SDF Java Project Report

CS24BTECH11036
Krishnan Ramanarayanan

April 2025

1 Aim of the project

The aim of my project is to implement an arbitrary precision arithmetic calculator.

2 My Approach

2.1 BigInteger

I have implemented a Java class which implements arithmetic operations on large integers stored in the string format.

- Definition of BigInteger.ZERO is present here
- Integers are stored as string and the operations are done on these strings to support large numbers.
- It is present in the same folder as the AFloat class code.

2.2 AInteger class

I have built a class called `BigInteger` to handle operations for very large integers. This class is defined in the same folder as the AInteger.

- It stores a big number inside using `BigInteger`.
- It lets you create objects in different ways:
  - Default constructor assigns the value to 0.
  - From a number written as a string
  - From another AInteger object
- It has math functions:
  - Add (`add`)

  - Subtract (`sub`)
  - Multiply (`mul`)
  - Divide (`div`) – checks for divide by zero
- It has extra functions to:
  - Get or set the value
  - Show the number as a string

2.3 BigDecimal

I have implemented a Java class which implements arithmetic operations on large floats stored in the string format.

- Definition of BigDecimal.ZERO is present here
- Floats are stored as string and the operations are done on these strings to support large numbers.
- It is present in the same folder as the AFloat class code.
- BigDecimal uses the functions of BigInteger to perform arithmetic operations.

2.4 AFloat class

`AFloat` is a Java class that helps you work with very big or very small decimal numbers. It uses the `BigDecimal` class which I have built inside the same folder to do this.

2.4.1 Why use AFloat?

Sometimes, numbers need to be very accurate, like in scientific or financial calculations. All the math is done with very high precision (up to 1000 digits).

2.4.2 What can AFloat do?

- It stores a decimal number safely.
- It lets you:
  - Initializes with 0,
  - Start with a number written as a string (like `"3.14159"`),
  - Copy the value of another object of the class `AFloat`.
- It has math functions:
  - `add` – adds two numbers,

  - `sub` – subtracts them,
  - `mul` – multiplies them,
  - `div` – divides them (but checks for divide by zero).

- It also has:
  - `getVal()` – get the number,
  - `setVal()` – change the number,
  - `toString()` – show the number in the String format.

2.5 MyInfArith class

`MyInfArith` is a Java class that helps us do math operations (like addition, subtraction, etc.) on very big or very small numbers. It works for both:

- **Integers** (whole numbers),
- **Decimals** (floating point numbers).

It uses other classes called `AInteger` and `AFloat` to do the actual math.

2.5.1 How does it work?

This class has one main method called `MyInfArithCalc`. It takes four words (called strings) as input:

- `type` – is it an `int` or a `float`?
- `operand` – what operation to do: `add`, `sub`, `mul`, or `div`.
- `value1` – the first number.
- `value2` – the second number.

It then:

1. Figures out if the type is `int` or `float`.
2. Creates two numbers using `value1` and `value2`.
3. Checks what operation to do (`add`, `sub`, etc.).
4. Performs the operation.
5. Returns the answer as a string.

2.5.2 What about errors?

- If you try to divide by zero, it gives an error message: "Division by zero error".
- If you type something wrong (like a wrong operator), it gives help text showing the correct way to use it.

2.5.3 What does the main method do?

The `main` method:

- Checks if the number of inputs is exactly 4. If not, it shows how to use the program.
- If inputs are okay, it calls `MyInfArithCalc` with those inputs.
- Prints the result.

2.6 python script

This Python script is used to run a Java program called `MyInfArith`. The Java program does big number math operations like add, subtract, multiply, and divide.

Why use this script?

Instead of typing long commands in the terminal to build and run the Java code, we can use this Python script to do everything in one go!

Steps the script follows

Step 1: Build the Java project

- It uses a tool called `Ant` to build a JAR file (which is a Java program in a packaged format).
- If the build fails, it shows an error message.

Step 2: Check the inputs

- If you pass 0 arguments, it will build the jar file and then run test cases present under MyInfArithTest
- The script expects 4 inputs from the command line:
  - The type of number (`int` or `float`),
  - The operation (`add`, `sub`, `mul`, or `div`),
  - The first number,
  - The second number.
- If you don't give exactly 0 or 4 inputs, it shows how to use the script and exits.

Step 3: Run the Java program

- If everything is okay, it runs the Java program with the inputs you gave.

Step 4: Show the result

- If the Java program has an error, it shows an error message.
- If it works, it shows the answer of the math operation.

2.7 Ant Script: build.xml

Goal

We want to take our Java code and turn it into a **JAR file** so it can be easily run. We use a tool called **Ant** to help us do this.

Project Setup

- The project is called `arbitraryarithmetic`.
- The main Java file we want to run is `MyInfArith`.
- The output file we want is a JAR file named `aarithmetic.jar`.

Important Folders

- `src/main/java`: Where our Java source code is.
- `build/classes`: Where the compiled `.class` files will go.
- `build/jar`: Where the final JAR file will be saved.

What Each Step Does

1. `init` – Create folders

This step makes sure that the folders for compiled code and the JAR file exist.

2. `compile` – Compile Java code

This step takes all the Java files from `src/main/java` and compiles them (turns them into `.class` files). It puts the compiled files in `build/classes`.

3. `jar` – Make the JAR file

Now that we have compiled code, we package it into a JAR file called `aarithmetic.jar`, and we tell it which class to run first (`MyInfArith`).

4. `clean` – Clean up

This step deletes the `build` folder and everything inside it. We use this when we want to start fresh.

3 Design Section

Design Decisions

The library was designed with the aim of providing a highly precise and flexible way to perform arithmetic operations on arbitrarily large or small integers and floating-point numbers. The primary design decisions were:

- **Wrapper Classes:** Two custom wrapper classes, `AInteger` and `AFloat`, were created to encapsulate `BigInteger` and `BigDecimal`, respectively. This abstraction allows for cleaner code.
- **Operation Handling:** The operations (addition, subtraction, multiplication, and division) are performed using a switch-case structure, which makes it easy to extend or modify the operations supported by the library.
- **Precision Handling:** For floating-point numbers, we ensured that operations are computed up to 30 decimal places, though it can handle more.
- **Error Handling:** Proper error messages were added for edge cases, such as division by zero. Invalid input handling is done via default messages, guiding the user to correct usage.
- **Command-Line Interface:** The program accepts four command-line arguments, including the type of number, the operation, and two operands. This provides flexibility for running operations directly from the command line or through a Python wrapper.

Class Descriptions:

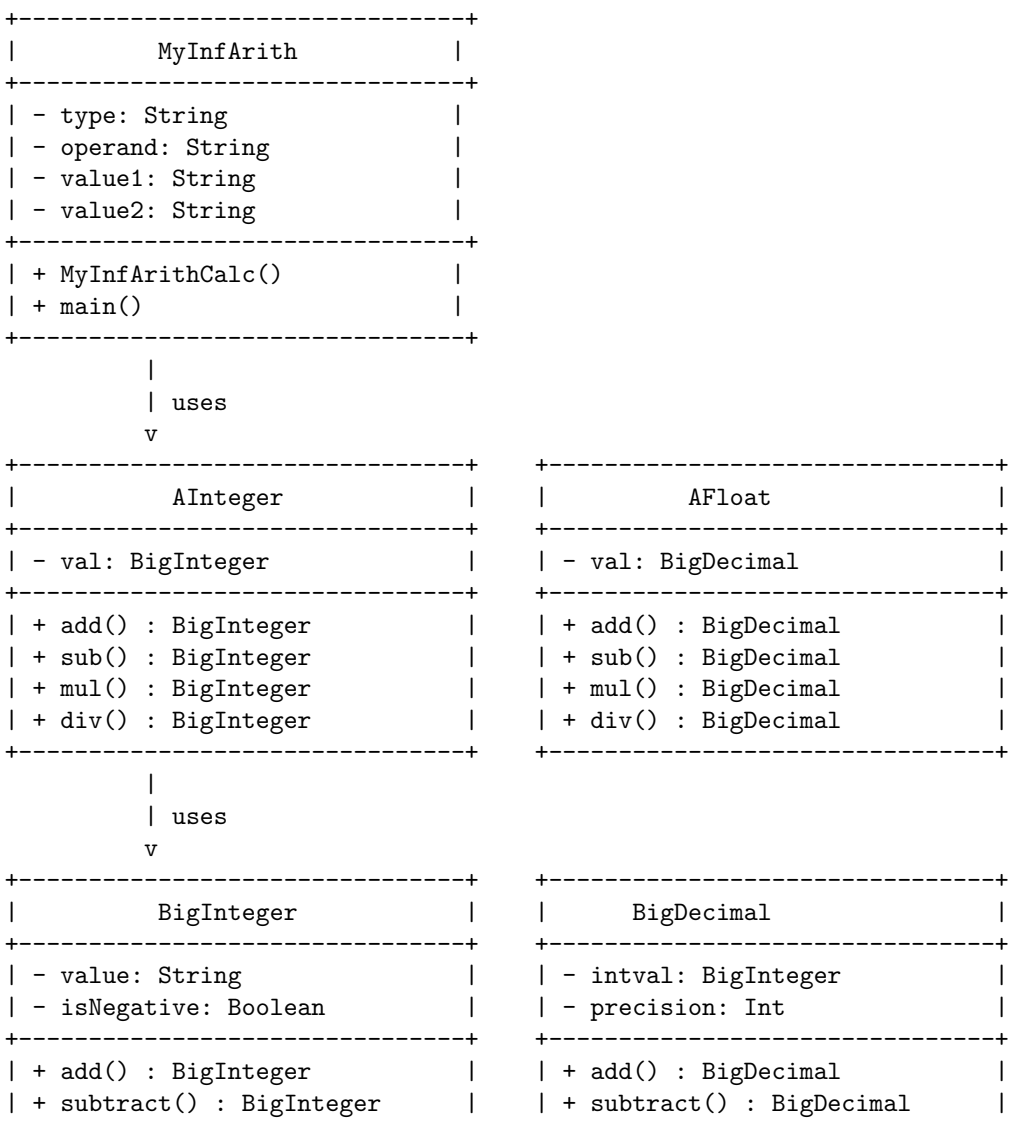
- `MyInfArith`: The main class responsible for accepting user input, performing the appropriate operation, and displaying the result. This class uses the `AInteger` and `AFloat` classes based on whether the operation is for integer or floating-point numbers.
- `BigInteger`: Implements arithmetic operations such as addition, subtraction, multiplication, and division for large integers in the string format.
- `BigFloat`: Implements arithmetic operations such as addition, subtraction, multiplication, and division for large decimals in the string format.
- `AInteger`: A wrapper class for `BigInteger`. This class encapsulates the `BigInteger` and provides methods for arithmetic operations such as addition, subtraction, multiplication, and division.
- `AFloat`: A wrapper class for `BigDecimal`. This class performs similar operations as `AInteger` but for floating-point numbers. Precision control is implemented in this class for decimal arithmetic.

- `MyInfArithTest`: A test class that verifies the correctness of the library by running predefined test cases and comparing the output against expected results.

- `python`$_{s}cript.py$: $APythonwrapperforeaseofusage, providingasimplifiedinterfacetoruntheJavacodefroma$

UML Class Diagram

The diagram below illustrates the relationships and methods within the main classes:



```
| + multiply() : BigInteger      |    | + multiply() : BigDecimal      |
| + divide() : BigInteger        |    | + divide() : BigDecimal        |
+--------------------------------+    +--------------------------------+
```

4 README

Overview

This library provides support for performing arithmetic operations on arbitrarily large or small integers and floating-point numbers.

1. Compilation and Setup

The project uses Apache Ant for build management. To compile the project and generate a runnable JAR file:

1. Ensure Apache Ant is installed and set up.
2. Run the following command in the root directory (where `build.xml` is located):

   ```
   ant jar
   ```

3. This will generate a file `aarithmetic.jar` in the `build/jar` directory.

2. Running the Program

To run the program using the generated JAR file, use the command:

```
java -jar build/jar/aarithmetic.jar <type> <operation> <operand1> <operand2>
```

Parameters:

- `<type>` — `int` for integers or `float` for floating-point numbers
- `<operation>` — `add`, `sub`, `mul`, or `div`
- `<operand1>` — the first number (as a string)
- `<operand2>` — the second number (as a string)

Example:

```
java -jar build/jar/aarithmetic.jar float div 25.5 4.2
```

3. Python Wrapper (Optional)

A helper Python script is provided to simplify execution:

```
python python_script.py <int/float> <add/sub/mul/div> <operand1> <operand2>
```

Make sure the Python script is placed in the correct directory and Java is installed on your system. This python script also builds the jar file using the ant file,

4. Notes

- Floating-point operations are computed up to a precision of 30 decimal places.
- Inputs must be valid numbers in string format.
- Supported operations: `add`, `sub`, `mul`, `div`.

5 Git Commits - Snapshots

6 Docker

7 Limitations of the Library

- **No Support for Complex Numbers:** The library only handles integers and floating-point numbers, not complex numbers (numbers with real and imaginary parts).
- **Basic Arithmetic Only:** It supports only four operations: addition, subtraction, multiplication, and division. There is no support for other mathematical functions like square root, power, logarithm, etc.
- **No Error Handling for Input Format:** If a non-numeric string is passed as input, the library will throw a runtime error. There's no built-in input validation or friendly error message for that.
- **Precision Limitation is Hardcoded:** The floating-point precision is fixed at 30 digits. There's no way to change this dynamically based on the user's need.
- **Slow Performance:** As my code uses strings to perform operations, it has not been optimized for performance.

8 Verification Approach

To verify the correctness of the arithmetic operations implemented in the `MyInfArith` library, a separate test class named `MyInfArithTest` was created. This class was used to systematically run and validate different test cases covering both integer and floating-point operations.

Testing Methodology

- The test class uses a method called `runtests()`, which takes a 5-element string array as input. The first four elements represent the inputs to the calculator function:
  - The number type (`int` or `float`)
  - The operation to perform (`add`, `sub`, `mul`, `div`)
  - The first operand
  - The second operand

  The fifth element is the expected result.
- The function internally calls the `MyInfArithCalc()` method and compares the actual result against the expected result.
- If the results match, it prints `Success`; otherwise, it prints an error message showing the input values, expected result, and actual output. This helps in debugging incorrect results or mismatches.

Test Case Coverage

- The test cases include addition, subtraction, multiplication, and division for both integers and floating-point numbers.
- Edge cases are also considered, such as:
  - Division by zero (for both `int` and `float`)
  - Division resulting in 0 (e.g., 25 divided by 123)
  - Floating-point precision checks for values with many decimal places
- Integer operations use very large numbers to test the capability of `BigInteger`.
- Floating-point operations check if `BigDecimal` handles high-precision outputs accurately.

9 Key Learnings from the overall project

- **Building BigInteger and BigDecimal:** Learned how to build my own `BigInteger` and `BigDecimal` classes to handle very large or very precise numbers without losing accuracy.
- **Object-Oriented Programming (OOP):** Implemented OOP concepts such as encapsulation (using private variables), constructors (including copy constructor), and method overriding (like `toString`).
- **Class Design for Arithmetic:** Designed reusable classes (`AFloat` and `AInteger`) that wrap around Java's core libraries, making arithmetic easier and more controlled.
- **Command-line Java Execution:** Learned how to pass command-line arguments into a Java program and handle them effectively.
- **Automation with Ant:** Gained hands-on experience using Apache Ant to automate the build process, compile code, and create runnable JAR files.
- **Interfacing Java with Python:** Explored how to run Java programs from Python scripts, which is useful for integrating multiple technologies.
- **Software Packaging:** Understood how to structure a Java project with separate folders for source code, build files, and JAR outputs.