Arbitrary Precision Arithmetic Library in Java

Maka Nehith

CS24BTECH11040

Abstract

This project presents the implementation of an arbitrary precision library in Java, designed to handle arithmetic operations involving large numbers which cannot be stored in primitive data types. The library consists of two Classes- AInteger and AFloat -which handle integer and floating point arithmetic respectively. The input numbers are stored as strings, allowing the library to process large numbers accurately without loss of precision. It supports addition, subtraction, multiplication and division of two numbers, which are implemented through algorithms that mimic the process used by humans to perform the arithmetic operations.

1 Introduction

In this project, implementation of arbitrary precision arithmetic library has been done in Java. It stores the input numbers as strings to avoid the loss of precision and supports addition, subtraction, multiplication and division on these numbers depending upon their type which can be either int or float. The two classes - AInteger and AFloat - are packaged into a jar file, facilitating the distribution and deployment of the library.

2 Requirements and Tools

This project has been implemented in the Java programming language. A package named arbitraryarithmetic has been created that includes the two class files, AFloat.class and AInteger.class. An Ant build.xml file has been written which automates compilation of the .java files and packages the class files into an aarithmetic.jar file, making it into a distributable library. For evaluation of the library, MyInfArith.java file and a Python script named run.py , have been written. The aarithmetic.jar file which can be created using the build.xml file, can be used for performing arithmetic operations on big numbers.

3 Implementation Details of AInteger class

This class performs arithmetic operations on integers. It supports addition, subtraction, multiplication, and division operations by taking numbers as input in their string representation. The class has a private attribute, **number**, which stores the string representation of the integer. A getter method, getInteger() can be used to access the attribute **number**. This attribute can be initialized during the instantiation of the object. If not provided, the default constructor initializes it to the string "0". A copy constructor has been included which can be used to create a copy of an existing AInteger object. The arithmetic operations have been implemented as static methods so that they can be used directly without instantiating an object. Many helper methods have been included in this class. The methods that perform the actual arithmetic operations make use of this helper functions to carry out the desired operation.

3.1 Constructors

The AInteger class includes several constructors, each designed for different use cases as described below.

- **Default constructor AInteger()** - Initializes an instance of this class with default number 0. In other words, the attribute, **number**, gets initialized to the string "0".
- **Constructor AInteger(String s)** - Initializes an instance of this class with the provided string **s**, as the integer in its string representation. That is, the attribute, **number**, is initialized to the provided string **s**.
- **Default constructor AInteger(AInteger AIntegerobject)** - creates an instance of the AInteger class by assigning the value of attribute, **number**, of the AIntegerobject to the instance's **number** attribute, thereby creating a copy of the AIntegerobject.

3.2 Helper Methods

The functionality of each helper method included in the AInteger class has been utilized multiple times throughout the class. As a result, each required functionality is implemented as a separate method, thereby modularizing the code in AInteger class. Each helper method has been made private, since it is called only by the methods declared inside the class.

The functionality of each helper method declared inside the AInteger class is as described below.

- **parse(String s)** – returns an instance of the AInteger class with its attribute, **number**, initialized to the string **s** using the constructor AInteger(string s). In other words, it converts the string **s** into an AInteger object.

- **Addition(String s1, String s2)** – This method performs the addition of two positive integers **s1** and **s2**.This method implements the algorithm for adding two positive integers represented as strings. The addition is performed by iterating over the strings **s1** and **s2** from right to left. At each step, corresponding digits are added along with any carry (excess) from the previous step. The last digit of the resulting sum is added to the start of the result, while the remaining part (if any) is carried over to the next iteration. This continues until both strings have been fully traversed. If one string is longer than the other, its remaining digits are processed in the same way–adding the carry to each digit, appending the last digit of the sum to the result, and updating the carry accordingly. After all digits are processed, any remaining carry is added to the result. Finally, the method returns the **string result**. This method is made static so that it can be used in the actual add method(addition of two integers) which is static.

- **subtract(String s1 , String s2)** – This method performs the subtraction of two positive integers, with the precondition that s1 represents a greater integer than s2. The subtraction is carried out by iterating over the strings s1 and s2 from right to left. At each step, corresponding digits are subtracted, and the difference is prepended to the result. If a digit in s1 is smaller than the corresponding digit in s2, it borrows from the preceding digit in s1. A flag is maintained to track whether a borrow was already taken from a digit, ensuring that the digit providing the borrow is decremented appropriately. This process continues until all digits of s2 have been processed. If s1 has remaining digits, they are also processed similarly. If a digit in s1 is negative due to a previous borrow, another borrow is taken from the digit

before it (adding 10 to the current digit), and the flag is updated accordingly. Once all digits are processed, the final result string is returned. This method is made static so that it can be used in the actual sub method(subtraction of two integers) which is static.

- **isGreater(String number1, String number2)** – This is a comparison function that checks whether number1 is greater than or equal to number2, returning true if number1 is greater than or equal to number2, and false otherwise. The inputs number1 and number2 must be non-negative integers represented as strings, with no leading zeros. The function first compares the lengths of number1 and number2. If number1 is longer, it returns true; if number2 is longer, it returns false. If both have the same length, the function compares their digits from left to right. It returns true as soon as a digit in number1 is greater than the corresponding digit in number2, and false if it is smaller. If no such condition is found (i.e., both numbers are equal), the function returns true. This method is made static so that it can be used in other static methods.

- **removeLeadingZeros(String number)** – This method removes leading zeros from an integer represented as a string and returns the cleaned string. It first determines whether the number is positive or negative by checking the first character. A flag is used to track the sign of the number. If the number is negative, the negative sign is removed for processing. The method then iterates through the digits from left to right until it finds the first non-zero digit. If no non-zero digit is found (i.e., the number is zero), it returns the string 0. Otherwise, it returns the substring starting from the first non-zero digit. If the number was originally negative, a minus sign is prepended to the result. This method is made static so that it can be used in other static methods.

- **multiplicationWithDigit(String number, String digit)** – This method performs the multiplication of a non-negative integer (represented as a string) with a single-digit number (also represented as a string), and returns the product as a string. The multiplication is done by iterating over the digits of the input number from right to left. At each step, the current digit is multiplied by the given single-digit number, and any carry (excess) from the previous step is added. The last digit of the resulting product is prepended to the result string, and the carry is updated to the remaining part of the product (excluding the last digit). This process continues until all digits have been processed. If there is any remaining carry after the final step, it is prepended to the result. This method is made static so that it can be used in multiply method which is static.

- **multiply(String number1 , String number2)** – This method implements the core algorithm for multiplying two non-negative integers represented as strings and returns the product as a string. It iterates over each digit of number2 from right to left, multiplying it with number1 using the multiplicationWithDigit method. For each iteration, the resulting partial product is appended with zeros based on the digit's position (i.e., the number of zeros equals the number of digits processed from the right excluding the current digit process). This shifted product is then added to an accumulated result using the Addition method. Finally, the result is returned. This method is made static so that it can be used in actual mul method(multiplication of two integers) which is static.

3.3 Arithmetic Operation Methods

The implementation of the actual methods that perform addition, subtraction, multiplication, and division, making use of the helper methods described above, is as follows:

- **add(AInteger number1 , AInteger number2)** – This method adds two integers represented by AInteger objects and returns the sum as a new AInteger instance. It begins by retrieving the numeric strings from both objects using the getInteger method and removing any leading zeros with the removeLeadingZeros method. Two flags are then used to determine the sign of each number. If a number is negative, its negative sign is removed so that the arithmetic can be performed on non-negative values using the Addition or subtract methods. If both numbers have the same sign, the Addition method is used to compute the result. If the signs differ, the subtract method is called to perform the subtraction. After the appropriate operation is completed, the correct sign is prepended to the result. The final result is then converted into an AInteger object using the parse method and returned.

- **sub(AInteger number1 , AInteger number2)** – This method subtracts two integers represented by AInteger objects and returns the result as a new AInteger object. It begins by retrieving the numeric strings from both objects using the getInteger method and removing any leading zeros with the removeLeadingZeros method. Two flags are then used to determine the sign of each number. If a number is negative, its negative sign is removed so that the arithmetic can be performed on non-negative values using the Addition or subtract methods. If both numbers have the same sign, the subtract method is used to perform subtraction. If the signs differ, the Addition method is called to perform the addition. After the appropriate operation is completed, the correct sign is prepended to the result. The final result is then converted into an AInteger object using the parse method and returned.

- **mul(AInteger number1 , AInteger number2)** – This method performs the multiplication of two integers represented by AInteger objects and returns the product as a new AInteger object. It begins by retrieving the numeric string from each object using the getInteger method, and then removes any leading zeros using the removeLeadingZeros method. If either one of them is zero, then an AInteger object instanstiated with string 0 is returned. If not, two flags are used to determine the sign of each number. If a number is negative, its negative sign is removed so that the arithmetic can be performed on non-negative values using the multiply method. Now, the multiplication is performed on the above obtained non-negative integers free from leading zeros, using the multiply function and correct sign is prepended to the result based on the flags. The final result is then converted into an AInteger object using the parse method and returned.

- **div(AInteger number1 , AInteger number2)** – This method performs the division of two integers represented by AInteger objects and returns the quotient as a new AInteger object. It begins by retrieving the numeric string from each object using the getInteger method, and then removes any leading zeros using the removeLeadingZeros method. If the divisor(number in number2 object) is zero, it throws an ArithmeticException saying "Division by zero Error". Next, it checks whether number1(number attribute in it) is less than number2(number attribute in it) using the isGreater method. If number1 is less than number2, then an AInteger object instanstiated with string 0 is returned. If not, two flags are used to determine the sign of each number. If a number is negative, its negative sign is removed so that we can perform the division of non-negative integers and prepend the correct sign to

the quotient. The core algorithm for division works by processing each digit of the dividend from left to right. At each step, the current digit is concatenated with the remainder from the previous step to form a temporary value, which is then compared with the divisor to determine the next quotient digit. Initially, a substring of the dividend is taken from the left, with a length equal to the number of digits in the divisor. This substring is compared to the divisor using the isGreater method. If it is greater than or equal to the divisor, the quotient digit (ranging from 1 to 9) is computed by counting how many times the divisor can be subtracted from the substring so that it becomes less than the divisor. This digit is then appended to the result. If the substring is smaller than the divisor, a zero is appended to the result. In both cases, the remainder from the subtraction of substring with product obtained by multiplication of quotient digit and divisor, is carried forward. In subsequent steps, this remainder is concatenated with the next digit of the dividend to form the new temporary value, and the process repeats. This continues until all digits of the dividend have been processed. After the iteration is complete, any leading zeros in the result are removed, and the correct sign is prepended based on the original signs of the inputs. Finally, the resulting string is converted into an AInteger object using the parse method and returned.

The four arithmetic operation methods mentioned above are made static so that they can be used even without instantiating an AInteger object.

3.4 UML Class Diagram for AInteger

AInteger
– number : String
+ AInteger ()
+ AInteger (s : String)
+ AInteger (numbercopy : AInteger)
+ getInteger () : String
+ parse (number : String) : AInteger
– Addition (s1 : String , s2 : String) : String
– subtract (s2 : String , s2 : String) : String
– isGreater (number1 : String , number2 : String) : boolean
– removeLeadingZeros (number : string) : string
– multiplicationWithDigit (number : String , digit : String) : String
– multiply (number1 : string , number2 : String) : String
+ add (number1 : AInteger , number2 : AInteger) : AInteger
+ sub (number1 : AInteger , number2 : AInteger) : AInteger
+ mul (number1 : AInteger , number2 : AInteger) : AInteger
+ div (number1 : AInteger , number2 : AInteger) : AInteger

Table 1: UML Class Diagram for AInteger

4 Implementation Details of AFloat class

This class performs arithmetic operations on floating point numbers. It supports addition, subtraction, multiplication, and division operations by taking numbers as input in their string representation. The methods in this class which perform arithmetic return results

with a precision of **30 decimal places**. The class has a private attribute, **number**, which stores the string representation of the floating point number. A getter method, getFloat() can be used to access the attribute **number**. This attribute can be initialized during the instantiation of the object. If not provided, the default constructor initializes it to the string "0.0". A copy constructor has been included which can be used to create a copy of an existing AFloat object. The arithmetic operations have been implemented as static methods so that they can be used directly without instantiating an object. Many helper methods have been included in this class. The methods that perform the actual arithmetic operations make use of this helper functions to carry out the desired operation.

4.1 Constructors

The AFloat class includes several constructors, each designed for different use cases as described below.

- **Default constructor AFloat()** - Initializes an instance of this class with default number 0.0 . In other words, the attribute, **number**, gets initialized to the string "0.0".
- **Constructor AFloat(String s)** - Initializes an instance of this class with the provided string **s**, as a floating point number in its string representation. That is, the attribute, **number**, is initialized to the provided string **s**.
- **Default constructor AFloat(AFloat AFloatobject)** - creates an instance of the AFloat class by assigning the value of attribute, **number**, of the AFloatobject to the instance's **number** attribute, thereby creating a copy of the AFloatobject.

4.2 Helper Methods

The functionality of each helper method included in the AFloat class has been utilized multiple times throughout the class. As a result, each required functionality is implemented as a separate method, thereby modularizing the code in AFloat class. Each helper method has been made private, since it is called only by the methods declared inside the class. They are also made static so that they can used in other static methods.

The functionality of each helper method declared inside the AFloat class is as described below.

- **parse(String s)** – returns an instance of the AFloat class with its attribute, **number**, initialized to the string **s** using the constructor AFloat(string s). In other words, it converts the string **s** into an AFloat object.

- **Addition(String s1, String s2)** – This method performs the addition of two positive integers **s1** and **s2**.This method implements the algorithm for adding two positive integers represented as strings. The addition is performed by iterating over the strings **s1** and **s2** from right to left. At each step, corresponding digits are added along with any carry (excess) from the previous step. The last digit of the resulting sum is added to the start of the result, while the remaining part (if any) is carried over to the next iteration. This continues until both strings have been fully traversed. If one string is longer than the other, its remaining digits are processed in the same way–adding the carry to each digit, appending the last digit of the sum to the result, and updating the carry accordingly. After all digits are processed, any remaining carry is added to the result. Finally, the method returns the **string result**.

- **subtract(String s1 , String s2)** – This method performs the subtraction of two positive integers, with the precondition that s1 represents a greater integer than s2. The subtraction is carried out by iterating over the strings s1 and s2 from right to left. At each step, corresponding digits are subtracted, and the difference is prepended to the result. If a digit in s1 is smaller than the corresponding digit in s2, it borrows from the preceding digit in s1. A flag is maintained to track whether a borrow was already taken from a digit, ensuring that the digit providing the borrow is decremented appropriately. This process continues until all digits of s2 have been processed. If s1 has remaining digits, they are also processed similarly. If a digit in s1 is negative due to a previous borrow, another borrow is taken from the digit before it (adding 10 to the current digit), and the flag is updated accordingly. Once all digits are processed, the final result string is returned.

- **isGreater(String number1, String number2)** – This is a comparison function that checks whether number1 is greater than or equal to number2, returning true if number1 is greater than or equal to number2, and false otherwise. The inputs number1 and number2 must be non-negative integers represented as strings, with no leading zeros. The function first compares the lengths of number1 and number2. If number1 is longer, it returns true; if number2 is longer, it returns false. If both have the same length, the function compares their digits from left to right. It returns true as soon as a digit in number1 is greater than the corresponding digit in number2, and false if it is smaller. If no such condition is found (i.e., both numbers are equal), the function returns true.

- **removeLeadingZeros(String number)** – This method removes leading zeros, if any, from a floating-point number represented as a string. It begins by checking the sign of the number. If the number is negative, the minus sign is removed, and a flag is set to indicate the negative sign. Next, the method identifies the index of the first non-zero digit before the decimal point by iterating over the characters of the number. If such a digit is found, a substring from that index to the end of the string is extracted. If there are no non-zero digits before the decimal point, the method extracts a substring starting one position before the decimal point to ensure a zero is placed before the decimal. The cleaned number is then reconstructed by prepending the appropriate sign based on the flag and returned. If the number contains no decimal point and consists entirely of zeros, the method returns the string "0".

- **removeEndingZeros(String number)** – This method removes trailing zeros from a floating-point number represented as a string. It first checks whether the number is a floating-point value by determining if it contains a decimal point. If no decimal point is found, it implies that the number is an integer so it is returned unchanged. If the number contains a decimal point, the method searches from the end of the string to locate the last non-zero digit after the decimal point. If no such digit is found, the method returns the substring from the start of the number up to (but not including) the decimal point. Otherwise, it returns the substring from the start of the number up to the position of the last non-zero digit.

- **convertToFloat(String number)** – This method converts the number, if it is an integer, to float by concatenating ".0" to the end of the number. To know whether number is an integer, the method checks the presence of decimal point in the number. It returns the float form of the number as string.

- **truncate(String number)** – This method truncates the number, if it is a floating

point number and has more than 30 digits after the decimal point, to 30 decimal places. Otherwise, the number is returned unchanged.

- **Addition(String s1, String s2)** – This method performs the addition of two positive integers **s1** and **s2**.This method implements the algorithm for adding two positive integers represented as strings. The addition is performed by iterating over the strings **s1** and **s2** from right to left. At each step, corresponding digits are added along with any carry (excess) from the previous step. The last digit of the resulting sum is added to the start of the result, while the remaining part (if any) is carried over to the next iteration. This continues until both strings have been fully traversed. If one string is longer than the other, its remaining digits are processed in the same way–adding the carry to each digit, appending the last digit of the sum to the result, and updating the carry accordingly. After all digits are processed, any remaining carry is added to the result. Finally, the method returns the **string result**.

- **subtract(String s1 , String s2)** – This method performs the subtraction of two positive integers, with the precondition that s1 represents a greater integer than s2. The subtraction is carried out by iterating over the strings s1 and s2 from right to left. At each step, corresponding digits are subtracted, and the difference is prepended to the result. If a digit in s1 is smaller than the corresponding digit in s2, it borrows from the preceding digit in s1. A flag is maintained to track whether a borrow was already taken from a digit, ensuring that the digit providing the borrow is decremented appropriately. This process continues until all digits of s2 have been processed. If s1 has remaining digits, they are also processed similarly. If a digit in s1 is negative due to a previous borrow, another borrow is taken from the digit before it (adding 10 to the current digit), and the flag is updated accordingly. Once all digits are processed, the final result string is returned.

- **add_int(String number1 , String number2)** – This method performs the addition of two integers represented as strings and returns the sum as a string. It begins by removing any leading zeros with the removeLeadingZeros method. Two flags are then used to determine the sign of each number. If a number is negative, its negative sign is removed so that the arithmetic can be performed on non-negative values using the Addition or subtract methods. If both numbers have the same sign, the Addition method is used to compute the result. If the signs differ, the subtract method is called to perform the subtraction. After the appropriate operation is completed, the correct sign is prepended to the result. The result is then returned.

- **sub_int(String number1 , String number2)** – This method performs the subtraction of two integers represented as strings and returns the result as a string. It begins by removing any leading zeros with the removeLeadingZeros method. Two flags are then used to determine the sign of each number. If a number is negative, its negative sign is removed so that the arithmetic can be performed on non-negative values using the Addition or subtract methods. If both numbers have the same sign, the subtract method is used to perform subtraction. If the signs differ, the Addition method is called to perform the addition. After the appropriate operation is completed, the correct sign is prepended to the result. The result is then returned.
- **multiplicationWithDigit(String number, String digit)** – This method performs the multiplication of a non-negative integer (represented as a string) with a single-digit number (also represented as a string), and returns the product as a string. The multiplication is done by iterating over the digits of the input number

from right to left. At each step, the current digit is multiplied by the given single-digit number, and any carry (excess) from the previous step is added. The last digit of the resulting product is prepended to the result string, and the carry is updated to the remaining part of the product (excluding the last digit). This process continues until all digits have been processed. If there is any remaining carry after the final step, it is prepended to the result. This method is made static so that it can be used in multiply method which is static.

- **multiply(String number1 , String number2)** – This method implements the core algorithm for multiplying two non-negative integers represented as strings and returns the product as a string. It iterates over each digit of number2 from right to left, multiplying it with number1 using the multiplicationWithDigit method. For each iteration, the resulting partial product is appended with zeros based on the digit's position (i.e., the number of zeros equals the number of digits processed from the right excluding the current digit process). This shifted product is then added to an accumulated result using the Addition method. Finally, the result is returned.

4.3 Arithmetic Operation Methods

The implementation of the actual methods that perform addition, subtraction, multiplication, and division, making use of the helper methods described above, is as follows:

- **add(AFloat number1 , AFloat number2)** – This method adds two floating-point numbers represented by AFloat objects, number1 and number2, and returns the result as a new AFloat object. It begins by retrieving the numeric strings from both objects using the getFloat method and removes any leading zeros using the removeLeadingZeros method. To perform the addition, both floating-point numbers are first converted into integer form. This is done by equalizing the number of digits after the decimal point in both numbers through zero-padding, followed by removing the decimal point. The resulting integer strings are then added using the add_int method. After the addition, the decimal point is reinserted into the result at the correct position — specifically, at a distance from the end equal to the number of decimal digits originally present. Any leading or trailing zeros are removed from the result, and the value is truncated using the truncate function. Finally, the result is converted into an AFloat object using the parse method and returned.

- **sub(AFloat number1 , AFloat number2)** – This method subtracts two floating-point numbers represented by AFloat objects, number1 and number2, and returns the result as a new AFloat object. It begins by retrieving the numeric strings from both objects using the getFloat method and removes any leading zeros using the removeLeadingZeros method. To perform the subtraction, both floating-point numbers are first converted into integer form. This is done by equalizing the number of digits after the decimal point in both numbers through zero-padding, followed by removing the decimal point. The resulting integer strings are then subtracted using the sub_int method. After the addition, the decimal point is reinserted into the result at the correct position — specifically, at a distance from the end equal to the number of decimal digits originally present. Any leading or trailing zeros are removed from the result, and the value is truncated using the truncate function. Finally, the result is converted into an AFloat object using the parse method and returned.

- **mul(AFloat number1 , AFloat number2)** – This method performs the multiplication of two floating-point numbers represented by AFloat objects, number1 and number2, returns the product as a new AFloat object. It begins by retrieving the numeric strings from both objects using the getFloat method and removes any leading zeros using the removeLeadingZeros method. Two flags are then used to determine the sign of each number. If a number is negative, its negative sign is removed so that the arithmetic operations can be performed on non-negative values and appropriate sign can be prepended to the result at the end. Both non-negative floating numbers are then converted into integer form by removing the decimal point. The resulting integer strings are multiplied using the multiply function. After the multiplication, the decimal point is inserted into the product at the correct position — specifically, at a distance from the end equal to the total number of decimal digits in both inputs (after leading zeros have been removed). Any leading or trailing zeros are then removed from the result. If the final result is "0", the string "0.0" is returned as an AFloat object using the parse method. Otherwise, correct sign is prepended to the final result. Finally, the result is truncated using the truncate function, converted into an AFloat object using the parse method and returned.

- **div(AFloat number1 , AFloat number2)** – This method performs the division of two floating numbers represented by AFloat objects, number1 and number2, returns the result as a new AFloat object. It begins by retrieving the numeric strings from both objects using the getFloat method, then removes any leading zeros using the removeLeadingZeros method and trailing zeros using the removeTrailingZeros method. The method checks whether number1 is zero. If so, the string "0.0" is returned as an AFloat object using the parse method. Otherwise, the division process begins. Two flags are used to determine the sign of each number. If a number is negative, its negative sign is removed so that the arithmetic operations can be performed on non-negative values and appropriate sign can be prepended to the result at the end. Both non-negative floating numbers are then converted into integer form by equalizing the number of digits after the decimal point in both numbers through zero-padding, followed by removing the decimal point. Division of these transformed integers simulates division of the original floating-point numbers. Now, the actual algorithm for the division begins. First, it compares the first digit in number1 to number2 using isGreater function. If the first digit is greater than or equal to number2 ,the quotient digit (ranging from 1 to 9) is computed by counting how many times number2 can be subtracted from that digit so that it becomes less than the divisor. This quotient digit is then appended to the result. If the first digit is less than number2, a zero is appended to the result. In both cases, the remainder from the subtraction is carried forward and concatenated with the next digit of number1 to form a new value for comparison. This process continues until all digits of number1 have been processed. If a non-zero remainder remains after processing all digits, a decimal point is appended to the result. The algorithm continues by treating the remainder as the new value, appending zeros to it, and computing quotient digits as before. This continues until either the remainder becomes zero or the desired number of decimal digits (30 decimal digits) is reached. Afterward, any leading zeros are removed, the appropriate sign is prepended, and the result is converted into an AFloat object using the parse method before being returned.

  In the above four methods, if the result is an integer, it is converted into a float using convertToFloat method. Also, these methods are made static so that they can be used even without instantiating an AFloat object.

4.4 UML Class Diagram for AFloat

AFloat
– number : String
+ AFloat ()
+ AFloat (s : String)
+ AFloat (numbercopy : AFloat)
+ getFloat () : String
+ parse (number : String) : AFloat
– Addition (s1 : String , s2 : String) : String
– subtract (s2 : String , s2 : String) : String
– isGreater (number1 : String , number2 : String) : boolean
– removeLeadingZeros (number : string) : string
– removeEndingZeros (number : string) : string
– add_int (s1 : String , s2 : String) : String
– sub_int (s1 : String , s2 : String) : String
– convertToFloat (number : string) : string
– truncate (number : string) : string
– multiplicationWithDigit (number : String , digit : String) : String
– multiply (number1 : string , number2 : String) : String
+ add (number1 : AFloat , number2 : AFloat) : AFloat
+ sub (number1 : AFloat , number2 : AFloat) : AFloat
+ mul (number1 : AFloat , number2 : AFloat) : AFloat
+ div (number1 : AFloat , number2 : AFloat) : AFloat

Table 2: UML Class Diagram for AFloat

5 Evaluation of the Library

Both classes, AInteger and AFloat, have been evaluated on various test cases using the MyInfArith.java file. This file takes four command-line arguments to output the result.

1. First argument is **type** of the number **<int/float>**
2. Second argument is the **operation** to be performed **<add/sub/mul/div>**
3. Third argument is the **first operand**
4. Fourth argument is the **second operand**

Ant build.xml file can also be used to evaluate the correctness of the library. It compiles AInteger.java and AFloat.java, packages the object files into aarithmetic.jar file and then compiles and runs MyInfArith.java with the jar file. When the Ant build.xml is run, it even re-compiles the files which are modified. The following command that can be used to run MyInfArith using the build.xml file.

```
ant runInfArith -Darg1=<type> -Darg2=<operation> -Darg3=<operand1> -Darg4=<operand2>
```

To delete the object files and aarithmetic.jar file, use the following command

```
ant clean
```

Below are some of the outputs produced when MyInfArith.java was executed with various test cases.

Input: `java MyInfArith int add 236500782249129494973109332402504293978326246711379838638440149 8`
Output: `665898614873800632956973176417 48`

Input: `java MyInfArith int sub 3116511674006599806495512758577577452423003463811444464538840 08`
Output: `-546287306263397813379509411254 31`

Input: `java MyInfArith int mul 1434416316044592994268069731232223017167694823904478474013730519`
Output: `330162008905899217578310782382075660760972861550182008086155118`

Input: `java MyInfArith int div 879272636528306057983395052167721149383525361708964745499835 8`
Output: `17804979`

Input: `java MyInfArith float div 879272636528306057983395052167721149383525361708964745499835 8.0`
Output: `17804979.09146998930296115952008787853 3`

Input: `java MyInfArith float add 84486723.420039 70974199.843732`
Output: `155460923.263771`

Input: `java MyInfArith float sub 840196454.51725 712586963.70283`
Output: `127609490.81442`

Input: `java MyInfArith float mul 6400251.9377695 2326541.6827934`
Output: `14890452913599.9717457253213`

Input: `java MyInfArith int div 2 0`
Output: `Division by zero Error`

These outputs have been verified using the BigInteger and BigDecimal classes which are used for handling large numbers.

6 How to Use the Library

The aarithmetic.jar file, created using the Ant build.xml script, serves as the library containing implementation of the arbitrary precision arithmetic. The aarithmetic.jar includes the compiled AInteger.class and AFloat.class files. Once the jar file gets created, it can be transferred to any machine and used as a library in other Java programs.

The contents of aarithmetic.jar can be displayed using the command in the terminal
`jar tf /<Path of the directory where aarithmetic.jar is present>/arrithmetic.jar`

The contents of aarithmetic.jar are as follows:

```
META-INF/
META-INF/MANIFEST.MF
arbitraryarithmetic/
arbitraryarithmetic/AFloat.class
```

```
arbitraryarithmetic/AInteger.class
```

To use this library in a java file, first the following import statements should be included in the java file:

- If we want to use AInteger.class, write the following import statement at the beginning of the java file

  ```
  import arbitraryarithmetic.AInteger;
  ```

- If we want to use AFloat.class, write the following import statement at the beginning of the java file

  ```
  import arbitraryarithmetic.AFloat;
  ```

Once we include the above import statements in the java file, we can use the arithmetic operation methods declared in the class which is imported, in the java file to perform required operation.

To compile the java file which uses the library(aarithmetic.jar), we need to execute the following command in the terminal window.

```
javac -cp /<path>/aarithmetic.jar <name>.java
```

Once the compilation is successful, we run the java file using the following command in the terminal. If the java file takes command line arguments, they can be added after the java file name in this command.

```
java -cp /<path>/aarithmetic.jar <name>.java
```

In the above two commands,
`<path>` – path of the directory in which the jar file is present
`<name>` – name of the java file using aarithmetic.jar

7 Limitations

- It does not work for the numbers given in scientific notation.
- The precision of the output of AFloat arithmetic operations is 30 decimal places. It does not round-off the result but truncates it.
- The length of the input number cannot be more than $2^{31} - 1$. This is because the maximum lemgth of a string in Java is **INT_MAX**.
- There is no input validation for the numbers in AFloat and AInteger classes.

8 Key Learnings

This project helped me to know about

- the git internals and its commands, and use them to push a repository to github and clone a repository from github,

- Ant, the build tool for Java projects,
- the way to write a report for a project in Latex, and
- Docker which is used to packaging applications and its dependencies into a portable container.

9 Conclusion

In conclusion, this project successfully completed the task of implementing an arbitrary precision arithmetic library in Java which can be used for handling big number computations.