**NAME: MUGO MIKE KARIUKI**

**REGISTRATION NUMBER: SCT221-0833/2021**

**UNIT: APPLICATION PROGRAMMING 2**

**UNIT CODE: BIT 2323**

**ASSIGNMENT**

**Question 1:**

using System;

class Calculator

{

static void Main()

{

Console.Write("Enter the first number: ");

double num1 = Convert.ToDouble(Console.ReadLine());

Console.Write("Enter the second number: ");

double num2 = Convert.ToDouble(Console.ReadLine());

Console.WriteLine("Choose an operation: +, -, \*, /");

char operation = Console.ReadKey().KeyChar;

Console.WriteLine();

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("Division by zero is not allowed.");

return;

}

break;

default:

Console.WriteLine("Invalid operation.");

return;

}

Console.WriteLine("The result is: " + result);

}

}

**Question 1a**

using System;

class AverageCalculator

{

public static double CalculateAverage(int[] scores)

{

if (scores.Length == 0)

{

return 0;

}

int sum = 0;

foreach (int score in scores)

{

sum += score;

}

return (double)sum / scores.Length;

}

static void Main()

{

int[] scores = { 90, 80, 100, 70 };

double average = CalculateAverage(scores);

Console.WriteLine("Average score: " + average);

}

}

**Question 1b**

using System;

class Product

{

public string Name { get; set; }

public double Price { get; set; }

// Default constructor

public Product()

{

Name = "Unknown";

Price = 0.0;

}

// Overloaded constructor

public Product(string name, double price)

{

Name = name;

Price = price;

}

public void Display()

{

Console.WriteLine($"Product Name: {Name}, Price: {Price}");

}

static void Main()

{

Product defaultProduct = new Product();

defaultProduct.Display();

Product customProduct = new Product("Laptop", 999.99);

customProduct.Display();

}

}

**Question 1c**

using System;

class Employee

{

public string Name { get; set; }

public int ID { get; set; }

public string Department { get; set; }

public double Salary { get; set; }

// Constructor

public Employee(string name, int id)

{

Name = name;

ID = id;

}

// Overloaded constructor

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

{

Department = department;

Salary = salary;

}

public void Display()

{

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

}

static void Main()

{

Employee emp1 = new Employee("Alice", 101);

emp1.Display();

Employee emp2 = new Employee("Bob", 102, "IT", 60000);

emp2.Display();

}

}

**Question 2:**

== operator: This operator is used to compare the values of the objects, but its behavior can vary depending on how it's overloaded. For value types (e.g., int, double), == compares the actual values. For reference types (e.g., string), == compares the memory references unless it is overridden to compare the actual contents (which is the case with string).

Equals() method: This method is used to compare the contents of objects. It’s a method that can be overridden in classes to provide specific equality logic. It is generally more reliable for content comparison.

**Question 2a:**

string str1 = "Hello";

string str2 = "Hello";

string str3 = new string(new char[] { 'H', 'e', 'l', 'l', 'o' });

Console.WriteLine(str1 == str2); // True - both refer to the same string in memory

Console.WriteLine(str1 == str3); // True - strings are compared by value

Console.WriteLine(str1.Equals(str3)); // True - Equals compares the contents

**Question 3:**

Common Language Runtime (CLR): The CLR is the execution engine for .NET programs. It handles memory management, thread management, exception handling, and security. It also provides a framework for the execution of .NET programs, allowing code to be executed across multiple languages.

Base Class Library (BCL): The BCL is a library of classes, interfaces, and value types that provide access to system functionality such as IO, threading, and collections. It simplifies development by providing a large set of pre-built functionalities.

**Question 3a**

using System;

using System.IO;

class FileOperations

{

static void Main()

{

string path = "books.txt";

// Writing to a file

using (StreamWriter writer = new StreamWriter(path))

{

writer.WriteLine("Book 1: C# in Depth");

writer.WriteLine("Book 2: Pro ASP.NET Core");

writer.WriteLine("Book 3: Clean Code");

}

// Reading from a file

using (StreamReader reader = new StreamReader(path))

{

string line;

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

{

Console.WriteLine(line);

}

}

}

}

**Question 4:**

Value Types: Value types are stored on the stack, and the variable directly holds the value. Examples include int, float, bool, and struct.

Reference Types: Reference types are stored on the heap, and the variable holds a reference (or address) to the actual object in memory. Examples include string, array, class, and interface.

**Question 4a**

using System;

class ValueVsReference

{

static void Main()

{

int valueType = 10;

int valueTypeCopy = valueType;

valueTypeCopy = 20;

Console.WriteLine("Original value type: " + valueType); // Output: 10

Console.WriteLine("Copied value type: " + valueTypeCopy); // Output: 20

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

int[] referenceTypeCopy = referenceType;

referenceTypeCopy[0] = 10;

Console.WriteLine("Original reference type: " + referenceType[0]); // Output: 10

Console.WriteLine("Copied reference type: " + referenceTypeCopy[0]); // Output: 10

}

}

**Question 5:**

Encapsulation: Encapsulation is the principle of bundling data (fields) and methods that operate on the data into a single unit, or class, and restricting access to some of the object's components. This is achieved using access modifiers like private, public, and protected.

**Question 5a**

using System;

class Person

{

// Private fields

private string name;

private int age;

// Public properties with validation

public string Name

{

get { return name; }

set { name = value; }

}

public int Age

{

get { return age; }

set

{

if (value < 0)

throw new ArgumentException("Age cannot be negative.");

age = value;

}

}

// Constructor

public Person(string name, int age)

{

Name = name;

Age = age;

}

public void Display()

{

Console.WriteLine($"Name: {Name}, Age: {Age}");

}

}

class Program

{

static void Main()

{

Person person = new Person("John Doe", 30);

person.Display();

// person.age = -5; // Error: Cannot access private field

person.Age = 25; // Valid access

person.Display();

}

}

**Question 6:**

**Question 6a**

using System;

class ArraySum

{

public static int SumTwoDimensionalArray(int[,] array)

{

int sum = 0;

foreach (int num in array)

{

sum += num;

}

return sum;

}

static void Main()

{

int[,] array = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

int sum = SumTwoDimensionalArray(array);

Console.WriteLine("Sum of elements: " + sum);

}

}

**Question 6b**

using System;

namespace ColorPickerApp

{

// Define an enum called Color with values Red, Green, and Blue

public enum Color

{

Red,

Green,

Blue

}

// Define a class called Shape

public class Shape

{

// Define a nested class called Circle

public class Circle

{

// Property to hold the color of the circle

public Color CircleColor { get; set; }

// Constructor to initialize the color of the circle

public Circle(Color color)

{

CircleColor = color;

}

// Method to display the color of the circle

public void DisplayColor()

{

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

}

}

}

class Program

{

static void Main(string[] args)

{

// Create a Circle object with the color Red

Shape.Circle myCircle = new Shape.Circle(Color.Red);

myCircle.DisplayColor();

// Create another Circle object with the color Blue

Shape.Circle anotherCircle = new Shape.Circle(Color.Blue);

anotherCircle.DisplayColor();

}

}

}

**Question 7**

try Block- The try block contains the code that may potentially throw an exception. If an exception occurs within this block, the control is transferred to the appropriate catch block.

catch Block- The catch block is used to handle the exception that occurs in the try block. Multiple catch blocks can be used to handle different types of exceptions, allowing for more specific exception handling.

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 caught. This block is typically used for cleanup activities, such as closing files, releasing resources, or resetting states.

### Best Practices

1. Avoid catching general exceptions like Exception. Instead, catch specific exceptions to handle each error appropriately. This helps in debugging and ensures that unexpected exceptions aren't swallowed silently.
2. The finally block should be used to release resources, such as closing files or database connections, ensuring they are properly cleaned up even if an exception occurs.
3. Never leave catch blocks empty. If you catch an exception, you should handle it or at least log it for future reference. Silent failures can lead to difficult-to-debug issues.
4. Logging exceptions is crucial for diagnosing problems. Ensure that caught exceptions are logged with relevant details like the exception message, stack trace, and any custom error messages.
5. If you catch an exception but can't handle it properly, consider rethrowing it to allow higher-level code to handle it. This can be done using throw;.
6. Only wrap code that might throw an exception in a try block. This makes it easier to identify the code that caused the exception and simplifies debugging.
7. Exceptions should be used for exceptional conditions, not for regular control flow. Using exceptions for normal operations can lead to performance issues and confusing code.

Potential Pitfalls

1. If exceptions are caught and not handled (e.g., caught and ignored or logged without further action), it can lead to subtle bugs and unpredictable behavior.

2. Catching generic exceptions like Exception or System. Exception can mask bugs and make debugging difficult. It's better to catch specific exceptions and handle them accordingly.

3. While finally is useful for cleanup, overusing it can lead to redundant code if not all operations require explicit cleanup.

**Question 7a**

using System;

class ExceptionHandling

{

static void Main()

{

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

try

{

// Attempt to access an index out of bounds

int number = numbers[3];

}

catch (IndexOutOfRangeException ex)

{

Console.WriteLine("Exception caught: " + ex.Message);

}

Console.WriteLine("Program continues...");

}

}

**Question 8**

using System;

class EvenOddCheck

{

static void Main()

{

Console.Write("Enter a number: ");

int number = Convert.ToInt32(Console.ReadLine());

if (number % 2 == 0)

{

Console.WriteLine("The number is even.");

}

else if (number % 2 != 0)

{

Console.WriteLine("The number is odd.");

}

if (number > 0)

{

Console.WriteLine("The number is positive.");

}

else if (number < 0)

{

Console.WriteLine("The number is negative.");

}

else

{

Console.WriteLine("The number is zero.");

}

}

}

**Question 8a:**

while Loop: Checks the condition before the loop body. Use when the number of iterations is not known.

do-while Loop: Checks the condition after the loop body. Use when the loop must execute at least once.

for Loop: Combines initialization, condition, and iteration in one statement. Use when the number of iterations is known.

**Question 8b**

using System;

class FactorialGenerator

{

static void Main()

{

Console.Write("Enter an odd number: ");

int number = int.Parse(Console.ReadLine());

if (number % 2 == 0)

{

Console.WriteLine("The number is not odd.");

return;

}

long factorial = 1;

for (int i = 1; i <= number; i++)

{

factorial \*= i;

}

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

}

}

**Question 8c:**

using System;

class TrianglePattern

{

static void Main()

{

Console.Write("Enter the number of rows: ");

int rows = int.Parse(Console.ReadLine());

for (int i = 1; i <= rows; i++)

{

for (int j = 1; j <= i; j++)

{

Console.Write("\*");

}

Console.WriteLine();

}

}

}

**Question 9:**

Threads allow concurrent execution in C# for better performance and responsiveness by running multiple operations simultaneously.

### **Thread Class vs. Task Class**

Thread Class - Represents a single thread of execution; provides low-level control.

Features - Manual management of thread lifecycle, synchronization, and priorities.

Use Case - Suitable for scenarios needing fine-grained thread control, such as custom thread pools.

Task Class - Represents an asynchronous operation; higher-level abstraction.

Features - Simplifies async programming with support for continuation, cancellation, and integration with async and await.

Use Case - Ideal for asynchronous and parallel operations, with simpler management

Key Differences:

1. Abstraction Level:

Thread: Low-level, manual control.

Task: High-level, simplified async programming.

1. Ease of Use:

Thread: Complex, requires manual synchronization.

Task: Simplified with async/await support.

1. Suitability:

Thread: For custom thread management.

Task: For easy asynchronous and parallel processing.

**Question 9a:**

using System;

using System.Threading;

class ThreadExample

{

static void PrintNumbers()

{

for (int i = 1; i <= 5; i++)

{

Console.WriteLine(i);

Thread.Sleep(1000); // Simulate work

}

}

static void Main()

{

Thread newThread = new Thread(PrintNumbers);

newThread.Start();

// Main thread continues

Console.WriteLine("Main thread continues...");

}

}

**Question 10**

using System;

using System.Net.Http;

using System.Threading.Tasks;

using Newtonsoft.Json.Linq;

class NewsAggregator

{

private static readonly HttpClient client = new HttpClient();

static async Task Main()

{

// Define the API endpoint and API key (replace with your actual API key)

string apiKey = "your\_api\_key"; // Replace with your NewsAPI key

string apiUrl = $"https://newsapi.org/v2/top-headlines?country=us&apiKey={apiKey}";

try

{

// Make the GET request

HttpResponseMessage response = await client.GetAsync(apiUrl);

response.EnsureSuccessStatusCode();

// Read and parse the JSON response

string responseBody = await response.Content.ReadAsStringAsync();

JObject json = JObject.Parse(responseBody);

// Extract articles from the JSON response

var articles = json["articles"];

foreach (var article in articles)

{

string title = article["title"].ToString();

string summary = article["description"]?.ToString() ?? "No description available";

// Display article title and summary

Console.WriteLine("Title: " + title);

Console.WriteLine("Summary: " + summary);

Console.WriteLine();

}

}

catch (HttpRequestException e)

{

Console.WriteLine("Request error: " + e.Message);

}

}

}

**Question 10a**

using System;

using System.IO;

class FileReadWrite

{

static void Main()

{

string inputPath = "input.txt";

string outputPath = "output.txt";

using (StreamReader reader = new StreamReader(inputPath))

using (StreamWriter writer = new StreamWriter(outputPath))

{

string line;

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

{

// Filter lines that contain the keyword "C#" and have more than 10 characters

if (line.Contains("C#") && line.Length > 10)

{

writer.WriteLine(line);

}

}

}

Console.WriteLine("File processing complete. Filtered lines have been written to output.txt.");

}

}

**Question 11:**

Packages in C# provide pre-built libraries or tools that can be easily integrated into projects to add functionality or simplify development tasks. NuGet is the package manager for .NET that allows developers to install, update, and manage these packages. To install a NuGet package, you can use the NuGet Package Manager in Visual Studio or the .NET CLI with the command `dotnet add package PackageName`. Packages simplify development by offering reusable components, reducing the need to write code from scratch, and ensuring code consistency by providing well-tested and maintained libraries. This helps streamline development processes, maintain high code quality, and focus on implementing core application logic rather than dealing with low-level details.

**Question 11a:**

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 ZipCode { get; set; }

}

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Address HomeAddress { get; set; }

public List<string> Hobbies { get; set; }

}

static void Main()

{

// Creating a complex object

var person = new Person

{

Name = "John Doe",

Age = 30,

HomeAddress = new Address

{

Street = "123 Main St",

City = "Anytown",

ZipCode = "12345"

},

Hobbies = new List<string> { "Reading", "Swimming", "Coding" }

};

// Serialize object to JSON

string json = JsonConvert.SerializeObject(person, Formatting.Indented);

Console.WriteLine("Serialized JSON:\n" + json);

// Deserialize JSON back to object

var deserializedPerson = JsonConvert.DeserializeObject<Person>(json);

Console.WriteLine("\nDeserialized Person:");

Console.WriteLine($"Name: {deserializedPerson.Name}, Age: {deserializedPerson.Age}");

Console.WriteLine($"Address: {deserializedPerson.HomeAddress.Street}, {deserializedPerson.HomeAddress.City}, {deserializedPerson.HomeAddress.ZipCode}");

Console.WriteLine("Hobbies: " + string.Join(", ", deserializedPerson.Hobbies));

}

}

**Question 12**

`List<T>`, `Queue<T>`, and `Stack<T>` are different data structures in C# with distinct use cases. `List<T>` is a dynamic array that allows random access and efficient insertion/removal at the end, making it ideal for scenarios where you need to frequently access elements by index or perform operations like sorting. `Queue<T>` operates on a first-in, first-out (FIFO) principle, suitable for tasks like processing tasks in order or implementing buffers. `Stack<T>` follows a last-in, first-out (LIFO) approach, which is useful for scenarios like reversing items or managing function calls (e.g., recursion). For instance, use `List<T>` for managing a collection of items with frequent access, `Queue<T>` for task scheduling or breadth-first search algorithms, and `Stack<T>` for implementing undo functionality or parsing expressions.

**Question 12a:**

using System;

using System.Collections.Generic;

class BankQueue

{

static void Main()

{

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

List<string> vipCustomers = new List<string> { "Alice", "Bob" };

// Add customers to the queue

queue.Enqueue("John");

queue.Enqueue("Jane");

queue.Enqueue("Mark");

// Handle VIP customers

foreach (var vip in vipCustomers)

{

queue.Enqueue(vip); // VIPs join the queue

}

Console.WriteLine("Processing queue:");

// Dequeue and serve customers

while (queue.Count > 0)

{

string currentCustomer = queue.Dequeue();

if (vipCustomers.Contains(currentCustomer))

{

Console.WriteLine($"VIP Customer {currentCustomer} is being served.");

}

else

{

Console.WriteLine($"Customer {currentCustomer} is being served.");

}

}

}

}

**Question 13**

Inheritance in C# allows a class (derived class) to inherit fields, properties, and methods from another class (base class), promoting code reuse and establishing a hierarchy. To implement inheritance, use the `:` symbol followed by the base class name in the derived class definition. Access modifiers such as `public`, `protected`, and `private` control the visibility of members. `public` members are accessible from any class, `protected` members are accessible only within the base class and derived classes, and `private` members are accessible only within the base class. For example, if `BaseClass` has a `protected` method `DoWork()`, a `DerivedClass` can access and extend this method, but it will not be accessible to other classes outside this hierarchy. This structure allows derived classes to build on the functionality of the base class while controlling the visibility and accessibility of members.

**Question 13a**

using System;

class Animal

{

public virtual void Speak()

{

Console.WriteLine("Animal speaks.");

}

}

class Dog : Animal

{

public override void Speak()

{

Console.WriteLine("Dog barks.");

}

}

class Cat : Animal

{

public override void Speak()

{

Console.WriteLine("Cat meows.");

}

}

class Bird : Animal

{

public override void Speak()

{

Console.WriteLine("Bird chirps.");

}

}

class Program

{

static void Main(string[] args)

{

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

foreach (var animal in animals)

{

animal.Speak(); // Polymorphism in action

}

}

}

**Question 14**

Polymorphism in C# allows objects of different derived classes to be treated as objects of a common base class, enabling a single interface to represent different underlying forms. It can be achieved through method overriding, where a base class defines a method and derived classes provide specific implementations using the `virtual` and `override` keywords, and through method overloading, where methods in the same class have the same name but different parameters. For example, consider a base class `Animal` with a method `Speak()`. Derived classes `Dog` and `Cat` can override `Speak()` to provide different implementations. This allows a `List<Animal>` to store various `Animal` types and call `Speak()` polymorphically, ensuring the correct method is executed based on the actual object type.

**Question 14a**

using System;

using System.Collections.Generic;

interface IDrive

{

void Drive();

}

class Vehicle : IDrive

{

public virtual void Drive()

{

Console.WriteLine("The vehicle is driving.");

}

}

class Car : Vehicle

{

public override void Drive()

{

Console.WriteLine("The car is driving.");

}

}

class Bike : Vehicle

{

public override void Drive()

{

Console.WriteLine("The bike is driving.");

}

}

class Program

{

static void Main(string[] args)

{

List<IDrive> vehicles = new List<IDrive> { new Car(), new Bike() };

foreach (var vehicle in vehicles)

{

vehicle.Drive(); // Polymorphism in action

}

}

}

**Question 15**

Abstraction in C# hides complex implementation details and exposes only essential features, allowing developers to interact with objects through simplified interfaces. It can be implemented using abstract classes, which provide a partial implementation and require derived classes to complete the implementation, and interfaces, which define a contract that classes must follow without providing any implementation. Abstraction simplifies code by enabling the design of flexible systems where components can interact through common interfaces or base classes, enhancing maintainability by allowing changes to underlying implementations without affecting the rest of the code. For example, using abstraction in a graphics application can simplify rendering different shapes by interacting with a common interface, making it easier to add new shapes or modify rendering logic without altering client code.

**Question 15a**

using System;

abstract class Shape

{

public abstract void Draw();

}

class Circle : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a Circle.");

}

}

class Square : Shape

{

public override void Draw()

{

Console.WriteLine("Drawing a Square.");

}

}

class Program

{

static void Main(string[] args)

{

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

foreach (var shape in shapes)

{

shape.Draw(); // Polymorphism with abstract classes

}

}

}

**Question 16**

**Question 16a**

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

for (int i = 0; i < array.Length; i++) {

Console.WriteLine(array[i]);

}

// Output:

// 1

// 2

// 3

// 4

// 5

**Question 16b**

string str1 = "Hello";

string str2 = "hello";

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

// Output: True

**Question 16c**

object obj1 = new object();

object obj2 = new object();

Console.WriteLine(obj1 == obj2);

// Output: False

**Question 16d**

int a = 5;

int b = 10;

Console.WriteLine(a += b);

// Output: 15

**Question 17**

**Question 17**

using System;

using System.Linq;

using System.Collections.Generic;

class ListMinMax

{

static void Main(string[] args)

{

List<int> numbers = new List<int> { 3, 5, 7, 2, 8, 10 };

if (numbers.Count == 0)

{

Console.WriteLine("List is empty.");

}

else

{

int min = numbers.Min();

int max = numbers.Max();

Console.WriteLine($"Smallest: {min}, Largest: {max}");

}

}

}

**Question 17a**

using System;

using System.Collections.Generic;

class SumIntegers

{

static void Main(string[] args)

{

HashSet<int> uniqueNumbers = new HashSet<int>();

int sum = 0;

int number;

Console.WriteLine("Enter integers (negative number to stop):");

while ((number = int.Parse(Console.ReadLine())) >= 0)

{

if (uniqueNumbers.Add(number)) // Only add unique numbers

{

sum += number;

}

}

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

}

}

**Question 17c**

using System;

class ReverseString

{

static void Main(string[] args)

{

Console.WriteLine("Enter a string:");

string input = Console.ReadLine();

char[] charArray = input.ToCharArray();

Array.Reverse(charArray);

string reversed = new string(charArray);

Console.WriteLine($"Reversed string: {reversed}");

}

}

**Question 17d**

using System;

using System.Collections.Generic;

class StudentGrades

{

static void Main(string[] args)

{

Dictionary<string, double> studentGrades = new Dictionary<string, double>();

studentGrades["Alice"] = 89.5;

studentGrades["Bob"] = 92.0;

studentGrades["Charlie"] = 78.5;

foreach (var student in studentGrades)

{

Console.WriteLine($"Student: {student.Key}, Grade: {student.Value}");

}

}

}

**Question 18**

Interfaces in C# define a contract that classes must adhere to, specifying methods and properties without implementing them. They promote loose coupling by allowing classes to interact through abstractions rather than concrete implementations, enabling easier maintenance and flexibility. This separation allows different components to be replaced or modified independently, fostering code reusability. For instance, you can create multiple implementations of an interface and interchange them without changing the code that relies on the interface, making your code more modular and adaptable to changes.

**Question 18a**

using System;

using System.Collections.Generic;

interface IDrive

{

void Drive();

}

class Car : IDrive

{

public void Drive()

{

Console.WriteLine("Car is driving.");

}

}

class Bike : IDrive

{

public void Drive()

{

Console.WriteLine("Bike is driving.");

}

}

class Program

{

static void Main(string[] args)

{

List<IDrive> vehicles = new List<IDrive> { new Car(), new Bike() };

foreach (var vehicle in vehicles)

{

vehicle.Drive(); // Polymorphism using interface

}

}

}

**Question 18b**

Abstract classes provide a base class definition for other classes to derive from, enforcing some shared behavior while allowing derived classes to implement specific details. Scenario where abstract classes are more appropriate: When you want to define some shared functionality in a base class but still force derived classes to implement their specific functionality. Abstract classes can also contain fields and constructor logic.

**Question 18c**

using System;

using System.Collections.Generic;

abstract class Animal

{

public abstract void MakeSound();

}

class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Dog barks.");

}

}

class Cat : Animal

{

public override void MakeSound()

{

Console.WriteLine("Cat meows.");

}

}

class Program

{

static void Main(string[] args)

{

List<Animal> animals = new List<Animal> { new Dog(), new Cat() };

foreach (var animal in animals)

{

animal.MakeSound(); // Polymorphism in action

}

}

}

**Question 19**

The Top-Down Approach - involves breaking down a system from the highest level (conceptual model) to lower levels (implementation details). This is beneficial for large-scale projects where overall architecture and modules need to be designed before diving into specific functions.

Example: Developing a web-based e-commerce platform where you start with defining high-level modules such as user management, product catalog, and order processing before implementing details like payment processing or inventory management.

**Question 19a**

Bottom-Up Approach - The Bottom-Up Approach starts with the implementation of individual, small, independent units that are gradually combined into larger subsystems. Example: Building a game engine, where you start by implementing low-level components like physics engines and graphics rendering before combining them into a larger game architecture.