Infinite Precision Arithmetic in Java

Project Report for CS1023

Software Development Fundamentals

Submitted by:

Karthikay Chandana

CS24BTECH11033

Date: April 29, 2025

1 Abstract

This Java library contains classes and methods to enable infinite precision arithmetic operations for both integers and floating-point numbers. The core classes, **AInteger** and **AFloat**, allow addition, subtraction, multiplication, and division on numbers of arbitrary length and precision, overcoming the limitations of Java's built-in numeric types. The approach is to store numbers as strings and perform arithmetic digit-wise, handling both integer and fractional parts, negative numbers, and arbitrary decimal precision.

2 Introduction

Accuracy is crucial in high-stakes computations, where minor inaccuracies can lead to major errors. Built-in Java types like `int`, `long`, `float`, and `double` have fixed ranges and are prone to overflow or precision loss when dealing with extremely large or small values. This library addresses these limitations by representing numbers as strings and performing arithmetic operations at the digit level, allowing for calculations with arbitrary size and precision.

Data range of standard data types in Java:

- int: $-2,147,483,648$ to $2,147,483,647$
- long: $-9,223,372,036,854,775,808$ to $9,223,372,036,854,775,807$
- float: $1.40239846 \times 10^{-45}$ to $3.40282347 \times 10^{38}$
- double: $4.94065645841246544 \times 10^{-324}$ to $1.79769313486231570 \times 10^{308}$

3 Features

3.1 AInteger Class

- Handles parsing and arithmetic for integers of any size.
- Each `AInteger` object stores its value as a string.
- Supports addition, subtraction, multiplication, and division.
- All operations are implemented digit-wise, similar to manual arithmetic.

3.2 AFloat Class

- Handles parsing and arithmetic for floating-point numbers of arbitrary precision.
- Each `AFloat` object stores its value as a string.
- Supports addition, subtraction, multiplication, and division up to 30 digits precision.
- Operations work by aligning decimal points, converting to scaled integers, and reinserting the decimal in the result.
- Uses `AInteger` internally for core arithmetic on the integer representations.

3.3 build.xml

- A build file that uses Apache Ant to automate the compilation and packaging of the Java project.
- The `clean` target deletes any previously compiled files and JARs for a fresh build.
- The `jar` target, which depends on `compile`, packages the compiled classes into a JAR file named `aarithmetic.jar` inside the `arbitraryarithmetic` folder and removes the temporary build directory afterward.

3.4 aarithmetic.jar

- A compiled Java archive (`.jar`) file that contains the `AInteger` and `AFloat` classes used for arbitrary-precision arithmetic.
- It was generated using the Apache Ant build system via the `build.xml` file.
- The `.jar` file is used as a dependency during compilation and execution of the main program.

3.5 MyInfArith.java

- The main driver class that accepts command-line arguments to perform arbitrary-precision arithmetic operations.
- Expects exactly four arguments: the number type (`int` or `float`), the operation (`add`, `sub`, `mul`, `div`), and two operands.
- Demonstrates the functionality of the arbitrary precision classes by providing a simple CLI interface.

3.6 codeRunner.py

- A Python script that acts as a wrapper to compile and run the Java class `MyInfArith` from the command line.
- Accepts four command-line arguments: number type (`int`/`float`), operation (`add`/`sub`/`mul`/`div`), and two operands.
- Simplifies testing by automating the build and execution steps from a single Python command.

4 Logic

4.1 AInteger Class

4.1.1 Input Validation

The `inputChecker()` method validates integer numbers by:

- Permitting an optional leading minus sign
- Rejecting invalid characters or decimals

4.1.2 Addition

- Align the numbers by padding with zeros to the right
- Perform digit-by-digit from right to left. Carries are handled as in manual addition
- Handle negative operands via reducing to subtraction

4.1.3 Subtraction

- Align the numbers by padding with zeros to the right
- Subtraction is also performed digit-wise, borrowing as required
- Handle negative operands via reducing to addition or reversing and subtracting

4.1.4 Multiplication

- Multiplication uses the classical long multiplication algorithm
- Each digit of one number is multiplied by each digit of the other, with results appropriately padded with zeros and summed continuously using the `AInteger.add()` method
- Negative numbers are handled by storing whether both the numbers are negative or only one is negative and accordingly appending − to the beginning of the product.

4.1.5 Division

- Division is implemented using a digit-wise approach similar to manual long division
- The algorithm builds the quotient one digit at a time by using `AInteger.mul()` and `AInteger.sub()` to find which digit (0 to 9) to append.
- The function terminates when the divisor becomes greater than the dividend
- Negative numbers are handled by storing whether both the numbers are negative or only one is negative and accordingly appending − to the beginning of the product.

4.1.6 Handling Signs and Zeros

- Preserve sign based on operand parity
- Removing leading zeros

4.2 AFloat Class

4.2.1 Input Validation

The `inputChecker()` method validates floating-point numbers by:

- Allowing at most one decimal point
- Permitting an optional leading minus sign
- Rejecting invalid characters or multiple decimals

4.2.2 Addition

- Align decimal points by padding with trailing zeros
- Remove decimals and treat as scaled integers
- Use `AInteger.add()` on the scaled values
- Reinsert decimal at the aligned position
- Handle negative operands via reducing to subtraction

4.2.3 Subtraction

- Align decimal points by padding with trailing zeros
- Convert to scaled integers and use `AInteger.sub()`
- Reinsert decimal at the aligned position
- Trim trailing zeros while preserving decimal precision
- Handle negative operands via reducing to addition or reversing and subtracting

4.2.4 Multiplication

- Remove decimals and count total fractional digits ($d_1 + d_2$)
- Multiply scaled integers using `AInteger.mul()`
- Insert decimal $d_1 + d_2$ places from the right
- Negative numbers are handled by storing whether both the numbers are negative or only one is negative and accordingly appending $-$ to the beginning of the product.

4.2.5 Division

- Scale numerator and denominator to integers by shifting the decimal point right until they become integers
- Perform long division similar to `AInteger.div()` operations but instead continue to append zeros to dividend for ensuring precision control (Default: 50 digits which get trimmed to 30)
- Insert decimal by using `AInteger.div()` to figure out where to insert
- Negative numbers are handled by storing whether both the numbers are negative or only one is negative and accordingly appending $-$ to the beginning of the product.

4.2.6 Handling Signs and Zeros

- Preserve sign based on operand parity
- Normalize results by:
  - Removing leading zeros before the decimal
  - Trimming trailing zeros after the decimal (And other digits if necessary to ensure only 30 digits after decimal point)
  - Ensuring single zero before decimals (e.g., `0.123`, not `.123`)
  - Ensuring single zero after decimals (e.g., `123.0`, not `123`)

5 Testing

Correctness of the functions of these classes was mainly tested manually, with comprehensive test cases for both integer and float operations.

5.1 AInteger

1. $80043 + 230 = 80273$
2. $-3 + 4 = 1$
3. $2987112 - 300292 = 2686820$
4. $11011 - (-29382) = 40393$
5. $2402726 * 8220887765 = 19752540776047390$
6. $8219 * (-29882) = -245600158$
7. $2987611761/1876155 = 1592$
8. $-7627221/(-887762) = 8$
9. $2/8 = 0$
10. $32/0 =$ Exception (Division by zero error)
11. $2.0 + 82 =$ Exception (Invalid input)
12. $20 - 20abc =$ Exception (Invalid input)

5.2 AFloat

1. $20.19862 + 198761.122111 = 198781.320731$
2. $-0.199211 + (-12) = -12.199211$
3. $90938281.19288128 - 12098871.1209838211 = 78839410.0718974589$
4. $20938832.20108 - 1000009009.0000900000 = -979070176.79901$
5. $99873.223232000000 * 1000992.1 = 99972307456.7684672$
6. $-1.01998881 * 091.988119 = -93.82685203294839$
7. $3000.109828/84.233 = 35.616798974273740695451901273847$
8. $-0.02288029/-0.00000009298882099098000 = 246054.20045297067362630391319951525 4$
9. $0.08828882/0.00000 =$ Exception (Division by zero error)
10. $-11877.0 - 92 + 297772.1 =$ Exception (Invalid input)
11. $230988.12345 * 2345.233.32 =$ Exception (Invalid input)
12. $123456789.987654321/\text{abcdefghi.ihgfedcba} =$ Exception (Invalid input)

All these answers can be verified to be correct.

6 Build and Run

6.1 Building the package (.JAR):

- `ant clean` followed by `ant`

6.2 Executing via MyInfArith:

- `javac MyInfArith.java`
- `java MyInfArith <int/float> <add/sub/mul/div> <operand1> <operand2>`

6.3 Executing via codeRunner.py:

- `python codeRunner.py <int/float> <add/sub/mul/div> <operand1> <operand2>`

6.4 Executing via aarbitraryarithmetic.jar:

- `javac -cp .:<add the absolute path to the .jar package here> <relative path to your desired file>`
- `java -cp .:<absolute path to .jar package> <relative path to your file>`