Contents

1. Practical Questions on Static Class

1. **Math Utility Class:**
   Create a static class called MathUtils that contains methods for basic mathematical operations such as addition, subtraction, multiplication, and division. Users should be able to call these methods directly without creating an instance of the class.
2. **Utility Class for String Manipulation:**
   Create a static class named StringUtils that provides methods for common string manipulation tasks such as reversing a string, converting a string to uppercase, and checking if a string is a palindrome.

2. Partial Class

1. **Partial Class Declaration:**
   Write a C# program that demonstrates the declaration of a partial class. Create two partial class files for a class named Person, with each file containing a different method. Compile and execute the program to demonstrate the functionality.
2. **Partial Class Inheritance:**
   Implement a C# program that showcases inheritance with partial classes. Define a base class **Vehicle** as a partial class with some properties and methods. Then create another partial class file that extends **Vehicle** named **Car**, adding specific properties and methods related to cars.
3. **Access Modifiers and Members:**
   Develop a C# program that illustrates the use of different access modifiers in partial classes. Create a partial class named **Employee** with private fields in one file and public properties accessing these fields in another file. Demonstrate accessing these members from another class outside the partial class.

3. Nested Class

1. **Address Book:** Create a class called **AddressBook** that represents a collection of contacts. Inside the **AddressBook** class, define a nested class named **Contact** to represent individual contacts. Implement methods to add, remove, and search for contacts in the address book.
2. **Math Operations:** Write a class called **MathOperations** that performs various mathematical operations. Inside the **MathOperations** class, define a nested class named **Geometry** that contains methods for calculating the area and perimeter of geometric shapes such as rectangles, circles, and triangles.
3. **File Manager:** Implement a class called **FileManager** for reading and writing files. Inside the **FileManager** class, define a nested class named **FileMetadata** to store information about files, such as file name, size, and creation date. Implement methods to retrieve file metadata and perform file operations.
4. **Employee Management System:** Create a class called **EmployeeManager** to manage employee data. Inside the **EmployeeManager** class, define a nested class named **Employee** to represent individual employees. Implement methods to add, remove, and update employee information.
5. **Shopping Cart:** Write a class called **ShoppingCart** for managing items in a shopping cart. Inside the **ShoppingCart** class, define a nested class named **CartItem** to represent individual items in the cart, including properties such as item name, quantity, and price. Implement methods to add, remove, and calculate the total cost of items in the cart.

4. Singleton Class

1. **Database Connection Singleton:** Write a database connection manager class called **DbConnectionManager** as a singleton to manage database connections for the application. The class should handle creating, opening, and closing database connections.
2. **User Session Singleton:** Create a user session manager class called **UserSession** as a singleton to manage user sessions in a web application. The class should handle creating, validating, and destroying user sessions.

5. Abstract Class

1. **Shape Hierarchy:** Create an abstract class called **Shape** with an abstract method **CalculateArea()**. Derive concrete classes such as **Circle**, **Rectangle**, and **Triangle** from **Shape** and implement the **CalculateArea()** method for each shape.

2. **Employee Management System:** Design an abstract class called **Employee** with properties like **Name**, **EmployeeID**, and an abstract method **CalculateSalary()**. Create concrete classes such as **FullTimeEmployee** and **ContractEmployee** that inherit from **Employee** and implement the **CalculateSalary()** method differently.

3. **Animal Hierarchy:** Define an abstract class called **Animal** with properties like **Name** and **Age**, and an abstract method **MakeSound()**. Create concrete classes such as **Dog**, **Cat**, and **Bird** that inherit from **Animal** and implement the **MakeSound()** method with appropriate sounds.

4. **Bank Account System:** Implement an abstract class called **Account** with properties like **AccountNumber**, **Balance**, and abstract methods like **Deposit()** and **Withdraw()**. Derive concrete classes such as **SavingsAccount** and **CheckingAccount** from **Account** and implement the abstract methods according to the specific account type.

5. **Vehicle Hierarchy:** Create an abstract class called **Vehicle** with properties like **Make**, **Model**, and an abstract method **Drive()**. Derive concrete classes such as **Car**, **Motorcycle**, and **Truck** from **Vehicle** and implement the **Drive()** method differently for each type of vehicle.

6. Sealed Class

1. **Math Library Extension:** Assume you have a math library with a class hierarchy representing different mathematical operations (e.g., **MathOperation**, **Addition**, **Subtraction**, **Multiplication**, **Division**). Seal the base **MathOperation** class to prevent further derivation and explain why sealing is appropriate in this context.