Java Project

Prajin Uppala

May 2025

Contents

1 Introduction

The objective of this project is to create a Java library to support basic arithmetic operations on very long integers and floating-point numbers. It uses string-based representations to perform addition, subtraction, multiplication, and division with arbitrary precision.

- The project includes three main components: AInteger for large integer operations, AFloat for floating-point numbers, and MyInfArith, a command-line tool for running the library.
- includes scripts for easy compilation (via Ant) and execution (with Python).
- Git was used for version control. Docker was used in this project to ensure platform-independent execution and to simplify the setup process for users.

2 Structure

The project consists of three main classes: AInteger and AFloat, which handle arbitraryprecision arithmetic operations on integers and floating-point numbers respectively, and MyInfArith, which serves as the entry point of the application and coordinates user input with the appropriate arithmetic logic.

2.1 AInteger

The main methods of this class are:

- `parse(String)`: Converts a string into an `AInteger` object.
- `removeLeadingZeros(String)`: Removes leading zeros from the given string.
- `removeLeadingZeros(StringBuilder)`: Removes leading zeros from the given `StringBuilder` object.
- `add(AInteger)`: Checks the signs of both numbers and calls either `add_strings_integer` or `subtract_strings_integer`, using non-negative string representations.
- `subtract(AInteger)`: Similar to `add`, it checks the signs and calls the appropriate method (`add_strings_integer` or `subtract_strings_integer`) based on the signs, using non-negative strings.
- `add_strings_integer(String, String)`: Adds two non-negative integer strings and returns the result.
- `subtract_strings_integer(String, String)`: Subtracts two non-negative integer strings and returns the result.
- `multiply(AInteger)`: Multiplies two integers using the partial products method. It uses digit-wise multiplication and handles the sign of the product appropriately, based on the signs of the operands.
- `division(AInteger)`: Performs integer division by using the long division method.
- `printValue()`: Prints the integer stored in the object.

2.2 AFloat

The main methods of this class are:

- `parse(String)`: Converts a string into an `AFloat` object.

- `removeTrailingZeros(String)`: Removes trailing zeros and any trailing decimal point from a floating-point string.
- `add_Float(AFloat)`: Handles sign logic and performs addition by calling either `add_strings_floating` or `substract_strings_floating`.
- `substract_Float(AFloat)`: Handles sign logic and performs subtraction by calling either `substract_strings_floating` or `add_strings_floating`.
- `add_strings_floating(String, String)`: Adds two non-negative floating-point numbers using the `add_strings_integer` method from `AInteger`.
- `substract_strings_floating(String, String)`: Subtracts two non-negative floating-point numbers using the `substract_strings_integer` method from `AInteger`.
- `multiply_Float(AFloat)`: Multiplies two floating-point numbers by converting them to integers, multiplying using `AInteger`, and reinserting the decimal point.
- `division_Float(AFloat)`: Divides two floating-point numbers with arbitrary precision by removing decimal points, adjusting precision by appending zeros, using `AInteger.division`, and reinserting the decimal point.
- `truncateTo30DecimalPlaces(StringBuilder)`: Truncates the given StringBuilder to 30 digits after decimal point.
- `printValue()`: Prints the floating value of the given AFloat object.

2.3 MyInfArith

: A command-line tool that lets users run the library from the terminal. It takes four inputs: the type of number (`int` or `float`), the operation (`add`, `sub`, `mul`, or `div`), and the two numbers to use. It then calls either `AInteger` or `AFloat` to do the calculation, depending on the type. This makes it easy to test the library without writing extra Java code.

3 Arithmetic Explanation for Integers

3.1 Addition: `add_strings_integer(num1, num2)`

- Remove leading zeros from both numbers.
- Swap the strings if `num1` is shorter than `num2`.
- Add digits from right to left using a loop, with carry tracking.
- If a carry remains at the end, insert it at the beginning.
- Clean the final result by removing any leading zeros.

3.2 Subtraction: `substract_strings_integer(num1, num2)`

- Remove leading zeros.
- Swap `num1` and `num2` if `num2 > num1`, and mark result as negative.
- Subtract digits from right to left, borrowing when necessary.
- Continue subtraction through remaining digits of the longer number.
- Trim leading zeros and add a minus sign if the result is negative.

3.3 Multiplication: `multiply(other)`

- Handle signs to determine if the result should be negative.
- Remove leading zeros from both numbers.

- Initialize an array of size `len1 + len2` to store intermediate results.
- Multiply each digit of `num2` with each digit of `num1` and add the result to the appropriate position in the array.
- Carry over any excess to the left positions.
- Convert the result array to a string, skipping leading zeros.
- Add a minus sign if the result is negative.

3.4 Division: `division(other)`

- Handle and determine the sign of the result.
- Remove leading zeros from both numbers.
- Throw an exception if dividing by zero.
- Return `0` if `num1 < num2`; return `1` or `-1` if they are equal.
- Simulate long division by iteratively bringing down digits and subtracting the divisor.
- Track how many times the divisor fits into each portion of the dividend.
- Append each quotient digit accordingly and trim any leading zeros.
- Add a minus sign if the result is negative.

4 Floating-Point Arithmetic Explanation

4.1 Addition: `add_strings_floating(num1, num2)`

- Ensure both numbers have a decimal point (append `.` if missing).
- Normalize both numbers by removing the decimal and padding the fractional parts with zeros to equal length.
- Perform integer addition using `add_strings_integer`.
- Pad result if necessary to accommodate fractional length.
- Reinsert the decimal at the correct position.
- Return the result after removing any trailing zeros.

4.2 Subtraction: `substract_strings_floating(num1, num2)`

- Normalize both numbers to have equal fractional length.
- Remove decimal points and perform integer subtraction via `substract_strings_integer`.
- Handle negative results by inserting padding after the negative sign.
- Reinsert the decimal at the correct position.
- Remove trailing zeros from the final result.

4.3 Multiplication: `multiply_Float(other)`

- Ensure both numbers contain decimal points.
- Remove decimal points and convert to `AInteger`.
- Multiply using `multiply` method of `AInteger`.
- Reinsert decimal at the combined fractional length position.
- Pad and truncate to 30 digits after the decimal.
- Return the result as a new `AFloat`.

4.4 Division: `division_Float(other)`

- Normalize input by ensuring decimal points and removing them.
- Append sufficient zeros to the numerator to ensure 30-digit precision.
- Perform integer division via `division()` of `AInteger`.
- Pad and reinsert decimal point to ensure 30 fractional digits.

- Remove trailing zeros and return as an `AFloat`.

5 UML Class Diagrams

The UML class diagram illustrates the library's structure.

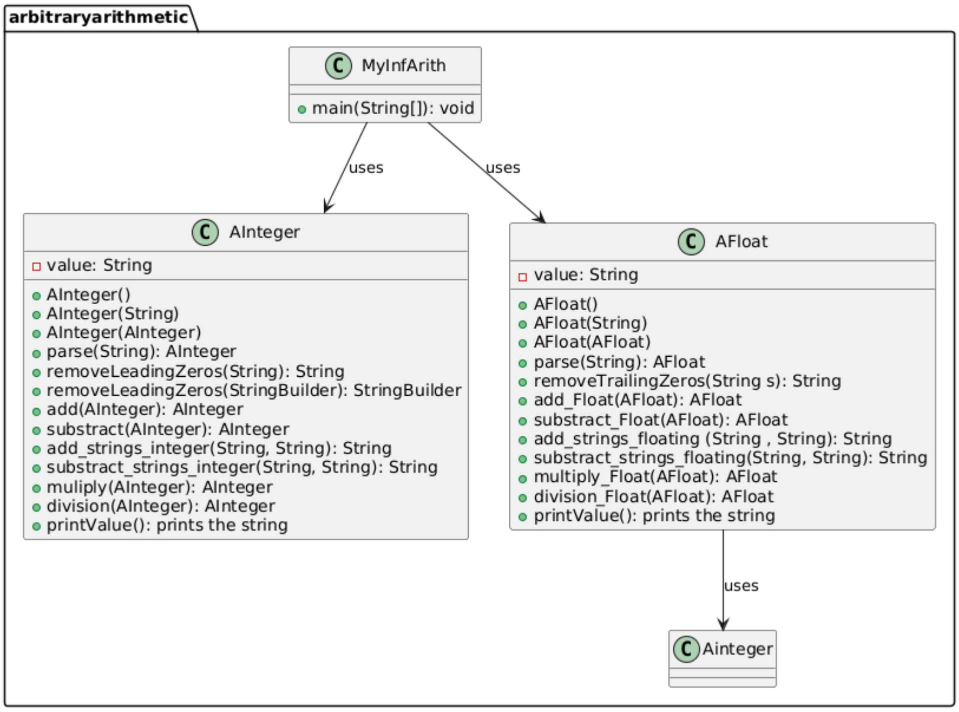
arbitraryarithmetic
C MyInfArith
main(String[]): void
uses
uses
C AInteger
value: String
AInteger()
AInteger(String)
AInteger(AInteger)
parse(String): AInteger
removeLeadingZeros(String): String
removeLeadingZeros(StringBuilder): StringBuilder
add(AInteger): AInteger
substract(AInteger): AInteger
add_strings_integer(String, String): String
substract_strings_integer(String, String): String
muliply(AInteger): AInteger
division(AInteger): AInteger
printValue(): prints the string
C AFloat
value: String
AFloat()
AFloat(String)
AFloat(AFloat)
parse(String): AFloat
removeTrailingZeros(String s): String
add_Float(AFloat): AFloat
substract_Float(AFloat): AFloat
add_strings_floating (String , String): String
substract_strings_floating(String, String): String
multiply_Float(AFloat): AFloat
division_Float(AFloat): AFloat
printValue(): prints the string
uses
C Ainteger

Figure 1: UML Class Diagram for Arbitrary Arithmetic Library

6 Usage of Library

6.1 Command Line Interface

The project provides a command-line interface through the `MyInfArith.java` class. This class parses the input arguments and invokes the appropriate methods in either the `AInteger` or `AFloat` classes.

Usage Pattern:

```
java MyInfArith <int/float> <add/sub/mult/div> <num1> <num2>
```

Examples:

- `java MyInfArith int add 164884844 -6846464646666`
  Output: The Answer is -6681579801822
- `java MyInfArith float div 244727.15202 75964.3891`
  Output: The Answer is 3.221603634537752111008551505615
- `java MyInfArith float sub 5.9585 -2.2565`
  Output: The Answer is 8.2150

6.2 MyInfArith.py

The Python script `MyInfArith.py` automates the compilation and execution process. It uses Apache Ant to compile the Java files and then executes the program with the provided input arguments.

Usage:

```
python MyInfArith.py <int/float> <add/sub/mult/div> <num1> <num2>
```

6.3 Build.xml

The `build.xml` file is the Ant build script that defines multiple targets for building and managing the project. Creates the `build` directory to store compiled class files.

- **clean:** Deletes all compiled files from the `build` directory.
- **compile:** Compiles all source files and outputs the class files into the `build` directory.

7 Conclusion

This arbitrary-precision arithmetic library provides a better solution for performing large number arithmetic operations in Java. By using string manipulation to handle large numbers, the library can support integers and floating-point numbers of arbitrary size.