Question 1: Method Overloading and Overriding in C#

Problem Statement: Create a C# class demonstrating method overloading and overriding. Show how virtual methods are overridden in derived classes.

Answer Code:

```
public class BaseClass {
    public virtual void Display() {
        Console.WriteLine("BaseClass.Display()");
    }

    public void Display(string message) {
        Console.WriteLine($"BaseClass.Display({message})");
    }
}

public class DerivedClass : BaseClass {
    public override void Display() {
        Console.WriteLine("DerivedClass.Display()");
    }

    public new void Display(string message) {
        base.Display(message);
        Console.WriteLine($"DerivedClass.Display({message})");
    }
}

public class Program {
    static void Main() {
        BaseClass obj = new DerivedClass();
        obj.Display();
        obj.Display("Hello");

        ((DerivedClass)obj).Display();
        ((DerivedClass)obj).Display("World");
    }
}
```

Explanation: This code demonstrates method overloading in the `BaseClass`, where two methods with the same name but different parameters exist. It also shows method overriding in the `DerivedClass`, where the `Display()` method is overridden using the `override` keyword, and the overloaded version of `Display(string message)` is hidden with the `new` keyword. The program showcases how virtual methods are resolved at runtime.

Difficulty Rating: 3/5 (Intermediate)

Question 2: Implement a Singleton Pattern in C#

Problem Statement: Create a thread-safe singleton class using lazy initialization.

Answer Code:

```
public sealed class Singleton {
    private static readonly Lazy<Singleton> _instance = new Lazy<Singleton>(() => new Singleton());

    private Singleton() { }

    public static Singleton Instance {
        get { return _instance.Value; }
    }

    public void DoWork() {
        Console.WriteLine("Singleton is working.");
    }
}

public class Program {
    static void Main() {
        Singleton.Instance.DoWork();
    }
}
```

Explanation: The singleton pattern ensures that a class has only one instance and provides a global point of access to it. This implementation uses `Lazy<T>` for thread-safe, lazy initialization, which is efficient and concise.

Difficulty Rating: 4/5 (Advanced)

Question 3: Implement a Factory Method Pattern

Problem Statement: Create a factory class that can create different types of products based on input parameters.

Answer Code:

```
public interface IProduct {
    void Produce();
}

public class ProductA : IProduct {
    public void Produce() => Console.WriteLine("Producing Product A.");
}

public class ProductB : IProduct {
    public void Produce() => Console.WriteLine("Producing Product B.");
}

public static class ProductFactory {
    public static IProduct CreateProduct(string type) {
        switch (type.ToUpper()) {
            case "A":
                return new ProductA();
            case "B":
                return new ProductB();
            default:
                throw new ArgumentException("Invalid product type.");
        }
    }
}

public class Program {
    static void Main() {
        IProduct product = ProductFactory.CreateProduct("A");
        product.Produce();

        product = ProductFactory.CreateProduct("B");
        product.Produce();
    }
}
```

Explanation: The factory method pattern is implemented using a static `ProductFactory` class that creates instances of different products based on the input type. This promotes loose coupling and makes it easy to add new product types without changing existing code.

Difficulty Rating: 4/5 (Advanced)

Question 4: Implement a Binary Search Tree

Problem Statement: Create a binary search tree and implement insertion and in-order traversal methods.

Answer Code:

```
public class Node {
    public int Value { get; set; }
    public Node Left { get; set; }
    public Node Right { get; set; }

    public Node(int value) {
        Value = value;
        Left = null;
        Right = null;
    }
}
```

```
public class BinarySearchTree {
    public Node Root { get; private set; }

    public void Insert(int value) {
        if (Root == null) {
            Root = new Node(value);
            return;
        }
        Insert(Root, value);
    }

    private void Insert(Node node, int value) {
        if (value < node.Value) {
            if (node.Left == null) {
                node.Left = new Node(value);
            } else {
                Insert(node.Left, value);
            }
        } else {
            if (node.Right == null) {
                node.Right = new Node(value);
            } else {
                Insert(node.Right, value);
            }
        }
    }

    public void InOrderTraversal() {
        InOrderTraversal(Root);
    }

    private void InOrderTraversal(Node node) {
        if (node != null) {
            InOrderTraversal(node.Left);
            Console.Write($"{node.Value} ");
            InOrderTraversal(node.Right);
        }
    }

    public static void Main() {
        BinarySearchTree bst = new BinarySearchTree();
        int[] values = { 5, 3, 7, 2, 4, 8 };
        foreach (int value in values) {
            bst.Insert(value);
        }
        Console.WriteLine("In-order traversal: ");
        bst.InOrderTraversal();
    }
}
```

Explanation: This code implements a binary search tree with insertion and in-order traversal methods. The `Insert` method places each new node in the correct position based on its value, and the `InOrderTraversal` method visits nodes in ascending order.

Difficulty Rating: 4/5 (Advanced)

Question 5: Implement a Custom Exception in C#

Problem Statement: Create a custom exception and use it to handle invalid input scenarios.

Answer Code:

```
public class InvalidInputException : Exception {
    public InvalidInputException(string message) : base(message) { }
}

public class InputValidator {
    public void Validate(int value) {
        if (value < 0) {
            throw new InvalidInputException("Negative values are not allowed.");
        }
        Console.WriteLine("Valid input.");
    }
}
```

```
public class Program {
    static void Main() {
        InputValidator validator = new InputValidator();
        try {
            validator.Validate(-1);
        } catch (InvalidInputException ex) {
            Console.WriteLine(ex.Message);
        }
    }
}
```

Explanation: This code defines a custom exception `InvalidInputException` derived from the base `Exception` class. The `InputValidator` class throws this exception when negative input is detected, which is caught and handled in the `Main` method.

Difficulty Rating: 3/5 (Intermediate)

Question 6: Implement a Generic Repository Pattern

Problem Statement: Create a generic repository pattern to handle CRUD operations on an entity.

Answer Code:

```
public interface IRepository<T> where T : class {
    void Add(T entity);
    void Remove(T entity);
    T GetById(int id);
}

public class Repository<T> : IRepository<T> where T : class {
    private List<T> _entities = new List<T>();

    public void Add(T entity) {
        _entities.Add(entity);
    }

    public void Remove(T entity) {
        _entities.Remove(entity);
    }

    public T GetById(int id) {
        return _entities.FirstOrDefault(e => e is dynamic d && d.Id == id);
    }
}

public class Customer {
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Program {
    static void Main() {
        IRepository<Customer> repository = new Repository<Customer>();
        Customer customer1 = new Customer { Id = 1, Name = "John" };
        repository.Add(customer1);
        Console.WriteLine(repository.GetById(1)?.Name);
    }
}
```

Explanation: The generic repository pattern is implemented with an interface `IRepository<T>` and a concrete class `Repository<T>`. It handles basic CRUD operations for any entity type. The `Customer` class is used as an example to demonstrate how the repository can be utilized.

Difficulty Rating: 4/5 (Advanced)

Question 7: Implement Asynchronous Programming with Async and Await

Problem Statement: Create an async method that fetches data from two sources concurrently.

Answer Code:

```
public class DataFetcher {
    public async Task<string> FetchDataAsync(string source) {
        Console.WriteLine($"Fetching data from {source}...");
        await Task.Delay(1000); // Simulate network delay
        return $"Data from {source}";
    }
}

public class Program {
    static async Task Main() {
        DataFetcher fetcher = new DataFetcher();
        var task1 = fetcher.FetchDataAsync("Source A");
        var task2 = fetcher.FetchDataAsync("Source B");

        Console.WriteLine("Waiting for tasks to complete...");
        string resultA = await task1;
        string resultB = await task2;

        Console.WriteLine(resultA);
        Console.WriteLine(resultB);
    }
}
```

Explanation: This code uses `async` and `await` to fetch data from two sources concurrently. The `FetchDataAsync` method simulates network delays using `Task.Delay`. By awaiting both tasks, the program efficiently waits for their completion without blocking.

Difficulty Rating: 4/5 (Advanced)

Question 8: Implement a Factory Method Pattern for Handling Different Product Types

Problem Statement: Create a factory class that can create different types of products based on input parameters.

Answer Code:

```
public interface IProduct {
    void Use();
}

public class ProductA : IProduct {
    public void Use() => Console.WriteLine("Using Product A.");
}

public class ProductB : IProduct {
    public void Use() => Console.WriteLine("Using Product B.");
}

public static class ProductFactory {
    public static IProduct CreateProduct(string type) {
        switch (type.ToUpper()) {
            case "A":
                return new ProductA();
            case "B":
                return new ProductB();
            default:
                throw new ArgumentException("Invalid product type.");
        }
    }
}

public class Program {
    static void Main() {
        IProduct productA = ProductFactory.CreateProduct("A");
        productA.Use();

        IProduct productB = ProductFactory.CreateProduct("B");
        productB.Use();
    }
}
```

Explanation: The `ProductFactory` class creates instances of different product types based on the input string. This implementation follows the factory method pattern, making it easy to add new product types without modifying existing code.

Difficulty Rating: 3/5 (Intermediate)

Question 9: Implement a Strategy Pattern for Sorting Algorithms

Problem Statement: Create different sorting strategies and apply them dynamically.

Answer Code:

```
public interface ISortStrategy {
    void Sort(int[] array);
}

public class BubbleSort : ISortStrategy {
    public void Sort(int[] array) {
        for (int i = 0; i < array.Length - 1; i++) {
            for (int j = 0; j < array.Length - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                }
            }
        }
    }
}

public class QuickSort : ISortStrategy {
    public void Sort(int[] array) {
        SortHelper(array, 0, array.Length - 1);
    }

    private void SortHelper(int[] array, int left, int right) {
        if (left < right) {
            int pivotIndex = Partition(array, left, right);
            SortHelper(array, left, pivotIndex - 1);
            SortHelper(array, pivotIndex + 1, right);
        }
    }

    private int Partition(int[] array, int left, int right) {
        int pivot = array[right];
        int i = left - 1;
        for (int j = left; j < right; j++) {
            if (array[j] <= pivot) {
                i++;
                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        int temp = array[i + 1];
        array[i + 1] = array[right];
        array[right] = temp;
        return i + 1;
    }
}

public class Sorter {
    private ISortStrategy _strategy;

    public void SetSortStrategy(ISortStrategy strategy) {
        _strategy = strategy;
    }

    public void Sort(int[] array) {
        _strategy.Sort(array);
    }
}

public class Program {
    static void Main() {
```

```
        int[] numbers = { 34, 7, 23, 32, 5, 62 };
        Sorter sorter = new Sorter();

        // Use BubbleSort
        sorter.SetSortStrategy(new BubbleSort());
        sorter.Sort(numbers.Clone() as int[]);
        Console.WriteLine("Bubble Sort: " + string.Join(", ", numbers));

        // Use QuickSort
        sorter.SetSortStrategy(new QuickSort());
        sorter.Sort(numbers.Clone() as int[]);
        Console.WriteLine("Quick Sort: " + string.Join(", ", numbers));
    }
}
```

Explanation: This code implements the strategy pattern with different sorting algorithms (`BubbleSort` and `QuickSort`). The `Sorter` class dynamically sets the strategy to use, allowing for flexible sorting behavior.

Difficulty Rating: 4/5 (Advanced)

Question 10: Implement a Chain of Responsibility Pattern

Problem Statement: Create handlers that process requests in a chain.

Answer Code:

```
public interface IHandler {
    void HandleRequest(int request);
}

public class ConcreteHandlerA : IHandler {
    private IHandler _nextHandler;

    public void SetNext(IHandler handler) {
        _nextHandler = handler;
    }

    public void HandleRequest(int request) {
        if (request < 10) {
            Console.WriteLine("ConcreteHandlerA handled request: " + request);
        } else if (_nextHandler != null) {
            _nextHandler.HandleRequest(request);
        }
    }
}

public class ConcreteHandlerB : IHandler {
    private IHandler _nextHandler;

    public void SetNext(IHandler handler) {
        _nextHandler = handler;
    }

    public void HandleRequest(int request) {
        if (request >= 10 && request < 20) {
            Console.WriteLine("ConcreteHandlerB handled request: " + request);
        } else if (_nextHandler != null) {
            _nextHandler.HandleRequest(request);
        }
    }
}

public class ConcreteHandlerC : IHandler {
    public void HandleRequest(int request) {
        Console.WriteLine("ConcreteHandlerC handled request: " + request);
    }
}

public class ChainBuilder {
    public static IHandler BuildChain() {
        IHandler handlerA = new ConcreteHandlerA();
        IHandler handlerB = new ConcreteHandlerB();
        IHandler handlerC = new ConcreteHandlerC();
```

```
        handlerA.SetNext(handlerB);
        handlerB.SetNext(handlerC);

        return handlerA;
    }
}

public class Program {
    static void Main() {
        IHandler chain = ChainBuilder.BuildChain();
        int[] requests = { 5, 15, 25 };

        foreach (int request in requests) {
            chain.HandleRequest(request);
        }
    }
}
```

Explanation: The chain of responsibility pattern is implemented with a series of handlers (`ConcreteHandlerA` , `ConcreteHandlerB` , `ConcreteHandlerC`) that process requests based on specific conditions. The handlers are linked together in a chain, and each handler decides whether to process the request or pass it along.

Difficulty Rating: 4/5 (Advanced)

Question 11: Implement a Command Pattern

Problem Statement: Create command objects that encapsulate actions to be performed.

Answer Code:

```
public interface ICommand {
    void Execute();
}

public class Light {
    public void TurnOn() => Console.WriteLine("Light is on.");

    public void TurnOff() => Console.WriteLine("Light is off.");
}

public class LightOnCommand : ICommand {
    private readonly Light _light;

    public LightOnCommand(Light light) {
        _light = light;
    }

    public void Execute() => _light.TurnOn();
}

public class LightOffCommand : ICommand {
    private readonly Light _light;

    public LightOffCommand(Light light) {
        _light = light;
    }

    public void Execute() => _light.TurnOff();
}

public class RemoteControl {
    private ICommand _command;

    public void SetCommand(ICommand command) {
        _command = command;
    }

    public void PressButton() {
        if (_command != null) _command.Execute();
    }
}

public class Program {
    static void Main() {
```

```
        Light light = new Light();
        RemoteControl remote = new RemoteControl();

        // Turn lights on
        remote.SetCommand(new LightOnCommand(light));
        remote.PressButton();

        // Turn lights off
        remote.SetCommand(new LightOffCommand(light));
        remote.PressButton();
    }
}
```

Explanation: The command pattern encapsulates a request as an object, allowing for logging, queuing, and undo operations. In this example, the `RemoteControl` class executes commands to turn a light on or off.

Difficulty Rating: 4/5 (Advanced)

Question 12: Implement a Decorator Pattern

Problem Statement: Create decorators to add functionality dynamically to an object.

Answer Code:

```
public interface IComponent {
    void Operation();
}

public class ConcreteComponent : IComponent {
    public void Operation() => Console.WriteLine("ConcreteComponent operation");
}

public abstract class Decorator : IComponent {
    protected IComponent _component;

    public Decorator(IComponent component) {
        _component = component;
    }

    public virtual void Operation() => _component.Operation();
}

public class DecoratorA : Decorator {
    public DecoratorA(IComponent component) : base(component) { }

    public override void Operation() {
        Console.WriteLine("DecoratorA added functionality before operation.");
        base.Operation();
        Console.WriteLine("DecoratorA added functionality after operation.");
    }
}

public class DecoratorB : Decorator {
    public DecoratorB(IComponent component) : base(component) { }

    public override void Operation() {
        Console.WriteLine("DecoratorB added functionality before operation.");
        base.Operation();
        Console.WriteLine("DecoratorB added functionality after operation.");
    }
}

public class Program {
    static void Main() {
        IComponent component = new ConcreteComponent();
        component = new DecoratorA(component);
        component = new DecoratorB(component);

        component.Operation();
    }
}
```

Explanation: The decorator pattern dynamically adds responsibilities to objects. Here, decorators `DecoratorA` and

`DecoratorB` add functionality before and after the operation of a concrete component.

Difficulty Rating: 4/5 (Advanced)

Question 13: Implement a Proxy Pattern

Problem Statement: Create a proxy to control access to a resource-intensive object.

Answer Code:

```
public interface IResource {
    void Load();
}

public class RealResource : IResource {
    public void Load() => Console.WriteLine("RealResource: Loading resource.");
}

public class ResourceProxy : IResource {
    private RealResource _realResource;
    public bool IsCached { get; set; } = false;

    public void Load() {
        if (!IsCached) {
            Console.WriteLine("ResourceProxy: Caching resource.");
            _realResource = new RealResource();
        }
        if (_realResource != null) {
            _realResource.Load();
        }
    }
}

public class Program {
    static void Main() {
        IResource proxy = new ResourceProxy();
        proxy.Load(); // Caches and loads resource
        proxy.IsCached = true;
        proxy.Load(); // Uses cached resource
    }
}
```

Explanation: The proxy pattern controls access to the `RealResource` object. It caches the resource after the first load, preventing repeated expensive operations.

Difficulty Rating: 3/5 (Intermediate)

Question 14: Implement a Flyweight Pattern

Problem Statement: Create a flyweight factory to reuse objects and reduce memory usage.

Answer Code:

```
public interface IFlyweight {
    void Operation(string extrinsic);
}

public class Flyweight : IFlyweight {
    private string _intrinsic;

    public Flyweight(string intrinsic) => _intrinsic = intrinsic;

    public void Operation(string extrinsic) {
        Console.WriteLine($"Flyweight: Displaying {extrinsic} with {_intrinsic}");
    }
}

public class FlyweightFactory {
    private Dictionary<string, IFlyweight> _flyweights = new();
```

```
    public FlyweightFactory() {
        // Initialize with default or common intrinsic states
    }

    public IFlyweight GetFlyweight(string intrinsic) {
        if (!_flyweights.ContainsKey(intrinsic)) {
            _flyweights[intrinsic] = new Flyweight(intrinsic);
        }
        return _flyweights[intrinsic];
    }

    public void PrintFlyweightCount() => Console.WriteLine($"Total flyweights: {_flyweights.Count}");
}

public class Program {
    static void Main() {
        FlyweightFactory factory = new();
        IFlyweight fw1 = factory.GetFlyweight("A");
        IFlyweight fw2 = factory.GetFlyweight("B");
        IFlyweight fw3 = factory.GetFlyweight("A");

        fw1.Operation("Extrinsic 1");
        fw2.Operation("Extrinsic 2");
        fw3.Operation("Extrinsic 3");

        factory.PrintFlyweightCount();
    }
}
```

Explanation: The flyweight pattern reduces memory usage by sharing common objects (flyweights). The `FlyweightFactory` reuses existing flyweights with the same intrinsic state, creating new ones only when necessary.

Difficulty Rating: 3/5 (Intermediate)

Question 15: Implement a Composite Pattern

Problem Statement: Create a structure where individual objects and compositions of objects are treated uniformly.

Answer Code:

```
public interface IComponent {
    void Add(IComponent component);
    void Remove(IComponent component);
    string GetName();
}

public class Leaf : IComponent {
    private string _name;

    public Leaf(string name) => _name = name;

    public void Add(IComponent component) =>
        throw new NotSupportedException("Leaf cannot have children.");

    public void Remove(IComponent component) =>
        throw new NotSupportedException("Leaf cannot remove children.");

    public string GetName() => _name;
}

public class Composite : IComponent {
    private List<IComponent> _children = new();

    public void Add(IComponent component) => _children.Add(component);

    public void Remove(IComponent component) => _children.Remove(component);

    public string GetName() => "Composite";

    public void Display(int indent = 0) {
        Console.WriteLine(new string(' ', indent * 2) + GetName());
        foreach (IComponent component in _children) {
            if (component is Composite composite)
                composite.Display(indent + 1);
```

```
            else
                Console.WriteLine(new string(' ', (indent + 1) * 2) + component.GetName());
        }
    }
}

public class Program {
    static void Main() {
        Composite root = new();
        Leaf leaf1 = new("Leaf 1");
        Leaf leaf2 = new("Leaf 2");
        Composite compositeChild = new();

        compositeChild.Add(new Leaf("Child Leaf 1"));
        compositeChild.Add(new Leaf("Child Leaf 2"));

        root.Add(leaf1);
        root.Add(compositeChild);

        root.Display();
    }
}
```

Explanation: The composite pattern treats individual objects (leaves) and compositions of objects uniformly. The `Composite` class can contain other components, allowing for tree-like structures to be created and displayed.

Difficulty Rating: 4/5 (Advanced)

Question 16: Implement a Template Method Pattern

Problem Statement: Create an abstract class with a template method and define the skeleton of an algorithm.

Answer Code:

```
public abstract class Coffee {
    public void PrepareRecipe() {
        BoilWater();
        BrewCoffeeGroundsInBoiledWater();
        PourInCup();
        AddCondiments();
    }

    protected abstract void BrewCoffeeGroundsInBoiledWater();

    protected abstract void AddCondiments();

    private void BoilWater() => Console.WriteLine("Boiling water.");

    protected virtual void PourInCup() => Console.WriteLine("Pouring into cup.");
}

public class Americano : Coffee {
    protected override void BrewCoffeeGroundsInBoiledWater() {
        Console.WriteLine("Drip brewing coffee grounds.");
    }

    protected override void AddCondiments() {
        Console.WriteLine("Adding sugar and milk.");
    }
}

public class Espresso : Coffee {
    protected override void BrewCoffeeGroundsInBoiledWater() {
        Console.WriteLine("Forcing boiling water through coffee grounds under pressure.");
    }

    protected override void AddCondiments() {
        Console.WriteLine("Adding nothing.");
    }

    protected override void PourInCup() => base.PourInCup();
}

public class Program {
```

```
    static void Main() {
        Coffee coffee = new Americano();
        coffee.PrepareRecipe();

        Console.WriteLine("\n");

        coffee = new Espresso();
        coffee.PrepareRecipe();
    }
}
```

Explanation: The template method pattern defines the skeleton of an algorithm in an abstract class, allowing subclasses to implement specific steps. Here, `Coffee` is the abstract class with a template method `PrepareRecipe`, and subclasses (`Americano`, `Espresso`) provide specific implementations for brewing coffee.

Difficulty Rating: 4/5 (Advanced)

Question 17: Implement a Visitor Pattern

Problem Statement: Create a visitor that can operate on elements of an object structure without changing their classes.

Answer Code:

```
public interface IElement {
    void Accept(IVisitor visitor);
}

public class ElementA : IElement {
    public void Accept(IVisitor visitor) => visitor.Visit(this);
}

public class ElementB : IElement {
    public void Accept(IVisitor visitor) => visitor.Visit(this);
}

public interface IVisitor {
    void Visit(ElementA element);
    void Visit(ElementB element);
}

public class ConcreteVisitor : IVisitor {
    public void Visit(ElementA element) => Console.WriteLine("ConcreteVisitor visiting ElementA.");
    public void Visit(ElementB element) => Console.WriteLine("ConcreteVisitor visiting ElementB.");
}

public class ObjectStructure {
    private List<IElement> _elements = new();

    public void Add(IElement element) => _elements.Add(element);

    public void Accept(IVisitor visitor) {
        foreach (IElement element in _elements) {
            element.Accept(visitor);
        }
    }
}

public class Program {
    static void Main() {
        ObjectStructure structure = new();
        structure.Add(new ElementA());
        structure.Add(new ElementB());

        IVisitor visitor = new ConcreteVisitor();
        structure.Accept(visitor);
    }
}
```

Explanation: The visitor pattern allows for operations to be performed on elements of an object structure without changing their classes. Here, `ConcreteVisitor` visits different element types and performs operations specific to each.

Difficulty Rating: 4/5 (Advanced)

Question 18: Implement a State Pattern

Problem Statement: Create state objects that encapsulate varying behavior based on the state of an object.

Answer Code:

```
public interface IState {
    void Handle();
}

public class StateA : IState {
    public void Handle() => Console.WriteLine("Handling State A.");
}

public class StateB : IState {
    public void Handle() => Console.WriteLine("Handling State B.");
}

public class Context {
    private IState _state;

    public void Transition(IState state) => _state = state;

    public void Request() {
        Console.WriteLine("Context requests handling.");
        _state.Handle();
    }
}

public class Program {
    static void Main() {
        Context context = new();
        IState stateA = new StateA();
        IState stateB = new StateB();

        context.Transition(stateA);
        context.Request();

        context.Transition(stateB);
        context.Request();
    }
}
```

Explanation: The state pattern allows an object to change its behavior when its internal state changes. Here, the `Context` object transitions between different states (`StateA` , `StateB`) and delegates handling to the current state.

Difficulty Rating: 3/5 (Intermediate)

Question 19: Implement a Model-View-Presenter (MVP) Pattern

Problem Statement: Create an MVP pattern implementation for a simple calculator.

Answer Code:

```
public interface IView {
    int NumberA { get; set; }
    int NumberB { get; set; }
    int Result { get; set; }

    void ShowResult();
}

public class CalculatorPresenter {
    private IView _view;

    public CalculatorPresenter(IView view) => _view = view;

    public void Add() {
        int result = _view.NumberA + _view.NumberB;
        _view.Result = result;
        _view.ShowResult();
```

```
    }
}

public class CalculatorView : IView {
    public int NumberA { get; set; }
    public int NumberB { get; set; }
    public int Result { get; set; }

    public void ShowResult() => Console.WriteLine($"Result: {Result}");
}

public class Program {
    static void Main() {
        IView view = new CalculatorView();
        CalculatorPresenter presenter = new(view);

        view.NumberA = 5;
        view.NumberB = 3;

        presenter.Add();
    }
}
```

Explanation: The MVP pattern separates the user interface (View), business logic (Presenter), and data (Model). Here, the `CalculatorPresenter` handles user input from the view and updates it with the result.

Difficulty Rating: 4/5 (Advanced)

Question 20: Implement a Publish-Subscribe Pattern

Problem Statement: Create a publish-subscribe pattern where subscribers are notified of events.

Answer Code:

```
public interface ISubscriber {
    void Update(string message);
}

public class Publisher {
    private List<ISubscriber> _subscribers = new();

    public void Subscribe(ISubscriber subscriber) => _subscribers.Add(subscriber);

    public void Unsubscribe(ISubscriber subscriber) => _subscribers.Remove(subscriber);

    public void NotifySubscribers(string message) {
        foreach (ISubscriber subscriber in _subscribers) {
            subscriber.Update(message);
        }
    }
}

public class SubscriberA : ISubscriber {
    public void Update(string message) => Console.WriteLine("Subscriber A received: " + message);
}

public class SubscriberB : ISubscriber {
    public void Update(string message) => Console.WriteLine("Subscriber B received: " + message);
}

public class Program {
    static void Main() {
        Publisher publisher = new();
        ISubscriber subscriberA = new SubscriberA();
        ISubscriber subscriberB = new SubscriberB();

        publisher.Subscribe(subscriberA);
        publisher.Subscribe(subscriberB);

        publisher.NotifySubscribers("Hello Subscribers!");

        publisher.Unsubscribe(subscriberA);
        publisher.NotifySubscribers("Only B receives this.");
    }
```

```
}
```

Explanation: The publish-subscribe pattern decouples publishers from subscribers. Here, `SubscriberA` and `SubscriberB` are notified when the publisher sends messages. Subscribers can be added or removed dynamically.

Difficulty Rating: 4/5 (Advanced)

Question 21: Implement a Dependency Injection in C#

Problem Statement: Use dependency injection to resolve dependencies and implement inversion of control.

Answer Code:

```
public interface ILogger {
    void Log(string message);
}

public class ConsoleLogger : ILogger {
    public void Log(string message) => Console.WriteLine("Console: " + message);
}

public class FileLogger : ILogger {
    public void Log(string message) => Console.WriteLine("File: " + message);
}

public class Service {
    private readonly ILogger _logger;

    public Service(ILogger logger) => _logger = logger;

    public void DoWork() {
        _logger.Log("Service is doing work.");
    }
}

public class Program {
    static void Main() {
        // Using ConsoleLogger
        ILogger logger1 = new ConsoleLogger();
        Service service1 = new(logger1);
        service1.DoWork();

        // Using FileLogger
        ILogger logger2 = new FileLogger();
        Service service2 = new(logger2);
        service2.DoWork();
    }
}
```

Explanation: Dependency injection and inversion of control are implemented by passing dependencies (loggers) to the `Service` class constructor. This decouples the service from specific logger implementations, making it more flexible and testable.

Difficulty Rating: 3/5 (Intermediate)

Question 22: Implement a UnitOfWork Pattern

Problem Statement: Create a unit of work pattern to manage data persistence.

Answer Code:

```
public interface IRepository<T> where T : class {
    void Add(T entity);
    void Remove(T entity);
}

public interface IUnitOfWork {
    IRepository<T> GetRepository<T>() where T : class;
}
```

```
public class Repository<T> : IRepository<T> where T : class {
    private List<T> _entities = new();

    public void Add(T entity) => _entities.Add(entity);

    public void Remove(T entity) => _entities.Remove(entity);
}

public class UnitOfWork : IUnitOfWork {
    private Dictionary<Type, object> _repositories = new();

    public IRepository<T> GetRepository<T>() where T : class {
        Type type = typeof(T);
        if (!_repositories.ContainsKey(type)) {
            _repositories[type] = new Repository<T>();
        }
        return (IRepository<T>)_repositories[type];
    }

    public void SaveChanges() {
        foreach (var repository in _repositories.Values) {
            if (repository is Repository<T> repo)
                Console.WriteLine($"Saving changes for {typeof(T).Name}: {repo.Entities.Count} entities.");
        }
    }
}

public class Program {
    static void Main() {
        IUnitOfWork unitOfWork = new UnitOfWork();

        IRepository<Customer> customerRepo = unitOfWork.GetRepository<Customer>();
        IRepository<Order> orderRepo = unitOfWork.GetRepository<Order>();

        Customer customer = new() { Id = 1, Name = "John" };
        Order order = new() { Id = 101, CustomerId = 1 };

        customerRepo.Add(customer);
        orderRepo.Add(order);

        unitOfWork.SaveChanges();
    }
}

public class Customer {
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Order {
    public int Id { get; set; }
    public int CustomerId { get; set; }
}
```

Explanation: The unit of work pattern manages a collection of repository objects and coordinates operations between them. Here, `UnitOfWork` provides repositories for different entity types and handles saving changes.

Difficulty Rating: 4/5 (Advanced)

Question 23: Implement a Specification Pattern

Problem Statement: Create specifications to encapsulate business rules for validation.

Answer Code:

```
public interface ISpecification<T> {
    bool IsSatisfiedBy(T candidate);
}

public class AndSpecification<T> : ISpecification<T> {
    private readonly ISpecification<T> _left;
    private readonly ISpecification<T> _right;

    public AndSpecification(ISpecification<T> left, ISpecification<T> right) {
```

```
        _left = left;
        _right = right;
    }

    public bool IsSatisfiedBy(T candidate) => _left.IsSatisfiedBy(candidate) &&
_right.IsSatisfiedBy(candidate);
}

public class AgeSpecification : ISpecification<Person> {
    private int _minimumAge;

    public AgeSpecification(int minimumAge) => _minimumAge = minimumAge;

    public bool IsSatisfiedBy(Person person) =>
        person.Age >= _minimumAge;
}

public class NameSpecification : ISpecification<Person> {
    private string _name;

    public NameSpecification(string name) => _name = name;

    public bool IsSatisfiedBy(Person person) =>
        string.Equals(person.Name, _name, StringComparison.OrdinalIgnoreCase);
}
```

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

public class Program {
    static void Main() {
        Person person = new() { Name = "John", Age = 25 };

        ISpecification<Person> ageSpec = new AgeSpecification(18);
        ISpecification<Person> nameSpec = new NameSpecification("John");

        ISpecification<Person> spec = new AndSpecification<Person>(ageSpec, nameSpec);

        if (spec.IsSatisfiedBy(person)) {
            Console.WriteLine("Person satisfies the specifications.");
        } else {
            Console.WriteLine("Person does not satisfy the specifications.");
        }
    }
}
```

Explanation: The specification pattern encapsulates business rules in reusable objects. Here, `AgeSpecification` and `NameSpecification` check if a person meets certain criteria, and `AndSpecification` combines them using logical AND.

Difficulty Rating: 4/5 (Advanced)

Question 24: Implement a Chain of Responsibility Pattern for Authentication

Problem Statement: Create handlers that check user credentials in a chain.

Answer Code:

```
public interface IHandler {
    bool HandleAuth(string username, string password);
}

public class UsernameValidator : IHandler {
    public bool HandleAuth(string username, string password) {
        if (string.IsNullOrEmpty(username)) {
            Console.WriteLine("Invalid username.");
            return false;
        }
        return true;
    }
}

public class PasswordValidator : IHandler {
```

```
    public bool HandleAuth(string username, string password) {
        if (password.Length < 8) {
            Console.WriteLine("Password too short.");
            return false;
        }
        return true;
    }
}

public class RoleValidator : IHandler {
    public bool HandleAuth(string username, string password) {
        if (username != "admin") {
            Console.WriteLine("Unauthorized role.");
            return false;
        }
        return true;
    }
}

public class AuthChainBuilder {
    public static IHandler BuildChain() {
        IHandler usernameValidator = new UsernameValidator();
        IHandler passwordValidator = new PasswordValidator();
        IHandler roleValidator = new RoleValidator();

        usernameValidator.HandleAuth = (u, p) => {
            if (string.IsNullOrEmpty(u)) {
                Console.WriteLine("Invalid username.");
                return false;
            }
            return passwordValidator.HandleAuth(u, p);
        };

        passwordValidator.HandleAuth = (u, p) => {
            if (p.Length < 8) {
                Console.WriteLine("Password too short.");
                return false;
            }
            return roleValidator.HandleAuth(u, p);
        };

        roleValidator.HandleAuth = (u, p) => {
            if (u != "admin") {
                Console.WriteLine("Unauthorized role.");
                return false;
            }
            Console.WriteLine("Authentication successful.");
            return true;
        };

        return usernameValidator;
    }
}

public class Program {
    static void Main() {
        IHandler authChain = AuthChainBuilder.BuildChain();

        // Test authentication
        bool result1 = authChain.HandleAuth("admin", "P@ssw0rd");
        Console.WriteLine("\n");

        bool result2 = authChain.HandleAuth("test", "1234");
    }
}
```

Explanation: The chain of responsibility pattern is used to validate user credentials in sequence. Each handler checks a specific condition and passes the request along if it's satisfied.

Difficulty Rating: 4/5 (Advanced)

Question 25: Implement a Threading Model with Thread-Safe Operations

Problem Statement: Create thread-safe operations using locks.

Answer Code:

```
public class Counter {
    private int _value;
    private readonly object _lock = new();

    public void Increment() {
        lock (_lock) {
            _value++;
            Console.WriteLine($"Counter incremented to {_value} by thread {Thread.CurrentThread.Name}");
        }
    }

    public void Decrement() {
        lock (_lock) {
            _value--;
            Console.WriteLine($"Counter decremented to {_value} by thread {Thread.CurrentThread.Name}");
        }
    }

    public int GetValue() {
        lock (_lock) {
            return _value;
        }
    }
}

public class Program {
    static void Main() {
        Counter counter = new();
        Thread[] threads = new Thread[5];

        for (int i = 0; i < 2; i++) {
            threads[i] = new Thread(() => counter.Increment()) { Name = "IncrementThread" + (i + 1) };
            threads[i].Start();
        }

        for (int i = 2; i < 4; i++) {
            threads[i] = new Thread(() => counter.Decrement()) { Name = "DecrementThread" + (i - 1) };
            threads[i].Start();
        }

        foreach (Thread thread in threads) {
            if (thread != null)
                thread.Join();
        }

        Console.WriteLine("Final counter value: " + counter.GetValue());
    }
}
```

Explanation: This code uses a lock to synchronize access to a counter's methods, ensuring thread-safe operations. Multiple threads increment and decrement the counter safely.

Difficulty Rating: 4/5 (Advanced)

Question 26: Implement a Custom Threading Synchronization Mechanism

Problem Statement: Create a custom synchronization mechanism to manage thread access.

Answer Code:

```
public class CustomSemaphore {
    private int _count;
    private readonly object _sync = new();

    public CustomSemaphore(int count) => _count = count;

    public void Enter() {
        lock (_sync) {
            while (_count <= 0) {
                Monitor.Wait(_sync);
            }
```

```
            _count--;
        }
    }

    public void Exit() {
        lock (_sync) {
            _count++;
            Monitor.Pulse(_sync);
        }
    }
}

public class Program {
    static void Main() {
        CustomSemaphore semaphore = new(3); // Allow 3 threads at a time
        Thread[] threads = new Thread[5];

        for (int i = 0; i < 5; i++) {
            int threadNumber = i + 1;
            threads[i] = new Thread(() => {
                semaphore.Enter();
                Console.WriteLine("Thread {0} entered the critical section.", Thread.CurrentThread.Name);
                Thread.Sleep(1000);
                Console.WriteLine("Thread {0} exiting the critical section.", Thread.CurrentThread.Name);
                semaphore.Exit();
            }) { Name = threadNumber.ToString() };
        }

        foreach (Thread thread in threads) {
            thread.Start();
        }

        foreach (Thread thread in threads) {
            thread.Join();
        }
    }
}
```

Explanation: The custom `CustomSemaphore` class controls access to a critical section by limiting the number of concurrent threads. It uses `Monitor.Wait` and `Monitor.Pulse` to manage thread synchronization.

Difficulty Rating: 4/5 (Advanced)

Question 27: Implement a Wait-notify Mechanism in Threading

Problem Statement: Use `Monitor.Wait` and `Monitor.Pulse` for thread synchronization.

Answer Code:

```
public class WaitNotifyExample {
    private int _value;
    private readonly object _lock = new();

    public void Produce() {
        lock (_lock) {
            while (_value > 0) {
                Monitor.Wait(_lock);
            }
            _value++;
            Console.WriteLine("Produced: " + _value);
            Monitor.Pulse(_lock);
        }
    }

    public void Consume() {
        lock (_lock) {
            while (_value <= 0) {
                Monitor.Wait(_lock);
            }
            _value--;
            Console.WriteLine("Consumed: " + _value);
            Monitor.Pulse(_lock);
        }
    }
```

```
}

public class Program {
    static void Main() {
        WaitNotifyExample example = new();
        Thread producer = new(() => {
            for (int i = 0; i < 3; i++) {
                example.Produce();
                Thread.Sleep(500);
            }
        }) { Name = "Producer" };

        Thread consumer = new(() => {
            for (int i = 0; i < 3; i++) {
                example.Consume();
                Thread.Sleep(500);
            }
        }) { Name = "Consumer" };

        producer.Start();
        consumer.Start();

        producer.Join();
        consumer.Join();
    }
}
```

Explanation: This example uses `Monitor.Wait` and `Monitor.Pulse` to synchronize producer and consumer threads. The producer increments a value, while the consumer decrements it.

Difficulty Rating: 4/5 (Advanced)

Question 28: Implement a ThreadPool with Custom Work Items

Problem Statement: Use the thread pool to execute custom work items.

Answer Code:

```
public class WorkItem {
    private int _id;

    public WorkItem(int id) => _id = id;

    public void Execute() {
        Console.WriteLine("Work item {0} is executing.", _id);
        Thread.Sleep(1000);
        Console.WriteLine("Work item {0} completed.", _id);
    }
}

public class Program {
    static void Main() {
        WorkItem[] items = { new(1), new(2), new(3) };

        foreach (WorkItem item in items) {
            ThreadPool.QueueUserWorkItem(_ => item.Execute());
        }

        Console.WriteLine("All work items queued.");
        Thread.Sleep(4000);
    }
}
```

Explanation: The thread pool efficiently manages threads for executing work items. In this example, multiple `WorkItem` instances are queued and executed asynchronously.

Difficulty Rating: 3/5 (Intermediate)

Question 29: Implement a Custom Thread Local Storage

Problem Statement: Use thread-local storage to store data specific to each thread.

Answer Code:

```
public class ThreadLocalStorageExample {
    private static readonly ThreadLocal<int> _threadData = new(() => 42);

    public void DisplayThreadData() {
        Console.WriteLine($"Thread {Thread.CurrentThread.Name} has data: {_threadData.Value}");
    }
}

public class Program {
    static void Main() {
        Thread[] threads = new Thread[3];

        for (int i = 0; i < threads.Length; i++) {
            int threadIndex = i + 1;
            threads[i] = new Thread(() => {
                ThreadLocalStorageExample example = new();
                example.DisplayThreadData();
            }) { Name = "Thread" + threadIndex };
        }

        foreach (Thread thread in threads) {
            thread.Start();
        }

        foreach (Thread thread in threads) {
            thread.Join();
        }
    }
}
```

Explanation: Thread-local storage allows each thread to have its own copy of data. Here, the static `ThreadLocal<int>` field provides a default value per thread.

Difficulty Rating: 3/5 (Intermediate)

Question 30: Implement a Synchronous and Asynchronous Singleton

Problem Statement: Create singleton instances with thread safety for both synchronous and asynchronous access.

Answer Code:

```
public class Singleton {
    private static Singleton _instance;
    private static readonly object _syncObject = new();

    private Singleton() { }

    public static Singleton Instance {
        get {
            if (_instance == null) {
                lock (_syncObject) {
                    if (_instance == null) {
                        _instance = new Singleton();
                    }
                }
            }
            return _instance;
        }
    }

    public static async Task<Singleton> GetAsyncInstance() {
        if (_instance == null) {
            await Task.Run(() => {
                lock (_syncObject) {
                    if (_instance == null) {
                        _instance = new Singleton();
                    }
                }
            });
        }
        return _instance;
```

```
    }
}

public class Program {
    static void Main() {
        // Synchronous singleton access
        Singleton syncSingleton = Singleton.Instance;

        // Asynchronous singleton access
        Task<Singleton> asyncTask = Singleton.GetAsyncInstance();
        Singleton asyncSingleton = asyncTask.Result;

        Console.WriteLine("Synchronized Singleton: " + syncSingleton.GetHashCode());
        Console.WriteLine("Asynchronous Singleton: " + asyncSingleton.GetHashCode());
    }
}
```

Explanation: This singleton implementation ensures thread safety for both synchronous and asynchronous access. The `Instance` property uses double-checked locking, while `GetAsyncInstance` ensures thread safety in asynchronous scenarios.

Difficulty Rating: 4/5 (Advanced)

Question 31: Implement a Custom Asynchronous Programming Model

Answer Code:

```
public class AsyncOperation {
    private Action _completedCallback;
    private readonly object _lock = new();

    public event EventHandler Completed;

    public void StartAsyncOperation(Action callback) {
        _completedCallback = callback;
        ThreadPool.QueueUserWorkItem(_ => PerformOperation());
    }

    private void PerformOperation() {
        // Simulate long-running operation
        Thread.Sleep(2000);
        Console.WriteLine("Async operation completed.");

        lock (_lock) {
            OnCompleted();
        }
    }

    private void OnCompleted() {
        Completed?.Invoke(this, EventArgs.Empty);
        if (_completedCallback != null) {
            _completedCallback();
        }
    }
}

public class Program {
    static void Main() {
        AsyncOperation operation = new();
        operation.Completed += (sender, args) => Console.WriteLine("Event handler notified.");

        operation.StartAsyncOperation(() => {
            Console.WriteLine("Callback executed.");
        });

        // Keep console alive
        Thread.Sleep(3000);
    }
}
```

Explanation: This custom asynchronous model uses `ThreadPool` for background operations and notifies both event handlers and callbacks upon completion.

Difficulty Rating: 4/5 (Advanced)

Question 32: Implement a Reactive Extensions (Rx) Observable

Answer Code:

```
public class RxExample {
    public IObservable<int> GetNumbersObservable() {
        return Observable.Create<int>(observer => {
            observer.OnNext(1);
            observer.OnNext(2);
            observer.OnCompleted();
            return () => Console.WriteLine("Subscription disposed.");
        });
    }
}

public class Program {
    static void Main() {
        RxExample rx = new();
        IDisposable subscription = null;

        var observable = rx.GetNumbersObservable();
        subscription = observable.Subscribe(
            x => Console.WriteLine("Received: " + x),
            ex => Console.WriteLine("Error: " + ex.Message),
            () => Console.WriteLine("Completed.")
        );

        // Simulate cancellation
        Thread.Sleep(1000);
        subscription.Dispose();

        Console.WriteLine("Subscription disposed.");
    }
}
```

Explanation: This example creates an Rx observable that emits numbers and handles subscriptions, errors, and completion. The subscription can be disposed to cancel observation.

Difficulty Rating: 4/5 (Advanced)

Question 33: Implement a Custom LINQ Provider

Answer Code:

```
public class MyEnumerable<T> : IEnumerable<T> {
    private readonly List<T> _data;

    public MyEnumerable(List<T> data) => _data = data;

    public IEnumerator<T> GetEnumerator() => _data.GetEnumerator();
    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

    public MyEnumerable<TResult> Select<TResult>(Func<T, TResult> selector) {
        return new MyEnumerable<TResult>(_data.Select(selector).ToList());
    }
}

public static class MyLinqExtensions {
    public static MyEnumerable<TResult> Select<T, TResult>(
        this IEnumerable<T> source,
        Func<T, TResult> selector
    ) {
        if (source is MyEnumerable<T> enumerable) {
            return enumerable.Select(selector);
        }
        throw new ArgumentException("Source must be of type MyEnumerable<T>.");
    }
}

public class Program {
    static void Main() {
```

```
        List<int> numbers = new() { 1, 2, 3, 4 };
        MyEnumerable<int> myEnumerable = new(numbers);

        var squaredNumbers = myEnumerable.Select(x => x * x);
        foreach (int num in squaredNumbers) {
            Console.WriteLine(num);
        }
    }
}
```

Explanation: This custom LINQ provider extends `MyEnumerable` with a `Select` method, demonstrating how to create a simple LINQ provider.

Difficulty Rating: 4/5 (Advanced)

Question 34: Implement a Custom Exception Handling Mechanism

Answer Code:

```
public class Operation {
    public void Perform() {
        try {
            Console.WriteLine("Performing operation...");
            Thread.Sleep(1000);
            throw new InvalidOperationException("An error occurred.");
        } catch (Exception ex) {
            OnError(ex);
        }
    }

    protected virtual void OnError(Exception ex) =>
        Console.WriteLine("Default error handler: " + ex.Message);
}

public class CustomOperation : Operation {
    protected override void OnError(Exception ex) =>
        Console.WriteLine("Custom error handler: " + ex.Message);
}

public class Program {
    static void Main() {
        Operation op1 = new();
        op1.Perform();

        CustomOperation op2 = new();
        op2.Perform();
    }
}
```

Explanation: This example shows a custom exception handling mechanism where exceptions are caught and handled in an `OnError` method, which can be overridden by derived classes.

Difficulty Rating: 3/5 (Intermediate)

Question 35: Implement a Custom Logging Mechanism

Answer Code:

```
public interface ILogger {
    void Log(string message);
}

public class ConsoleLogger : ILogger {
    public void Log(string message) =>
        Console.WriteLine("[Console] " + message);
}

public class FileLogger : ILogger {
    public void Log(string message) =>
        File.AppendAllText("log.txt", DateTime.Now.ToString() + ": " + message + Environment.NewLine);
```

```
}

public class LogManager {
    private static ILogger _logger;

    public static ILogger Logger {
        get => _logger;
        set => _logger = value ?? throw new ArgumentNullException(nameof(value));
    }

    static LogManager() {
        // Default logger
        _logger = new ConsoleLogger();
    }
}

public class Program {
    static void Main() {
        LogManager.Logger.Log("This is a log message.");

        // Change logger
        LogManager.Logger = new FileLogger();
        LogManager.Logger.Log("Another log message.");

        // Check if log.txt was created
        if (File.Exists("log.txt")) {
            Console.WriteLine("Log file exists.");
            string[] lines = File.ReadAllLines("log.txt");
            foreach (string line in lines) {
                Console.WriteLine(line);
            }
        }
    }
}
```

Explanation: This custom logging mechanism allows switching between different loggers (console and file) at runtime.

Difficulty Rating: 3/5 (Intermediate)

Question 36: Implement a Custom Task Scheduler

Answer Code:

```
public class SimpleTask {
    public string Name { get; set; }
    public Action Execute { get; set; }
}

public class SimpleTaskScheduler {
    private Queue<SimpleTask> _taskQueue = new();
    private Thread _schedulerThread;
    private volatile bool _running;

    public SimpleTaskScheduler() {
        _schedulerThread = new Thread(Run) { IsBackground = true };
        _schedulerThread.Start();
    }

    public void ScheduleTask(SimpleTask task) {
        if (task == null) throw new ArgumentNullException(nameof(task));

        lock (_taskQueue) {
            _taskQueue.Enqueue(task);
            Monitor.Pulse(_taskQueue);
        }
    }

    private void Run() {
        _running = true;
        while (_running) {
            lock (_taskQueue) {
                while (_taskQueue.Count == 0) {
                    Monitor.Wait(_taskQueue);
                }
```

```
                SimpleTask task = _taskQueue.Dequeue();
                Task.Run(task.Execute).GetAwaiter().GetResult(); // Wait for task to complete
            }
        }
    }

    public void Stop() {
        _running = false;
        lock (_taskQueue) {
            Monitor.Pulse(_taskQueue);
        }
    }
}

public class Program {
    static void Main() {
        SimpleTaskScheduler scheduler = new();

        // Create and schedule tasks
        for (int i = 0; i < 5; i++) {
            scheduler.ScheduleTask(new SimpleTask {
                Name = "Task" + (i + 1),
                Execute = () => Console.WriteLine("Executing Task {0}", Thread.CurrentThread.Name)
            });
        }

        // Stop after some time
        Thread.Sleep(3000);
        scheduler.Stop();

        Console.WriteLine("Scheduler stopped.");
    }
}
```

Explanation: This custom task scheduler uses a queue to manage tasks and runs them in the background. Tasks are executed on separate threads.

Difficulty Rating: 4/5 (Advanced)

Question 37: Implement a Custom Dependency Injection Container

Answer Code:

```
public interface IService {
    void Execute();
}

public class ServiceA : IService {
    public void Execute() => Console.WriteLine("Service A executed.");
}

public class ServiceB : IService {
    public void Execute() => Console.WriteLine("Service B executed.");
}

public class DependencyInjector {
    private Dictionary<Type, object> _services = new();

    public void Register<TService, TImplementation>()
        where TService : class
        where TImplementation : class, TService {
            _services[typeof(TService)] = new TImplementation();
    }

    public TService Resolve<TService>() where TService : class {
        return (TService)_services.GetType(typeof(TService)).Value;
    }
}

public class Program {
    static void Main() {
        DependencyInjector injector = new();
        injector.Register<IService, ServiceA>();
        injector.Register<IService, ServiceB>(); // Will override
```

```
        IService service = injector.Resolve<IService>();
        service.Execute();
    }
}
```

Explanation: This custom dependency injection container allows registering and resolving services, though it has limitations (e.g., only one instance per service type).

Difficulty Rating: 3/5 (Intermediate)

Question 38: Implement a Custom Data Access Layer

Answer Code:

```
public interface IRepository<T> {
    void Add(T entity);
    T Get(int id);
}

public class Repository<T> : IRepository<T> where T : IEntity {
    private List<T> _entities = new();

    public void Add(T entity) => _entities.Add(entity);

    public T Get(int id) {
        foreach (T entity in _entities) {
            if (entity.Id == id) {
                return entity;
            }
        }
        return default;
    }
}

public interface IEntity {
    int Id { get; set; }
}

public class Customer : IEntity {
    public int Id { get; set; }
    public string Name { get; set; }
}

public class Program {
    static void Main() {
        IRepository<Customer> repo = new Repository<Customer>();
        Customer customer = new() { Id = 1, Name = "John" };
        repo.Add(customer);
        Customer retrievedCustomer = repo.Get(1);

        if (retrievedCustomer != null) {
            Console.WriteLine("Retrieved customer: " + retrievedCustomer.Name);
        } else {
            Console.WriteLine("Customer not found.");
        }
    }
}
```

Explanation: This custom data access layer uses a generic repository pattern to manage entity storage and retrieval.

Difficulty Rating: 3/5 (Intermediate)

Question 39: Implement a Custom Validation Framework

Answer Code:

```
public interface IValidator {
    bool Validate(object target);
```

```
}

public class StringLengthValidator : IValidator {
    private int _maxLength;

    public StringLengthValidator(int maxLength) => _maxLength = maxLength;

    public bool Validate(object target) {
        if (target is string s) {
            return s.Length <= _maxLength;
        }
        return false;
    }
}

public class ValidatorEngine {
    private readonly List<IValidator> _validators = new();

    public void AddValidator(IValidator validator) => _validators.Add(validator);

    public bool Validate(object target) {
        foreach (IValidator validator in _validators) {
            if (!validator.Validate(target)) {
                return false;
            }
        }
        return true;
    }
}

public class Program {
    static void Main() {
        ValidatorEngine engine = new();
        IValidator validator1 = new StringLengthValidator(5);
        IValidator validator2 = new StringLengthValidator(10);

        engine.AddValidator(validator1);
        engine.AddValidator(validator2);

        string input = "Hello";
        bool isValid = engine.Validate(input);

        Console.WriteLine("Input is " + (isValid ? "valid" : "invalid"));
    }
}
```

Explanation: This custom validation framework uses validators to check input against specific rules. The `ValidatorEngine` executes all validations and returns the result.

Difficulty Rating: 3/5 (Intermediate)

Question 40: Implement a Custom Message Broker

Answer Code:

```
public interface IMessageBroker {
    void SendMessage(string message);
    string ReceiveMessage();
}

public class InMemoryMessageBroker : IMessageBroker {
    private Queue<string> _messages = new();
    private readonly object _sync = new();

    public void SendMessage(string message) {
        lock (_sync) {
            _messages.Enqueue(message);
            Monitor.Pulse(_sync);
        }
    }

    public string ReceiveMessage() {
        lock (_sync) {
            while (_messages.Count == 0) {
```

```
                Monitor.Wait(_sync);
            }
            return _messages.Dequeue();
        }
    }
}

public class Program {
    static void Main() {
        IMessageBroker broker = new InMemoryMessageBroker();

        // Start receiver thread
        Thread receiverThread = new(() => {
            while (true) {
                string message = broker.ReceiveMessage();
                Console.WriteLine("Received: " + message);
            }
        }) { IsBackground = true };
        receiverThread.Start();

        // Send messages
        for (int i = 0; i < 5; i++) {
            broker.SendMessage("Message " + (i + 1));
            Thread.Sleep(500);
        }

        // Keep console alive
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

Explanation: This in-memory message broker uses a queue to store messages and synchronize between sender and receiver threads.

Difficulty Rating: 4/5 (Advanced)

Question 41: Implement a Custom Expression Evaluator

Answer Code:

```
public class ExpressionEvaluator {
    public double Evaluate(string expression) {
        // Simplified evaluation for addition and subtraction
        string[] parts = expression.Split(new[] { ' ', '+', '-' }, StringSplitOptions.RemoveEmptyEntries);
        double result = 0;
        bool isAddition = true;

        for (int i = 0; i < parts.Length; i++) {
            if (i == 0) {
                result = double.Parse(parts[i]);
                continue;
            }

            if (parts[i - 1] == "+") {
                result += double.Parse(parts[i]);
            } else if (parts[i - 1] == "-") {
                result -= double.Parse(parts[i]);
            }
        }

        return result;
    }
}

public class Program {
    static void Main() {
        ExpressionEvaluator evaluator = new();

        string expression1 = "5 + 3 - 2";
        double result1 = evaluator.Evaluate(expression1);
        Console.WriteLine("Result of " + expression1 + ": " + result1);

        string expression2 = "10 - 4 + 6";
```

```
        double result2 = evaluator.Evaluate(expression2);
        Console.WriteLine("Result of " + expression2 + ": " + result2);
    }
}
```

Explanation: This custom evaluator processes simple arithmetic expressions involving addition and subtraction.

Difficulty Rating: 3/5 (Intermediate)

Question 42: Implement a Custom Code Generation Tool

Answer Code:

```
public class CodeGenerator {
    public void GenerateClass(string className) {
        string code = $@"
using System;

public class {className}
{{
    public int Id {{ get; set; }}
    public string Name {{ get; set; }}

    public {className}()
    {{
        // Default constructor
    }}
}}";

        File.WriteAllText($"{className}.cs", code);
        Console.WriteLine($"Generated class {className} in file {className}.cs");
    }
}

public class Program {
    static void Main() {
        CodeGenerator generator = new();
        generator.GenerateClass("Person");
        Console.WriteLine("Check the generated file.");
    }
}
```

Explanation: This code generator creates a C# class with properties and constructors based on the input name.

Difficulty Rating: 3/5 (Intermediate)

Question 43: Implement a Custom Memory Management Mechanism

Answer Code:

```
public class ObjectPool<T> where T : new() {
    private Queue<T> _pool = new();
    private readonly object _sync = new();

    public TGetObject() {
        lock (_sync) {
            if (_pool.Count > 0) {
                return _pool.Dequeue();
            } else {
                Console.WriteLine("Creating new object.");
                return new T();
            }
        }
    }

    public void ReturnObject(T obj) {
        lock (_sync) {
            _pool.Enqueue(obj);
            Console.WriteLine("Returned object to pool.");
        }
    }
```

```
    public void ClearPool() {
        lock (_sync) {
            _pool.Clear();
            Console.WriteLine("Pool cleared.");
        }
    }
}

public class Program {
    static void Main() {
        ObjectPool<Customer> pool = new();
        Customer[] customers = new Customer[5];

        for (int i = 0; i < customers.Length; i++) {
            customers[i] = pool.GetObject();
        }

        foreach (Customer customer in customers) {
            pool.ReturnObject(customer);
        }

        pool.ClearPool();
    }
}

public class Customer {
    public int Id { get; set; }
    public string Name { get; set; }
}
```

Explanation: This object pool manages a collection of objects, reusing them when possible to reduce memory overhead.

Difficulty Rating: 4/5 (Advanced)

Question 44: Implement a Custom Progress Reporting Mechanism

Answer Code:

```
public interface IProgressReporter {
    void Report(int progress);
}

public class ConsoleProgressReporter : IProgressReporter {
    public void Report(int progress) =>
        Console.WriteLine("Progress: " + progress + "%");
}

public class ProgressManager {
    private IProgressReporter _reporter;

    public ProgressManager(IProgressReporter reporter) =>
        _reporter = reporter ?? throw new ArgumentNullException(nameof(reporter));

    public void SimulateProgress() {
        for (int i = 0; i <= 100; i += 25) {
            Thread.Sleep(300);
            _reporter.Report(i);
        }
    }
}

public class Program {
    static void Main() {
        IProgressReporter reporter = new ConsoleProgressReporter();
        ProgressManager manager = new(reporter);
        manager.SimulateProgress();
    }
}
```

Explanation: This progress reporting mechanism uses an interface to allow different output methods, such as console or GUI.

Difficulty Rating: 3/5 (Intermediate)

Question 45: Implement a Custom Configuration Management

Answer Code:

```
public class ConfigurationManager {
    private Dictionary<string, string> _config = new();

    public void Load(string filePath) {
        if (File.Exists(filePath)) {
            string[] lines = File.ReadAllLines(filePath);
            foreach (string line in lines) {
                if (!string.IsNullOrEmpty(line)) {
                    string[] parts = line.Split(new[] { '=' }, 2);
                    if (parts.Length == 2) {
                        _config[parts[0].Trim()] = parts[1].Trim();
                    }
                }
            }
        }
    }

    public string Get(string key) => _config.TryGetValue(key, out string value) ? value : null;
}

public class Program {
    static void Main() {
        ConfigurationManager config = new();
        string filePath = "config.txt"; // Ensure this file exists with key=value lines

        config.Load(filePath);

        string value = config.Get("LogLevel");
        Console.WriteLine("Log Level: " + (value ?? "Not found"));
    }
}
```

Explanation: This configuration manager reads key-value pairs from a file and allows retrieving values by key.

Difficulty Rating: 3/5 (Intermediate)

Question 46: Implement a Custom Event Bus

Answer Code:

```
public class EventBus {
    private Dictionary<string, List<Action<EventArgs>>> _events = new();

    public void Subscribe(string eventName, Action<EventArgs> handler) {
        if (!_events.ContainsKey(eventName)) {
            _events[eventName] = new List<Action<EventArgs>>();
        }
        _events[eventName].Add(handler);
    }

    public void Unsubscribe(string eventName, Action<EventArgs> handler) {
        if (_events.ContainsKey(eventName)) {
            _events[eventName].Remove(handler);
        }
    }

    public void Publish(string eventName, EventArgs args = null) {
        if (_events.ContainsKey(eventName)) {
            foreach (Action<EventArgs> handler in _events[eventName]) {
                handler?.Invoke(args);
            }
        }
    }
}

public class Program {
    static void Main() {
```

```
        EventBus bus = new();

        // Subscribe
        bus.Subscribe("DataUpdated", args =>
            Console.WriteLine("Data updated event received."));

        // Publish
        bus.Publish("DataUpdated", new EventArgs());

        // Unsubscribe
        bus.Unsubscribe("DataUpdated", args =>
            Console.WriteLine("Unsubscribed handler."));

        // Publish again (no handlers)
        bus.Publish("DataUpdated", new EventArgs());

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

Explanation: This event bus allows subscribing to and publishing events with callbacks.

Difficulty Rating: 3/5 (Intermediate)

Question 47: Implement a Custom File Processing Pipeline

Answer Code:

```
public interface IFileProcessor {
    void Process(string filePath);
}

public class StreamReaderProcessor : IFileProcessor {
    public void Process(string filePath) =>
        File.ReadAllLines(filePath).ToList().ForEach(line =>
            Console.WriteLine("Read: " + line));
}

public class FileWriterProcessor : IFileProcessor {
    public void Process(string filePath) =>
        File.AppendAllLines(filePath, new[] { DateTime.Now.ToString() });
}

public class FilePipeline {
    private List<IFileProcessor> _processors = new();

    public void AddProcessor(IFileProcessor processor) =>
        _processors.Add(processor);

    public void Process(string filePath) {
        foreach (IFileProcessor processor in _processors) {
            processor.Process(filePath);
        }
    }
}

public class Program {
    static void Main() {
        FilePipeline pipeline = new();
        IFileProcessor reader = new StreamReaderProcessor();
        IFileProcessor writer = new FileWriterProcessor();

        pipeline.AddProcessor(reader);
        pipeline.AddProcessor(writer);

        string filePath = "input.txt";
        if (File.Exists(filePath)) {
            pipeline.Process(filePath);
        } else {
            Console.WriteLine("File not found.");
        }
    }
}
```

Explanation: This file processing pipeline executes a series of processors on a given file.

Difficulty Rating: 3/5 (Intermediate)

Question 48: Implement a Custom Memory Leak Detector

Answer Code:

```
public class MemoryLeakDetector {
    private long _startMemory;
    private readonly string _objectName;

    public MemoryLeakDetector(string objectName) {
        _objectName = objectName;
        GC.Collect();
        _startMemory = GetTotalMemory();
    }

    public void CheckForLeak() {
        GC.Collect();
        long currentMemory = GetTotalMemory();
        if (currentMemory > _startMemory) {
            Console.WriteLine($"Potential memory leak detected in {_objectName}.");
        } else {
            Console.WriteLine("No memory leak detected.");
        }
    }

    private long GetTotalMemory() =>
        GC.GetTotalMemory(false);
}

public class Program {
    static void Main() {
        MemoryLeakDetector detector = new("Test Object");

        // Simulate memory usage
        List<byte[]> memoryLeak = new();
        for (int i = 0; i < 1000; i++) {
            memoryLeak.Add(new byte[100000]);
        }

        detector.CheckForLeak();

        // Cleanup
        memoryLeak.Clear();
        GC.Collect();
        detector.CheckForLeak();

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

Explanation: This detector checks for memory leaks by comparing memory usage before and after certain operations.

Difficulty Rating: 3/5 (Intermediate)

Question 49: Implement a Custom Code Coverage Tool

Answer Code:

```
public interface ICodeCoverage {
    void Start();
    void Stop();
}

public class SimpleCodeCoverage : ICodeCoverage {
    private Dictionary<string, int> _hitCount = new();

    public void Start() =>
        AppDomain.CurrentDomain.ReflectionOnlyAssemblyResolve += HandleReflectionAssemblyLoad;
```

```
    public void Stop() {
        AppDomain.CurrentDomain.ReflectionOnlyAssemblyResolve -= HandleReflectionAssemblyLoad;

        foreach (var kvp in _hitCount) {
            Console.WriteLine($"{kvp.Key} was hit {kvp.Value} times.");
        }
    }

    private void HandleReflectionAssemblyLoad(object sender, ResolveEventArgs args) {
        Assembly assembly = args.RequestingAssembly;
        if (assembly == null) return;

        foreach (Module module in assembly.Modules) {
            foreach (Type type in module.GetTypes()) {
                if (type.IsInterface || type.IsAbstract) continue;

                foreach (MethodInfo method in type.GetMethods(BindingFlags.Instance | BindingFlags.Public)) {
                    _hitCount[method.ToString()] =
                        _hitCount.GetValueOrDefault(method.ToString(), 0) + 1;
                }
            }
        }
    }
}

public class Program {
    static void Main() {
        ICodeCoverage coverage = new SimpleCodeCoverage();
        coverage.Start();

        // Simulate code execution
        for (int i = 0; i < 5; i++) {
            Console.WriteLine("Hello, World!");
        }

        coverage.Stop();
    }
}
```

Explanation: This simple code coverage tool tracks how many times methods are called during execution.

Difficulty Rating: 4/5 (Advanced)

Question 50: Implement a Custom Web Crawler

Answer Code:

```
public class WebCrawler {
    private readonly HashSet<string> _visitedUrls = new();
    private int _depthLimit;

    public WebCrawler(int depthLimit) => _depthLimit = depthLimit;

    public async Task CrawlAsync(string url, int currentDepth = 0) {
        if (currentDepth > _depthLimit || !_visitedUrls.Add(url)) return;

        try {
            Console.WriteLine("Crawling: " + url);
            var html = await GetHtmlAsync(url);

            foreach (string link in ExtractLinks(html)) {
                await CrawlAsync(link, currentDepth + 1);
            }
        } catch (Exception ex) {
            Console.WriteLine("Error crawling " + url + ": " + ex.Message);
        }
    }

    private async Task<string> GetHtmlAsync(string url) {
        using HttpClient client = new();
        HttpResponseMessage response = await client.GetAsync(url);
        if (response.IsSuccessStatusCode) {
            return await response.Content.ReadAsStringAsync();
```

```
        }
        throw new Exception("Failed to fetch URL.");
    }

    private IEnumerable<string> ExtractLinks(string html) {
        var regex = new Regex(@"<a\s+href=[""'](.[^""']+)[""']>", RegexOptions.IgnoreCase);
        return regex.Matches(html).Cast<Match>().Select(m => m.Groups[1].Value);
    }
}

public class Program {
    static async Task Main() {
        WebCrawler crawler = new(2); // Depth limit of 2
        await crawler.CrawlAsync("https://example.com");
    }
}
```

Explanation: This web crawler asynchronously explores websites up to a specified depth using HTML parsing.

Difficulty Rating: 4/5 (Advanced)

Question 51: Implement a Custom Search Engine

Answer Code:

```
public class Document {
    public string Content { get; set; }
}

public interface ISearchEngine {
    List<Document> Search(string query);
}

public class SimpleSearchEngine : ISearchEngine {
    private readonly List<Document> _documents = new();

    public void AddDocument(Document document) =>
        _documents.Add(document);

    public List<Document> Search(string query) {
        return _documents.Where(d =>
            d.Content.Contains(query, StringComparison.OrdinalIgnoreCase))
                         .ToList();
    }
}

public class Program {
    static void Main() {
        ISearchEngine engine = new SimpleSearchEngine();
        Document doc1 = new() { Content = "Hello, World!" };
        Document doc2 = new() { Content = "Welcome to C#" };
        Document doc3 = new() { Content = "C# Programming" };

        engine.AddDocument(doc1);
        engine.AddDocument(doc2);
        engine.AddDocument(doc3);

        List<Document> results = engine.Search("C#");
        foreach (Document doc in results) {
            Console.WriteLine(doc.Content);
        }
    }
}
```

Explanation: This simple search engine indexes documents and allows searching by keyword.

Difficulty Rating: 3/5 (Intermediate)

Question 52: Implement a Custom Machine Learning Model

Answer Code:

```
public class LinearRegressor {
    private double _slope;
    private double _intercept;

    public void Train(double[][] inputs, double[] outputs) {
        double sumX = 0, sumY = 0, sumXY = 0, sumXX = 0;
        int n = inputs.Length;

        for (int i = 0; i < n; i++) {
            sumX += inputs[i][0];
            sumY += outputs[i];
            sumXY += inputs[i][0] * outputs[i];
            sumXX += Math.Pow(inputs[i][0], 2);
        }

        _slope = (n * sumXY - sumX * sumY) / (n * sumXX - Math.Pow(sumX, 2));
        _intercept = (sumY / n) - (_slope * sumX / n);
    }

    public double Predict(double x) => _intercept + (_slope * x);
}

public class Program {
    static void Main() {
        double[][] inputs = { new[] { 1 }, new[] { 2 }, new[] { 3 }, new[] { 4 }, new[] {5} };
        double[] outputs = { 2, 3.5, 4, 6, 7 };

        LinearRegressor model = new();
        model.Train(inputs, outputs);

        double prediction = model.Predict(6);
        Console.WriteLine("Predicted value for x=6: " + prediction);
    }
}
```

Explanation: This linear regression model trains on input-output pairs and predicts outputs based on a new input.

Difficulty Rating: 4/5 (Advanced)

Question 53: Implement a Custom Natural Language Processing Tool

Answer Code:

```
public class TextAnalyzer {
    public string[] Analyze(string text) {
        var words = Regex.Matches(text, @"\b\w+\b")
                         .Cast<Match>()
                         .Select(m => m.Value)
                         .ToArray();

        string[] stemmed = new string[words.Length];
        for (int i = 0; i < words.Length; i++) {
            stemmed[i] = StemWord(words[i]);
        }

        return stemmed;
    }

    private string StemWord(string word) {
        // Simple stemming by removing common suffixes
        string[] suffixes = new[] { "ing", "ed", "ly" };
        foreach (string suffix in suffixes) {
            if (word.EndsWith(suffix)) {
                return word.Substring(0, word.Length - suffix.Length);
            }
        }
        return word;
    }
}

public class Program {
    static void Main() {
        TextAnalyzer analyzer = new();
```

```
        string text = "Processing information quickly is important.";

        string[] results = analyzer.Analyze(text);
        foreach (string word in results) {
            Console.WriteLine(word);
        }
    }
}
```

Explanation: This text analyzer tokenizes input and performs basic stemming on each word.

Difficulty Rating: 3/5 (Intermediate)

Question 54: Implement a Custom Image Processing Library

Answer Code:

```
public class ImageProcessor {
    public static Bitmap ResizeImage(Bitmap image, int width, int height) {
        using (Bitmap resized = new(width, height)) {
            using (Graphics g = Graphics.FromImage(resized)) {
                g.InterpolationMode = InterpolationMode.HighQualityBicubic;
                g.DrawImage(image, 0, 0, width, height);
            }
            return resized;
        }
    }

    public static Bitmap GrayscaleBitmap(Bitmap image) {
        for (int y = 0; y < image.Height; y++) {
            for (int x = 0; x < image.Width; x++) {
                Color c = image.GetPixel(x, y);
                int gray = (c.R + c.G + c.B) / 3;
                image.SetPixel(x, y, Color.FromArgb(c.A, gray, gray, gray));
            }
        }
        return image;
    }
}

public class Program {
    static void Main() {
        try {
            using Bitmap image = new("test.jpg");

            // Resize
            Bitmap resizedImage = ImageProcessor.ResizeImage(image, 200, 200);
            resizedImage.Save("resized.jpg", ImageFormat.Jpeg);

            // Grayscale
            Bitmap grayscaleImage = ImageProcessor.GrabarScaleBitmap(image);
            grayscaleImage.Save("grayscale.jpg", ImageFormat.Jpeg);

            Console.WriteLine("Processing completed.");
        } catch (Exception ex) {
            Console.WriteLine("Error: " + ex.Message);
        }
    }
}
```

Explanation: This image processing library includes methods for resizing and converting images to grayscale.

Difficulty Rating: 4/5 (Advanced)

Question 55: Implement a Custom Audio Processing Module

Answer Code:

```
public class AudioPlayer {
    public async Task PlayAsync(string filePath) {
        using SoundPlayer player = new(filePath);
```

```
        await Task.Run(() =>
            Console.WriteLine("Playing audio..."));
        player.Play();
    }
}

public class Program {
    static async Task Main() {
        AudioPlayer player = new();
        await player.PlayAsync("test.wav");
        Console.WriteLine("Audio played.");
    }
}
```

Explanation: This audio player uses the `SoundPlayer` class to play audio files asynchronously.

Difficulty Rating: 3/5 (Intermediate)

Question 56: Implement a Custom Video Processing Tool

Answer Code:

```
public class VideoProcessor {
    public async Task ConvertVideoAsync(string inputPath, string outputPath) {
        await Process.Start("ffmpeg", $"-i {inputPath} {outputPath}")
                .ConfigureAwait(false);
    }
}

public class Program {
    static async Task Main() {
        VideoProcessor processor = new();
        await processor.ConvertVideoAsync("input.mp4", "output.avi");
        Console.WriteLine("Conversion completed.");
    }
}
```

Explanation: This video processor uses FFmpeg to convert video formats asynchronously.

Difficulty Rating: 3/5 (Intermediate)

Question 57: Implement a Custom Bioinformatics Tool

Answer Code:

```
public class DNASequence {
    private string _sequence;

    public DNASequence(string sequence) =>
        _sequence = ValidateSequence(sequence);

    private string ValidateSequence(string sequence) {
        foreach (char c in sequence.ToUpper()) {
            if (!"ATCG".Contains(c)) {
                throw new ArgumentException("Invalid DNA sequence.");
            }
        }
        return sequence;
    }

    public string GetComplement() {
        char[] complement = new char[_sequence.Length];
        for (int i = 0; i < _sequence.Length; i++) {
            switch (_sequence[i]) {
                case 'A':
                    complement[i] = 'T';
                    break;
                case 'T':
                    complement[i] = 'A';
                    break;
                case 'C':
```

```
                    complement[i] = 'G';
                    break;
                case 'G':
                    complement[i] = 'C';
                    break;
                default:
                    throw new InvalidOperationException();
            }
        }
        return new string(complement);
    }
}

public class Program {
    static void Main() {
        DNASequence sequence = new("ATGC");
        string complement = sequence.GetComplement();
        Console.WriteLine("Original: " + sequence.ToString());
        Console.WriteLine("Complement: " + complement);
    }
}
```

Explanation: This tool processes DNA sequences, ensuring validity and computing their complements.

Difficulty Rating: 4/5 (Advanced)

Question 58: Implement a Custom Weather Analysis Tool

Answer Code:

```
public class WeatherAnalyzer {
    public double CalculateAverageTemperature(List<double> temperatures) =>
        temperatures.Average();

    public double FindMaximumTemperature(List<double> temperatures) =>
        temperatures.Max();
}

public class Program {
    static void Main() {
        List<double> temps = new() { 23.5, 24.1, 22.8, 25.3 };
        WeatherAnalyzer analyzer = new();

        double avg = analyzer.CalculateAverageTemperature(temps);
        double max = analyzer.FindMaximumTemperature(temps);

        Console.WriteLine("Average: " + avg);
        Console.WriteLine("Maximum: " + max);
    }
}
```

Explanation: This tool calculates average and maximum temperatures from a list.

Difficulty Rating: 2/5 (Beginner)

Question 59: Implement a Custom Social Media Analysis Tool

Answer Code:

```
public class TweetAnalyzer {
    public int CountHashtags(string tweetText) {
        if (string.IsNullOrEmpty(tweetText)) return 0;

        int count = 0;
        bool inHashtag = false;
        foreach (char c in tweetText) {
            if (c == '#') {
                inHashtag = true;
                count++;
            } else if (!char.IsLetterOrDigit(c)) {
                inHashtag = false;
```

```
            }
        }
        return count;
    }

    public List<string> ExtractHashtags(string tweetText) {
        var hashtags = new List<string>();
        string currentHashtag = null;

        for (int i = 0; i < tweetText.Length; i++) {
            if (tweetText[i] == '#') {
                currentHashtag = String.Empty;
                for (int j = i + 1; j < tweetText.Length; j++) {
                    if (char.IsLetterOrDigit(tweetText[j])) {
                        currentHashtag += tweetText[j];
                    } else {
                        break;
                    }
                }

                if (currentHashtag != null && currentHashtag.Length > 0) {
                    hashtags.Add(currentHashtag);
                }
                i += currentHashtag?.Length ?? 0;
            }
        }

        return hashtags;
    }
}

public class Program {
    static void Main() {
        string tweet = "#CSharp is awesome! #Programming #Code";
        TweetAnalyzer analyzer = new();

        int hashtagCount = analyzer.CountHashtags(tweet);
        Console.WriteLine("Total hashtags: " + hashtagCount);

        List<string> extracted = analyzer.ExtractHashtags(tweet);
        Console.WriteLine("Extracted hashtags:");
        foreach (string hashtag in extracted) {
            Console.WriteLine("#" + hashtag);
        }
    }
}
```

Explanation: This tool analyzes tweets to count and extract hashtags.

Difficulty Rating: 3/5 (Intermediate)

Question 60: Implement a Custom Recommendation Engine

Answer Code:

```
public class Recommender {
    private Dictionary<string, List<string>> _users = new();

    public void AddUser(string userId, params string[] items) =>
        _users[userId] = new List<string>(items);

    public List<string> GetRecommendations(string userId) {
        if (!_users.ContainsKey(userId)) return new List<string>();

        var recommendations = new Dictionary<string, int>();
        foreach (string user in _users.Keys) {
            if (user == userId) continue;

            int[] common = FindCommonItems(userId, user);
            if (common.Length > 0) {
                foreach (string item in _users[user]) {
                    if (!_users[userId].Contains(item)) {
                        recommendations[item] =
                            recommendations.GetValueOrDefault(item, 0) + common.Length;
```

```
                    }
                }
            }
        }

        return recommendations.OrderByDescending(kvp => kvp.Value)
                              .Select(kvp => kvp.Key)
                              .ToList();
    }

    private int[] FindCommonItems(string user1, string user2) {
        return _users[user1].Intersect(_users[user2]).ToArray();
    }
}

public class Program {
    static void Main() {
        Recommender recommender = new();

        recommender.AddUser("User1", "A", "B", "C");
        recommender.AddUser("User2", "B", "C", "D");
        recommender.AddUser("User3", "A", "D", "E");

        List<string> recommendations = recommender.GetRecommendations("User1");
        Console.WriteLine("Recommendations for User1:");
        foreach (string item in recommendations) {
            Console.WriteLine(item);
        }
    }
}
```

Explanation: This recommendation engine suggests items based on common interests among users.

Difficulty Rating: 4/5 (Advanced)

Question 61: Implement a Custom Cryptocurrency Tracker

Answer Code:

```
public interface ICryptoTracker {
    void Track(string symbol);
    void StopTracking(string symbol);
}

public class SimpleCryptoTracker : ICryptoTracker {
    private Dictionary<string, bool> _tracking = new();

    public void Track(string symbol) =>
        _tracking[symbol] = true;

    public void StopTracking(string symbol) =>
        _tracking.Remove(symbol);

    public List<string> GetActiveSymbols() =>
        _tracking.Where(kvp => kvp.Value)
                 .Select(kvp => kvp.Key)
                 .ToList();
}

public class Program {
    static void Main() {
        ICryptoTracker tracker = new SimpleCryptoTracker();
        tracker.Track("BTC");
        tracker.Track("ETH");

        List<string> active = tracker.GetActiveSymbols();
        Console.WriteLine("Tracking:");
        foreach (string symbol in active) {
            Console.WriteLine(symbol);
        }

        tracker.StopTracking("ETH");
        active = tracker.GetActiveSymbols();
        Console.WriteLine("\nNow tracking:");
```

```
        foreach (string symbol in active) {
            Console.WriteLine(symbol);
        }
    }
}
```

Explanation: This tracker manages a list of cryptocurrencies being tracked.

Difficulty Rating: 3/5 (Intermediate)

Question 62: Implement a Custom Quantum Computing Simulator

Answer Code:

```
public class QuantumBit {
    private double _alpha;
    private double _beta;

    public QuantumBit(double alpha, double beta) {
        if (Math.Abs(alpha * alpha + beta * beta - 1) > 1e-9) {
            throw new ArgumentException("Alpha and beta must satisfy alpha² + beta² = 1.");
        }
        _alpha = alpha;
        _beta = beta;
    }

    public double Alpha => _alpha;
    public double Beta => _beta;

    public void ApplyHadamard() {
        double newAlpha = (_alpha + _beta) / Math.Sqrt(2);
        double newBeta = (_alpha - _beta) / Math.Sqrt(2);
        _alpha = newAlpha;
        _beta = newBeta;
    }

    public override string ToString() =>
        $"|ψ⟩ = {_alpha.ToString("F2")} |0⟩ + {_beta.ToString("F2")} |1⟩";
}

public class Program {
    static void Main() {
        QuantumBit qubit = new(1.0 / Math.Sqrt(2), 1.0 / Math.Sqrt(2));
        Console.WriteLine("Initial state:");
        Console.WriteLine(qubit);

        qubit.ApplyHadamard();
        Console.WriteLine("\nAfter Hadamard:");
        Console.WriteLine(qubit);
    }
}
```

Explanation: This simulator models a qubit and performs a Hadamard gate operation.

Difficulty Rating: 4/5 (Advanced)

Question 63: Implement a Custom Neural Network

Answer Code:

```
public class Neuron {
    private double[] _weights;
    private double _bias;

    public Neuron(int inputCount) {
        _weights = new double[inputCount];
        Random random = new();
        for (int i = 0; i < inputCount; i++) {
            _weights[i] = random.NextDouble() * 2 - 1;
        }
```

```
        _bias = random.NextDouble() * 2 - 1;
    }

    public double Sigmoid(double x) =>
        1 / (1 + Math.Exp(-x));

    public double FeedForward(double[] inputs) {
        if (inputs.Length != _weights.Length) {
            throw new ArgumentException("Mismatched input and weights dimensions.");
        }

        double sum = 0;
        for (int i = 0; i < inputs.Length; i++) {
            sum += inputs[i] * _weights[i];
        }
        return Sigmoid(sum + _bias);
    }

    public void Train(double[] inputs, double target, double learningRate) {
        double output = FeedForward(inputs);
        double error = target - output;
        for (int i = 0; i < inputs.Length; i++) {
            _weights[i] += learningRate * error * output * (1 - output) * inputs[i];
        }
        _bias += learningRate * error * output * (1 - output);
    }
}

public class Program {
    static void Main() {
        Neuron neuron = new(2);
        double[][] inputs = { new[] { 0, 0 }, new[] { 0, 1 },
                              new[] { 1, 0 }, new[] { 1, 1 } };
        double[] targets = { 0, 1, 1, 1 };

        for (int epoch = 0; epoch < 1000; epoch++) {
            foreach (var pair in inputs.Zip(targets)) {
                neuron.Train(pair.First, pair.Second, 0.1);
            }
        }

        Console.WriteLine("Predictions after training:");
        foreach (double[] input in inputs) {
            double output = neuron.FeedForward(input);
            Console.WriteLine(string.Join(" ", input.Select(x => x.ToString("F2"))) +
                              " → " + output.ToString("F4"));
        }
    }
}
```

Explanation: This neural network simulates a neuron with training capabilities using the sigmoid activation function.

Difficulty Rating: 4/5 (Advanced)

Question 64: Implement a Custom Natural Language Generator

Answer Code:

```
public class NLGEngine {
    public string GenerateIntroduction(string name, int age) =>
        $"Hello! My name is {name}, and I am {age} years old.";

    public string GenerateConclusion(string topic) =>
        $"In conclusion, {topic} is an interesting subject worth exploring further.";
}

public class Program {
    static void Main() {
        NLGEngine engine = new();
        string intro = engine.GenerateIntroduction("Alice", 30);
        string conclusion = engine.GenerateConclusion("Artificial Intelligence");

        Console.WriteLine(intro);
        Console.WriteLine(conclusion);
```

```
    }
}
```

Explanation: This tool generates natural language text based on input parameters.

Difficulty Rating: 2/5 (Beginner)

Question 65: Implement a Custom Sentiment Analysis Tool

Answer Code:

```
public class SentimentAnalyzer {
    public enum Sentiment { Positive, Neutral, Negative }

    public Sentiment Analyze(string text) {
        string[] positiveWords = { "happy", "good", "great" };
        string[] negativeWords = { "sad", "bad", "terrible" };

        int positiveCount = 0, negativeCount = 0;
        string[] words = text.Split(new[] { ' ', ',', '.', '!', '?' }, StringSplitOptions.RemoveEmptyEntries);

        foreach (string word in words) {
            if (positiveWords.Contains(word.ToLower())) positiveCount++;
            else if (negativeWords.Contains(word.ToLower())) negativeCount++;
        }

        if (positiveCount > negativeCount) return Sentiment.Positive;
        else if (negativeCount > positiveCount) return Sentiment.Negative;
        else return Sentiment.Neutral;
    }
}

public class Program {
    static void Main() {
        string text = "I had a great day but the weather was bad.";
        SentimentAnalyzer analyzer = new();
        Sentiment sentiment = analyzer.Analyze(text);

        Console.WriteLine("Sentiment: " + sentiment);
    }
}
```

Explanation: This tool analyzes text to determine if it's positive, neutral, or negative based on predefined keywords.

Difficulty Rating: 3/5 (Intermediate)

Question 66: Implement a Custom Database Migration Tool

Answer Code:

```
public class DbMigrator {
    public void Migrate(string sourceConnectionString, string destinationConnectionString) {
        using SqlConnection source = new(sourceConnectionString);
        using SqlConnection dest = new(destinationConnectionString);

        string[] tables = GetTables(source);
        foreach (string table in tables) {
            CopyTable(source, dest, table);
        }
    }

    private string[] GetTables(SqlConnection connection) {
        List<string> tables = new();
        using SqlCommand command = new("SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES", connection);
        using SqlDataReader reader = command.ExecuteReader();
        while (reader.Read()) {
            tables.Add(reader.GetString(0));
        }
        return tables.ToArray();
    }
```

```
    private void CopyTable(SqlConnection source, SqlConnection dest, string tableName) {
        string query = $"SELECT * FROM {tableName}";
        using SqlCommand sourceCommand = new(query, source);
        using SqlDataReader reader = sourceCommand.ExecuteReader();

        string insertQuery = $"INSERT INTO {tableName} VALUES ({{0}})";
        using SqlCommand destCommand = new(insertQuery, dest);

        while (reader.Read()) {
            object[] values = new object[reader.FieldCount];
            reader.GetValues(values);
            destCommand.Parameters.Clear();
            for (int i = 0; i < values.Length; i++) {
                destCommand.Parameters.AddWithValue($"@p{i}", values[i]);
            }
            destCommand.ExecuteNonQuery();
        }
    }
}

public class Program {
    static void Main() {
        string source = "source_connection_string";
        string destination = "destination_connection_string";

        DbMigrator migrator = new();
        migrator.Migrate(source, destination);
    }
}
```

Explanation: This tool migrates data from one database to another by copying tables.

Difficulty Rating: 4/5 (Advanced)

Question 67: Implement a Custom Machine Learning Pipeline

Answer Code:

```
public class MLPipeline {
    public void Train(string dataPath, string modelPath) {
        // Simulate training
        Console.WriteLine("Training model...");
        Thread.Sleep(2000);
        // Save model
        File.WriteAllText(modelPath, "Model data");
    }

    public void Predict(string modelPath, string inputData) {
        // Simulate prediction
        Console.WriteLine("Running inference...");
        Thread.Sleep(1000);
        // Output prediction
        Console.WriteLine("Prediction: " + GenerateRandomResult());
    }

    private string GenerateRandomResult() {
        Random random = new();
        return random.Next(0, 2) switch {
            0 => "Positive",
            1 => "Negative",
            _ => throw new InvalidOperationException()
        };
    }
}

public class Program {
    static void Main() {
        MLPipeline pipeline = new();
        string dataPath = "data.csv";
        string modelPath = "model.bin";

        pipeline.Train(dataPath, modelPath);
        pipeline.Predict(modelPath, "input_data");
    }
```

```
}
```

Explanation: This pipeline simulates training a model and making predictions.

Difficulty Rating: 3/5 (Intermediate)

Question 68: Implement a Custom Robotic Process Automation Tool

Answer Code:

```
public class RPAEngine {
    public void Automate(string workflow) {
        switch (workflow.ToLower()) {
            case "login":
                PerformLogin();
                break;
            case "logout":
                PerformLogout();
                break;
            default:
                throw new ArgumentException("Invalid workflow.");
        }
    }

    private void PerformLogin() {
        Console.WriteLine("Attempting login...");
        // Simulate login actions
        Thread.Sleep(1000);
        Console.WriteLine("Login successful.");
    }

    private void PerformLogout() {
        Console.WriteLine("Attempting logout...");
        // Simulate logout actions
        Thread.Sleep(1000);
        Console.WriteLine("Logout successful.");
    }
}

public class Program {
    static void Main() {
        RPAEngine engine = new();
        engine.Automate("login");
        engine.Automate("logout");
    }
}
```

Explanation: This tool automates login and logout workflows.

Difficulty Rating: 3/5 (Intermediate)

Question 69: Implement a Custom Game AI

Answer Code:

```
public interface IGameAI {
    int MakeMove(int[] board);
}

public class TicTacToeAI : IGameAI {
    public int MakeMove(int[] board) {
        // Simulate a simple AI strategy
        for (int i = 0; i < board.Length; i++) {
            if (board[i] == 0) return i;
        }
        throw new InvalidOperationException("Board full.");
    }
}

public class Program {
    static void Main() {
```

```
        IGameAI ai = new TicTacToeAI();
        int[] board = { 1, -1, 0, 0, 1, 0, 0, 0, -1 };
        int move = ai.MakeMove(board);
        Console.WriteLine("AI moves to index: " + move);
    }
}
```

Explanation: This AI makes a move in Tic-Tac-Toe based on the first available spot.

Difficulty Rating: 3/5 (Intermediate)

Question 70: Implement a Custom Bug Tracking System

Answer Code:

```
public class Bug {
    public int Id { get; set; }
    public string Title { get; set; }
    public string Status { get; set; }
}

public class BugTracker {
    private List<Bug> _bugs = new();

    public void AddBug(string title) =>
        _bugs.Add(new Bug { Id = _bugs.Count + 1, Title = title, Status = "Open" });

    public void CloseBug(int id) {
        var bug = _bugs.Find(b => b.Id == id);
        if (bug != null) bug.Status = "Closed";
    }

    public List<Bug> GetBugs(string status = null) =>
        _bugs.Where(b => string.IsNullOrEmpty(status) || b.Status == status)
             .ToList();
}

public class Program {
    static void Main() {
        BugTracker tracker = new();
        tracker.AddBug("Login issue");
        tracker.AddBug("UI bug");

        List<Bug> openBugs = tracker.GetBugs("Open");
        Console.WriteLine("Open bugs:");
        foreach (Bug bug in openBugs) {
            Console.WriteLine($"#{bug.Id}: {bug.Title}");
        }

        tracker.CloseBug(1);
        List<Bug> closedBugs = tracker.GetBugs("Closed");
        Console.WriteLine("\nClosed bugs:");
        foreach (Bug bug in closedBugs) {
            Console.WriteLine($"#{bug.Id}: {bug.Title}");
        }
    }
}
```

Explanation: This tool tracks bugs with their statuses and allows adding, closing, and retrieving them.

Difficulty Rating: 3/5 (Intermediate)

Question 71: Implement a Custom Supply Chain Management Tool

Answer Code:

```
public class Inventory {
    private Dictionary<string, int> _items = new();

    public void AddItem(string name, int quantity) =>
        _items[name] += quantity;
```

```
    public void RemoveItem(string name, int quantity) =>
        _items[name] -= quantity;

    public int GetQuantity(string name) =>
        _items.TryGetValue(name, out int qty) ? qty : 0;

    public void PrintReport() {
        Console.WriteLine("Inventory Report:");
        foreach (var kvp in _items) {
            Console.WriteLine($"{kvp.Key}: {kvp.Value}");
        }
    }
}

public class Program {
    static void Main() {
        Inventory inv = new();
        inv.AddItem("Laptop", 10);
        inv.AddItem("Phone", 5);

        Console.WriteLine(inv.GetQuantity("Laptop"));
        inv.RemoveItem("Phone", 2);

        inv.PrintReport();
    }
}
```

Explanation: This tool manages inventory with add, remove, and report features.

Difficulty Rating: 3/5 (Intermediate)

Question 72: Implement a Custom Risk Management Tool

Answer Code:

```
public class RiskAssessor {
    public string AssessRisk(double probability, double impact) {
        if (probability >= 0.8 && impact >= 0.8) return "High";
        else if (probability >= 0.5 && impact >= 0.5) return "Medium";
        else return "Low";
    }
}

public class Program {
    static void Main() {
        RiskAssessor assessor = new();
        string riskLevel = assessor.AssessRisk(0.7, 0.6);
        Console.WriteLine("Risk Level: " + riskLevel);
    }
}
```

Explanation: This tool assesses risk based on probability and impact scores.

Difficulty Rating: 2/5 (Beginner)

Question 73: Implement a Custom Compliance Management System

Answer Code:

```
public class Policy {
    public string Name { get; set; }
    public DateTime ExpiryDate { get; set; }
}

public class ComplianceChecker {
    private List<Policy> _policies = new();

    public void AddPolicy(Policy policy) =>
        _policies.Add(policy);
```

```
    public bool IsCompliant() {
        foreach (Policy policy in _policies) {
            if (policy.ExpiryDate < DateTime.Now) return false;
        }
        return true;
    }

    public List<Policy> GetExpiredPolicies() =>
        _policies.Where(p => p.ExpiryDate < DateTime.Now)
                 .ToList();
}

public class Program {
    static void Main() {
        ComplianceChecker checker = new();

        checker.AddPolicy(new Policy { Name = "Data Protection", ExpiryDate = DateTime.Now.AddDays(-1) });
        checker.AddPolicy(new Policy { Name = "Access Control", ExpiryDate = DateTime.Now.AddDays(30) });

        bool compliant = checker.IsCompliant();
        Console.WriteLine("System is compliant: " + compliant);

        List<Policy> expired = checker.GetExpiredPolicies();
        Console.WriteLine("\nExpired policies:");
        foreach (Policy policy in expired) {
            Console.WriteLine(policy.Name);
        }
    }
}
```

Explanation: This system tracks policy expirations and checks compliance.

Difficulty Rating: 3/5 (Intermediate)

Question 74: Implement a Custom Customer Relationship Management Tool

Answer Code:

```
public class Customer {
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime LastContacted { get; set; }
}

public class CRM {
    private List<Customer> _customers = new();

    public void AddCustomer(string name) =>
        _customers.Add(new Customer { Id = _customers.Count + 1, Name = name, LastContacted = DateTime.Now });

    public void UpdateLastContact(int id) {
        var customer = _customers.Find(c => c.Id == id);
        if (customer != null) customer.LastContacted = DateTime.Now;
    }

    public List<Customer> GetInactiveCustomers(TimeSpan threshold) =>
        _customers.Where(c => DateTime.Now - c.LastContacted > threshold)
                  .ToList();
}

public class Program {
    static void Main() {
        CRM crm = new();
        crm.AddCustomer("Alice");
        crm.AddCustomer("Bob");

        Thread.Sleep(1000);
        crm.UpdateLastContact(2);

        List<Customer> inactive = crm.GetInactiveCustomers(TimeSpan.FromMinutes(1));
        Console.WriteLine("Inactive customers:");
        foreach (Customer c in inactive) {
            Console.WriteLine(c.Name);
        }
```

```
    }
}
```

Explanation: This tool manages customer relationships with add, update, and query features.

Difficulty Rating: 3/5 (Intermediate)

Question 75: Implement a Custom Project Management Tool

Answer Code:

```
public class Task {
    public int Id { get; set; }
    public string Title { get; set; }
    public bool IsComplete { get; set; }
}

public class ProjectManager {
    private List<Task> _tasks = new();

    public void AddTask(string title) =>
        _tasks.Add(new Task { Id = _tasks.Count + 1, Title = title });

    public void MarkComplete(int id) {
        var task = _tasks.Find(t => t.Id == id);
        if (task != null) task.IsComplete = true;
    }

    public List<Task> GetIncompleteTasks() =>
        _tasks.Where(t => !t.IsComplete)
              .ToList();
}

public class Program {
    static void Main() {
        ProjectManager pm = new();
        pm.AddTask("Design UI");
        pm.AddTask("Write Code");

        pm.MarkComplete(1);

        List<Task> incomplete = pm.GetIncompleteTasks();
        Console.WriteLine("Incomplete tasks:");
        foreach (Task task in incomplete) {
            Console.WriteLine(task.Title);
        }
    }
}
```

Explanation: This tool manages tasks with add, complete, and retrieval features.

Difficulty Rating: 3/5 (Intermediate)

Question 76: Implement a Custom Email Marketing Tool

Answer Code:

```
public class Campaign {
    public string Name { get; set; }
    public int Sent { get; set; }
    public int Opened { get; set; }
}

public class EmailMarketingTool {
    private List<Campaign> _campaigns = new();

    public void CreateCampaign(string name) =>
        _campaigns.Add(new Campaign { Name = name, Sent = 0, Opened = 0 });

    public void SendEmails(int campaignId, int count) {
        var campaign = _campaigns.Find(c => c.Name == "Campaign" + campaignId);
```

```
        if (campaign != null) {
            campaign.Sent += count;
            int opened = new Random().Next(0, count + 1);
            campaign.Opened += opened;
        }
    }

    public void Report(int campaignId) {
        var campaign = _campaigns.Find(c => c.Name == "Campaign" + campaignId);
        if (campaign != null) {
            Console.WriteLine($"Campaign {campaign.Name}:");
            Console.WriteLine("Emails Sent: " + campaign.Sent);
            Console.WriteLine("Emails Opened: " + campaign.Opened);
        } else {
            Console.WriteLine("Campaign not found.");
        }
    }
}

public class Program {
    static void Main() {
        EmailMarketingTool tool = new();
        tool.CreateCampaign("Campaign1");
        tool.SendEmails(1, 100);

        tool.Report(1);
    }
}
```

Explanation: This tool manages email campaigns, tracks sent and opened emails.

Difficulty Rating: 3/5 (Intermediate)

Question 77: Implement a Custom Business Intelligence Tool

Answer Code:

```
public class SalesData {
    public string Region { get; set; }
    public int Revenue { get; set; }
}

public interface IBITool {
    void AddData(SalesData data);
    int GetTotalRevenue();
}

public class BusinessIntelligenceTool : IBITool {
    private List<SalesData> _data = new();

    public void AddData(SalesData data) =>
        _data.Add(data);

    public int GetTotalRevenue() =>
        _data.Sum(d => d.Revenue);
}

public class Program {
    static void Main() {
        IBITool tool = new BusinessIntelligenceTool();

        tool.AddData(new SalesData { Region = "North", Revenue = 1000 });
        tool.AddData(new SalesData { Region = "South", Revenue = 1500 });

        int total = tool.GetTotalRevenue();
        Console.WriteLine("Total Revenue: $" + total);
    }
}
```

Explanation: This tool aggregates sales data to compute total revenue.

Difficulty Rating: 3/5 (Intermediate)

Question 78: Implement a Custom Customer Feedback System

Answer Code:

```
public class Review {
    public int Id { get; set; }
    public string Text { get; set; }
}

public class FeedbackSystem {
    private List<Review> _reviews = new();

    public void SubmitReview(string text) =>
        _reviews.Add(new Review { Id = _reviews.Count + 1, Text = text });

    public List<Review> GetRecentReviews(int count) =>
        _reviews.TakeLast(count).ToList();
}

public class Program {
    static void Main() {
        FeedbackSystem system = new();

        system.SubmitReview("Great service!");
        system.SubmitReview("Need improvement.");

        List<Review> recent = system.GetRecentReviews(2);
        foreach (Review rev in recent) {
            Console.WriteLine(rev.Text);
        }
    }
}
```

Explanation: This system collects and retrieves customer reviews.

Difficulty Rating: 3/5 (Intermediate)

Question 79: Implement a Custom Human Resources Management Tool

Answer Code:

```
public class Employee {
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Salary { get; set; }
}

public class HRManager {
    private List<Employee> _employees = new();

    public void AddEmployee(string name, decimal salary) =>
        _employees.Add(new Employee { Id = _employees.Count + 1, Name = name, Salary = salary });

    public decimal GetTotalSalary() =>
        _employees.Sum(e => e.Salary);
}

public class Program {
    static void Main() {
        HRManager hr = new();

        hr.AddEmployee("Alice", 50000m);
        hr.AddEmployee("Bob", 60000m);

        decimal total = hr.GetTotalSalary();
        Console.WriteLine("Total Salary: $" + total);
    }
}
```

Explanation: This tool manages employee data and calculates total salaries.

Difficulty Rating: 3/5 (Intermediate)

Question 80: Implement a Custom Executive Dashboard

Answer Code:

```
public class Metric {
    public string Name { get; set; }
    public double Value { get; set; }
}

public class Dashboard {
    private List<Metric> _metrics = new();

    public void AddMetric(string name, double value) =>
        _metrics.Add(new Metric { Name = name, Value = value });

    public void DisplayMetrics() {
        Console.WriteLine("Dashboard Metrics:");
        foreach (Metric m in _metrics) {
            Console.WriteLine($"{m.Name}: {m.Value:N2}");
        }
    }
}

public class Program {
    static void Main() {
        Dashboard dashboard = new();

        dashboard.AddMetric("Revenue", 100000);
        dashboard.AddMetric("Active Users", 5000);

        dashboard.DisplayMetrics();
    }
}
```

Explanation: This dashboard displays business metrics in a formatted manner.

Difficulty Rating: 2/5 (Beginner)

Question 81: Implement a Custom Energy Management System

Answer Code:

```
public class EnergyMonitor {
    public void TrackUsage(string device, double usage) {
        Console.WriteLine($"{device} energy usage: {usage:F2} kWh");
    }
}

public class Program {
    static void Main() {
        EnergyMonitor monitor = new();
        monitor.TrackUsage("AC", 5.2);
        monitor.TrackUsage("Heater", 3.8);
    }
}
```

Explanation: This system tracks energy usage for devices.

Difficulty Rating: 2/5 (Beginner)

Question 82: Implement a Custom Water Management Tool

Answer Code:

```
public class WaterLevelSensor {
    public bool IsWaterLow { get; private set; }
    public bool IsWaterHigh { get; private set; }
```

```
    public void UpdateLevel(double level) {
        if (level < 20) {
            IsWaterLow = true;
            IsWaterHigh = false;
        } else if (level > 80) {
            IsWaterLow = false;
            IsWaterHigh = true;
        } else {
            IsWaterLow = false;
            IsWaterHigh = false;
        }
    }
}

public class Program {
    static void Main() {
        WaterLevelSensor sensor = new();

        sensor.UpdateLevel(15);
        Console.WriteLine("Water Low: " + sensor.IsWaterLow);

        sensor.UpdateLevel(90);
        Console.WriteLine("Water High: " + sensor.IsWaterHigh);
    }
}
```

Explanation: This tool monitors water levels with low and high thresholds.

Difficulty Rating: 3/5 (Intermediate)

Question 83: Implement a Custom Waste Management System

Answer Code:

```
public class TrashCompactor {
    public void Compact() =>
        Console.WriteLine("Compacting trash...");

    public void Empty() =>
        Console.WriteLine("Emptying trash...");
}

public class Program {
    static void Main() {
        TrashCompactor compactor = new();
        compactor.Compact();
        compactor.Empty();
    }
}
```

Explanation: This system manages trash with compacting and emptying functions.

Difficulty Rating: 2/5 (Beginner)

Question 84: Implement a Custom Air Quality Monitoring System

Answer Code:

```
public class AirSensor {
    public double PM2_5 { get; private set; }
    public double CO2 { get; private set; }

    public void UpdateReadings(double pm, double co2) {
        PM2_5 = Math.Round(pm, 1);
        CO2 = Math.Round(co2, 0);
    }

    public void DisplayReadings() {
        Console.WriteLine($"PM2.5: {PM2_5} µg/m³");
        Console.WriteLine($"CO2: {CO2} ppm");
    }
```

```
}

public class Program {
    static void Main() {
        AirSensor sensor = new();
        sensor.UpdateReadings(12.345, 867.5);
        sensor.DisplayReadings();
    }
}
```

Explanation: This tool monitors air quality and displays readings.

Difficulty Rating: 3/5 (Intermediate)

Question 85: Implement a Custom Lighting Control System

Answer Code:

```
public class LightController {
    private bool _isOn;
    private int _brightness;

    public void TurnOn() =>
        _isOn = true;

    public void TurnOff() =>
        _isOn = false;

    public void SetBrightness(int level) {
        if (level < 0 || level > 100) throw new ArgumentException("Brightness must be between 0 and 100.");
        _brightness = level;
    }

    public override string ToString() =>
        $"Light is {(IsOn ? "on at {Brightness}% brightness" : "off")}";
}

public class Program {
    static void Main() {
        LightController controller = new();
        controller.SetBrightness(50);
        controller.TurnOn();

        Console.WriteLine(controller.ToString());
    }
}
```

Explanation: This system controls lighting with on/off and brightness adjustment.

Difficulty Rating: 3/5 (Intermediate)

Question 86: Implement a Custom Heating and Cooling System

Answer Code:

```
public class HVACController {
    private double _temperature;
    private Mode _mode;

    public enum Mode { Heat, Cool, Off }

    public HVACController(double initialTemp) =>
        _temperature = initialTemp;

    public void SetTargetTemperature(double temp) =>
        _temperature = Math.Round(temp, 1);

    public void SetMode(Mode mode) =>
        _mode = mode;

    public override string ToString() =>
```

```
        $"Current temp: {_temperature:F1}°C, Mode: {_mode}";
}

public class Program {
    static void Main() {
        HVACController h = new(22.5);
        h.SetMode(HVACController.Mode.Cool);
        h.SetTargetTemperature(25);

        Console.WriteLine(h.ToString());
    }
}
```

Explanation: This tool controls heating and cooling systems.

Difficulty Rating: 3/5 (Intermediate)

Question 87: Implement a Custom Security Monitoring System

Answer Code:

```
public class SecurityCamera {
    public bool IsRecording { get; private set; }

    public void StartRecording() =>
        IsRecording = true;

    public void StopRecording() =>
        IsRecording = false;
}

public class Program {
    static void Main() {
        SecurityCamera camera = new();
        camera.StartRecording();

        Console.WriteLine("Camera recording: " + camera.IsRecording);
    }
}
```

Explanation: This system controls a security camera's recording state.

Difficulty Rating: 2/5 (Beginner)

Question 88: Implement a Custom Energy Efficiency Calculator

Answer Code:

```
public class EnergyEfficiencyCalculator {
    public double Calculate(double usage, double production) =>
        Math.Round(production / usage * 100, 2);
}

public class Program {
    static void Main() {
        EnergyEfficiencyCalculator calc = new();
        double efficiency = calc.Calculate(100, 80);
        Console.WriteLine("Efficiency: " + efficiency + "%");
    }
}
```

Explanation: This tool calculates energy efficiency as a percentage.

Difficulty Rating: 2/5 (Beginner)

Question 89: Implement a Custom Smart Meter Reader

Answer Code:

```
public class SmartMeter {
    public double ElectricityUsage { get; private set; }
    public double GasUsage { get; private set; }

    public void UpdateReadings(double elec, double gas) {
        ElectricityUsage = Math.Round(elec, 2);
        GasUsage = Math.Round(gas, 2);
    }

    public override string ToString() =>
        $"Electricity: {ElectricityUsage:F2} kWh\nGas: {GasUsage:F2} m³";
}

public class Program {
    static void Main() {
        SmartMeter meter = new();
        meter.UpdateReadings(123.456, 789.012);
        Console.WriteLine(meter.ToString());
    }
}
```

Explanation: This tool reads and displays smart meter data.

Difficulty Rating: 3/5 (Intermediate)

Question 90: Implement a Custom Home Automation Controller

Answer Code:

```
public classHomeController {
    public void TurnOnLights() =>
        Console.WriteLine("Turning on lights...");

    public void AdjustTemperature(double temp) =>
        Console.WriteLine($"Adjusting temperature to {temp:F1}°C...");
}

public class Program {
    static void Main() {
        HomeController controller = new();
        controller.TurnOnLights();
        controller.AdjustTemperature(23.5);
    }
}
```

Explanation: This tool automates home devices with basic control functions.

Difficulty Rating: 2/5 (Beginner)

Question 91: Implement a Custom Structural Engineering Tool

Answer Code:

```
public class LoadCalculator {
    public double CalculateBendingMoment(double force, double distance) =>
        Math.Round(force * distance, 2);
}

public class Program {
    static void Main() {
        LoadCalculator calc = new();
        double moment = calc.CalculateBendingMoment(100, 5);
        Console.WriteLine("Bending Moment: " + moment + " Nm");
    }
}
```

Explanation: This tool calculates bending moments in structural engineering.

Difficulty Rating: 2/5 (Beginner)

Question 92: Implement a Custom HVAC Design Tool

Answer Code:

```
public class HVACDesigner {
    public double CalculateRequiredCapacity(double area, double efficiency) =>
        Math.Round(area * 1000 / efficiency, 2);
}

public class Program {
    static void Main() {
        HVACDesigner designer = new();
        double capacity = designer.CalculateRequiredCapacity(150, 0.8);
        Console.WriteLine("Required Capacity: " + capacity + " BTU/h");
    }
}
```

Explanation: This tool calculates HVAC system capacity based on area and efficiency.

Difficulty Rating: 2/5 (Beginner)

Question 93: Implement a Custom Road Safety Analysis Tool

Answer Code:

```
public class AccidentAnalyzer {
    public double CalculateSeverityIndex(int injuries, int fatalities) =>
        (injuries * 10 + fatalities * 100);
}

public class Program {
    static void Main() {
        AccidentAnalyzer analyzer = new();
        double severity = analyzer.CalculateSeverityIndex(5, 1);
        Console.WriteLine("Severity Index: " + severity);
    }
}
```

Explanation: This tool calculates the severity index of road accidents.

Difficulty Rating: 2/5 (Beginner)

Question 94: Implement a Custom Urban Planning Tool

Answer Code:

```
public class LandUsePlanner {
    public double CalculateDensity(int population, int area) =>
        Math.Round(population / area, 2);
}

public class Program {
    static void Main() {
        LandUsePlanner planner = new();
        double density = planner.CalculateDensity(100000, 50);
        Console.WriteLine("Population Density: " + density + " per km²");
    }
}
```

Explanation: This tool calculates population density for urban planning.

Difficulty Rating: 2/5 (Beginner)

Question 95: Implement a Custom Construction Material Calculator

Answer Code:

```
public class MaterialEstimator {
    public double CalculateConcreteRequired(double area, double thickness) =>
        Math.Round(area * thickness / 1000, 2);
}

public class Program {
    static void Main() {
        MaterialEstimator estimator = new();
        double concrete = estimator.CalculateConcreteRequired(100, 0.5);
        Console.WriteLine("Concrete Needed: " + concrete + " m³");
    }
}
```

Explanation: This tool estimates concrete required for construction projects.

Difficulty Rating: 2/5 (Beginner)

Question 96: Implement a Custom Environmental Impact Assessment Tool

Answer Code:

```
public class EIAEvaluator {
    public double CalculateCarbonFootprint(double emissions, double reduction) =>
        Math.Round(emissions * (1 - reduction / 100), 2);
}

public class Program {
    static void Main() {
        EIAEvaluator evaluator = new();
        double footprint = evaluator.CalculateCarbonFootprint(100, 20);
        Console.WriteLine("Carbon Footprint: " + footprint + " tons CO2");
    }
}
```

Explanation: This tool calculates the carbon footprint with reduction.

Difficulty Rating: 2/5 (Beginner)

Question 97: Implement a Custom Structural Integrity Monitor

Answer Code:

```
public class StructuralMonitor {
    public bool IsSafe(double stress, double yield) =>
        stress < yield * 0.8;
}

public class Program {
    static void Main() {
        StructuralMonitor monitor = new();
        bool safe = monitor.IsSafe(200, 300);
        Console.WriteLine("Structure Safe: " + safe);
    }
}
```

Explanation: This tool checks if a structure is within safe stress limits.

Difficulty Rating: 2/5 (Beginner)

Question 98: Implement a Custom Building Energy Audit Tool

Answer Code:

```
public class EnergyAuditor {
```

```
    public double CalculateEnergyEfficiency(double usage, double potential) =>
        Math.Round((usage / potential) * 100, 2);
}

public class Program {
    static void Main() {
        EnergyAuditor auditor = new();
        double efficiency = auditor.CalculateEnergyEfficiency(50, 100);
        Console.WriteLine("Energy Efficiency: " + efficiency + "%");
    }
}
```

Explanation: This tool calculates a building's energy efficiency percentage.

Difficulty Rating: 2/5 (Beginner)

Question 99: Implement a Custom Construction Cost Estimator

Answer Code:

```
public class CostEstimator {
    public double CalculateTotalCost(double materials, double labor) =>
        materials + labor * 1.15;
}

public class Program {
    static void Main() {
        CostEstimator estimator = new();
        double cost = estimator.CalculateTotalCost(50000, 30000);
        Console.WriteLine("Total Cost: $" + cost);
    }
}
```

Explanation: This tool estimates construction costs including labor markup.

Difficulty Rating: 2/5 (Beginner)

Question 100: Implement a Custom Road Network Planner

Answer Code:

```
public class RoadPlanner {
    public int CalculateRequiredLanes(int vehiclesPerHour) =>
        (int)Math.Ceiling(vehiclesPerHour / 2000.0);
}
```

```
public class Program {
    static void Main() {
        RoadPlanner planner = new();
        int lanes = planner.CalculateRequiredLanes(8000);
        Console.WriteLine("Required Lanes: " + lanes);
    }
}
```

Explanation: This tool calculates the number of traffic lanes needed based on vehicle volume.

Difficulty Rating: 2/5 (Beginner)

To solve this problem, we need to implement a road network planner that calculates the number of traffic lanes required based on the vehicle volume per hour. The goal is to determine how many lanes are necessary to handle the traffic efficiently.

Approach

1. **Problem Analysis**: The problem requires us to determine the number of lanes needed based on the number of vehicles passing through an area per hour. Each lane can handle a certain capacity, and we need to calculate how many such capacities fit into the given vehicle volume.
2. **Intuition**: If each lane can handle a maximum of 2000 vehicles per hour, then the number of lanes required can be found by dividing the total vehicle volume by this capacity and rounding up to ensure we have enough lanes.

3. **Algorithm Selection**: We'll use a straightforward division and ceiling operation to calculate the number of lanes. This ensures that any partial requirement results in an additional lane.
4. **Complexity Analysis**: The solution involves a simple arithmetic operation, which has constant time complexity O(1).

Solution Code

```
public class RoadPlanner {
    public int CalculateRequiredLanes(int vehiclesPerHour) =>
        (int)Math.Ceiling(vehiclesPerHour / 2000.0);
}

public class Program {
    static void Main() {
        RoadPlanner planner = new();
        int lanes = planner.CalculateRequiredLanes(8000);
        Console.WriteLine("Required Lanes: " + lanes);
    }
}
```

Explanation

- **RoadPlanner Class**: This class contains a method `CalculateRequiredLanes` that takes an integer `vehiclesPerHour` as input.
- **Method Logic**: The method calculates the number of lanes by dividing `vehiclesPerHour` by 2000.0 (the capacity per lane) and then rounding up using `Math.Ceiling`. This ensures that even if there's a remainder, we account for it by adding an extra lane.
- **Program Class**: The `Main` method demonstrates the usage of the `RoadPlanner` class by calculating and printing the required lanes for a given vehicle volume.

This approach ensures that we efficiently determine the number of lanes needed, guaranteeing smooth traffic flow without under-provisioning.