Arbitrary Precision Arithmetic Library

Amogh Bindal
ES23BTECH11004

May 2, 2025

1 Introduction

While developing these files, my primary goal was to implement arbitrary-precision arithmetic for both integers and floating-point numbers in Java. I started with the AInteger class to handle large integers, including negative values, and then extended this functionality to floating-point numbers with the AFloat class.

For AInteger, I focused on representing numbers as strings to bypass the limitations of Java's built-in numeric types. I implemented methods for basic arithmetic operations-addition, subtraction, multiplication, and division-by simulating manual algorithms (similar to how calculations are done by hand). Special care was taken to manage leading zeros, negative numbers, and edge cases like division by zero. The class also includes utility methods for parsing and normalizing input.

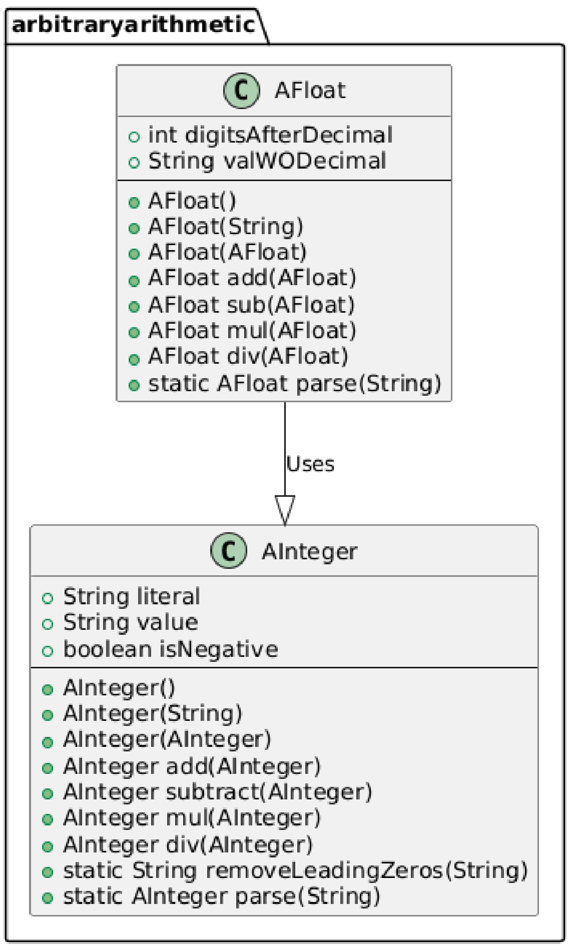
Building on AInteger, I created the AFloat class to support arbitrary-precision floating-point arithmetic. This class inherits from AInteger and adds fields to track the number of digits after the decimal point and the value without the decimal. Arithmetic operations in AFloat align the decimal points before performing calculations using the underlying integer logic from AInteger. The results are then formatted correctly with the decimal point. This design allows for precise calculations with very large or very small floating-point numbers, handling sign and scale adjustments as needed.

2 Design Overview

This project consists of two core classes: `AInteger` and `AFloat`, built to perform arbitrary-precision integer and floating-point arithmetic respectively. Key design features include:

- Support for large numbers without precision loss.
- Manual implementation of basic arithmetic: addition, subtraction, multiplication, and division.
- Separation of integer and decimal components in `AFloat` to aid high-precision operations.
- Used Basic Multiplicatoin and long Division algorithms to carry out the calculations

UML Diagram



3 How to Use the Library

3.1 How to run in a java file

```
1. Import the classes in your Java project:
   import arbitraryarithmetic.AInteger;
   import arbitraryarithmetic.AFloat;

2. Create objects:
   AInteger a = new AInteger("12345678901234567890");
   AFloat b = new AFloat("3.1415926535");

3. Use methods:
   AInteger result = a.add(new AInteger("100"));
   AFloat fresult = b.mulF(new AFloat("2.5"));

4. Run the main methods in each class for examples.
```

3.2 Running using Docker

To run the project using Docker, we first create a Docker image that uses Java 21 and Apache Ant. The Dockerfile compiles the Java code using Ant and sets the built JAR file as the container entry point.

- **Build the image:**

```
docker build -t myinfarith .
```

- **Pull from DockerHub**

```
docker image pull amoghbindal/myinfarith
```

- **Run the container:**

```
docker run -it amoghbindal/myinfarith int add 123 456
```

This command runs the image and gives us access to the interactive shell in the image.

3.3 Running using Python and CLI

A Python script named `run_myinfarith.py` is provided to compile the project using Ant and execute the generated JAR file with command-line arguments.

- **Run the program:**

```
python run_myinfarith.py float mul 3.14 2.0
```

This internally runs:

- `ant jar` to build the project.
- `java -jar dist/MyInfArith.jar` with provided arguments.

3.4 Running using Ant

The project can also be built and executed directly using Apache Ant. A `build.xml` file is included which automates the compilation and packaging.

- **Build the JAR file:**

```
ant jar
```

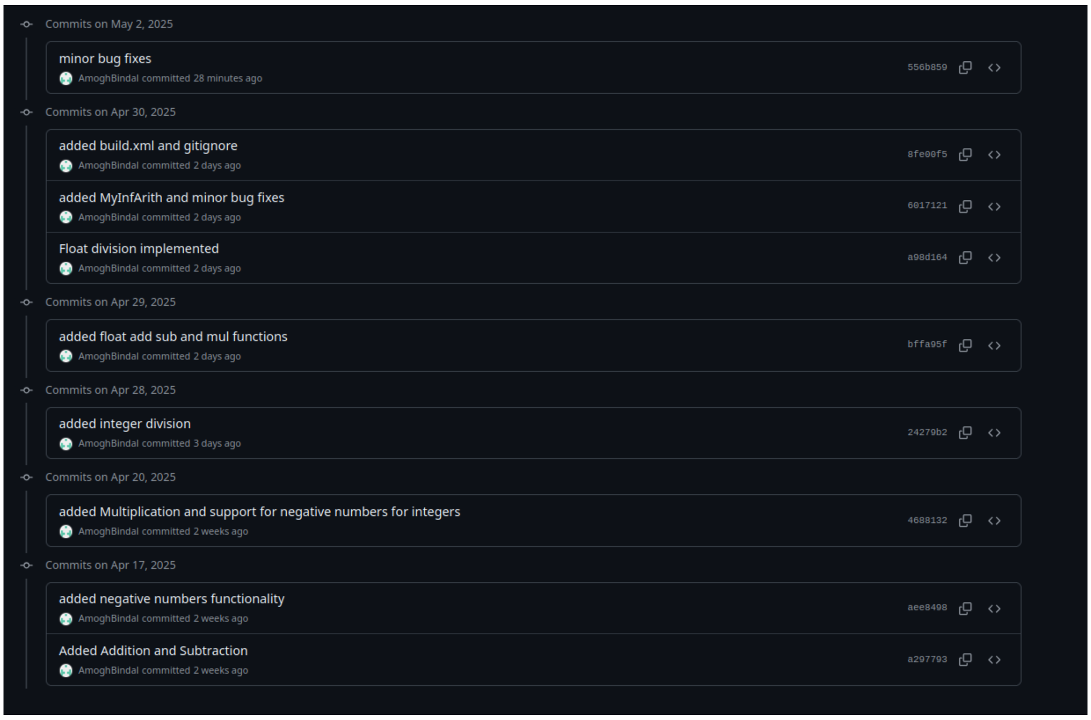
- **Run the program:**

```
java -jar dist/MyInfArith.jar int sub 100 45
```

The build process compiles all Java source files into the `build/` directory and creates a JAR file in the `dist/` directory.

4 Git Commit Snapshot

- Please refer to the included image `gitlog.png` for a snapshot of commit history.



```
Commits on May 2, 2025
  minor bug fixes                                                        556b859
  AmoghBindal committed 28 minutes ago
Commits on Apr 30, 2025
  added build.xml and gitignore                                          8fe00f5
  AmoghBindal committed 2 days ago
  added MyInfArith and minor bug fixes                                   6017121
  AmoghBindal committed 2 days ago
  Float division implemented                                             a98d164
  AmoghBindal committed 2 days ago
Commits on Apr 29, 2025
  added float add sub and mul functions                                  bffa95f
  AmoghBindal committed 2 days ago
Commits on Apr 28, 2025
  added integer division                                                 24279b2
  AmoghBindal committed 3 days ago
Commits on Apr 20, 2025
  added Multiplication and support for negative numbers for integers     4688132
  AmoghBindal committed 2 weeks ago
Commits on Apr 17, 2025
  added negative numbers functionality                                   aee8498
  AmoghBindal committed 2 weeks ago
  Added Addition and Subtraction                                         a297793
  AmoghBindal committed 2 weeks ago
```

5 Limitations

- Performance may degrade with extremely large inputs due to use of string-based arithmetic.
- Floating point division is manually padded for precision but not dynamically adaptive.
- There are better/faster methods to implement for division and multiplication
- No input validation for malformed strings beyond basic error checks.
- Leading zeros of decimal values not removed
- Can't change precision of the decimal easily and it is 30 rightnow which may not be enough sometimes9
- The function a.add(b) will not not update the value of a. (It is very easy to solve by using a set method to set the value of 2 objects to be equal we can set the attributes of the result after each operation equal to the instances attributes.)

6 Verification Approach

- Manual test cases written in `main()` methods for both classes.
- Verifications include negative inputs, large numbers, and zero-edge cases.

- The outputs of operations were cross-verified with Java's built-in `BigInteger` and `BigDecimal`.

7 Key Learnings

This project provided valuable insights into both software development and tooling practices. The key takeaways include:

- **Java Build Automation with Ant:** Using Apache Ant to compile and package the project helped in understanding XML-based build configuration, and how to automate repetitive compilation and packaging tasks.
- **Containerization with Docker:** Creating a Dockerfile to encapsulate the Java runtime, Ant environment, and project code emphasized the importance of reproducibility and portability in software deployment.
- **CLI Application Design:** Designing a command-line interface for invoking arithmetic operations improved understanding of argument parsing and user input validation in Java.
- **Cross-language Integration:** Writing a Python script to build and run the Java project demonstrated how scripting languages can be used to automate and streamline workflows.
- **Version Control Best Practices:** Using Git to track changes across multiple files and features improved code organization and encouraged modular, incremental development.