CSC 413 Project 2 Documentation
Summer 2024

Ulices Gonzalez

923328897

413.01

https://github.com/csc413-SFSU-SU2024/interpreter-ulicessgg.git

Table of Contents

1 Introduction

1.1 Project Overview

The Interpreter is a program that runs through a language that is a simplified version of Java. It makes use of what's called a byte code which are essentially commands for Interpreter to take and follow the processes that are included in each byte code. The interpreter takes x.cod files and splits up line by line in the program with the first part of the line being the byte code being called and the following part of the line being used to perform the commands inside.

1.2 Technical Overview

The Interpreter program runs through a mock language X which is a simplified version of Java that reads only the x.cod file type. Fifteen different bytecodes are created to ensure the interpreter is able to run each with three primary functions, init, execute, and toString with some having additional helper functions exclusive to them. The x.cod files are split up line by line in the program at run time with the first string of the line being the byte code token and the following string of the line being used as arguments for their respective init and execute functions. As each line is read the program handles the byte code called until all lines have been read and each byte code is done with its process. This is done both by the loader and code table working in tandem. For example, using the factorial.x.cod file will allow the interpreter to ask a user for a number and then calculate its factorial ex. !6 = 720. The interpreter also allows users to see the memory handling aspect of the program with Verbose mode which shows each byte code followed by information on the current state of the runtime stack.

1.3 Summary of Work Completed

To get The Interpreter up and running I began work on the runtime stack class by implementing its primary functions that the virtual machine class would use. From here I then created the abstraction for the byte code class and then its subclasses which then was followed by working on the Code Table now that the byte codes were implemented partially. In order to make sure I can start testing the program I then completed the byte code loader to make sure the files would be read at runtime and then begin working on each byte code and its corresponding virtual machine functions that would then link it to the runtime stack. I completed every byte code except the ones relying on resolve address which I then began working on alongside its dependent byte codes. To finish off my work I then reimplemented some of my runtime stack functions specifically popFrame and added exception handling. In the virtual machine, i then reimplemented its pop function as I had made it a generic function rather than what was specified in the documentation.

2 Development Environment

Version of Java Used: Oracle OpenJDK Version 22

IDE Used: Intellij IDEA 2023

3 How to Build/Import Your Project

1. In the command line enter
    a. git clone https://github.com/csc413-SFSU-SU2024/interpreter-ulicessgg.git
    b. or download the zip folder from the repository linked
2. If using Intellij you can run the IDE and on the top right click Open.
    a. From here navigate to the directory in which the interpreter folder is present and click ok
3. Once the project folder is open navigate to the interpreter folder
    a. Before building you must create a new configuration for the interpreter
        i. In the top right of the IDE to the left of the run button click on the arrow beside Current File
        ii. Click on Edit Configurations
        iii. Click on the + symbol and select application
        iv. Fill in the configuration info with the following
            - Name: Interpreter
            - Run on: Local Machine
            - Under Build and run select
                - java 22 as the SDK of 'interpreter'
            - Main class: interpreter.Interpreter
            - Program arguments: Any .cod file can be used
                - Examples: factorial.verbose.cod, factorial.x.cod, fib.x.cod, functionArgsTest.cod
            - Wokring Directory: …\interpreter-ulicessgg
                - … represents the path to \interpreter-ulicessgg
    b. To build Ctrl/CMND+F9

4 How to Run Your Project

1. After building the project you can run the calculator with the following
    a. To run with keystrokes use Shift+F10 or
    b. To run with the mouse and the IDE UI click the run button on the top right

5 Assumptions Made

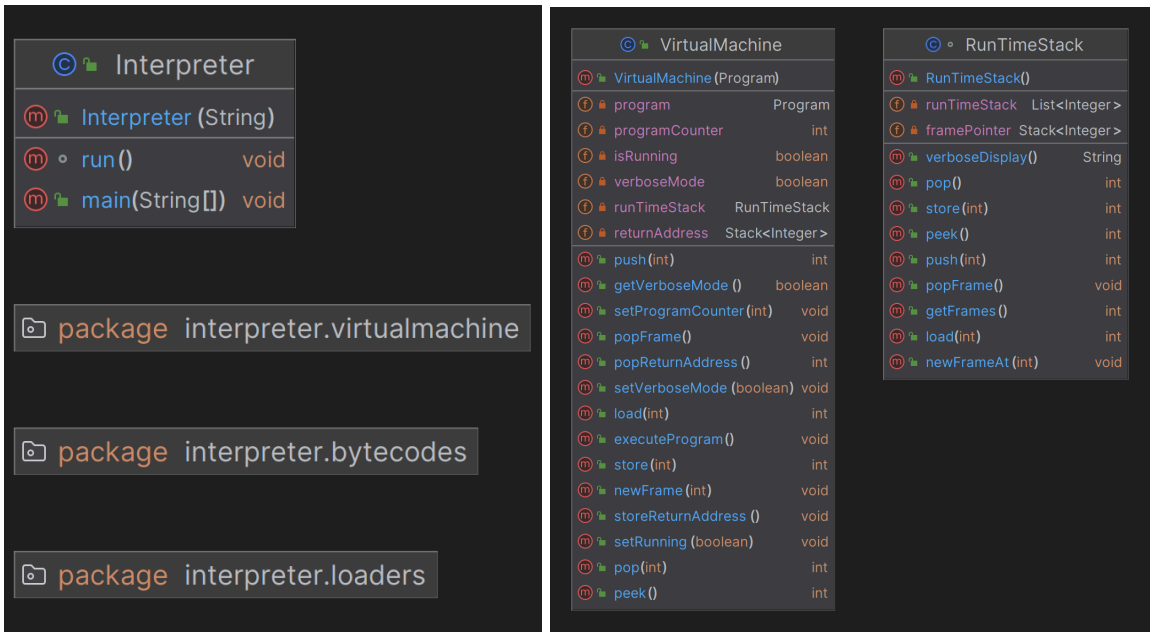
- No Decimals will be input
- No Negative integers will be accepted
- ByteCodes are created using ByteCode Abstract Class
- ByteCode abstraction provides three functions to subclasses, init, execute, and toString
- ByteCode subclasses will either use only two of its provided functions or use all of them with helper functions implemented based on necessity
- ByteCode init will skip over the first string in the list passed by loadCodes
- LabelCode, GoToCode, CallCode, and FalseBranchCode are reliant on the implementation of resolveAddress
- resolveAddress does two passes the first being to find instances of LabelCode which will populate a HashMap allowing for the addresses of GoToCode, CallCode, and FalseBranchCode to be resolved for further use
- .cod files will instruct the interpreter on a line-by-line basis after being read by loadsCode and each instance of a ByteCode alongside an argument will then reach the RunTimeStack
- ByteCodes access VirtualMachine.java through VirtualMachine vm
- VirtualMachine objects access RunTimeStack.java through RunTimeStack runTimeStack
- Interpreter.java is not edited
- InvalidProgramException.java is not edited
- Only .cod files can allow the interpreter to run
- A new Run Configuration needs to be created before running the interpreter

6 Implementation Discussion

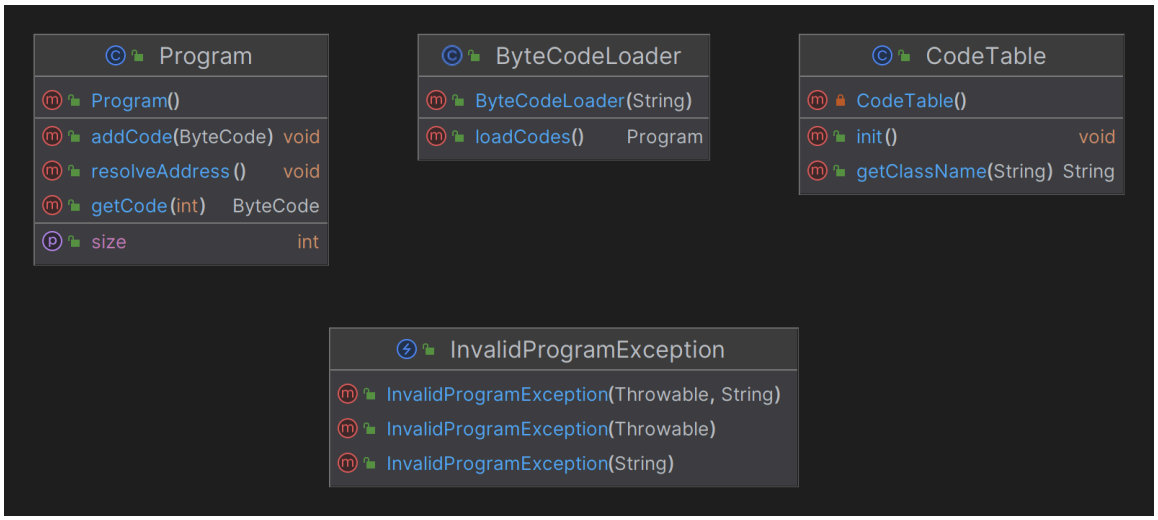
6.1 Design Choices

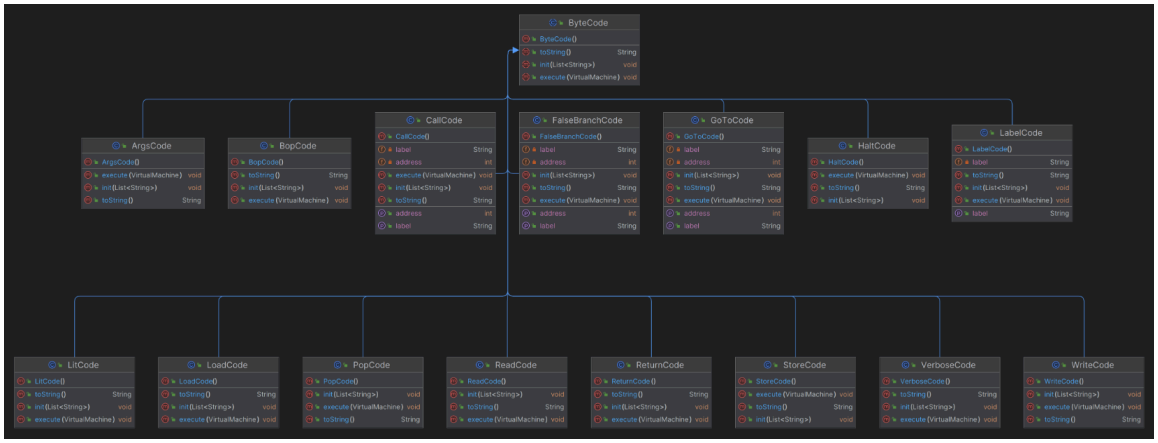
- Only positive non-zero integers are accepted when the interpreter prompts for input
- Code Table uses a HashMap to store Tokens and Class Names
- ByteCodes are implemented using an abstract class ByteCode
- ByteCodes check List of Strings passed by the loadsCode function for being empty prior to them being used
- Some ByteCodes have helper functions implemented to run their execute functions
- ByteCode subclasses make use of toString function for Verbose Mode
- All ByteCodes except HaltCode will output during Verbose Mode
- Exception handling is implemented into RunTimeStack functions
- BopCode uses a switch statements to handle all the different operators possible for equations
- Verbose Mode is handled by the VerboseCode Class which sets a boolean value in the VirtualMachine Class
- When enabled Verbose Mode will call the verboseDisplay RunTimeStack Function from inside the Virtual Machine
- Both RunTimeStack and VirtualMachine Functions will return values for ByteCodes to make use of
- VerboseCode handles the declaration of the boolean that turns Verbose Mode on and off

6.2 Class Diagram



6.3 Class Diagram (continued)





7 Project Reflection

The Interpreter was a bit of a challenge for me it was a straightforward project to understand for the most part but I mostly struggled with implementing functions that were alterations of widely used generic functions such as Pop and Push for example. This project did build off a lot of calculator projects we did before this so getting the implementation done for the abstract class for the bike code and its many subclasses was easy. I do have to admit I did struggle because I missed a few specific notes in the documentation we were provided most notably with the popcode since I had not realized that you need to take an argument to specify the number of elements being popped from the runtime stack. After having reimplemented some of the functions that I had mistakenly left finished it was a breeze from there with exception handling since I was able to easily identify where I should go and make sure that it wouldn't cause any errors in my project. Overall I think I did okay but there is a lot of room for improvement since I was trying to make sure that I met the requirements expected of us in this project just to make sure I played it safe and didn't overextend myself when I wasn't completely done.

8 Project Conclusion/Results

The Interpreter now works as intended with all its subclasses being properly implemented and able to run the factorial and Fibonacci test files provided to us. It is properly running through each file line by line without any issues and any exceptions being thrown. The only issue that potentially might come is from the verbose mode as it doesn't match exactly the outcome provided to us but it is outputting the correct information that it's intended to show to the users when it's on.