Assignment 2 The Interpreter

Kenneth Surban
Student ID: 913399830
CSC413, Summer 2018

https://github.com/csc413-01-su18/csc413-p2-ksurban

Project Overview:

For this project, we were provided with a far from complete version of the Interpreter along which was required to be completed by us. The basic algorithm of the project was implement the classes: Program, ByteCodeLoader, ByteCode with all other ByteCode subclasses, Virtual Machine, and RunTimeStack in order to create a program that could decipher and interpret bytecodes. The general flow consisted of using a CodeTable class, a class that initialized a hashmap of the ByteCodes and their respective classes; a ByteCode class for bytecodes to properly execute their methods; a ByteCodeLoader class, a class that would a parse our desired file and load all the ByteCodes; a Program class which held all the ByteCodes and gave them their appropriate addresses; a Virtual Machine class that executed the ByteCodes; and then a RunTimeStack class to perform operations on the Stack.

Technical Overview:

Using the Interpreter class, we instantiate a new Interpreter object and pass in the name of our File. Via the defined Interpreter construct, a initialization of our static HashMap in CodeTable occurs and then a ByteCodeLoader object instantiated with our file name. The file is then parsed each line and then StringTokenized. Using the codeTable, the first token is used as a key to return its respective class name, and then the second is the argument of the bytecode. Each byteCode is instantiated, initialized with their respective arguments, and then added to our program object. The program object is loaded and the byteCodes are given their address in which is used in order to correctly access specific byteCodes instructions. After, we execute the program in the Virtual Machine class, and the byteCodes are ran. These byteCodes have their own unique method on how they will operate; meaning correcting the address, returning or going to a specific instruction, performing mathematical operations on the stack, or terminating the program. In order to execute some these byteCodes, a RunTimeStack is used to push or pop arguments, variables, and user inputs. After a byteCode is performed, the operation is outputted along with a visual of where the framePointer points, and what's on the RunStack. Once every byteCode object is finished performing their operation, the project will stop.

Work Completed:

The work that was completed was the implementation of the ByteCodeLoader, Program, ByteCode, ByteCode subclasses, Virtual Machine, and most of the RunTimeStack class. I tested both the factorial.x.cod and the fib.x.cod files and my project returned the correct outputs. The only thing that I couldn't get to work was a function in the RunTimeStack called dump. Apparently, my dump function only seemed to work on some inputs. I know for factorial.x.cod inputs: 1 and 2 worked without any errors, but when I tried 8 I ran into an ExceptionOutOfBound error. The same happened for fib.x.cod where I tried 1 and 2, but I got another ExceptionOutOfBoundError when I tried 3. Not sure where I went wrong but I really clutched as many points I can get as I only got my project(minus Dump) to work in the last several hours before the submission time.

Development Environment:

Version of Java Used:

I used java version: Java 9 SE.

IDE Used:

The IDE I used to complete this project was Eclipse Version: Oxygen.3a Release (4.7.3a).

How to build or import the project in the IDE used:

- Open Eclipse and you will be asked for the directory of your desired workspace
- Eclipse will open a workspace and may automatically import the project if it is found in that directory
- If the project is not automatically imported, manually import the project by clicking on "File", then "Import…". This opens up a small window asking to select the project file or directory.
- Select "General" and then "Projects from Folder or Archive"
- Navigate to the directory of the project and then click "Finish" at the bottom of the window

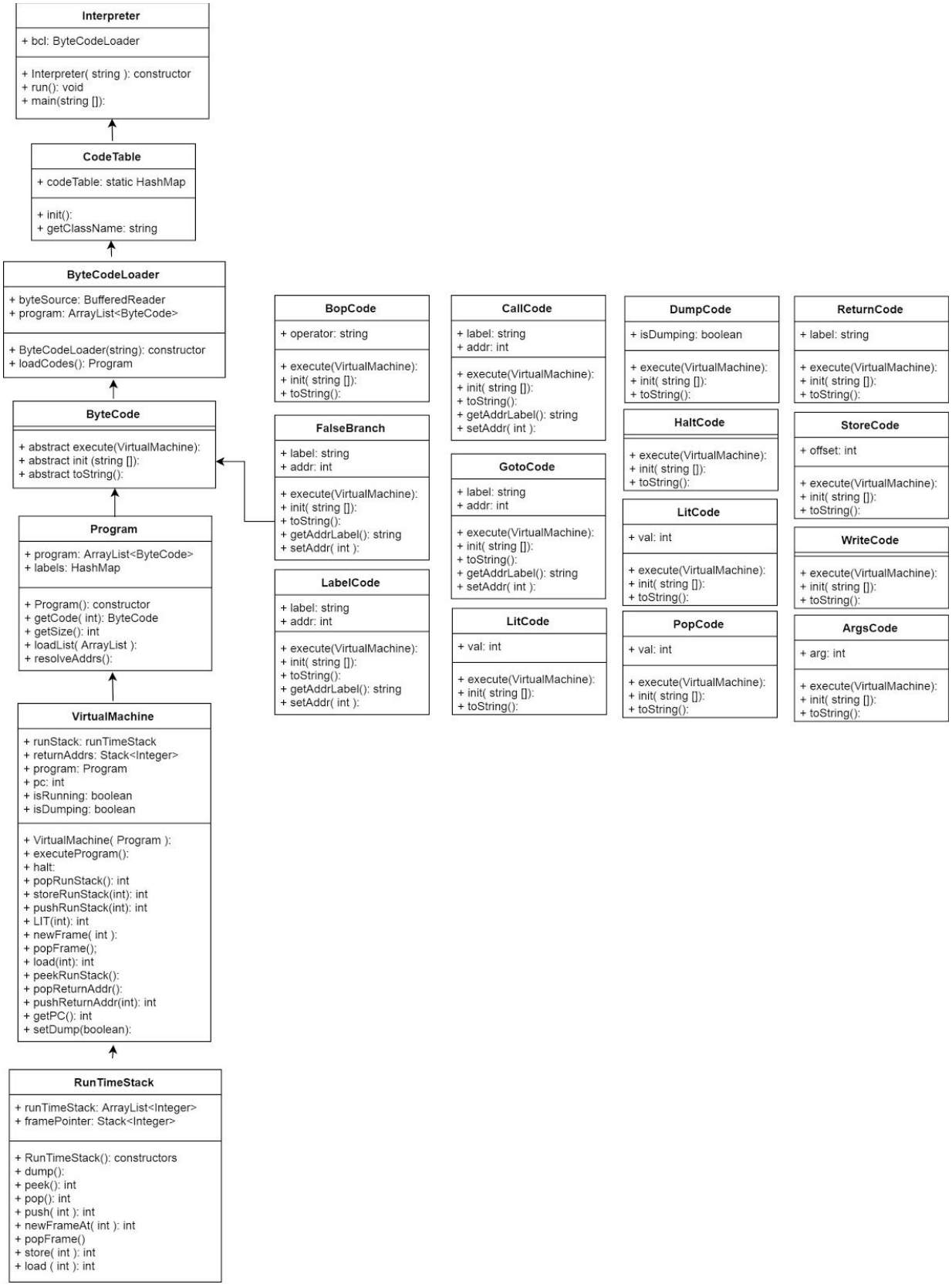
How to run the project:

- After successfully importing the project into your workspace, on the top and on the same row as where you would "File", go to "Run" and a drop down list will appear
- Select "Run Configurations…". A window titled "Run Configurations" will popup
- On the left hand side, you'll see a list of options (Gradle Project, Java Applet, Java Application, …). Double click "Java Application" and a new configuration titled "New_configuration" will be made for you.
- In the "Main" tab, under Project select your project and then under main class select Search where you will be prompted a list of main classes with a main class. For this project, select "Interpreter - interpreter"
- Next in the Arguments tab, under "Program Arguments" type the desired file you would like to run ("factorial.x.cod", "fib.x.cod"). If you'd like to change the desired file later, simply erase the previous input and replace with the name of your new desired file.
- Finally, click apply and then run. The window will close and the project will run.
- Once this configuration is made, you can press the green circle with a white play arrow inside of it if you'd like to run the same configuration again.
-

Assumptions Made:

- I assumed that the resolve Addresses of each bytecode were as linear as line codes that way it would be easier to return (e.g. Line 1: GOTO Read = 1, LABEL Read = 2)
- I assumed the user will only input integers

Implementation:



In the main figure above, we see the whole UML implementation of the project. The Interpreter class initializes a static HashMap of the CodeTable, which takes a string as its key and the value as its respective class. The ByteCodeLoader is then called to parse the file and use tokens to instantiate and initialize the class of the parsed token. The Program class is then used to give the ByteCodes their appropriate address for when they should be called. The Virtual Machine runs the program, manages the flow of which line should be executed, and contains execution methods for ByteCode classes. The RunTimeStack handles the pushing and popping of all the variables, arguments, and inputs, and contains method.

A key concept to understanding the implementation of ByteCodeLoader and Program class is how ByteCodes are very linear, meaning that typically after a line is performed then so will the next one. With this concept, we would just have to parse every line and then give each ByteCode an address number according to their line number. This way we would have an arrayList which we could directly access each ByteCode by their index so for ByteCodes such as CALL or RETURN, we could specify the ByteCode we want to run.

Project results and conclusion:

The project works except for the dump method in the RunTimeStack class. As stated in the overview of the project, the dump method only worked for some inputs. For ones that didn't work the compiler returned an error ExceptionOutOfBounds.

Overall, I would have to say this project was very challenging to understand the overall algorithm. I had to reread the project guidelines several times to understand the workflow and the purpose of each class. I should really build a sanctuary for the gods who invented debuggers or I would not finished. With that said, iterating through sixty or fifty byteCodes through the debugger did feel like I was losing mind haha…

I wish I had more time or had started sooner. This project was definitely of different caliber compared to the Evaluator assignment. I think had I spent another couple hours at the debugger I possibly could have finished the dump method. I am pretty frustrated I didn't finish, but glad I was able to see implement most of what was asked of me. This project really hurt my brain.