Assignment 1: Banking System Design

Objective: Build an application using OOP principles with a focus on Composition Over Inheritance.

Requirements:

1. **Account Types:**
   - Savings Account: Allows deposits and withdrawals.
   - Checking Account: Allows deposits and withdrawals, but has an overdraft limit.
   - Fixed Deposit Account: Allows deposits, no withdrawals allowed until maturity.

2. **Common Features:**
   - All accounts should support basic operations like checking the balance and displaying the account details.

3. **Composition Over Inheritance:**
   - Implement the solution using composition rather than relying heavily on inheritance.
   - Avoid using a deep hierarchy of classes for different account types.

4. **Future Extensibility:**
   - Design the system in a way that allows for easy addition of new account types without modifying existing code.
   - Consider potential new account types that might be introduced in the future.

5. **Behavioural Flexibility:**
   - Ensure that the system can handle changes in behaviour for existing account types without causing cascading changes throughout the codebase.

Implementation Guidelines:

1. **Account Class:**
   - Create an `Account` class that serves as the base class for all account types.
   - Use interfaces for common features shared among different account types.

2. **Composition:**
   - Implement the interfaces and use as the composition in the Account type to achieve the concretion.

3. **Encapsulation:**
   - Don't forget about the encapsulation, isolate your account specific members (ex. the overdraft limit) and use data hiding to protect them.

4. **Client Code:**
   - Create a sample client code that demonstrates the use of your banking system with different account types.
   - Showcase how the system remains flexible in the face of changes or additions.

Assignment 2: Generic Operations Utility

Objective: Create the `ICalculationUtility` interface that supports generic types and provide separate implementations for string and numeric calculations.

Requirements:

1. **ICalculationUtility Interface (Generic):**
   - Create the `ICalculationUtility` generic interface.
   - The interface should encompass separate methods for addition, subtraction, multiplication, division (with `ref` keyword if needed), and quotient/remainder (with `out` keyword).

2. **Numeric Implementation:**
   - Implement the `ICalculationUtility` interface and for numeric operations.

3. **String Implementation:**
   - Implement the `ICalculationUtility` interface that will be applicable to strings.
   - You can simply throw the NotSupportedException for the operations that are not commonly used for string type.

4. **Client Code:**
   - Write a sample console application that demonstrates the usage of both numeric and string operations using the generic `ICalculationUtility` interface.