ASSIGNMENT 2
The Interpreter

Hadia Andar
915397384
CSC 413, Summer 2018
GITHUB LINK: https://github.com/csc413-01-su18/csc413-p2-handar
Collaborated with: Stephanie Santana

Introduction

a. Project Overview

The purpose of this assignment is to code an interpreter for a mock language called "X-language". This program must translate the given compiled x.cod files into commands that are stored into a list. Then it uses a virtual machine to run the commands. If it is implemented correctly, the interpreter can pass in x.cod files and translate each part of the file through the program as arguments. It will be able to complete through the bytecode operations list and show the correct answers for the three tests we were given: fibonacci and two factorial calculations.

b. Technical Overview

We were provided the skeleton of the code, and had to fill the rest in given the information given. The code that was fully given to us was the Interpreter class and CodeTable class. The ByteCodeLoader class, Program class, and VirtualMachine class was only given to us with some of the structure filled in. We were free to create any other methods and classes that we felt was appropriate to execute the program.

ByteCodeLoader's main function was loadCodes(). This function read the .x.cod source file that we provided to the program in Netbeans. It fed in each line in the file and interpreted (translated) it into commands that are named ByteCodes. These ByteCodes are put into an instance of the Program class.

Program's main function was resolveAddress(). This function returned the addresses that were inside of ByteCodes that were used to jump to different labels in the code. Each label is converted to its appropriate addresses so VirtualMachine would know what to set its program counter to so it can execute when the function exits.

VirtualMachine controlled the program; all of the operations had to go through it. We weren't given any of the methods as we had to write our own. One of the most important methods was the dump function which specified when to check if the boolean dump was true or false. Another important method was executeProgram(), which set the program up and running.

c. Summary of work completed. The x.cod files given to us are translated to produce the correct output for the test cases of the fibonacci series and recursive factorials.

Development environment

a. Java: 1.8.0_161; Java HotSpot(TM) 64-Bit Server VM 25.161-b12
b. Netbeans IDE 8.2

How to build or import your game in the IDE you used.

Importing:

File -> New Project -> Java with existing sources -> select existing sources -> Finish

For running the EvalutorTester.java → right click → run file

1. If the EvaluatorTester.java outputs the correct answers, then you can run the whole project
2. You can run the project from the Run Tab on menu Bar (select Run Program) or press the play button on the toolbar

For running the x.cod files:

File -> Project Properties (projectname) -> categories -> select run -> set interpreter.interpreter as main class and your arguments are the path to the given x.cod files (an example of my path is C:\Users\handa\csc413-p2-handar-master\factorial.x.cod) -> select OK

- You have to edit the argument path every time you want to test a different x.cod file
-

Building and Running Project:

From the menu: Run -> Run Project

-The output window allow you to enter an integer

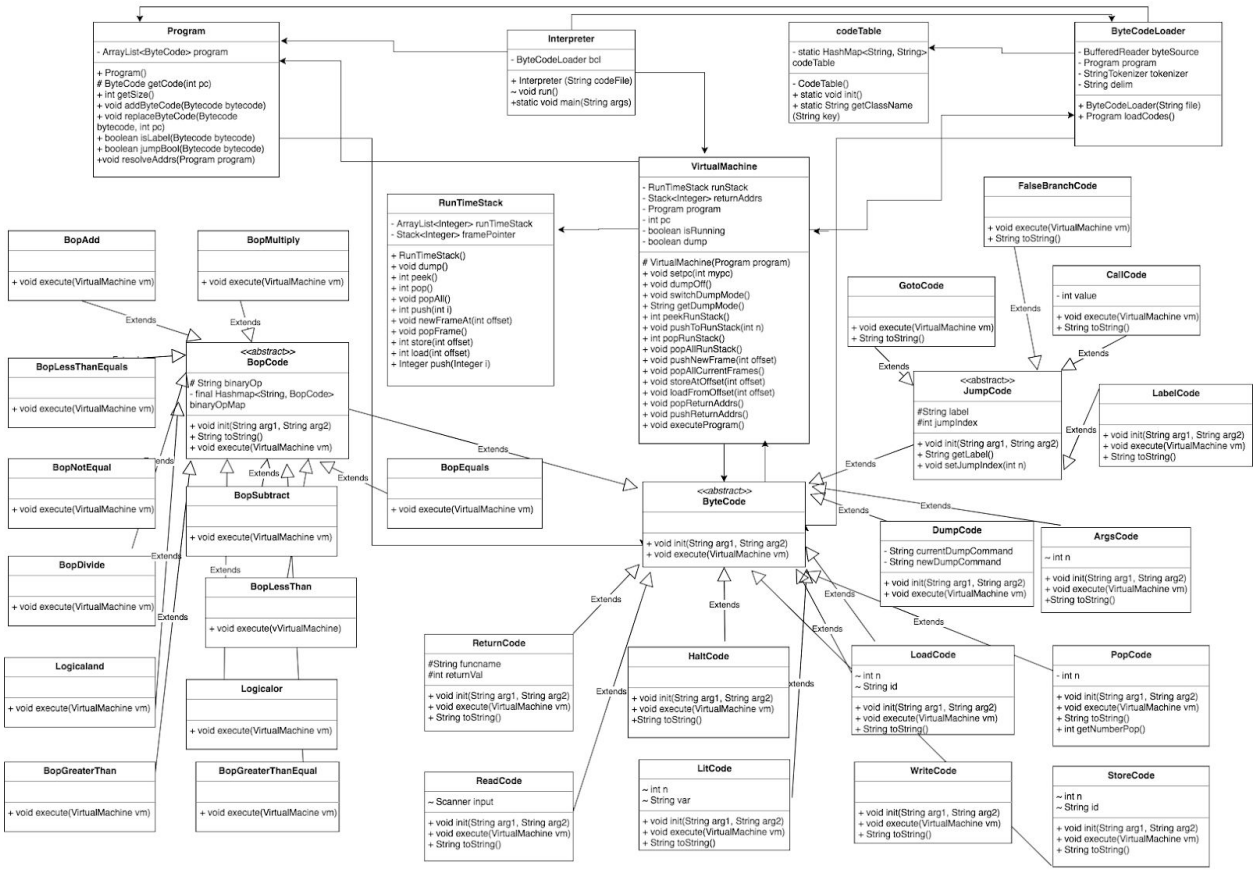
-After entering an integer, the build should be successful and output the correct output for each test case

Assumptions Made when designing and implementing your project?:

- The x.cod files should be written correctly and be compiled right. The logic must be correct because if the files are not written correctly, then the output will produce unexpected, incorrect results.
- User must give valid input that fit the scope of the program.
- Expected input is of integer type

Implementation Discussion

We used the abstract ByteCode class to make extended classes for each type of ByteCode and implemented them. The subclasses inherited the fields of the ByteCode class, which made it really easy to organize the code and run, call, and instantiate them. The extended subclasses inherit the fields of the abstract ByteCode class. This makes it easier to organize and instantiate the bytecodes. I was inspired by the ByteCode and CodeTable class, especially with how they worked together with a Hashmap, so I did that with my BopCode class. I made BopCode abstract and extending from ByteCode and made extended bop subclasses for it. I also made a new abstract class called JumpCode that extended from ByteCode and had CallCode, LabelCode, FalseBranchCode, and GotoCode extend from it. I did this because it reminded me of how the MIPS assembly language is organized with their jump labels. The diagram below will show my implementation for the interpreter project:



Project reflection

Ever since I started using Hashmaps, I've been inclined to use it more and more in my codes. I can see how amazingly helpful it is at keeping my code neat and organized. Some of the setbacks and learning curves that I encountered was being careful to not break encapsulation, making sure I was following the design expectations that were in the documentation, and catching exceptions. I had a lot of trouble with loadCodes() in ByteCodeLoader because Interpreter.java was giving me an exception error on it and it was suggesting for me to add an

IOException in my Interpreter class, which I was not supposed to touch. I eventually fixed it by adjusting my try-catch and the exceptions that were on it. I took a lot of time designing and implementing my classes because although the project was structured in the documentation, there was a lot of freedom to interpret and make the project how I wanted it to be. The hardest part was knowing where to start, because I felt overwhelmed first. Time management and organization was very important. I did not like how each extended ByteCode class thew red errors for not being implemented so I tried implementing all of those first, but I should have done the classes in the order that the documentation said to, because then I would have been less overwhelmed.

Project Conclusion and Results

The x.cod files successfully ran and produced the correct output, which shows that the interpreter was able to successfully translate the files.