Arbitrary Precision Arithmetic Library in Java

CS1023 - Software Development Fundamentals (SDF) Project

Shiva Chethan-CS24BTECH11057

1st May 2025

Overview : Arbitrary precision arithmetic enables calculations with numbers of any size and precision, unrestricted by typical hardware limits. This project implements an arithmetic engine in Java to accurately perform operations on extremely large integers and floating-point numbers. By representing numbers as strings and designing custom algorithms, it avoids overflow and rounding errors common in standard fixed-precision arithmetic. The system supports fundamental operations such as addition, subtraction, multiplication, and division for both integer and floating-point values.

Contents

1 Introduction

The objective of this project is to develop a Java library capable of handling arithmetic operations on numbers of arbitrary size and precision, avoiding limitations of Java's built-in primitive types. Using string manipulation, we bypass overflow constraints and achieve high precision in floating point calculations. This project also emphasizes sound software engineering practices such as modular design, comprehensive documentation, testing, and version control.

2 Design

2.1 Design Decisions

- **Character-Level Arithmetic:** Instead of relying on built-in numerical types, the library processes numbers as character strings. This technique eliminates the limitations imposed by fixed-size datatypes and supports computations with theoretically unlimited size and precision.
- **Input Refinement:** To ensure consistency, all user-provided numbers are cleaned and standardized. This includes eliminating insignificant zeros and formatting the numbers uniformly before any operations are applied.
- **Internal Sign Encoding:** The sign of a number is decoupled from its digits and stored separately using a boolean flag. This separation simplifies arithmetic logic by allowing operations to be performed on absolute values with sign corrections applied afterward.
- **Precision-Controlled Decimals:** For floating-point calculations, the system performs decimal alignment by equalizing the length of fractional parts. This allows digit-by-digit operations without risking misalignment or data loss.
- **Graceful Failure Mechanisms:** Rather than failing silently or producing incorrect results, the library actively detects and reports erroneous conditions—such as invalid inputs or division by zero—through meaningful exception messages.

2.2 Project Architecture

The project is organized as follows:

- `AInteger` and `AFloat`: Core classes containing logic for all arithmetic operations.
- `MyInfArith`: Entry point for CLI-based usage, handling argument parsing and invoking appropriate methods.
- `src/main/java/arbitraryarithmetic`: Contains main source code.
- `src/test/java/arbitraryarithmetic`: Contains JUnit tests.

2.3 Project Structure

The directory structure of the project is as follows:

```
root/
 RAW/
    AFloat.class
    AFloat.java
    AInteger.class
    AInteger.java
 myinfarith/
    MyInfArith.class
    MyInfArith.java
 src/
    main/java/arbitraryarithmetic/
        AFloat.class
        AFloat.java
        AInteger.java
 target/
    classes/arbitraryarithmetic/
       AFloat.class
       AInteger.class
    maven-archiver/
       pom.properties
    maven-status/maven-compiler-plugin/
       createdFiles.lst
       inputFiles.lst
    aarithmetic.jar
 MyInfArith.py
 pom.xml
 README.md
```

3 Implementation

3.1 Algorithms

- **Addition/Subtraction:** I have used manual borrow and addition logic. Addition of number with opposite signs are subtracted and same signs are added and a minus sign is added at last.
- **Multiplication:** Multiplying the multiplicand to each digit of the second number and adding reuired zeroes.
- **Division:** Long division with subtracting repeatedly and obtaining remainder. Zero is added to remainder and repeatedly subtracted to get 30 digit precision.

3.2 Edge Case Handling

- Division by zero throws `ArithmeticException` when divided by zero.
- Results like `-00.000` are normalized to `0.0`.
- Outputs having trailing zeroes or leading zeroes are removed.

4 Limitations

- Precision is supported up to 30 digits.
- A number like $\frac{1}{10^{31}} = 0.000\ldots$ (31 digits) is too small to display a non-zero digit within the precision limit.
- Positive numbers should not include a '+' sign. Any number of leading or trailing zeroes will be correctly handled and normalized.

5 Verification

- All arithmetic operations tested by test cases generated by ChatGpt and manual verification.
- Verified correctness for large inputs, zeros, negative numbers.

6 Key Learnings

- Learnt how to write long lines of code in Java and debug them.
- Learned how to use Maven Git Latex and import packages.
- Practiced writing clean code and checking every edge case.

7 References

- Apache Maven Documentation
- ChatGPT for helping in maven.