University of Computer Science of Iasi

Authors:

Popa Stefan - ISS1

Ionita Mihail-Catalin - ISS1

Craciun Mihai-Cosmin - ISS1

Professors:

Lect.dr. Radulescu Vlad

Chapter 1

Abstract

The goal of this project is to develop an application which allows carrying out operations with very big numbers (positive integers). The software development was performed in 3 stages: application development - in which we designed and implemented a system that satisfies the specifications, unit testing - made use of the unit testing tools in order to test the code developed in phase 1 and use of assertions in order to check the preconditions, postconditions, and invariants for the operations implemented during phase 1.

1.1 Steps

1.1.1 Application Development phase

The main functionality of application should be to be able to parser and evaluate expressions of type (a+b)∗2;a=2222222222;b=5 that should support variables and operand with an increased number of digits which in normal circumstances would essentially overflow any positive integer known type, for instance Integer type of 64 bytes in Java allows for a maximum value equal to 9,223,372,036,854,775,807. The application should allow for inputting the string expression from the keyboard or from a known-structure JSON file.

For supporting the functionality mentioned above the following aspects are covered by the architecture:

1. Choose a mode: automatic (get values from file) or interactive (get value from command line)
2. Finding the token in the string expression (a token can be a numeric value, operator, variable)

3. Parsing the tokens and their respective values

4. Convert the parsed expression into the RPN (reverse polish notation), to be easier to be evaluated

5. Evaluate the expression and output the result of the expression as a BigInt instance.

For this design, the application architecture was divided into 3 parts and each member was assigned to one.

Craciun Mihai-Cosmin - Classes and modules responsible for low-level computing and operations (BigInt)
Ionita Mihail-Catalin - Classes and modules responsible for Parser logic as well as custom data types
Popa Stefan - Dorin - Classes and modules responsible for abstraction of modes (automatic/interactive) and serialization/deserialization of objects

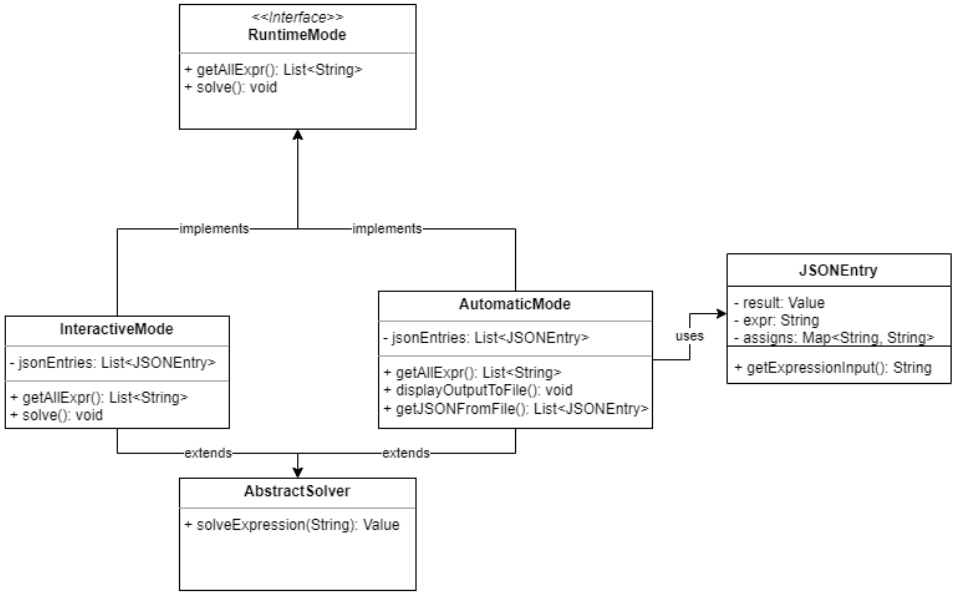

Fig.1: Facade module of modes

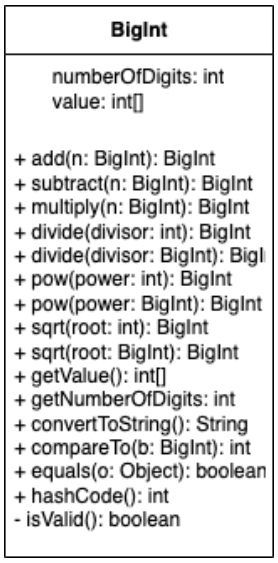

Fig.2: BigInt class methods

Modes Module

In Figure 1 it's presented the Facade of the application - high level API that hides the implementation underneath. Following this architecture we can instantiate an unknown mode at compile time, but explicit at runtime. This design pattern is known as Factory Method. Given the benefits of it, we concluded that this approach is the best fit in order to satisfy the constraints of the application.

Parser Module

The parser module of the application is built such as to be able to solve the following problems in expression evaluation:

1. Tokenization
2. Expression mathematical evaluation

In the process of tokenization the parser will use regular expressions to scope up any known combination that could mean a token and place them in a stack structure for later use.

In the process of mathematical evaluation the patser will use the generated stack in the previous step to evaluate the operator, numeric values using rules such as precedence of the operators instantiated by parenthesis. To achieve this goal the parser will converse the

expression from it's infixed formed such as a + b to an easier formed called reverse polished notation such as a,b,+. After the expression was converted to the RPN it becomes easier to pop subsequently elements from the top of the stack and evaluate their meaning, update variables in case of assignment, etc.

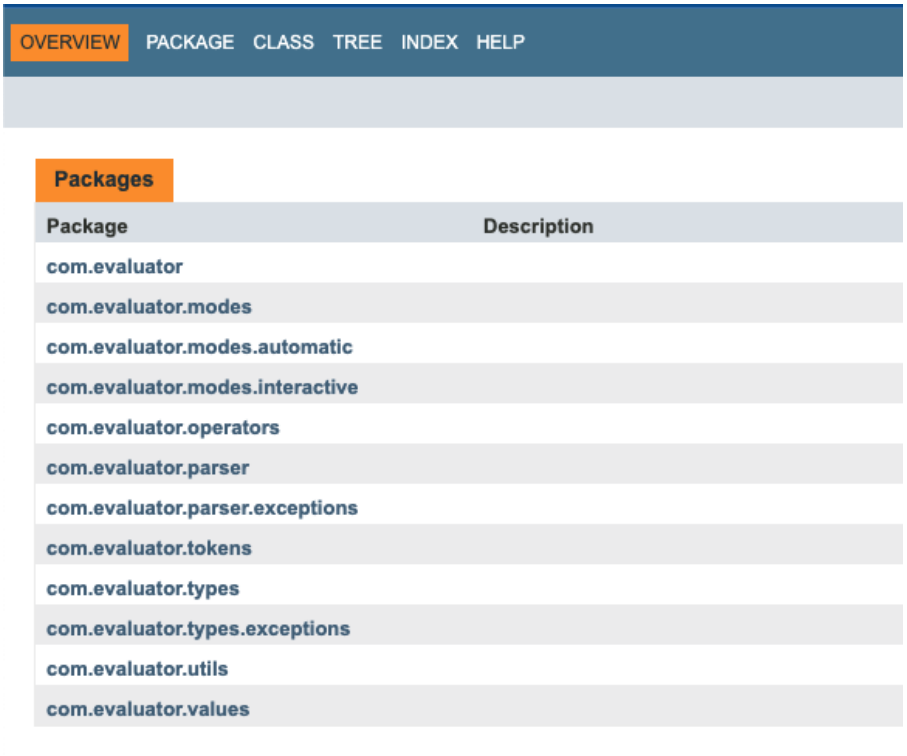

Fig.2 Documentation generated by javadoc after compilation through

The codebase binding with the comments blocks

```
/**
 * Gets a BigInt as a parameter and returns the sum of the current value with the given parameter
 * @param n The number to be added to the current value
 * @throws InvalidNumberFormatException in case resulting value is not valid
 * @throws MaximumNumberOfDecimalExceededException in case resulted value have more than MAX_NUMBER_OF_DIGITS digits
 * @return The sum represented as a new BigInt
 */
public BigInt add(BigInt n) throws InvalidNumberFormatException, MaximumNumberOfDecimalExceededException {
    assert this.isValid();
    assert n.isValid();
```

Fig.3 javadoc style inline comments blocks for methods

```java
/**
 * The BigInt class
 *
 * @author Craciun Mihai-Cosmin
 * @since 01.05.2022
 */
public class BigInt implements Comparable<BigInt>{
```

Fig.4 javadoc style inline comment block for classes

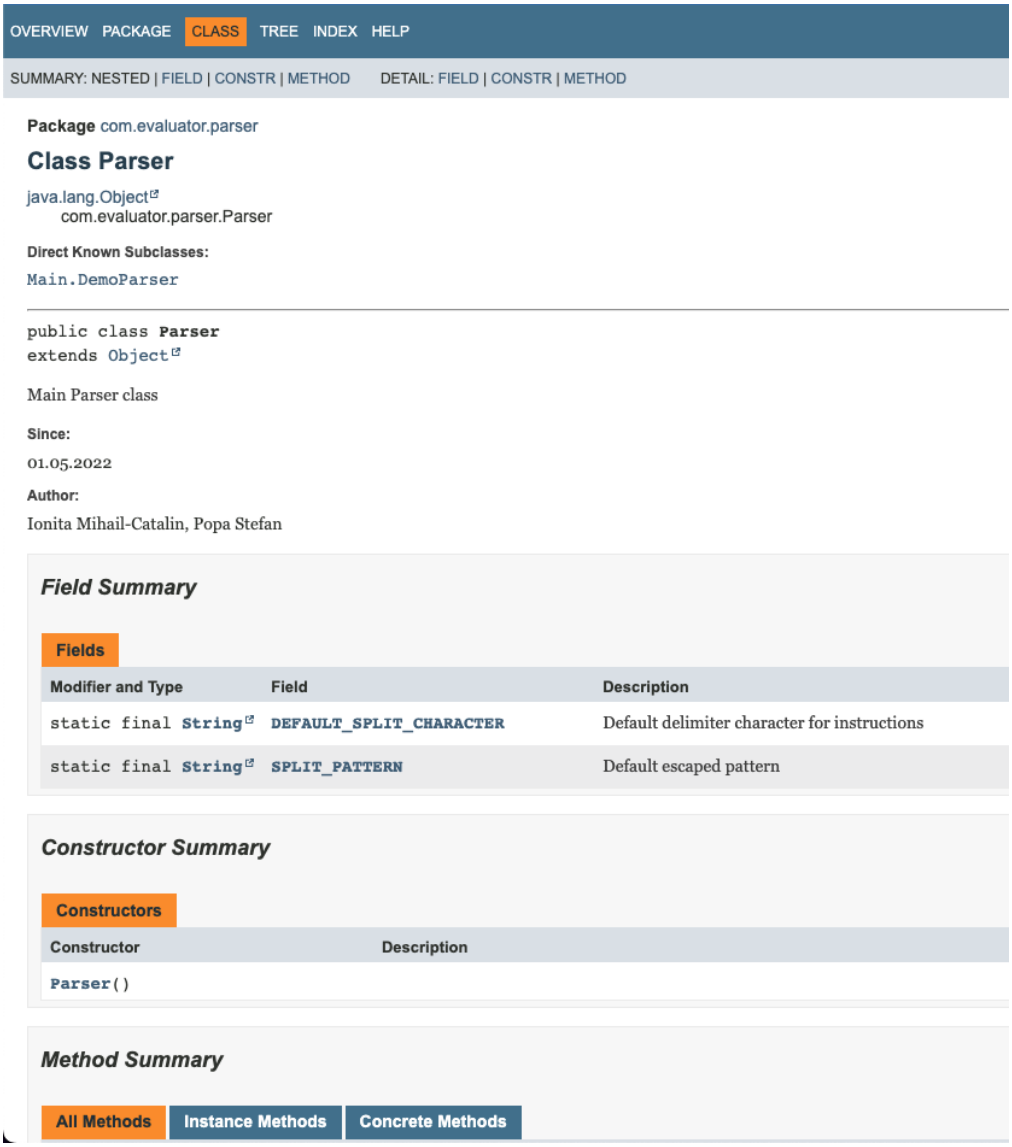

Fig 4. Class documentation generated by javadoc

BigInt Module

The BigInt class contains all the low level operations inside it and mimics the BigInteger native class in it's behavior meaning the following:

- The class it's immutable meaning that once it is initialized the child fields cannot be changed.
- The value is stored as an array of digits inside an integer array starting in reverse order (eg. number "1234" will be stored as [4,3,2,1]) in order to make the mathematical operations easier to compute.

The class contains multiple ways to initialize it but the most common one is using the String constructor that receives a number represented as string. It has to be noted that the BigInt class has a soft limit value for the number of digits that is 100.000 in order to keep the computational time manageable for testing. The algorithms used for basic operations are fast, the class being able to compute 2^{10000} under 1 second and multiplying to numbers of 50.000 digits under 15 seconds.

The class contains the basic comparison between itself and another instance of BigInt, a representation in String format and instance equality.

The class also contains a self validation method used later in the assertion phase that makes sure that the class is valid under any circumstances.

1.1.2 Unit testing phase

The goal of this stage is to discover and fix the defects introduced in phase 1 - application development in which some implementation errors might be introduced.
For this section a master test plan was made which will be described below.
The tools used for this purpose were:

1. Junit - Java unit testing framework that's one of the best test methods for regression testing. An open-source framework, it is used to write and run repeatable automated tests.
2. Mockito - a mocking framework, Java-based library that is used for effective unit testing of Java applications. Mockito is used to mock interfaces so that a dummy functionality can be added to a mock interface that can be used in unit testing.

Testing strategies:

- Equivalence classes and class border testing - Multiply two numbers until an error is thrown because the result is too big (out of border)
- Test cases are based on classes
- Test cases going through each program loop
- Test cases where each precondition is not satisfied
- Test cases of unusual conditions - strange inputs / state of the program
- Test cases with very big numbers
- Test cases checking the correctness of results
- Test cases for I/O Errors

For this phase we chose a **non-incremental** Testing Approach. Each module is independent from the rest of the application. (Unit)
Specific code was written to simulate other modules (mocking).
All the unit tests are tested in parallel resulting in a code coverage of over 80%.
For each module of the application a bottom-up testing approach was used - testing begins with the modules situated at the lowest level. Using this approach, the critical modules are tested as soon as possible.

1.1.3 Assertions phase

For adding assertions that are validated at runtime while the software is running the native support for assert instruction inside the Java programming language was used. To ensure that the parser and its auxiliary components such as: serialization/ deserialization module are working as intended specific boolean conditions were supplied and used with the assert instruction.
The assertions were explicitly included in application code in order to check implementation constraints at runtime.

What it was intended to assert at runtime mostly waters down to:

- Preconditions - condition that must be true when entering a method
- Postconditions - condition that must be true when exiting a method
- Loop Invariants - condition that must be true at any time for a certain object

In addition to the second phase, assertions allow checking of the state of the object throughout the execution of the method.
Fault sniffers are added with high specificity to the core logic of the algorithms (Parser and BigInt) in which we have a high risk of errors.

Because the architecture is making use of OOP Polymorphism and Abstraction the preconditions of methods from derived classes (InteractiveMode, AutomaticMode) should not be stronger than the precondition of parent class (AbstractSolver).

Furthermore, some assertions are used to check the relation between input and output values. For example we check if a result of a computation is correct before writing it to the file or showing it to the screen.