Arbitrary Precision Arithmetic Library
Software Development Fundamentals (SDF) Project
CS1023

Prashant - CS24BTECH11028

May 2025

Contents

1 Introduction

This report describes the design, implementation, and testing of an arbitrary-precision arithmetic library developed for the Software Development Fundamentals (SDF) course (CS1023, Jan–May 2025). The project includes a Java package called arbitraryarithmetic, which has two main classes: AInteger for big integers and AFloat for big floating-point numbers. These classes support basic operations like addition, subtraction, multiplication, and division. The library can be used from the command line using the MyInfArith.java program, or packaged as a aarithmetic.jar file for use in other projects. This report explains the design choices, how the code was written, how to use the library, its limitations, how it was tested, and the main lessons learned during the project.

2 Design Decisions

The design of the arbitrary-precision arithmetic library focuses on keeping things simple, correct, and following object-oriented programming (OOP) rules. Here are the main design choices:

2.1 Class Structure

- **Package**: The library is organized in the `arbitraryarithmetic` package, containing `AInteger` and `AFloat` classes, ensuring modularity and reusability.
- **AInteger**: Represents arbitrary-precision integers, storing the number as a string to handle large values without relying on external libraries.
- **AFloat**: Represents arbitrary-precision floating-point numbers, also stored as strings, with operations split into integer and fractional parts.
- **MyInfArith**: A separate class in the default package for command-line execution, parsing inputs and invoking `AInteger` or `AFloat` operations.

2.2 Operation Implementation

- **Static Methods**: Arithmetic operations (`addition`, `subtraction`, `multiplication`, `division`) are implemented as static methods in `AInteger` and `AFloat`, called via `MyInfArith`. This deviates from the project requirement for operator overloading but simplifies the initial implementation.
- **String-Based Arithmetic**: Both classes use string representations and digit-by-digit arithmetic to avoid native type limitations, ensuring no round-off errors for integers and controlled precision (30 decimal digits) for floating-point division.
- **Negative Numbers**: Handled by checking for a leading minus sign, adjusting operations accordingly (e.g., redirecting to subtraction for mixed signs in addition).

2.3 Error Handling

- **Input Validation**: `MyInfArith` validates command-line arguments (type, operation, operands) and catches exceptions for invalid formats or division by zero.
- **Division by Zero**: Both classes throw an `ArithmeticException` with the message "Division by zero error," as required by test cases.

3 Implementation Details

The implementation consists of three main Java files, organized as follows:

3.1 AInteger.java

- **Constructors**: Default (initializes to "0"), string-based, and copy constructor.
- **Methods**: `parse` creates an instance from a string. Arithmetic operations (`addition`, `subtraction`, `multiplication`, `division`) process digits in arrays, handling carries and negative numbers.
- **Key Logic**: Uses digit-by-digit arithmetic (e.g., schoolbook addition/subtraction, long multiplication, iterative division) to ensure arbitrary precision.

3.2 AFloat.java

- **Constructors**: Default (initializes to "0.0"), string-based, and copy constructor.
- **Methods**: `parse` creates an instance. Arithmetic operations split numbers into integer and fractional parts, leveraging `AInteger` for core calculations.
- **Precision**: Division truncates to 30 decimal digits, with padding to align fractional parts.

3.3 MyInfArith.java

- **Main Method**: Parses command-line arguments, invokes `AInteger` or `AFloat` operations, and handles errors (e.g., invalid type, operation, or division by zero).
- **Error Handling**: Uses try-catch for `ArithmeticException` and `NumberFormatException`.

3.4 Build System

The project uses Ant for building. The `build.xml` file defines targets for compilation and packaging the library into `aarithmetic.jar`. Key commands include:

- Compile: `ant compile`
- Package: `ant jar`

4 Usage Instructions (README)

The following is an excerpt from the project's `README.md`:

```
# Arbitrary Precision Arithmetic Library
A Java library for arbitrary-precision arithmetic on integers and floating-point numb

## Compilation
Run: ‘ant compile‘
Package: ‘ant jar‘

## Command-Line Usage
Execute: ‘java -cp aarithmetic.jar MyInfArith <int/float> <add/sub/mul/div> <op1> <op
Example: ‘java -cp aarithmetic.jar MyInfArith int add 123 456‘
Output: ‘579‘

## Library Usage
Include ‘aarithmetic.jar‘ in your project to use ‘AInteger‘ and ‘AFloat‘ classes.
Example:
- Import ‘arbitraryarithmetic.AInteger‘
- Create instances: ‘AInteger a = AInteger.parse("123");‘
- Perform operations: ‘AInteger sum = AInteger.addition(a, AInteger.parse("456"));‘
- Output: ‘579‘

## Requirements
- Java 11 or higher
- Ant for building
```

5 Verification Approach

The implementation was tested against the provided test cases to ensure correctness:

- **Integer Addition**: `java MyInfArith int add 236500782249129494973109332402504293978326246711379838638440149 8`
  Output: `665898614873800632956973176417 48`
- **Integer Subtraction**: `java MyInfArith int sub 3116511674006599806495512758577577452423003463811444464538840 08`
  Output: `-546287306263397813379509411254 31`
- **Integer Multiplication**: `java MyInfArith int mul 1434416316044592994268069731232223017167694823904478474013730519`
  Output: `330162008905899217578310782382075660760972861550182008086155118`
- **Integer Division**: `java MyInfArith int div 879272636528306057983395052167721149383525361708964745499835 8`
  Output: `17804979`

- **Float Division**: `java MyInfArith float div 879272636528306057983395052167721 1.0 4938352536170896474549983 58`
  Output: `17804979.09146998930296115952008787853 3`
- **Division by Zero**: `java MyInfArith int div 20 0`
  Output: `Division by zero error`

Testing was automated using a Python script that compiles the project with Ant and runs test cases, ensuring consistent results across all provided inputs.

6 Limitations

- **Operator Overloading**: The implementation uses static methods . This simplifies the initial implementation but reduces OOP adherence.
- **Performance**: Digit-by-digit arithmetic is computationally expensive for very large numbers compared to using optimized libraries.
- **Input Validation**: Limited validation for malformed inputs (e.g., non-numeric strings or invalid floating-point formats) may cause unexpected behavior.
- **Floating-Point Precision**: While division truncates to 30 decimal digits, other operations may produce longer fractional parts, which are not consistently truncated.

7 Key Learnings

- **Arbitrary-Precision Arithmetic**: Gained a deep understanding of digit-by-digit arithmetic algorithms and their implementation challenges.
- **OOP Principles**: Learned the importance of encapsulation and modularity, though the use of static methods highlighted areas for improvement in operator overloading.
- **Tool Usage**: Mastered Ant for build automation, Git for version control, and LaTeX for professional documentation.
- **Testing and Debugging**: Developed skills in writing automated tests and handling edge cases (e.g., negative numbers, division by zero).
- **Team Collaboration**: Coordinated tasks effectively, using Git for collaborative development and ensuring consistent documentation.

8 Conclusion

The arbitrary-precision arithmetic library successfully implements the required functionality for integer and floating-point operations, meeting most test cases. While the use of static methods deviates from the operator overloading requirement, the implementation ensures correctness and arbitrary precision. Future improvements include adopting instance methods, optimizing performance, and enhancing input validation. The project provided valuable experience in OOP, software engineering tools, and large-scale arithmetic computation.