Tyler Hsieh
920216320

Assignment 4 Documentation

Github Repository

https://github.com/sfsu-csc-413-fall-2022-roberts/assignment-4---interpreter-Tyler9648

Project Introduction and Overview

For this project, we had to create an interpreter for our x language compiler. The interpreter project required me to create and define classes for each ByteCode, implement CodeTable, RunTimeStack, VirtualMachine, ByteCodeLoader, and Program.

Scope of Work

Task	Completed
Add ByteCode classes	Yes
ARGS	Yes
BOP	Yes
CALL	Yes
DUMP	Yes
FALSEBRANCH	Yes
GOTO	Yes
HALT	Yes
LABEL	Yes
LIT	Yes
LOAD	Yes
POP	Yes
READ	Yes
RETURN	Yes
STORE	Yes
WRITE	Yes
Program	Yes
VirtualMachine	Yes
CodeTable	Yes
ByteCodeLoader	Yes
RunTimeStack	Yes

Execution and Development Environment

I used Intellij IDE to code and compile everything for my program using MacOS, and WindowsOS,.

Compilation Result

```
sfs-wifi-dhcp-10-143-142-105:assignment-4---interpreter-Tyler9648 Tyler$
javac interpreter/bytecode/*.java
sfs-wifi-dhcp-10-143-142-105:assignment-4---interpreter-Tyler9648 Tyler$
javac interpreter/Interpreter.java
sfs-wifi-dhcp-10-143-142-105:assignment-4---interpreter-Tyler9648 Tyler$ java
interpreter.Interpreter sample_files/simple.x.cod
DUMP ON
[][]
GOTO start<<1>>
[][]
LIT 0 i                          inti
[0][0]
LIT 0 j                          intj
[0][0]
LOAD 0 i                         <load i>
[0, 0][0, 0]
LOAD 1 j                         <load j>
[0, 0][0, 0]
BOP +
[0, 0][0, 0]
LIT 7
[0, 0][0, 0]
BOP +
[0, 0][0, 0]
STORE 0 i                                i = 0
[7][7]
LOAD 0 i                         <load i>
[7, 0][7, 0]
ARGS 1
[7][7][7][7]
CALL Write                               Write
[7][7][7][7]
LOAD 0 dummyFormal                       <load dummyFormal>
[7, 0][7][7, 0][7]
Writing: 7
```

```
WRITE
[7, 0][7][7, 0][7]
RETURN
[7][7]
STORE 1 j                                   j = 1
[7][7]
POP 2
[][]
```

Program runs as expected with no errors.

Assumptions

I had to assume that the interpreter.java given by the instructor was already fully working and required no change. I also assumed that the byteCode folder was meant for byteCode subclasses for each byteCode type.

Implementation

ByteCode sub classes

I added byteCode subclasses for each bytecode. This includes ARGS, BOP, CALL, DUMP, FALSEBRANCH, POP, GOTO, HALT, LABEL, LIT, LOAD, POP, READ, STORE, RETURN, and WRITE. Each subclass contained atleast an init(), execute(), getString(), and getByteCode(). Depending on the purpose of the byteCode, some also additional getters and/or setters. The execute() method is the most unique method between subclasses and calls methods from the VirtualMachine.

CodeTable

For my code table, I had init() put each byteCode value into a HashMap <String, String> with the first string being the raw byteCode value, and the second string being the class name.

ByteCodeLoader

For the ByteCodeLoader, I had the loadcode() method use a while loop to read each line of the input file, and put them into a HashMap which would be used for the Program init() parameter.

Program

My program's init() takes in the HashMap of the file's bytecode created by ByteCodeLoader. The init method takes every line of code from the file as a byteCode subclass corresponding to what each byteCode it is, and stores it inside a static vector. A HashMap stores the addresses of Label bytecodes, and the HashMap parameter that's taken in is copied to a static HashMap. The init() method also updates the addressMap HashMap for Label bytecodes and manages the addresses for Call, FalseBranch, and GoTo bytecodes.

RunTimeStack

RunTimeStack serves a purpose in the program almost exact to that of a normal stack object. RunTimeStack has a peek(), pop(), and push() method like a normal stack. In addition, I also added a dump() class which prints out the runStack and returns all other dump info as a string.

For loading variables onto the stack and storing values from the stack into variables, I had to implement the store and load methods, which interact with the runStack.

I also had to implement the newFrameAt() and popFrame() methods to manage the framePointer stack.

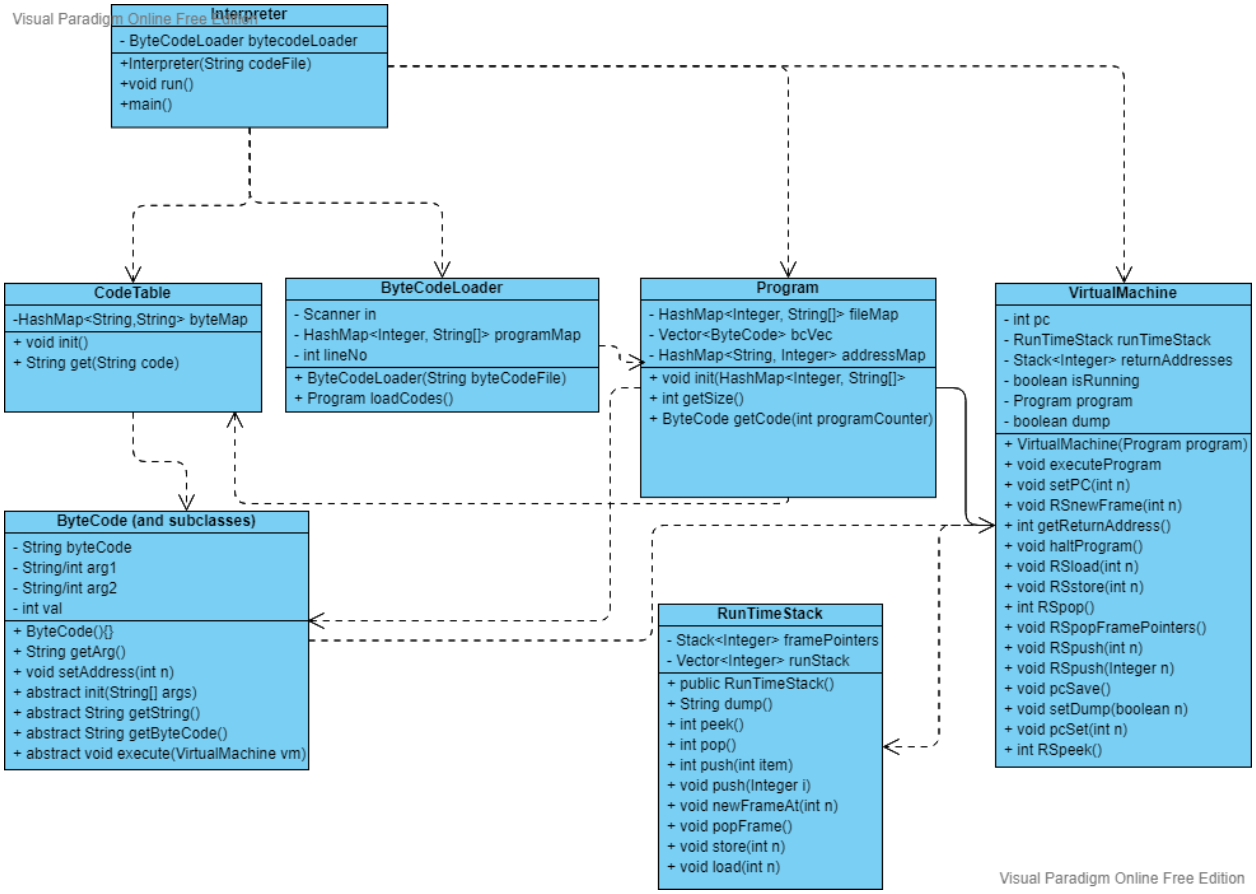
VirtualMachine

The virtualMachine is what simulates the code running, more specifically, the executeProgram() method. This class creates an instance of RunTimeStack and a stack for the return addresses. ExecuteProgram() executes the byteCodes line by line, and dumps if dump is on.
VirtualMachine also has methods for each respective method inside of RunTimeStack.

Code Organization

I kept the original file structure given by the professor as it is already organized. ByteCode subclasses are all kept under the byteCode folder.

Class Diagram



Results and Conclusions

This project helped me further understand and manage scopes when using OOP, and how to simulate a runtime environment in java using objects. It also helped me better my understanding of stacks and what makes them different from vectors.

Challenges

Some of the challenges that I ran into while are not realizing that I had to check the entire byteCode vector for label and setting their addresses before setting the args/addresses of other byteCodes that might call the label such as GoTo.

I also had a lot of trouble with RunTimeStack and storing objects as I'd frequently get out of bound errors. I solved this by checking if the position that the object will be stored into is within the size of the runStack.

There were many small syntax errors such as not using proper declaration for vectors and such.