****

**OOP & Data Structures - Assignment**

**Task 1 (Code Already Provided): Design a Zoo Management System Using OOP**

**Problem Statement:**

Your task is to design and implement classes for a simple zoo simulator. The simulator will model different types of animals, their habitats within a zoo, and the zoo itself. The simulator will demonstrate key object-oriented programming (OOP) principles such as inheritance, encapsulation, polymorphism, and abstraction.

The simulator will also demonstrate key data structures in C# including Lists, and Dictionaries.

The classes you will need to create include:

* Animal: This is an abstract class that serves as a base class for specific types of animals. Each animal should have a name, an age, a species, and a unique ID. The Animal class should implement a method Eat().
* ISoundBehaviour: An interface that defines a method MakeSound(). This represents the animal's ability to make a sound.
* Lion, Elephant, Monkey, and Fish: These are derived classes from Animal and implement ISoundBehaviour (except for Fish, which does not make a sound). Each class should override the Eat() method appropriately.
* Habitat: This class represents a specific habitat in the zoo. Each habitat has a name and a list of animals that live in it.
* Zoo: This class represents the entire zoo. It should contain a List of habitats and a Dictionary that acts as an inventory to quickly look up animals by their unique IDs. It should also contain a Dictionary that keeps count of each type of animal present in the zoo. The Zoo class should also have methods to FeedAllAnimals(), MakeAllAnimalsSound(), and GetAnimalCountByType() which will use the Dictionary of animal types.

In your main program, create an instance of a Zoo, some Habitats, and several Animals. Add the animals to appropriate habitats and the habitats to the zoo. Demonstrate the functionality of your classes by feeding all animals, making all sound-capable animals make a sound, and looking up specific animals using the zoo inventory.

Your tasks include:

1. Implementing the list in the Habitat class so animals can be added to the same habitat.
2. Implementing the Dictionary in the Zoo class to keep track of animal counts by type, and to provide a fast look-up of animals by their unique IDs.
3. Demonstrating the usage of these data structures by creating methods in the Zoo class to feed all animals, make all sound-capable animals make a sound, and provide a count of each type of animal present in the zoo.
4. Testing your implementation by creating several animals, adding them to habitats, adding the habitats to a zoo, and then calling the methods in the Zoo class.

When creating the animals, habitats, and zoo, print out appropriate messages to show what is being created. Also, when an animal is added to a habitat, print out a message.

Implement the above classes, and use appropriate data structures where necessary. Utilize OOP principles effectively to ensure your code is clean, modular, and efficient.

**Task 2: Create a Bank Account Management System**

**Objective:** The goal of this project is to design a comprehensive Bank Account Management System using the core concepts of Object-Oriented Programming (OOP) - Abstraction, Encapsulation, Polymorphism, and Inheritance. The design should effectively utilize interfaces and display an understanding of relationships such as Association, Aggregation, Dependency, and Realization/Implementation. Moreover, you will be using collections like Lists and Dictionaries to store and manipulate the data of your system.

**Task Details:**

1. Abstraction: You will design a BankAccount class as an abstraction of a bank account. The class should encapsulate the basic properties of a bank account like accountNumber, accountHolderName, and balance, along with methods that operate on these properties.

2. Encapsulation: Ensure data security by declaring accountNumber, accountHolderName, and balance as private data members in the BankAccount class. Provide public methods for accessing and modifying these properties to uphold encapsulation.

3. Polymorphism: Your system will illustrate polymorphism by Method Overriding and Method Overloading.

* Method Overriding: Create subclasses SavingsAccount, CheckingAccount, LoanAccount that inherit from the superclass BankAccount. Override the CalculateInterest() method in each subclass to define a unique interest calculation.
* Method Overloading: Overload the Deposit() and Withdraw() methods in the BankAccount class to support different ways of depositing or withdrawing money.

4. Inheritance: Implement inheritance by designing SavingsAccount, CheckingAccount, and LoanAccount as subclasses that inherit from the superclass BankAccount.

5. Interfaces: Create an ITransaction interface with abstract methods ExecuteTransaction(decimal amount) and PrintTransaction(). Have SavingsAccount, CheckingAccount, LoanAccount classes implement this interface.

6. Collections:

* List: For each BankAccount object, maintain a list of transactions, where each transaction is an object of a class (e.g., Transaction class) holding relevant transaction details.
* Dictionary: In the Bank class, maintain a dictionary of all the bank accounts. The account number will be the key, and the corresponding BankAccount object will be the value.

7. Relationships:

* Association: The Bank class is associated with the BankAccount class as it manages the different bank accounts.
* Aggregation: The Bank class will contain a dictionary of BankAccount objects, representing an aggregation relationship.
* Dependency: The BankAccount objects are dependent on the Bank object, as the bank account objects need the bank object for their creation.
* Realization/Implementation: The SavingsAccount, CheckingAccount, LoanAccount classes realize the ITransaction interface.

8. Additional Features: Extend your system with additional features:

* Transaction History: Implement a feature that allows account holders to view their transaction history.
* Different Transaction Types: Extend the types of transactions beyond deposit and withdrawal. Consider including bank charges, interest accrual, etc.

Remember to follow necessary error checking and validation steps while dealing with financial data. The goal of this project is to provide you a concept about how things work in real-world applications. Implementing a robust, error-free financial system requires careful consideration and extensive testing beyond the scope of this project. Happy coding!