CSC 413 Project 2 - The Interpreter

Summer 2024

Bryan Lee

922649673

Section 1-R4

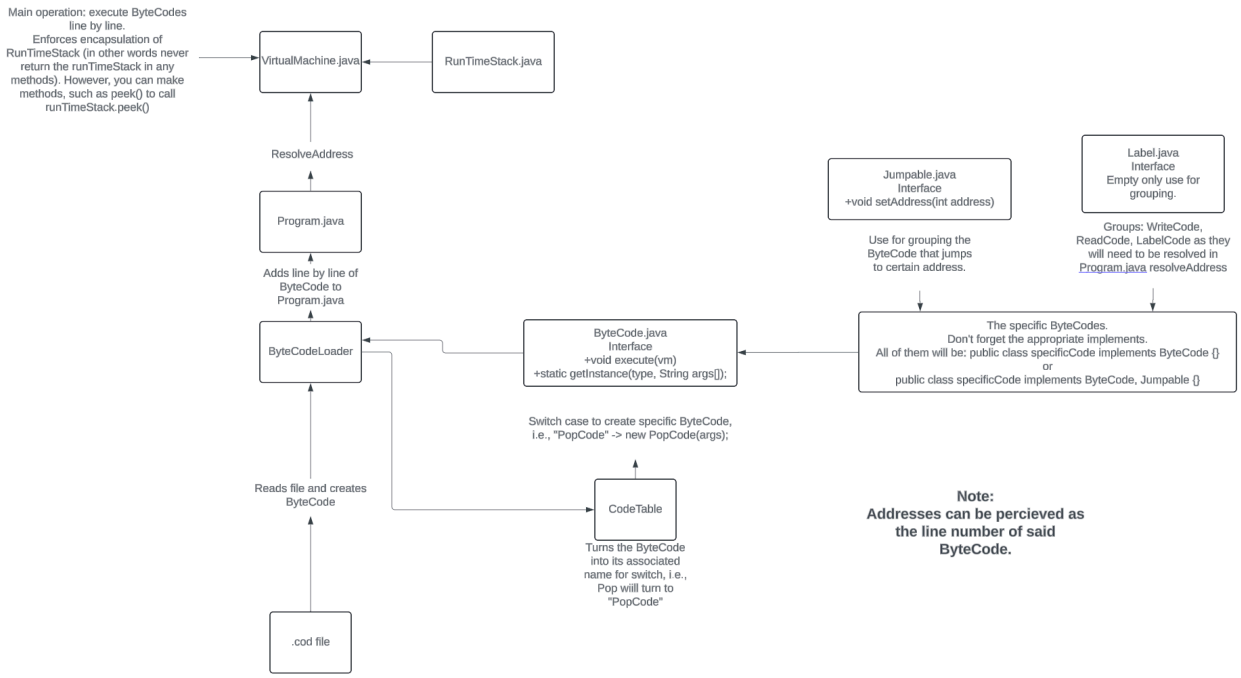
https://github.com/csc413-SFSU-SU2024/interpreter-BryanL43/tree/main

Introduction

Project Overview:
This program simulates an interpreter that translates a mock coding language called X, a simplified version of the popular Java coding language. The program uses the given X commands to simulate certain computer functions, such as calculating the Fibonacci sequence or retrieving the factorial for a certain number.

Technical Overview:
This project simulates a virtual machine that executes machine code, referred to as bytecodes, for a mock programming language called X. The virtual machine processes "cod" files, which contain these bytecodes—the compiled object code of the X language–that offers the ability to create variables, functions, input/output operations, conditional statements, and computations. The bytecodes control and manage the execution process of the runtime stack, simulating the step-by-step process most programming languages use.

Summary of work completed:
A critical aspect that helped me in the development process of this assignment was carefully reading the instructions, particularly the bytecode implementations. I used my foundation from the machine structure course to help interpret the breakdown of each bytecode's action. One of my unique ways of interpreting the project was perceiving the program's execution on a line-by-line basis. It was apparent when I saw the program counter variable in the virtual machine class. Treating the bytecode as its line helped me realize that jumping to a label simply moves the program's index to said address/location. To summarize the program, I've also continuously developed a diagram to visualize the process throughout the development. Here is the diagram/plan that I've made:



(Better visibility in the documentation > Interpreter_Diagram_Plan.png)

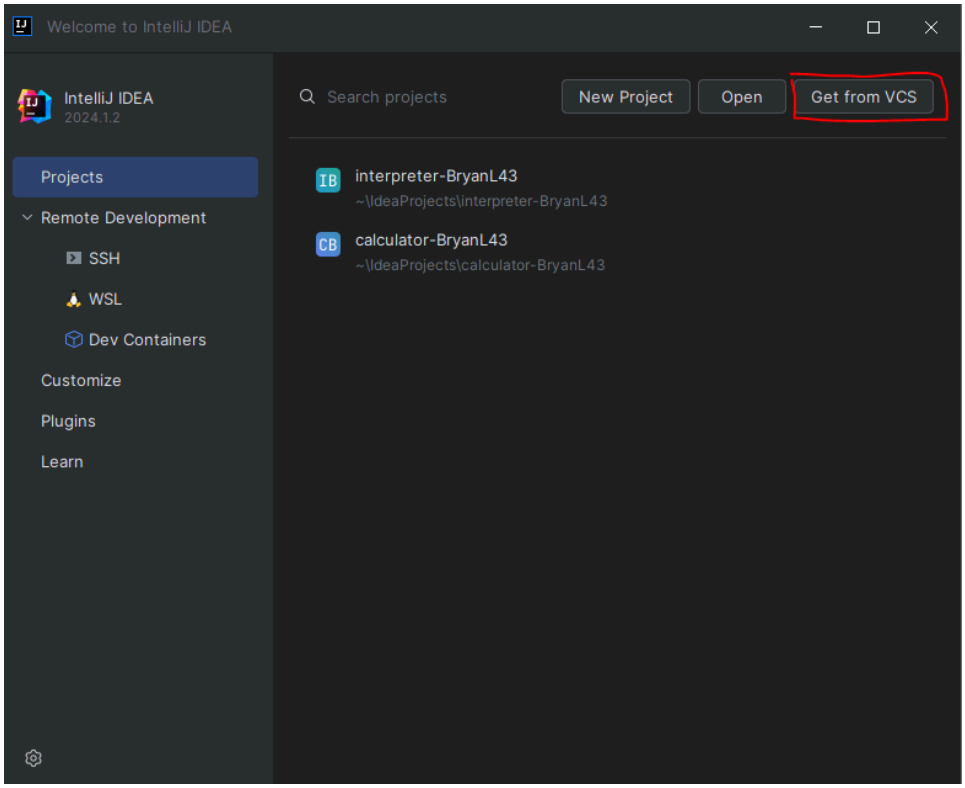
Development Environment

a) The version of Java used: Oracle OpenJDK 21.0.2
b) IDE used: IntelliJ

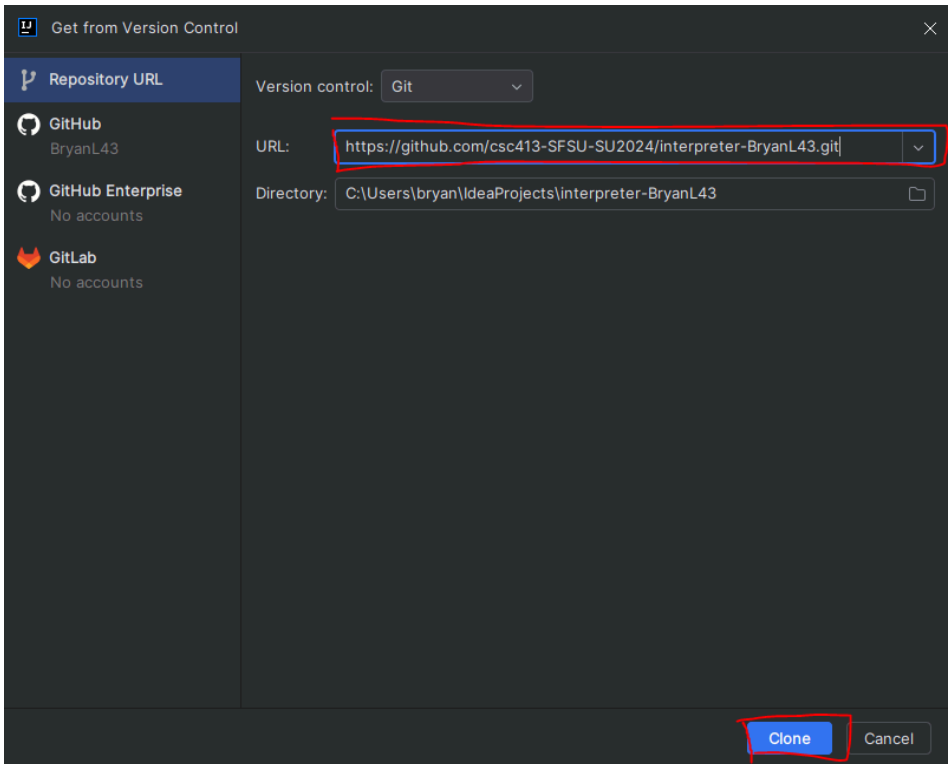
How to run/build the project

Cloning from IntelliJ:

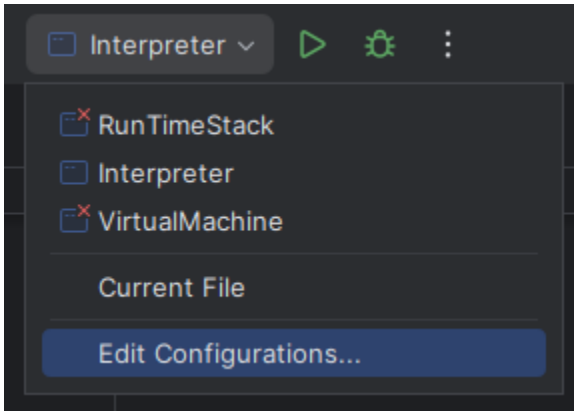
Step 1: Click “Get from VCS”



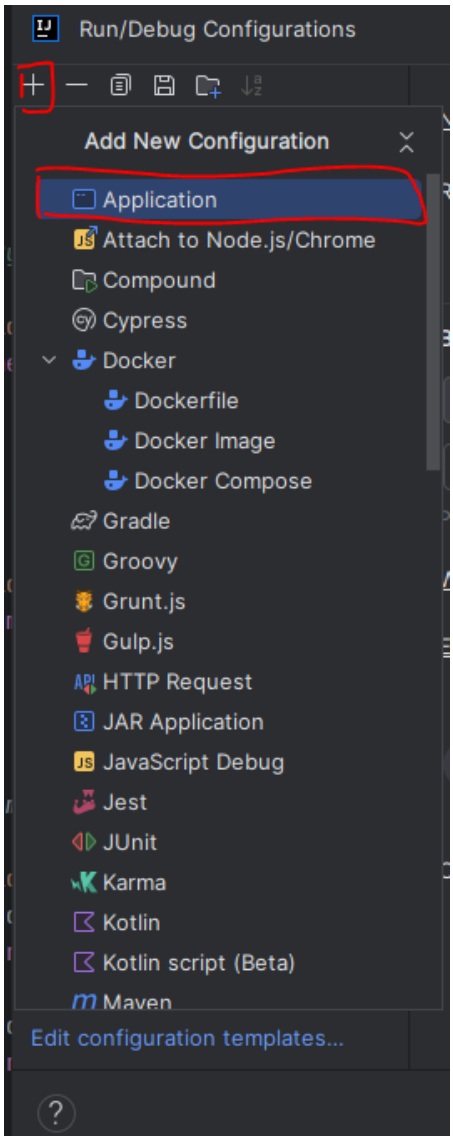
Step 2: paste the copied git link and click clone.



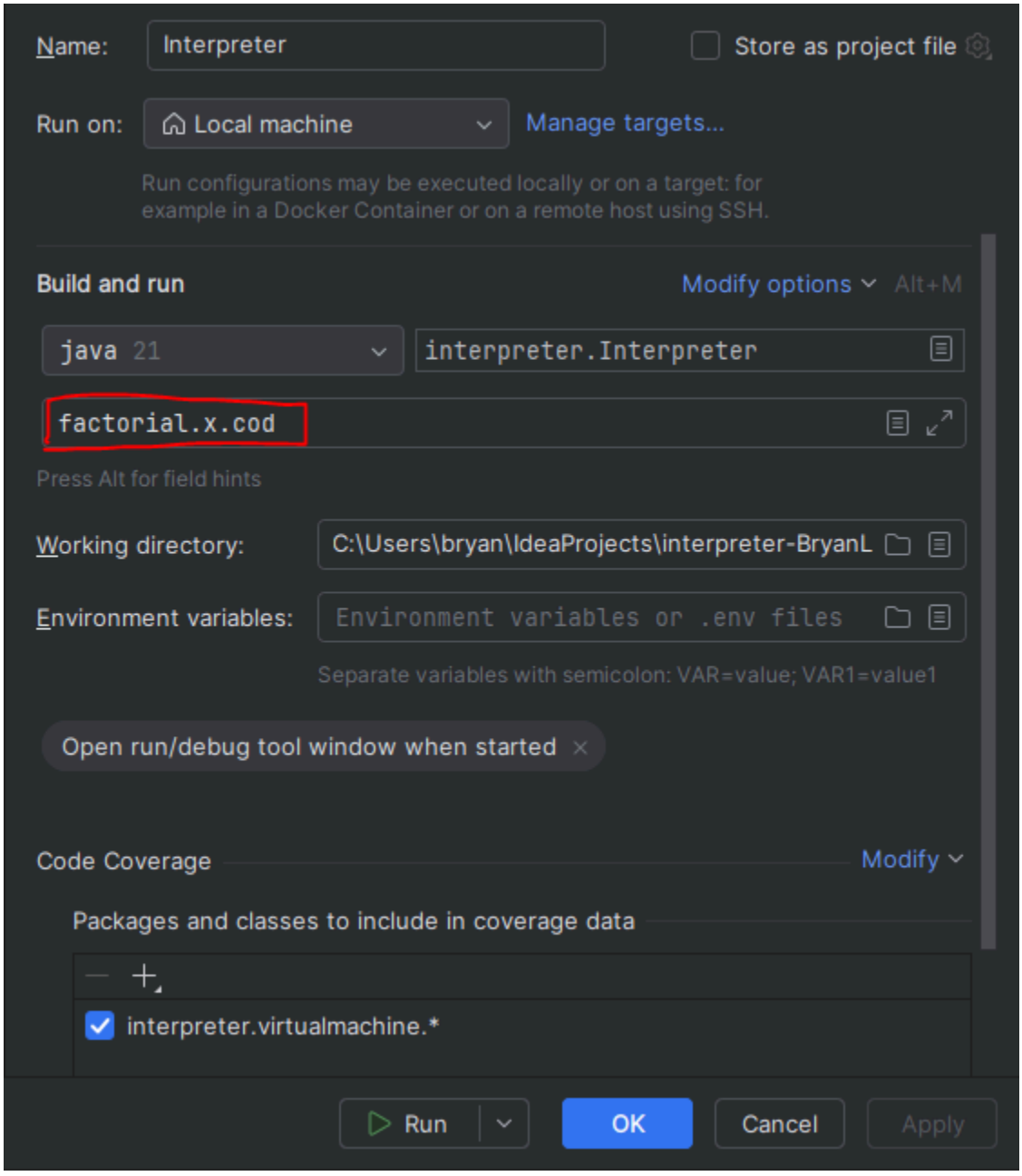
Step 3: Open up the configuration.



Step 4: Add a new configuration.



Step 5: Add the name, Java version, class name, and, most importantly, the red highlight part for the cod file you want to execute. Click Apply and OK to save the change. Now, you can run and compile the code.



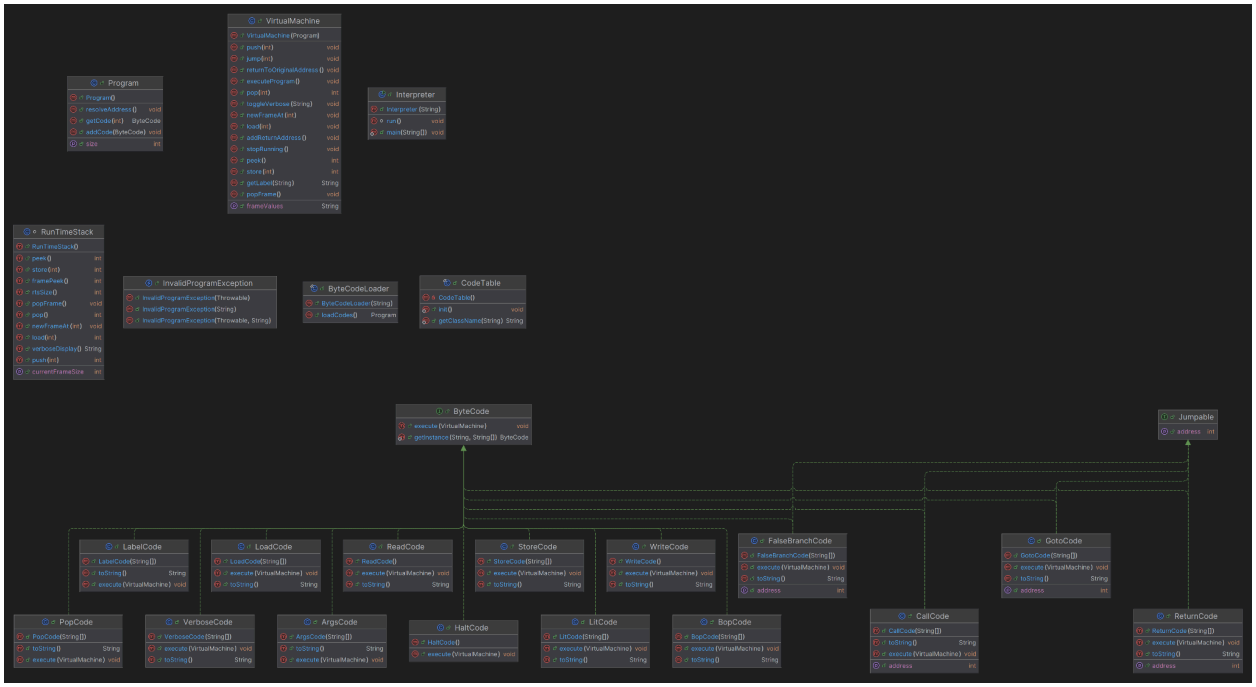
Assumption Made

An assumption I made was that interfaces could group abstractions, such as bytecodes and bytecode operations that require jumping. The abstraction helped override signature methods in each bytecode and assign their unique functionalities. Another assumption I made was the strict enforcement of the runtime stack's encapsulation within the virtual machine. The encapsulation means I cannot access the runtime stack directly from the bytecodes and must do so through custom methods in the virtual machine class. It also helps limit the access point to the runtime stack and prevent any unnecessary data from leaking out.

Implementation Discussion

Design choice:
A design choice that dictated my entire project was having an abstract class called ByteCode. The class contains a method to instantiate the individual specific ByteCodes, which the ByteCodeLoader will use. The design allowed me to parse the “cod” file’s bytecodes into a string and instantiate an object with it. Another design I chose was grouping the bytecode, which requires jumping under a unique abstraction. The grouping allows the Program class to resolve the jumping address efficiently to its associated index. The other design choice mentioned in the assumption-made part is where I enforce encapsulation of the runtime stack within the virtual machine class.

UML diagram:



A better-quality image is located in the documentation > UML_Interpreter.png.

Project Reflection

The project was enjoyable, but I had to plan quite a bit because I could only test the program once it was near completion. This brings me back to when I first learned the foundation of computer science. I habitually only tested code when it was near completion, which developed my debugging skills. The project provided a practical, hands-on application of the theories of OOP that I rarely used in 210 and 220. The project required me to draw forth my strategies for succeeding in operating systems to develop an efficient and working program while abstracting many cases before any code implementation. It is a nice change of pace from the earlier computer science courses, where I seek a solution without much thought.

Project Conclusion and Results

Initially, this project was annoying due to the long requirements for implementing the individual bytecodes. However, once I grasped a basic design, I could code all the pieces without the need for testing. Surprisingly, I only ran into one bug that stumped me a little: the BOP operation, where I popped the first operand first, but it should have been the second. However, it surprised me that my code was working flawlessly despite my rapid pace on this project. I could understand the entire project with just the instruction pdf and the lecture on the runtime stack and its frames. The program was an excellent refresher on my abstraction and encapsulation skills. Overall, most of the project was tedious work and consisted of repeats of the same interface overrides. It seemed daunting at the start but immediately became quite interesting.