Matrix Calcultion Project Report

Yu Lu 2364294 (Biomedical Statistics)

How I understand the question

Task 1: Matrix Class Implementation

I need to create a constructor that validates the input matrix and then initializes the instance variable data with a deep copy of the provided matrix.

Task 2: Accessor Method

I need to ensure that the data (input matrix) is deeply copied rather than merely referencing the original data. This guarantees that the matrix passed to the constructor or method is fully duplicated, so any modifications to the original data will not affect the copied matrix.

Task 3 - Task 5: two matrix's calculation

I should design a project to calculate two matrices' addition, subtraction and multiplication correctly (pass tests):

- Test Valid Addition: Passed
- Test Valid Subtraction: Passed
- Test Valid Multiplication: Passed

Task 6 - Task 7: one matrix's transpose or multiplication

I should design a project to calculate a matrix's transposition and determinant(size of 2×2 and 3×3) correctly (pass tests):

- Test Valid Transposition: Passed
- Test Valid Determinant: Passed

Handle edge completely

I should design a project to handle all possible edge situation using exceptions appropriately:

- Test Invalid Constructor (null data): Passed
- Test Invalid Constructor (jagged array): Passed
- Test Invalid Addition (different sizes): Passed
- Test Invalid Subtraction (different sizes): Passed
- Test Invalid Multiplication (incompatible sizes): Passed
- Test Invalid Determinant (non-square matrix): Passed

Why I believe my solution is the best approach.

Clear structure

This diagram with corresponding description demonstrates my clear structure of this project. According to UML rules in JAVA program, I use "-" to represent private attribute or method and "+" to represent public attribute or method.

Access Modifier	AbstractMatrix	Description
-	double[][] data	Attribute to store matrix data
	AbstractMatrix(double[][] data)	Constructor to initialize matrix with data
+	void ValidData(double[][] data)	Validate the matrix data
+	boolean isSameRowLength(double[][] data)	Check if all rows have the same length
+	double[][] getData()	Method to get a deep copy of the matrix data
+	double[][] DeepCopy(double[][] data)	Create a deep copy of the matrix
+	void SameSize-Data(AbstractMatrix AnotherData)	Validate if matrices have the same size for addition and subtraction
+	abstract Matrix **add**(AbstractMatrix AnotherData)	Method to add two matrices

Access Modifier	AbstractMatrix	Description
+	abstract Matrix **subtract**(AbstractMatrix AnotherData)	Method to subtract two matrices
+	void ValidMultiply(AbstractMatrix AnotherData)	Validate if matrices can be multiplied
+	abstract Matrix **multiply**(AbstractMatrix AnotherData)	Method to multiply two matrices
+	double[][] TwoMatricesOperation(AbstractMatrix AnotherData, String OperationType)	Perform an operation (add, subtract, multiply) between matrices
+	abstract Matrix **transpose()**	Method to transpose the matrix
+	void SquareMatrix()	Validate if the matrix is square for determinant calculation
+	abstract double **determinant()**	Method to calculate the determinant of the matrix

Access Modifier	Matrix (extends AbstractMatrix)	Description
	Matrix(double[][] data)	Constructor to initialize matrix with data
+	Matrix **add**(AbstractMatrix AnotherData)	Method to add another matrix
+	Matrix **subtract**(AbstractMatrix AnotherData)	Method to subtract another matrix
+	Matrix **multiply**(AbstractMatrix AnotherData)	Method to multiply with another matrix
+	Matrix **transpose**()	Method to transpose the matrix
+	double **determinant**()	Method to calculate the determinant of the matrix

Robust Data Handling

The constructor ensures robust data management:

- **Validation**: The `ValidData` method ensures that only valid input matrices are accepted, preventing invalid states.
- **Deep Copy**: The input matrix is deep-copied to prevent unintended modifications to the original data, ensuring the integrity of the matrix object's state.

Matrix class inheriting AbstractMatrix

The `AbstractMatrix` class serves as a reliable, reusable (by inheriting from `AbstractMatrix`, the `Matrix` class can reuse the code for data validation, deep copying, and other utility methods without rewriting them) and clean-code(inheritance and some nice functions that I designed simplifies the design by allowing me to define a general matrix behavior in AbstractMatrix and then specialize it in my subclass Matrix) foundation for advanced matrix operations, ensuring efficiency and maintainability across various linear algebra applications. It provides a flexible and reusable structure for matrix operations:

- **Abstract Methods**: Operations like `add`, `subtract`, and `multiply` enforce a consistent interface across implementations.
- **Concrete Methods**: Utility methods like `TwoMatricesOperation` reduce redundancy by providing reusable logic across subclasses.
- **Extensibility**: Specialized matrices in my future linear algebra learning journey can be introduced without altering existing code, ensuring adaptability for future requirements.

Polymorphism

The use of polymorphism in these matrix classes enhances code flexibility and reusability, accommodating various matrix types and operations while adhering to object-oriented design principles.

- Abstract Classes and Methods:

  AbstractMatrix is an abstract class defining abstract methods like add, subtract, multiply, transpose, and determinant. These methods are meant to be implemented by subclasses, allowing for different behaviors.

- Method Overriding:

  The Matrix class overrides these abstract methods from AbstractMatrix, demonstrating polymorphism by providing specific implementations that can be further customized by other subclasses.

- Constructor and Data Handling:

  Matrix uses super(data) to call AbstractMatrix's constructor, ensuring data validation and deep copying are performed, maintaining consistency across subclasses.

- Polymorphic Method Invocation:

  TwoMatricesOperation in Matrix accepts AbstractMatrix types, allowing polymorphic behavior. It handles operations like addition and subtraction, demonstrating how subclasses can implement different logic.

Focused Functionality

Each function in the code is designed to perform one specific task, which is clean and clear to understand.

- **Modular Design**: Edge-checking and calculation are handled in separate functions, such as `ValidData` for validation and `TwoMatricesOperation` for computations. This ensures the code is easy to test, debug, and extend without unintended side effects.

Clear and Informative Error Messaging

- **Detailed Feedback**: Issues such as dimension mismatches or non-square matrices for calculations are explicitly flagged with clear error messages.
- **Usability**: These specific messages help users quickly identify and resolve input calculation issues.

Readability and Maintainability

- **Descriptive Method Names**: Methods like `ValidData`, `DeepCopy`, and `SameSizeData` are self-explanatory, making the code easy to understand for future developers.
- **Logical Organization**:

  Input validation and initialization occur in the constructor.

  Utility methods like `ValidMultiply` and `TwoMatricesOperation` are logically placed to support core operations.

Method comes followed by the preparing works(validation test) mentioned above.

Inheritance helps in organizing code into a hierarchical structure, which can make it easier to understand and navigate.

- **Inline Comments**: Critical steps are documented to clarify intent and functionality, enhancing maintainability.

Scalability

- **Flexible Structure**: The `AbstractMatrix` class supports straightforward extensions(as discussed before), such as adding new operations or handling specialized matrix types.
- **Prepared for Advanced Use**: Common operations like addition, subtraction, multiplication, and transposition provide a solid foundation, while the determinant calculation currently limited to 3×3 can be extended using techniques like recursive computation or high-level matrix decomposition method as advanced linear algebra concepts are introduced.

Conclusion

In conclusion, the Matrix Calculation Project has been designed to provide a robust and efficient framework for performing various matrix operations. The implementation of the AbstractMatrix class and its extension, the Matrix class, ensures a clear structure and maintainability, adhering to object-oriented principles and design patterns.