Arbitrary-Precision Arithmetic Library in Java

Rishika Shreya Fadi
CS24BTECH11052

Contents

1 Introduction

The goal of this project is to implement an arbitrary-precision arithmetic library in Java,supporting both integer and floating-point arithmetic.
In standard programming environments, arithmetic operations are limited by the fixed size of built-in data types such as `int`, `long`, or `double`. These limitations can lead to overflow or loss of precision when dealing with very large or highly precise numbers.

Arbitrary-precision arithmetic provides a way to perform calculations on numbers of any size or precision, constrained only by the system's memory. This project focuses on designing and implementing an arbitrary-precision arithmetic library in Java.
The goal is to support addition, subtraction, multiplication, and division for both integers and floating-point numbers without relying on Java's built-in `BigInteger` or `BigDecimal` classes. The library is designed from the ground up using string-based representations and custom logic to perform digit-level arithmetic operations.

Through this project, the underlying mechanisms of number representation and arithmetic could be understood clearly, and software engineering principles such as modular design, testing, and documentation to ensure project was well-structured and easy-to-maintain.

The purpose of this report is to document the design, implementation, and testing of an arbitrary-precision arithmetic library in Java. It provides a detailed explanation of the algorithms/functions that I used while building the library. The report also highlights the software engineering practices employed, such as modular design and testing.
Through this documentation, the goal is to share insights into the process of building a high-precision arithmetic system from scratch.

2 Design

The project follows a Maven-based Java structure with clear separation of source code, tests, and build outputs.

2.1 Project Structure

```
final_proj
├── src
│   └── main
│       └── java
│           ├── arbitraryarithmetic
│           │   ├── AFloat.java
│           │   └── AInteger.java
│           └── MyInfArith.java
├── target
│   ├── classes
│   │   ├── arbitraryarithmetic
│   │   │   ├── AFloat.class
│   │   │   └── AInteger.class
│   │   └── MyInfArith.class
│   ├── generated-sources
│   ├── maven-archiver
│   ├── maven-status
│   └── arbitrary-arithmetic-1.0-SNAP.jar
├── documentation
│   ├── report.tex
│   └── report.pdf
├── compile_and_run.py
└── pom.xml
```

- `AInteger.java` & `AFloat.java`: Core classes implementing **arbitrary-precision arithmetic** for integers and floating-point numbers, respectively.
- `MyInfArith.java`: **Entry point** for CLI execution, handling user input and invoking `AInteger`/`AFloat` operations.

- `src/main/java/arbitraryarithmetic`: Directory for **main source code**
- `target/`: Maven's build output (compiled classes, JAR artifacts, and metadata)
- `pom.xml`: Maven configuration file defining dependencies and build settings
- `compile_and_run.py`: Utility script to automate compilation/execution
- `documentation/report.tex`: Project documentation in LaTeX

2.2 UML Class Diagram

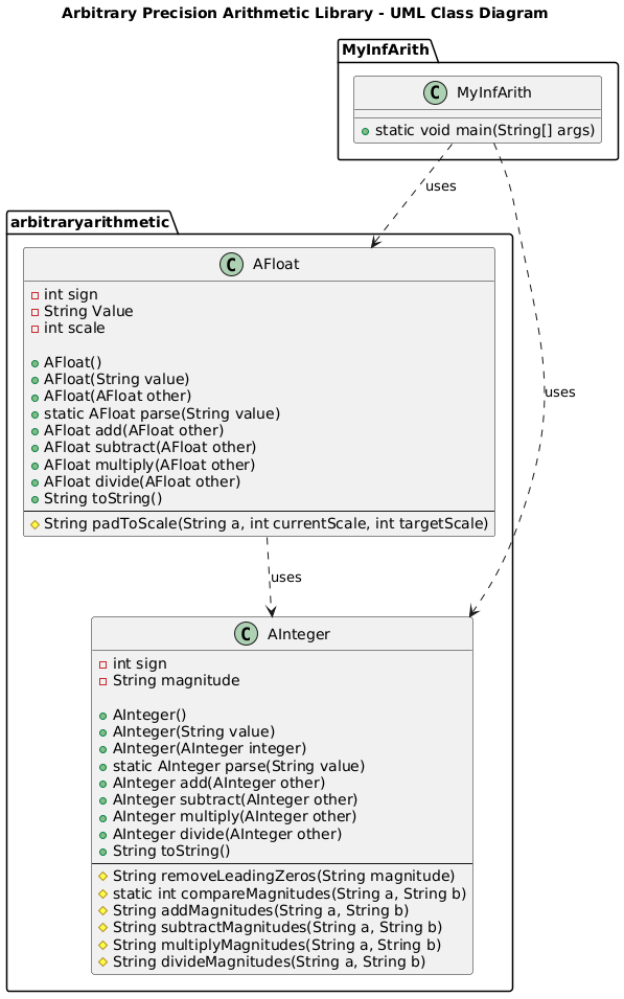

Figure 1: UML Class Diagram for `MyInfArith , AInteger and AFloat`

3 Implementation

3.1 Integer arithmetic

- **Addition**
  - Processes digits **right-to-left**
  - Sums digits + carry from prior step
  - Stores last digit of sum, carries forward the rest
  - Reverses final result
  - helper-function `addMagnitudes` is used.
- **Subtraction**
  - Works **right-to-left**
  - Subtracts digits + borrow from prior step
  - If result < 0, adds 10 and sets borrow $= 1$
  - Trims leading zeros from result
  - helper-functions `subtractMagnitudes` and `compareMagnitudes` are used.
- In Addition if one of the numbers is negative then subtraction logic is implemented
- In Subtraction if only one of the numbers is negative then addition logic is implemented.
- **Multiplication**
  - Processes digits right-to-left, multiplying each digit pair
  - Stores results in a temporary array with proper digit positioning
  - Handles carries automatically by splitting products into tens/units places
  - Converts the final array to a string, trimming leading zeros.
  - Sets sign positive if operands have same sign, negative otherwise.
  - helper-functions `multiplyMagnitudes` is used.
- **Division**
  - Performs digit-by-digit long division on the magnitude strings.(repeated subtraction and remainder tracking)
  - Processes the dividend left-to-right, building a "current" number

- Repeatedly subtracts the divisor from "current" to count quotients
- Appends each quotient digit and continues with the remainder
- Returns "0" if divisor > dividend, otherwise the quotient string
- uses `subtractMagnitudes` and `compareMagnitudes` as helper-functions.

3.2 Floating-point arithmetic

Basic logic: remove the decimal points and perform the operations on the numbers as if they were integers and depending on the operation add the decimal point in the result.

- **Addition**
  - Equalizes decimal places by padding the shorter number with trailing zeros
  - Treats both numbers as integers (removes decimal points)
  - Uses `AInteger.add()` to perform exact integer addition
  - Restores decimal position to the maximum scale among the numbers.
  - Trims trailing zeros and limits to 30 decimal places
  - no sign handling needs to be done because it is handled in `AInteger`.
- **Subtraction**
  - same padding with zeroes as addition.
  - Treats both numbers as integers.
  - Uses `AInteger.subtract()`.
  - restores decimal position to the maximum scale among the two numbers.
  - Trims trailing zeros and limits to 30 decimal places
- **Multiplication**
  - Sums the scales of both operands
  - Removes decimals and treats as integers
  - Uses `AInteger.multiply()`.
  - Applies combined scale (decimals from both numbers).
  - Sets sign positive if operands have same sign, negative otherwise
- **Division**

- Pads numerator with 30 zeros for decimal precision
- final scale: relative scale difference between operands
- Uses `AInteger.divide()` on scaled integers.
- Trims trailing zeroes.
- Enforces 30 decimal place maximum.
- Handles division-by-zero error explicitly

3.3 Edge Case Handling

- Empty or Null input throws `IllegalArgumentException`.
- Invalid arguments like (a123, 456- etc) throws `IllegalArgumentExcpetion`.
- Inputs with irregular formatting (eg, .5, -.5, 000006) are all modified to proper inputs using regeX.
- +0, +0.0, -0, -0.0, -00000000.. inputs like these are standardized to 0.
- Division by Zero throws `ArithmeticException` error.

4 Testing and Validation

- Each method (e.g., add, subtract) was tested individually with predefined inputs and expected outputs.(uploaded on Google Classroom)
- For larger values, outputs were compared against Java's BigInteger or BigDecimal classes to ensure accuracy.
- Multiple test cases were written to cover a wide range of scenarios, such as:
  - Adding two large positive integers, one positive one negative, two negative integers.
  - Subtracting a smaller number from a larger one and vice versa.
  - Subtracting a positive number from a negative number and vice versa.
  - Performing floating-point addition with varying decimal lengths.
  - Handling negative numbers and zero in different operations.
- In all cases tested, the output matched the expected results

5 Conclusion

5.1 What I learned

- I gained a deeper understanding of OOPS concepts such as modularity, method overloading etc.
- I improved my ability to write code in a more organized and clean way, making it easier to understand, test, and maintain.
- Gained a bit more experience in Git and Docker.
- Used maven for the first time.

5.2 Conclusion

This project helped me strengthen my programming skills and gain hands-on experience with practices in software development.