
                foreach (Book book in books)
                {
                    writer.WriteLine($"{book.Title},{book.Author}");
                }
            }
        }
        catch (Exception ex)
    }
}

```



```

        {
            Console.WriteLine("Error writing books: " + ex.Message);
        }
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        Library library = new Library();

        // Read existing books from file
        List<Book> books = library.ReadBooks();

        // Add new books
        books.Add(new Book { Title = "The Lord of the Rings", Author = "J.R.R. Tolkien"
    });
        books.Add(new Book { Title = "Pride and Prejudice", Author = "Jane Austen" });

        // Write updated list to file
        library.WriteBooks(books);

        Console.WriteLine("Books saved successfully!");
    }
}

```

Question 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

- **Directly hold data:** Value type variables store their data directly within their memory location.
- **Memory allocation:** Typically allocated on the stack, which is faster for memory access.
- **Examples:** int, double, bool, char, struct, enum.

```
int x = 10;
```

```
int y = x; // y is now a copy of x, so changes to y don't affect x
```

```
y = 20;
```

```
Console.WriteLine(x); // Output: 10
```

Reference Types

- **Hold references:** Reference type variables store a reference (memory address) to the actual data, which is located on the heap.
- **Memory allocation:** Allocated on the heap, which allows for dynamic memory allocation but is slower than stack access.
- **Examples:** string, class, array, interface.

```
class Person
```

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

```
Person p1 = new Person() { Name = "John" };
```

```
Person p2 = p1; // p2 is now a reference to the same object as p1
```

```
p2.Name = "Jane";
```

```
Console.WriteLine(p1.Name); // Output: Jane (because p1 and p2 reference the same object)
```

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

- ☐ If he's tracking simple data like employee IDs, salaries, or boolean flags, value types would be suitable for performance and efficiency.
- ☐ For complex data structures like employee information with multiple properties, reference types (classes) would be appropriate to encapsulate related data.

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;
```

```
namespace ValueAndReferenceTypes
```

```
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            // Value types  
            int num1 = 10;  
            int num2 = num1; // Creates a copy of num1
```

```

Console.WriteLine("Value types:");
Console.WriteLine($"num1: {num1}");
Console.WriteLine($"num2: {num2}");
num2 = 20;
Console.WriteLine($"num1 after changing num2: {num1}");
Console.WriteLine($"num2 after changing num2: {num2}");

// Reference types
string str1 = "Hello";
string str2 = str1; // Both refer to the same object
Console.WriteLine("\nReference types:");
Console.WriteLine($"str1: {str1}");
Console.WriteLine($"str2: {str2}");
str2 = "World";
Console.WriteLine($"str1 after changing str2: {str1}");
Console.WriteLine($"str2 after changing str2: {str2}");

// Comparing memory addresses
int[] intArray1 = { 1, 2, 3 };
int[] intArray2 = intArray1; // Both refer to the same array object
string[] stringArray1 = { "a", "b", "c" };
string[] stringArray2 = stringArray1; // Both refer to the same array object
Console.WriteLine("\nComparing memory addresses:");
Console.WriteLine($"intArray1 and intArray2: {Object.ReferenceEquals(intArray1,
intArray2)}");
    Console.WriteLine($"stringArray1 and stringArray2:
{Object.ReferenceEquals(stringArray1, stringArray2)}");
}
}
}

```

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 a fundamental principle of object-oriented programming (OOP) that involves bundling data (attributes or fields) and methods (functions) that operate on that data within a single unit, typically a class.

How encapsulation applies to classes and objects:

- **Classes:** A class serves as a blueprint for creating objects. It encapsulates data members (fields) and member functions (methods) that operate on those data members.
- **Objects:** An object is an instance of a class. When an object is created, it encapsulates its own data within its instance variables.

Controlling access to fields and methods:

C# provides access modifiers to control the visibility of class members:

- **public:** Accessible from anywhere.
 - **private:** Accessible only within the class.
 - **protected:** Accessible within the class and its derived classes.
 - **internal:** Accessible within the same assembly.
- 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 with validation
    public string Name
    {
        get { return name; }
        set
        {
            if (!string.IsNullOrEmpty(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 ArgumentException("Age must be non-negative.");
        }
    }
}

// Constructor to initialize the Person object
public Person(string name, int age)
{
    Name = name; // Using the property to set the value
    Age = age;   // Using the property to set the value
}

}

class Program
{
    static void Main()
    {
        try
        {
            // Creating a Person object with valid data
            Person person = new Person("Maria", 25);
            Console.WriteLine($"Name: {person.Name}, Age: {person.Age}");

            // Attempting to set invalid age
            person.Age = -5; // This will throw an exception
        }
        catch (ArgumentException 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.

A single-dimensional array is a collection of elements of the same data type stored in contiguous memory locations.

Use Case: Storing a list of student IDs, a collection of product prices, or an array of names.

A jagged array is an array of arrays, where each element is a single-dimensional array. This means that the length of each inner array can be different.

Use Case: Representing a matrix with varying row lengths, storing student grades for different subjects, or handling data with irregular structures.

Key Differences

- **Structure:** Single-dimensional arrays have a fixed length and all elements are of the same type. Jagged arrays are arrays of arrays, allowing for different lengths in each inner array.
- **Memory Allocation:** Single-dimensional arrays allocate contiguous memory for all elements. Jagged arrays allocate memory for each inner array independently.

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.**

```

public static int SumArrayElements(int[][] array)
{
    int sum = 0;

    for (int i = 0; i < array.Length; i++)
    {
        if (array[i] != null)
        {
            for (int j = 0; j < array[i].Length; j++)
            {
                sum += array[i][j];
            }
        }
    }

    return sum;
}

```

- 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;
public enum Color
{
    Red,
    Green,
    Blue
}
public class Shape
{
    public class Circle
    {
        public Color Color { get; set; }

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

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.

Understanding try, catch, and finally

C# provides a robust mechanism for handling exceptions using the try, catch, and finally blocks.

- **try:** Encloses code that might throw an exception.
- **catch:** Handles specific exceptions that occur within the try block. Multiple catch blocks can be used for different exception types.
- **finally:** Executes code regardless of whether an exception occurs or not, often used for cleanup operations like closing files or database connections.

Best Practices

- **Catch specific exceptions:** Instead of catching the generic `Exception`, catch specific exceptions to provide more targeted handling.
- **Use finally for cleanup:** Ensure resources are released, regardless of whether an exception occurs.
- **Rethrow exceptions:** If you cannot handle an exception, rethrow it to let outer blocks handle it.
- **Avoid empty catch blocks:** This can hide errors.

Potential Pitfalls

- **Overuse of try-catch:** Excessive use can hinder performance and readability.
- **Ignoring exceptions:** Swallowing exceptions without proper handling can lead to unexpected behavior.
- **Incorrect exception types:** Catching the wrong exception type can prevent correct error handling.

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 ListManagementApp
```

```
{
```

```
    static void Main()
```

```
    {
```

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

```
        try
```

```
        {
```

```
            // Outer try-catch block to handle general exceptions
```

```
            try
```

```
            {
```

```
                // Attempt to access an element outside the bounds of the array
```

```
                Console.WriteLine("Accessing element at index 10:");
```

```
                int outOfBoundsElement = numbers[10];
```

```
            }
```

```
        catch (IndexOutOfRangeException ex)
```

```
        {
```

```
            // Handle index out of range exception
```

```
            Console.WriteLine($"Error: {ex.Message}");
```

```
        }
```



```

// Inner try-catch block for another potential exception
try
{
    // Simulate a divide by zero scenario
    Console.WriteLine("Performing division:");
    int result = 10 / 0;
    Console.WriteLine($"Result: {result}");
}
catch (DivideByZeroException ex)
{
    // Handle divide by zero exception
    Console.WriteLine($"Error: {ex.Message}");
}
catch (Exception ex)
{
    // General exception handling (catching any other unexpected exceptions)
    Console.WriteLine($"Unexpected error: {ex.Message}");
}
finally
{
    // Code in finally block always executes, regardless of exceptions
    Console.WriteLine("Execution completed.");
}
}

```

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;

namespace NumberChecker
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Write("Enter a number: ");
            int number = int.Parse(Console.ReadLine());
            if (number == 0)
            {
                Console.WriteLine("The number is zero.");
            }
            else if (number > 0)
            {
                if (number % 2 == 0)
                {
                    Console.WriteLine("The number is even and positive.");
                }
                else
                {
                    Console.WriteLine("The number is odd and positive.");
                }
            }
            else
            {
                if (number % 2 == 0)
                {
                    Console.WriteLine("The number is even and negative.");
                }
                else
                {
                    Console.WriteLine("The number is odd and negative.");
                }
            }
        }
    }
}
```

```

        {
            Console.WriteLine("The number is even and negative.");
        }
        else
        {
            Console.WriteLine("The number is odd and negative.");
        }
    }
}
}
}

```

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

A while loop repeatedly executes a block of code as long as a specified condition is true. The condition is checked before each iteration.

```

int counter = 1;
while (counter <= 5)
{
    Console.WriteLine($"Counter: {counter}");
    counter++;
}

```

Use Case: When the number of iterations is unknown or depends on a dynamic condition.

Do-While Loop

A do-while loop is similar to a while loop, but the condition is checked after the loop body is executed at least once.

```

int count = 0;
do
{

```

```
    Console.WriteLine(count);  
    count++;  
} while (count < 5);
```

Use Case: When you need to execute a block of code at least once, regardless of the initial condition.

For Loop

A for loop is typically used when the number of iterations is known beforehand. It provides a more concise way to iterate over a sequence of values.

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

Use Case: When the number of iterations is predetermined or when iterating over collections or arrays.

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;  
  
namespace FactorialCalculator  
{  
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Console.Write("Enter a number: ");  
            int n = int.Parse(Console.ReadLine());  
            int[] factorials = CalculateFactorials(n);  
            Console.WriteLine("Factorials of odd numbers up to " + n + ":");  
            foreach (int factorial in factorials)  
            {  
                Console.WriteLine(factorial);  
            }  
        }  
    }  
}
```

```

    }
}
static int[] CalculateFactorials(int n)
{
    int[] factorials = new int[(n + 1) / 2]; // Pre-allocate array for efficiency
    int index = 0;
    for (int num = 1; num <= n; num += 2)
    {
        int factorial = 1;
        for (int i = 1; i <= num; i++)
        {
            factorial *= i;
        }
        factorials[index++] = factorial;
    }
    return factorials;
}
}
}

```

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 TrianglePatterns
{
    static void Main()
    {
        int n = 5; // Number of rows for the triangle

        // Right-angled triangle
    }
}

```

```

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

A thread represents an independent execution path within a process. It's a fundamental unit of concurrency in .NET.

```
using System;
```

```
using System.Threading;
```

```

class ThreadExample
{
    static void ThreadMethod()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine("Thread: " + i);
            Thread.Sleep(100);
        }
    }
    static void Main()
    {
        Thread thread = new Thread(ThreadMethod);
        thread.Start();

        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine("Main: " + i);
            Thread.Sleep(100);
        }
    }
}

```

Tasks

A task represents a unit of work that can be scheduled for execution. It's a higher-level abstraction than threads and often provides better performance and easier management.

```

using System;
using System.Threading.Tasks;

```

```

class TaskExample
{
    static async Task Main()
    {
        Task task = Task.Run(() =>
        {
            for (int i = 0; i < 5; i++)
            {
                Console.WriteLine("Task: " + i);
                Task.Delay(100).Wait();
            }
        });

        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine("Main: " + i);
            await Task.Delay(100);
        }

        await task;
    }
}

```

Key Differences

- **Thread management:** Threads require explicit management for creation, starting, stopping, and joining. Tasks are managed by the Task Parallel Library (TPL), providing a higher-level abstraction.
- **Resource utilization:** Threads are more resource-intensive than tasks, as each thread requires its own stack and kernel resources.

When to Use Which

- **Threads:** Use threads when you need fine-grained control over thread management or when interoperability with existing code requires thread-based APIs.
- **Tasks:** Use tasks for most asynchronous operations, as they provide a simpler and more efficient way to manage concurrent work.

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.

```
using System;
using System.Threading;
namespace ThreadExample
{
    class Program
    {
        static int sharedValue = 0;
        static object lockObject = new object();
        static void ThreadMethod()
        {
            for (int i = 0; i < 5; i++)
            {
                lock (lockObject)
                {
                    sharedValue++;
                    Console.WriteLine("Thread: Shared value = " + sharedValue);
                }
                Thread.Sleep(100);
            }
        }

        static void Main()
        {
            Thread t = new Thread(ThreadMethod);
            t.Start();
            t.Join();
            Console.WriteLine("Main: Shared value = " + sharedValue);
        }
    }
}
```

```

Thread thread = new Thread(ThreadMethod);
thread.Start();
for (int i = 0; i < 5; i++)
{
    lock (lockObject)
    {
        sharedValue--;
        Console.WriteLine("Main: Shared value = " + sharedValue);
    }
    Thread.Sleep(100);
}
thread.Join(); // Wait for the thread to finish
Console.WriteLine("Final shared value: " + sharedValue);
}
}
}

```

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.Text.Json;
using System.Threading.Tasks;
namespace NewsAggregator
{
    public class Article
    {
        public string Title { get; set; }
        public string Summary { get; set; }
    }
}

```

```
class Program
{
    static async Task Main()
    {
        string apiKey = "YOUR_API_KEY"; // Replace with your actual API key
        string apiUrl = "https://newsapi.org/v2/top-headlines?country=us&apiKey=" + apiKey;

        using (var client = new HttpClient())
        {
            try
            {
                var response = await client.GetAsync(apiUrl);
                response.EnsureSuccessStatusCode();

                var content = await response.Content.ReadAsStringAsync();
                var newsResponse = JsonSerializer.Deserialize<NewsResponse>(content);

                foreach (var article in newsResponse.Articles)
                {
                    Console.WriteLine($"Title: {article.Title}");
                    Console.WriteLine($"Summary: {article.Summary}");
                    Console.WriteLine();
                }
            }
            catch (Exception ex)
            {
                Console.WriteLine($"Error fetching news: {ex.Message}");
            }
        }
    }
}
```

```

    }
}
public class NewsResponse
{
    public List<Article> Articles { get; set; }
}
}

```

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;
using System.Linq;
namespace FileProcessing
{
    class Program
    {
        static void Main(string[] args)
        {
            string inputFile = "input.txt";
            string outputFile = "output.txt";
            string[] keywords = { "keyword1", "keyword2" };
            int minLength = 10;
            try
            {
                string[] lines = File.ReadAllLines(inputFile);
                using (StreamWriter writer = new StreamWriter(outputFile))
                {
                    foreach (string line in lines)
                    {
                        if (line.Length >= minLength && keywords.Any(k => line.Contains(k)))

```

```

        {
            writer.WriteLine(line);
        }
    }
}

Console.WriteLine("File processing completed successfully.");
}
catch (IOException 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 (or NuGet packages) are essentially libraries or assemblies of pre-compiled code that encapsulate functionality. They serve as a mechanism to share code between projects and developers.

- **Code Reusability:** Packages promote code reuse by providing pre-built components that can be incorporated into multiple projects.
- **Dependency Management:** They simplify managing dependencies between different parts of an application or between your application and external libraries.
- **Distribution:** Packages facilitate the distribution of code and updates to a wider audience.
- **Version Control:** NuGet supports versioning, allowing developers to manage different versions of packages and their dependencies.

Installing and Using a NuGet Package

NuGet is the package manager for the .NET ecosystem. To install a package:

1. **Open Visual Studio:** Create a new project or open an existing one.

2. **Access NuGet Package Manager:** This can be done through the Visual Studio UI (Tools -> NuGet Package Manager -> Manage NuGet Packages for Solution) or using the Package Manager Console.
3. **Search for a Package:** Use the search bar to find the desired package.
4. **Install Package:** Select the package and click Install. NuGet will download and install the package, adding necessary references to your project.

Simplifying Development and Ensuring Consistency

Packages significantly simplify development by:

- **Accelerating development:** Providing pre-built components saves time and effort.
- **Improving code quality:** By using well-tested and maintained packages, you can enhance the overall quality of your code.
- **Promoting consistency:** Using standardized packages helps maintain consistent coding practices across projects.

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.Text.Json;
```

```
namespace JsonSerialization
```

```
{
```

```
    public class Person
```

```
    {
```

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

```
        public int Age { get; set; }
```

```
        public Address Address { get; set; }
```

```
    }
```

```
    public class Address
```

```
    {
```

```
        public string Street { get; set; }
```

```
        public string City { get; set; }
```

```
    }
```

```

class Program
{
    static void Main(string[] args)
    {
        // Create a Person object with nested Address
        Person person = new Person
        {
            Name = "John Doe",
            Age = 30,
            Address = new Address
            {
                Street = "123 Main St",
                City = "Anytown"
            }
        };

        // Serialize to JSON
        string jsonString = JsonSerializer.Serialize(person);
        Console.WriteLine("Serialized JSON: " + jsonString);

        // Deserialize from JSON
        Person deserializedPerson = JsonSerializer.Deserialize<Person>(jsonString);

        Console.WriteLine("Deserialized Person: " + deserializedPerson.Name + ", " +
            deserializedPerson.Age + ", " + deserializedPerson.Address.Street + ", " +
            deserializedPerson.Address.City);
    }
}

```

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

- ☐ **List<T>** for dynamic arrays with random access,
- ☐ **Queue<T>** for FIFO operations
- ☐ **Stack<T>** for LIFO operations.

Comparison and Appropriate Scenarios:

1. **List<T>**:
 - **Best For:** When you need random access, frequent modifications, or want to store a collection of items where the order matters.
 - **Example:** A contact list in a phone app.
2. **Queue<T>**:
 - **Best For:** When you need to process items in the order they were added, like task scheduling or resource management.
 - **Example:** A job queue for a multi-threaded server.
3. **Stack<T>**:
 - **Best For:** When you need to backtrack, reverse a process, or manage nested operations, such as method call stacks or browser history.
 - **Example:** Implementing undo functionality in a text editor.

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;

public class Person

{

public string Name { get; set; }

public bool IsVIP { get; set; }

public Person(string name, bool isVIP)

{

 Name = name;

 IsVIP = isVIP;

}

}

public class BankQueue

{

private Queue<Person> normalQueue = new Queue<Person>();

private Queue<Person> vipQueue = new Queue<Person>();


```
public void Enqueue(Person person)
{
    if (person.IsVIP)
    {
        vipQueue.Enqueue(person);
    }
    else
    {
        normalQueue.Enqueue(person);
    }
}

public Person Dequeue()
{
    if (vipQueue.Count > 0)
    {
        return vipQueue.Dequeue();
    }
    else
    {
        return normalQueue.Dequeue();
    }
}

public bool IsEmpty()
{
    return normalQueue.Count == 0 && vipQueue.Count == 0;
}

}

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

```

{
    BankQueue bankQueue = new BankQueue();
    bankQueue.Enqueue(new Person("Alice", false));
    bankQueue.Enqueue(new Person("Bob", true));
    bankQueue.Enqueue(new Person("Charlie", false));
    while (!bankQueue.IsEmpty())
    {
        Person person = bankQueue.Dequeue();
        Console.WriteLine($"{person.Name} is being 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 you to create new classes (derived classes) based on existing classes (base classes). The derived class inherits the properties and methods of the base class and can add its own members or modify inherited members.

Access Modifiers in Inheritance

Access modifiers determine the visibility of members within a class and its derived classes.

- **public:** Accessible from anywhere.
- **protected:** Accessible within the class and its derived classes.
- **private:** Accessible only within the class.
- **internal:** Accessible within the same assembly.
- **protected internal:** Accessible within the same assembly or derived classes in other assemblies.

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;

```

public class Animal
{

```

```
public virtual void Speak()
{
    Console.WriteLine("Animal makes a sound");
}
}
public class Dog : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Woof!");
    }
}
public class Cat : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Meow!");
    }
}
public class Bird : Animal
{
    public override void Speak()
    {
        Console.WriteLine("Chirp!");
    }
}
class Program
{
    static void Main(string[] args)
    {
```

```

Animal[] animals = { new Dog(), new Cat(), new Bird() };

foreach (Animal animal in animals)
{
    animal.Speak();
}
}

```

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

□ **Polymorphism** in C# allows methods to be overridden in derived classes, enabling dynamic method invocation at runtime.

□ **Method Overriding** is the primary mechanism of achieving runtime polymorphism, where the derived class provides a specific implementation of a method defined in the base class.

```

public class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("Animal makes a sound");
    }
}

```

```

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

```

□ **Method Overloading** is another form of polymorphism but is resolved at compile-time, allowing multiple methods with the same name but different parameters to coexist in the same class.

```

public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public double Add(double a, double b)
    {
        return a + b;
    }
}

```

```
    }  
}
```

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

```
using System;
```

```
interface IDriveable
```

```
{  
    void Drive();  
}
```

```
class Vehicle
```

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

```
class Car : Vehicle, IDriveable
```

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

```
class Bike : Vehicle, IDriveable
```

```
{  
    public void Drive()  
    {  
        Console.WriteLine("Bike is moving");  
    }  
}
```

```

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

        foreach (IDriveable vehicle in vehicles)
        {
            vehicle.Drive();
        }
    }
}

```

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 is a fundamental concept in object-oriented programming that focuses on essential features while hiding implementation details. It simplifies complex systems by providing a higher-level view of functionality.

Abstract Classes

An abstract class is a base class that cannot be instantiated directly. It serves as a blueprint for derived classes. It can contain both abstract and non-abstract methods.

```

abstract class Shape
{
    public abstract double CalculateArea();
}

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

```

```

    public override double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }
}

```

Interfaces

An interface is a contract that defines a set of methods, properties, indexers, and events that a class must implement. It provides a blueprint for classes without providing implementation details.

```

interface IShape
{
    double CalculateArea();
}

class Rectangle : IShape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public double CalculateArea()
    {
        return Width * Height;
    }
}

```

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;

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

```

```
}
```

```
public class Circle : Shape
```

```
{
```

```
    public double Radius { get; set; }
```

```
    public Circle(double radius)
```

```
    {
```

```
        Radius = radius;
```

```
    }
```

```
    public override void Draw()
```

```
    {
```

```
        Console.WriteLine($"Drawing a circle with radius {Radius}");
```

```
    }
```

```
    public double CalculateArea()
```

```
    {
```

```
        return Math.PI * Radius * Radius;
```

```
    }
```

```
}
```

```
public class Square : Shape
```

```
{
```

```
    public double Side { get; set; }
```

```
    public Square(double side)
```

```
    {
```

```
        Side = side;
```

```
    }
```

```
    public override void Draw()
```

```
    {
```



```

        Console.WriteLine($"Drawing a square with side {Side}");
    }
    public double CalculateArea()
    {
        return Side * Side;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Shape[] shapes = { new Circle(5), new Square(4) };

        foreach (Shape shape in shapes)
        {
            shape.Draw();
            if (shape is Circle circle)
            {
                Console.WriteLine($"Circle area: {circle.CalculateArea()}");
            }
            else if (shape is Square square)
            {
                Console.WriteLine($"Square area: {square.CalculateArea()}");
            }
        }
    }
}

```

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]); }`

The output of the code:

1
2
3
4
5

a Predict the output of the following code: string str1 = "Hello"; string str2 = "hello"; Console.WriteLine(str1.Equals(str2, StringComparison.OrdinalIgnoreCase));

The output of the code is: True

b Predict the output of the following code: object obj1 = new object(); object obj2 = new object(); Console.WriteLine(obj1 == obj2);

The output of the code is: False

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

The output of the code is: 15

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.Collections.Generic;
```

```
using System.Linq;
```

```
public class Program
```

```
{
```

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

```
    {
```

```
        int[] numbers = { 5, 2, 8, 1, 9 };
```

```
        (int min, int max) = FindMinMax(numbers);
```

```
        Console.WriteLine("Smallest number: " + min);
```

```
        Console.WriteLine("Largest number: " + max);
```

```
    }
```

```
    public static (int min, int max) FindMinMax(int[] numbers)
```

```
    {
```

```

        if (numbers == null || numbers.Length == 0)
        {
            return (int.MaxValue, int.MinValue); // Handles empty list
        }
        int min = numbers[0];
        int max = numbers[0];
        for (int i = 1; i < numbers.Length; i++)
        {
            if (numbers[i] < min)
            {
                min = numbers[i];
            }
            else if (numbers[i] > max)
            {
                max = numbers[i];
            }
        }

        return (min, max);
    }
}

```

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;
namespace SumOfUniqueIntegers
{
    class Program
    {

```

```

static void Main(string[] args)
{
    List<int> numbers = new List<int>();
    int number;

    Console.WriteLine("Enter integers (negative number to stop):");
    do
    {
        number = int.Parse(Console.ReadLine());
        if (number >= 0)
        {
            if (!numbers.Contains(number))
            {
                numbers.Add(number);
            }
        }
    } while (number >= 0);
    int sum = numbers.Sum();
    Console.WriteLine("Sum of unique integers: " + 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;

enum DayOfWeek
{
    Sunday,
    Monday,
    Tuesday,

```

```

    Wednesday,
    Thursday,
    Friday,
    Saturday
}
class Program
{
    static void Main()
    {
        // Example days
        DayOfWeek today = DayOfWeek.Sunday;
        DayOfWeek tomorrow = DayOfWeek.Monday;

        // Print the day and whether it is a weekend or weekday
        Console.WriteLine($"Today is: {today}");
        Console.WriteLine($"Is today a weekend? {IsWeekend(today)}");

        Console.WriteLine($"Tomorrow is: {tomorrow}");
        Console.WriteLine($"Is tomorrow a weekend? {IsWeekend(tomorrow)}");
    }
    static bool IsWeekend(DayOfWeek day)
    {
        // Check if the day is Saturday or Sunday
        return day == DayOfWeek.Saturday || day == DayOfWeek.Sunday;
    }
}

```

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()
    {
        // Prompt the user for input
        Console.Write("Enter a string: ");
        string input = Console.ReadLine();

        // Reverse the string and print the result
        string reversed = ReverseString(input);
        Console.WriteLine($"Reversed string: {reversed}");
    }

    static string ReverseString(string str)
    {
        // Convert the string to a character array
        char[] charArray = str.ToCharArray();

        // Reverse the character array
        Array.Reverse(charArray);

        // Convert the reversed character array back to a string
        return new string(charArray);
    }
}
```

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()
    {
        // Create a dictionary to store student grades
        Dictionary<int, Student> studentGrades = new Dictionary<int, Student>();
        // Add students to the dictionary
        studentGrades.Add(101, new Student { Name = "Alice", Grade = 95.5 });
        studentGrades.Add(102, new Student { Name = "Bob", Grade = 89.0 });
        studentGrades.Add(103, new Student { Name = "Charlie", Grade = 76.3 });
        // Retrieve and print student information
        Console.WriteLine("Student Grades:");
        foreach (var kvp in studentGrades)
        {
            Console.WriteLine($"Student ID: {kvp.Key}, Name: {kvp.Value.Name}, Grade: {kvp.Value.Grade}");
        }
        // Demonstrate retrieving a specific student's information
        int studentIdToFind = 102;
        if (studentGrades.TryGetValue(studentIdToFind, out Student student))
        {
            Console.WriteLine($"Details for Student ID {studentIdToFind}:");
            Console.WriteLine($"Name: {student.Name}");
            Console.WriteLine($"Grade: {student.Grade}");
        }
        else
        {
            Console.WriteLine($"Student ID {studentIdToFind} not found.");
        }
    }
}

```

```

}
// Define a class to hold student details
class Student
{
    public string Name { get; set; }
    public double Grade { get; set; }
}

```

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

Purpose: An interface in C# defines a contract that a class must adhere to. It outlines a set of methods, properties, indexers, and events that a class must implement. Interfaces provide a way to achieve abstraction by focusing on what a class can do rather than how it does it.

Benefits:

- **Loose Coupling:** Interfaces promote loose coupling between classes, meaning changes to one class have minimal impact on others. This improves code maintainability and testability.
- **Polymorphism:** Interfaces enable polymorphism, allowing objects of different types to be treated as if they were of the same type. This leads to more flexible and adaptable code.
- **Code Reusability:** Interfaces can be used as a basis for creating reusable components.

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

```

using System;
using System.Collections.Generic;
interface IDrive
{
    void Drive();
}
class Car : IDrive
{
    public void Drive()
    {

```



```

        Console.WriteLine("Car is driving");
    }
}
class Bike : IDrive
{
    public void Drive()
    {
        Console.WriteLine("Bike is moving");
    }
}
class Program
{
    static void Main(string[] args)
    {
        List<IDrive> vehicles = new List<IDrive> { new Car(), new Bike() };

        foreach (IDrive vehicle in vehicles)
        {
            vehicle.Drive();
        }
    }
}

```

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

Abstract Classes

An abstract class is a base class that cannot be instantiated directly. It serves as a blueprint for derived classes.

It can contain both abstract and non-abstract methods, fields, and properties.

Interfaces

An interface is a contract that defines a set of methods, properties, indexers, and events that a class must implement. It provides a blueprint for classes without providing implementation details.

When to Use Which

- **Abstract class:** Use when there's a clear inheritance hierarchy and shared functionality among derived classes.
- **Interface:** Use when there's no inherent relationship between classes but they need to share a common behavior or when multiple inheritance is required.

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;
```

```
public abstract class Animal
```

```
{
```

```
    public abstract void MakeSound();
```

```
}
```

```
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(string[] args)
    {
        List<Animal> animals = new List<Animal> { new Dog(), new Cat() };
        foreach (Animal animal in animals)
        {
            animal.MakeSound();
        }
    }
}

```

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

Top-down design starts with breaking down a complex problem into smaller, more manageable sub-problems.

This approach focuses on defining the overall system structure and its components before delving into the specifics of implementation.

Benefits of Top-Down Approach

- **Clearer project vision:** By defining the high-level components first, the project's goals and objectives become clearer.
- **Improved organization:** Breaking down the system into modules promotes better organization and code structure.

Example: E-commerce Application

A large-scale e-commerce application can benefit significantly from a top-down approach. The initial focus would be on defining the core modules:

- **User management:** Handling user registration, authentication, and authorization.
- **Product catalog:** Managing product information, categories, and inventory.
-

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

In a bottom-up approach, development starts with the most basic components and gradually builds upon them to create more complex systems.

This approach is often used in software development when dealing with low-level functionalities or libraries.