SDF Project-1 Report

Sriraj Vasanta
CS24BTECH11066

May 2025

1 Introduction

The CS1023 course project involved developing a custom arbitrary-precision arithmetic library in Java, designed to deepen understanding of numerical computation beyond Java's built-in BigInteger and BigDecimal classes.
This library enables precise manipulation of integers exceeding Integer.MAX_VALUE ($2^{32} - 1$) and floating-point numbers with up to 30 decimal places. Unlike standard libraries, it prioritizes pedagogical value by requiring ground-up implementation of core algorithms and data structures for arbitrary-precision arithmetic

2 Design

2.1 Overview

- **Sign handling:** Sign has been handled separately by removing it at the start and then adding the proper sign at the end.
- **Input validation:** This involves checking if alphabets, more than two decimal points, or more that two minus signs are present in the input.
- **Exception handling:** The use of argprse library in MyInfArith.py handles most of the errors caused by invalid inputs. The other errors are handled manually.
- **Optimisation:** AInteger.java methods were invoked to solve the operations in AFloat.java. Division was also slightly optimized by comparing the string length.

2.2 Files description

- **AInteger.java -** Handles operations between arbitrarily large integers.
- **AFloat.java -** Handles operation between floating-points with arbitrary precision.
- **MyInfArith.py -** Imports both the classes to simpler use.

3 File Structure

```
Actual/
|-- arbitraryarithmetic/
|    |--> AInteger.java
|    |--> AInteger.class
|    |--> AFloat.java
|    |--> AFloat.class
|--reports/
|    |--> Report.pdf
|    |--> Report.tex
|-- .gitignore
|-- Dockerfile
|-- MyInfArith.java
|-- MyInfArith.py
|-- README.md
|-- build.xml
```

4 Implementation Logic

4.1 AInteger.java

4.1.1 Addition

- For numbers of the same sign, traditional digit-by-digit addition was implemented. Carries were properly managed and propagated to the subsequent digits.
- For numbers of the opposite sign the subtraction method of the class was used.[1]
- The final sign was determined and added after the numerical operation. For example, the sum of two negative numbers was computed as the sum of their absolute values, with a negative sign appended to the result.

4.1.2 Subtraction

- For numbers of the same sign typical digit-by-digit subtraction was implemented (properly storing the carry and using it for the next digits).
- For numbers of the opposite sign addition method was used.[1]
- As with addition, the final sign was determined at the end of the operation based on the original operands.

[1]This approach leverages Java's flexibility, which allows a method to be called before it is defined, provided it exists somewhere in the same file.

4.1.3 Multiplication

- The absolute values of the numbers were for the core multiplication logic. The sign of the final result was determined and stored separately (using a boolean flag), and applied after the multiplication.
- Traditional long multiplication was implemented: each digit of the second number was multiplied with the first number, and all partial products were accumulated appropriately.

4.1.4 Division

- The sign of the result was handled similarly to multiplication.
- Long division was implemented to perform the operation.
- Leading zeroes were removed before starting the operation to simplify calculations. For instance, if the length of the numerator is less than that of the denominator (after stripping leading zeroes), the result is directly returned as ‘0’.

4.1.5 UML diagram for AInteger.class

AInteger
`- int_str : String`
`+ AInteger()`
`+ AInteger(str_int : String)`
`+ AInteger(other_int : AInteger)`
`+ return_string() : String`
`+ add(other_int : AInteger) : AInteger`
`+ sub(other_int : AInteger) : AInteger`
`+ mul(other_int : AInteger) : AInteger`
`+ div(other_int : AInteger) : AInteger`
`+ static parse(str_int : String) : AInteger`
`# int_mod_compare(s1, s2) : int`
`# sub_helper_int(int_1, int_2) : String`
`# add_helper_int(int_1, int_2) : String`
`# mul_helper_int(int_1, int_2) : String`
`# div_helper_int(int_1, int_2) : String`
`# remove_leading_zeros(s) : String`

4.2 AFloat.java

4.2.1 Addition

- Both floating-point numbers were converted to integers by removing the decimal point and padding with appropriate zeroes. The resulting integers were then added using the method defined in AInteger.java.

- The sign of the result was determined and applied in the same manner as in AInteger.java.
- StringBuilder was used for efficient string manipulation (e.g., removing and re-inserting the decimal point), minimizing the creation of intermediate string objects.

4.2.2 Subtraction

- Similar to addition, both floats were treated as integers by removing the decimal point and padding with zeroes. The subtraction method from AInteger.java was then used.
- Final sign determination and immutability were handled consistently with the addition method.

4.2.3 Multiplication

- Decimal points were removed from both operands, converting them into integers. Multiplication was then performed using the method in AInteger.java.
- The operand with fewer digits after the decimal point was padded with zeroes to match the other. This facilitated accurate decimal placement in the final result.

4.2.4 Division

- Both floats were converted into integers by removing the decimal points and padding with zeroes. Division was then performed using the method from AInteger.java.
- Padding ensured equal decimal precision for easier placement of the decimal point in the quotient.
- A mechanism was implemented to handle 30-decimal precision in the decimal portion of the result.

4.2.5 UML diagram for AFloat.class

AFloat
`- float_str : String`

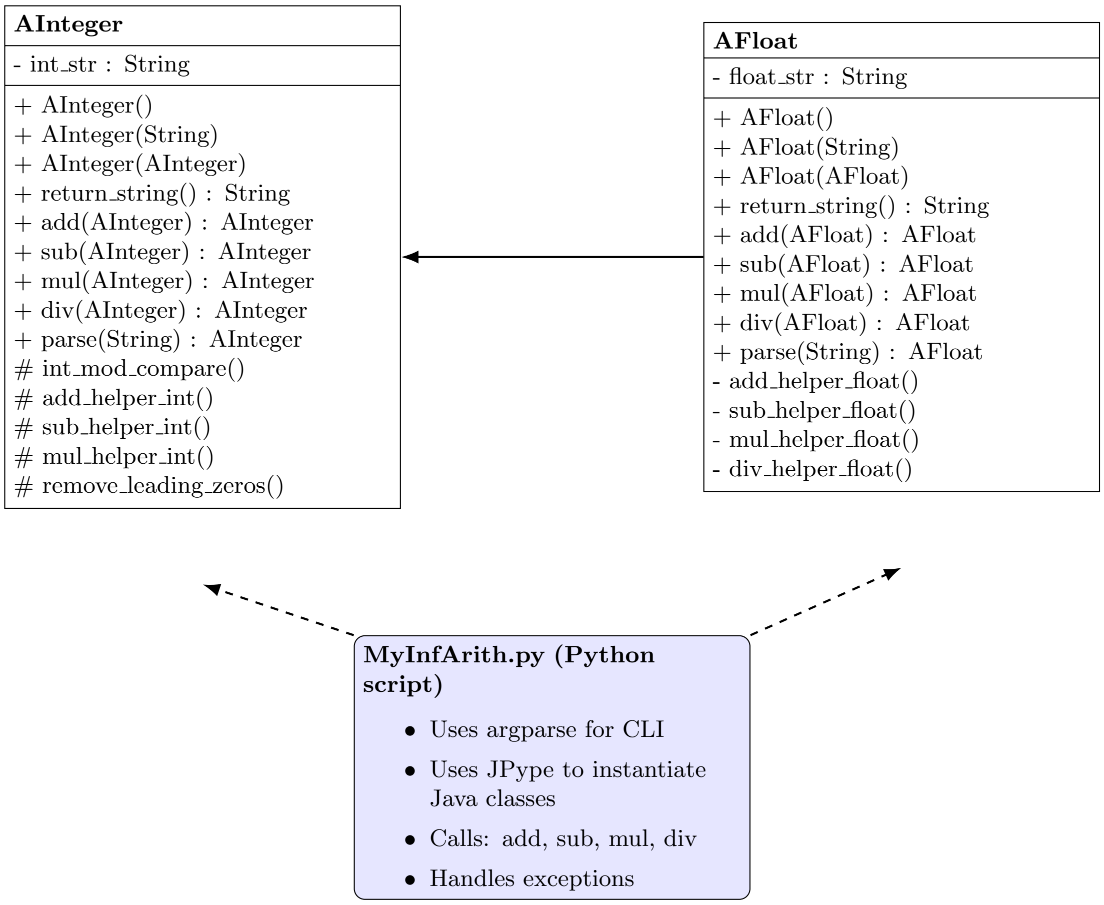
`+ AFloat()`
`+ AFloat(str_float : String)`
`+ AFloat(other_float : AFloat)`
`+ return_string() : String`
`+ add(other_float : AFloat) : AFloat`
`+ sub(other_float : AFloat) : AFloat`
`+ mul(other_float : AFloat) : AFloat`
`+ div(other_float : AFloat) : AFloat`
`+ static parse(str_float : String) : AFloat`

`# sub_helper_float(float_1, float_2) : String`
`# add_helper_float(float_1, float_2) : String`
`# mul_helper_float(float_1, float_2) : String`
`# div_helper_float(float_1, float_2) : String`

4.3 MyInfArith.py

- The argparse library was utilized to handle command-line arguments for the Python script. By leveraging the choices parameter, it significantly reduces the likelihood of invalid inputs and associated errors.
- JPype was employed to seamlessly import and interact with Java classes within the Python environment, enabling cross-language integration.
- While argparse's choices feature prevents many common input errors, additional validation was implemented to handle edge cases-such as verifying whether input strings can be correctly parsed as integers or floats.

5 UML diagram



6 README: Usage of the library

6.1 Overview

This project presents a custom Java library designed for arbitrary-length and high precision floating-point arithmetic, built from scratch without relying the Java's built-in classes.

6.2 Python CLI

Make sure that the current directory is the project directory terminal

```
cd projectdirectory
```

Run the build file using Python CLI

terminal

```
python MyInfArith int add 45 3
```

The output is as follows

terminal

```
48
```

6.3 Running through docker docs

Steps for using dockerhub:

- Use the dockerfile in the repo

  terminal

  ```
  docker build -t imagename .
  ```

  This opens a shell of the docker container

  terminal

  ```
  docker run -it imagename bash
  ```

- To push

  terminal

  ```
  docker tag imagename username/officialimagename:tag
  ```

  terminal

  ```
  docker push username/officialimagename:tag
  ```

7 Git Commits

terminal

```
* 1aa3857 (HEAD -> master, origin/master) Stop tracking
p.py
* f355022 Stop tracking p.py
* f54f243 Built a docker container
* 24dabab added ant and README.md
* 417fb1d Merge remote-tracking branch 'origin/master'
merging git
|
| * 4feb80a Create README.md
* | c71b5b2 added ant file
|/
* eea42ab Completed the automation
* 20e70e1 Modified AFloat to handle int operands
* 5546bc3 Appropriate operations done
* df1f2e6 Added required arguments and checking the
validity of inputs
* 6e76bbb Added the division by zero exception in AFloat
class
* 294e23c Tidied up the code a bit
* e799b10 Deleting the java automation files
* 6162083 Delete directory My_Project
* 2ab8b1d Python automation file added
* eaa5ed4 Comments added to AFloat.java, and its add and
sub methods were changed
* 125014e Comments added to AInteger.java
* e0db5c4 Float division method added
* 793ad72 Float multiplication method added
* fa0ae7e Float addition method added
* f7c4bc7 Float subtraction method added
```

8 Verification

- Manual testing of corner cases such as large integers beyond standard int range.
- Comparison with Java's BigInteger and BigDeciaml for validation.

9 Key Learnings

- Understanding of argparse library and its benefits , and Jpyp library used to bridge Python and Java.
- Working with **ant** automation system
- Using **Docker**
- Generating UML diagrams in LaTex.
- Basic idea of Markdown language.

10 Possible improvements

- Instead of fixing the precision to 30 decimals, taking an input for precision from the user would have made it more flexible.