Arbitrary Precision Arithmetic Library in Java

Nayan Modak

May 2, 2025

Contents

1 Introduction

This report describes the development and design of an arbitrary-precision arithmetic library in Java, created for the Software Development Fundamentals (CS1023) course. The library features two main classes: `AInteger` and `AFloat`, which represent large integers and floating-point numbers, respectively. The goal of this library is to provide a simple and efficient solution for performing arithmetic operations on numbers of arbitrary size.
The range of standard data types in java are :

- int : -2147483648 to 2147483648
- long : -9223372036854775808 to 9223372036854775808
- float : 1.40239846e-45 to 3.40282347e+38
- double : 4.94065645841246544e-324 to 1.79769313486231570e+308

2 Project Objectives

The key objectives of this project are:

- Implement an arbitrary-precision arithmetic library that handles large integers and floating-point numbers.
- Provide basic arithmetic operations (addition, subtraction, multiplication, and division) for large numbers.
- Ensure that calculations are performed with high precision, even for numbers that exceed the standard limits of Java's native types.
- Implement custom exception handling for errors such as division by zero, invalid input, and arithmetic overflow.
- Design the library with modularity in mind, using object-oriented principles to ensure maintainability and extensibility.
- Provide a simple command-line interface (CLI) to interact with the library, allowing users to perform operations via a terminal.
- Develop unit tests to verify the correctness of the operations and edge cases.

3 Design Decisions

3.1 Class Structure

The library consists of two main classes:

- `AInteger`: This class handles the representation and operations for large integers. It allows for high-precision arithmetic on integer values without losing precision, even for extremely large numbers. The class contains two constructors isNegative of Boolean type which tracks the sign of the interger whereas other is String Vlaue which contains the value of the string.
- `AFloat`: This class handles the representation and operations for floating-point numbers. It supports arbitrary precision for both the integer and fractional parts, ensuring that operations on floating-point numbers are accurate to the desired precision. Similar to AInteger isNegative is adopted along with Integral part and fraction part seperately.

Both classes use string-based representations to avoid the limitations of native Java numeric types. Java's native types like `int` and `double` are limited in size and cannot handle extremely large numbers. By using strings, we can manipulate numbers digit by digit, ensuring that no precision is lost.

3.2 String-Based Representation

The library uses strings to represent numbers, which allows for the handling of arbitrarily large numbers. Each number is stored as a string of digits, and arithmetic operations are implemented by processing each digit individually. This approach avoids the overflow issues inherent in fixed-precision data types like integers and floating-point numbers in Java.

For example, the number `12345678901234567890` can be represented as a string and used directly in arithmetic operations, regardless of its size. This allows the library to handle numbers with thousands or even millions of digits.

3.3 Arithmetic Operations

The core operations supported by the library are:

- **Addition/Subtraction**: These operations are implemented by simulating manual addition and subtraction, taking care of carries and borrows as necessary. The process involves iterating through each digit from right to left, adding or subtracting corresponding digits and adjusting for carries or borrows.

- **Multiplication**: This operation is implemented using the standard long multiplication algorithm. Each digit of one number is multiplied by each digit of the other number, and the results are accumulated accordingly. This ensures that multiplication is performed accurately, even for very large numbers.
- **Division**: The division operation is implemented using the long division algorithm. The dividend is divided by the divisor digit by digit, and the quotient and remainder are computed step by step. The division process ensures that the precision is maintained throughout the calculation.

3.4 Exception Handling

The library includes custom exceptions to handle errors during arithmetic operations:

- `DivisionByZeroException`: This exception is thrown when an attempt is made to divide by zero. Division by zero is undefined in arithmetic, so the library raises an exception to prevent such operations.
- `InvalidInputException`: This exception is thrown for unsupported or invalid operations. For example, attempting to perform arithmetic with non-numeric input or unsupported operations would trigger this exception.

Custom exception handling ensures that errors are caught early and handled appropriately, providing better feedback to the user and maintaining the integrity of the calculations.

4 Implementation Details

4.1 AInteger Class

The `AInteger` class provides methods for parsing, storing, and performing arithmetic on large integers. The class supports:

- **Addition, Subtraction, and Multiplication**: The methods for these operations simulate manual calculation, ensuring that the operations are performed digit by digit.
- **Division**: The division operation uses a step-by-step long division algorithm to divide the number by another, returning both the quotient and remainder.
- **String-Based Representation**: The integer is stored as a string, allowing for manipulation of individual digits.

4.2 AFloat Class

The `AFloat` class handles floating-point numbers, which are represented as strings with both an integer and a fractional part. The class supports:

- **Addition/Subtraction/Multiplication/Division**: These operations are implemented by first aligning the decimal points of both numbers and then performing the operation on the integer and fractional parts separately.
- **Precision Handling**: The class maintains high precision, ensuring that operations on floating-point numbers remain accurate to a specified number of decimal places.
- **Rounding and Conversion**: The class includes methods to round results to a desired precision and to convert numbers between scientific and decimal notation.

4.3 Command-Line Interface (CLI)

A simple CLI is provided through the `MyInfArith` class. The CLI allows users to interact with the library by entering numbers and selecting operations from the terminal. The user can perform basic arithmetic operations, view results, and handle errors via exception messages.

5 Testing and Verification

5.1 Unit Tests

Unit tests were created using JUnit to verify the correctness of each method in the library. The tests ensure that:

- Arithmetic operations like addition, subtraction, multiplication, and division work correctly.
- Edge cases, such as division by zero, negative numbers, and very large numbers, are handled appropriately.
- Floating-point precision is maintained across all operations, and rounding is performed correctly when necessary.

6 Limitations

6.1 Limitations

The current implementation has a few limitations:

- **Performance**: Due to the use of string manipulation, the library may experience slower performance for very large numbers compared to native types. Optimizing performance could be a focus for future versions.
- **Memory Usage**: Storing numbers as strings increases memory usage, especially for very large numbers. A more memory-efficient representation might be necessary for extremely large numbers.
- **Limited Operations**: The library currently supports basic arithmetic operations. Future versions could include support for more advanced operations like exponentiation, logarithms, and trigonometric functions.

7 Conclusion

The arbitrary-precision arithmetic library provides a robust solution for performing high-precision arithmetic operations on very large integers and floating-point numbers. Using string-based representations, the library ensures that precision is maintained even for large inputs. The implementation is thoroughly tested, and future improvements could optimize performance and add additional functionality.

8 Key Learnings

- **String-Based Arithmetic**: One of the key learnings from this project was the importance of using string-based representations for arbitrary-precision arithmetic. This approach allows for accurate calculations on numbers of any size, overcoming the limitations of native data types.
- **Manual Arithmetic Simulation**: The implementation of manual algorithms for addition, subtraction, multiplication, and division was a valuable learning experience. It reinforced the importance of understanding the underlying mathematical operations and how to implement them at a low level.
- **Performance Trade-offs**: Although the library provides high precision, I learned that string-based arithmetic can be slower and more memory-intensive than using native data types. Performance optimization remains an area for future improvement.