Brar, Harnoor

Project Interpreter

INTRODUCTION

a. PROJECT REVIEW

For this project, we had to implement an interpreter for a machine mock language into a simplified version of Java. This project revolves around 3 files, the factorial, Fibonacci sequence, and a test, which basically tests all the functions of our code. This code asks user to input a number to perform the code for factorial and Fibonacci series. The code also outputs the current process that is going on too.

b. TECHNICAL OVERVIEW

This project takes the input from the user and perform rest of the code based on that. The project reads the machine mock code, and stores the data in a Hashmap, which is implemented in CodeTable. The main classes for this project are Virtual Machine, Program, and Runtime Stack. Virtual Machine is like the middle man between Program, Run Time Stack and the ByteCode classes. These classes don't have any authority over each other, they communicate through Virtual Machine. ByteCodes is basically an interface which tells us the functions usage, but not implementation.

c. SUMMARY OF WORK COMPLETED

From my understanding, the code works very well and is up to the mark. However, for sure it can still be changed to a better code. There are some methods which I think are not at their best. For example, the method "dump()" in RunTime stack class, I used a lot of "if" statements to implement it, so I think it could've been implemented better.

DEVELOPMENT ENVIRONMENT

a. Version of Java used – 14.0.2

b. IDE used – IntelliJ

BUILDING PROJECT

a. To build the project, you first need to clone my repository from https://github.com/csc413-02-FA2020/csc413-p2-brarharry . After you done cloning it, open your IDE(preferably IntelliJ). Once the IDE is open, click on the import project, and then select the location where you cloned the repository to. Once found, expand csc413-p1-brarharry, and then click on “calculator” and press “OK”, the project will be loaded.

RUNNING PROJECT

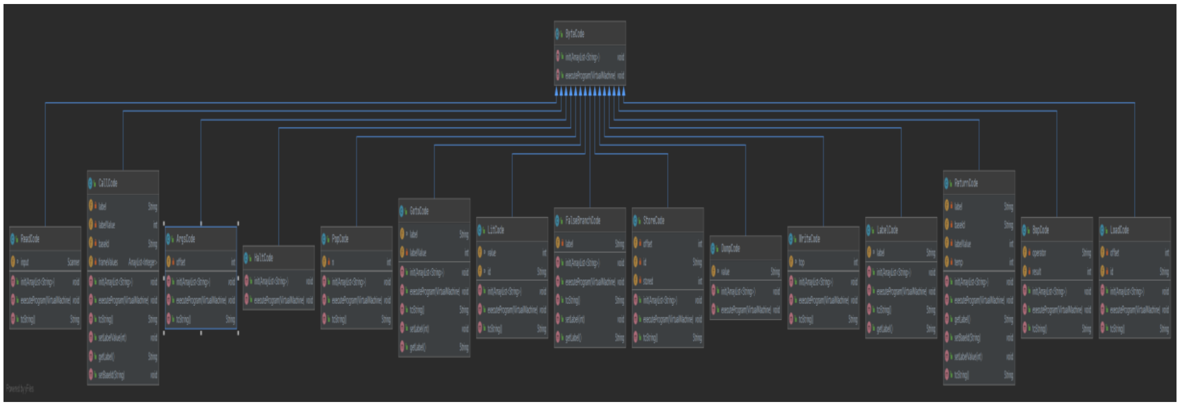
a. Once, the project is loaded, expand all the classes in main directory we have our main method and pretty much the heart of the code. This is our Interpreter class. To run the code of factorial, run the main method of Interpreter class, don’t mind the errors. Then, navigate to the top tab and expand the “Interpreter” tab. Click on “edit configuration”, and put the name of the file in “Arguments” section, and then go back and our code is ready to run. For Fibonacci and Args Test class, we need to do the same process. The output will ask you to enter an integer, and then the code will run based on that integer.

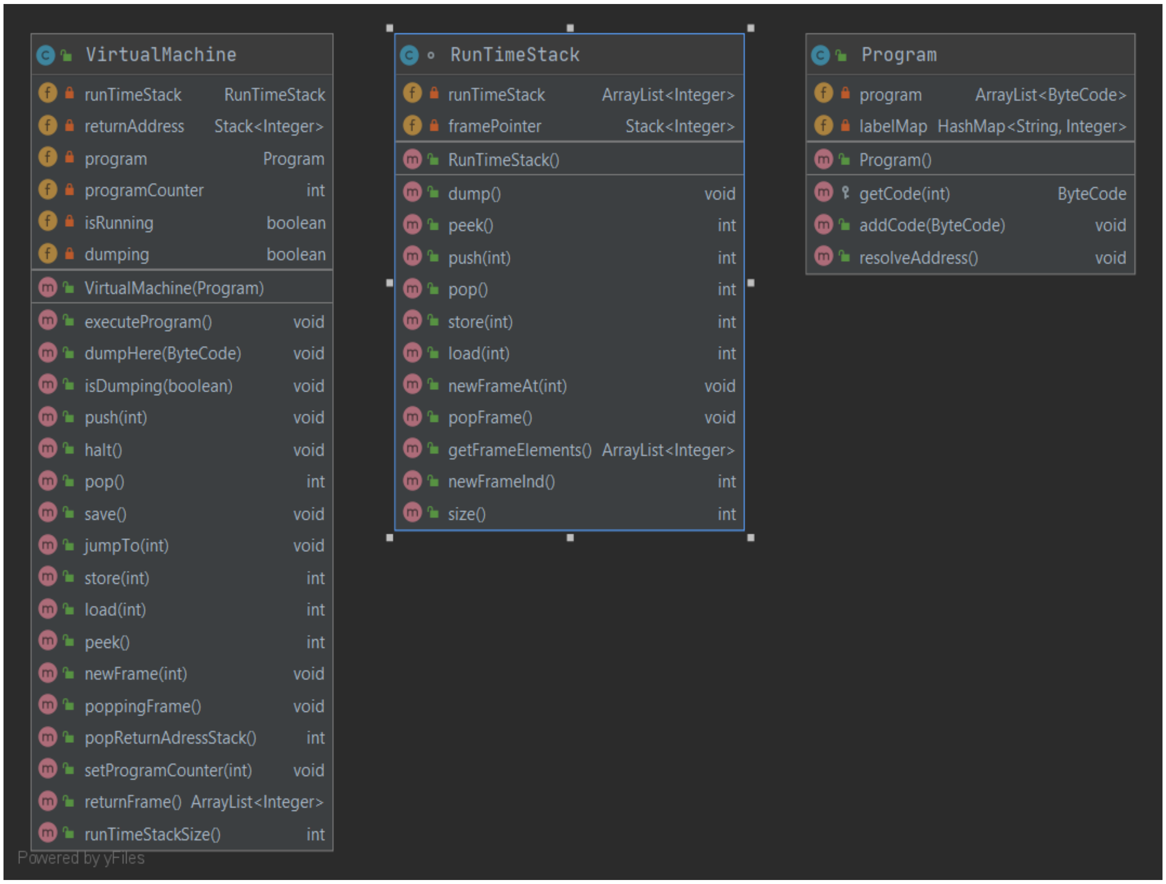
ASSUMPTIONS MADE

First and biggest assumption is that the code written in “cod” file is right, i.e. there are no errors in that code. Also, this project runs for very simple functions, if there is a new function added in the “cod” files, then the project will throw an error. The ByteCode is an abstract class, which is only there to tell us generally what the other classes like PopCode, FalseBranchCode etc. are doing. The implementation is not done in ByteCode class, but the child classes.

IMPLEMENT DISCUSSION

The files that I implemented are Program, RunTimeStack, VirtualMachine and all the files in bytecode directory. In my Program class, the most important method is “resolveAdress”, this method resolves the addresses of the machine language. Whenever, the machine language tells us to jump from one statement to other. I, first, put all the labels in a hashmap with labels as the keys, and their indexes as the value. Then, if the instance of the bytecode belongs to “GotoCode”, “CallCode”, “FalseBranchCode” or “ReturnCode” and then we going to grab their label and compare it with our hashmap and grab the appropriate value to get the address. I used “if” statements to check if the current bytecode belongs to either of this class using “instanceof”. In my RunTimeStack class, I implemented lot of simple functions like “pop”, “peek” etc. but the important function is “dump”. This method tells us what the code is doing in simple language. I used “if” statements in it too. The RunTimeStack has class has a stack, an ArrayList in it. The arrayList takes care of all the functions and pushes the result in it. The stack keeps check of the Activation Records of arrayList. In simple language, the stack stores pointers which basically point to the starting point of each frame in the arrayList, which is the runtime stack. The Virtual Machine is basically the middle man between our run time stack and the bytecodes. The byte codes and runtime stack classes don’t interact with each other at all. They interact through our virtual machine. Then is the bytecodes, which has all the functions that are used in the “cod” files. These classes revolve around two main methods that are “init” and “executeProgram”. These two classes’ implementation are different for each class. Also, each class has “toString” which basically prints whatever is going on in that class.





VirtualMachine	
runTimeStack	RunTimeStack
returnAddress	Stack<Integer>
program	Program
programCounter	int
isRunning	boolean
dumping	boolean
VirtualMachine(Program)	
executeProgram()	void
dumpHere(ByteCode)	void
isDumping(boolean)	void
push(int)	void
halt()	void
pop()	int
save()	void
jumpTo(int)	void
store(int)	int
load(int)	int
peek()	int
newFrame(int)	void
poppingFrame()	void
popReturnAdressStack()	int
setProgramCounter(int)	void
returnFrame()	ArrayList<Integer>
runTimeStackSize()	int

RunTimeStack	
runTimeStack	ArrayList<Integer>
framePointer	Stack<Integer>
RunTimeStack()	
dump()	void
peek()	int
push(int)	int
pop()	int
store(int)	int
load(int)	int
newFrameAt(int)	void
popFrame()	void
getFrameElements()	ArrayList<Integer>
newFrameInd()	int
size()	int

Program	
program	ArrayList<ByteCode>
labelMap	HashMap<String, Integer>
Program()	
getCode(int)	ByteCode
addCode(ByteCode)	void
resolveAddress()	void

Powered by yFiles

PROJECT REFLECTION

From my understanding, the code works very well and is up to the mark. However, for sure it can still be changed to a better code. There are some methods which I think are not at their best. For example, the method “dump()” in RunTime stack class, I used a lot of “if” statements to implement it, so I think it could’ve been implemented better.

PROJECT CONCLUSION

The program build, runs, and executes safely. The user can enter pretty much any integer, and the program runs fine. One thing that I would point out in this code that really helped me complete it was the debugger tool in IntelliJ, I was having a hard time figuring out where the problem in my code was, but debugger tool helped me get over it.