CSC 413 Project Documentation

Fall 2018

Cory Lewis

917359162

CSC413.01

https://github.com/csc413-02-fa18/csc413-p2-mecharmor.git

Table of Contents

1 Introduction

1.1 Project Overview

This application is an Interpreter for language X. When programming, developers will 'compile' their code so that it can be executed. Essentially what is happening here is the high-level programming language is getting converted into a file containing a list of instructions. This interpreter takes a file of instructions and executes them. So, in basic terms, this is a program to run programs.

1.2 Technical Overview

The **Interpreter.java** class is the driver file for this program, it is the file that directs the rest of the program. The bytecode package contains **1** Abstract class that has **15** child Byte Code classes. Each Child Byte Code class contains an init and execute method which set the required values and execute whatever operation that specific Byte Code oversees. The **CodeTable.java** class is used to essentially Map the language X commands with the respective class files (hence why we use a HashMap). The **ByteCodeLoader.java** (loadCodes) class oversees reading the text file and creating new instances of the read Byte Code which invokes the resolveAddrs() method before returning. **Program.java** holds the ArrayList containing ALL the instantiated Byte Codes and also contains a method resolveAddrs() which sets the memory address for commands that jump to LABELS. **RunTimeStack.java** manages the runTimeStack and framePointer stack which are used to keep track of function calls and variables. **VirtualMachine.java** loops through all the Byte Codes contained within **Program.java** and executes them. Important pieces of information regarding the Virtual Machine is that it keeps a counter (PC) that keeps the virtual machine on track to execute a specific Byte Code. The returnAddrs stack in Virtual Machine oversees storing the counter when a function is called (This is so the "RETURN" byte code can pop that value and set the new PC).

1.3 Summary of Work Completed

I created the necessary byte code files within bytecode package that language 'X' uses. I implemented the loadCodes() method within **ByteCodeLoader.java**. A helper method for resolveAddrs() was added to **CodeTable.java** which helps me verify byteCodes that need address resolution. resolveAddrs() was implemented within **Program.java** which iterates through all the read byte codes and checks to see if the correct memory address needs to be set for it. **RunTimeStack.java** has many helper methods that control the usage of the runTimeStack and framePointer stack. **VirtualMachine.java** implementation involves many get and set methods with some additional methods that invoke methods within **RunTimeStack.java**. An if statement was added to the executeProgram() method to verify if dumping was enabled or not.

2 Development Environment

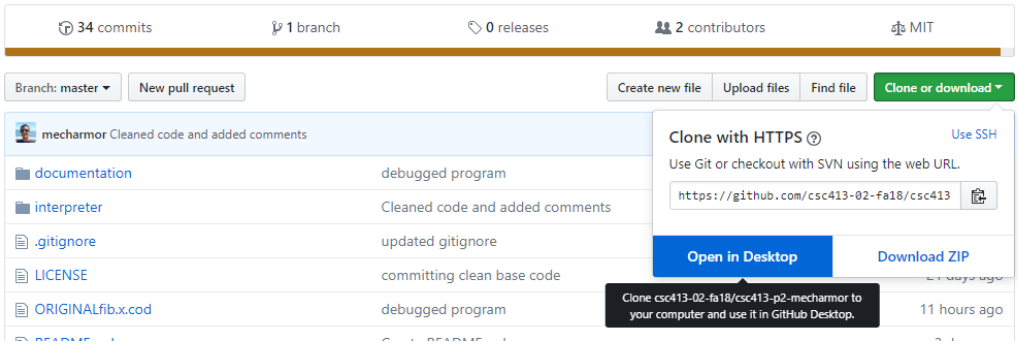
Java Version Used: **jdk-10.0.2**

intelliJ IDE used for development. Version: **2018.2.3**

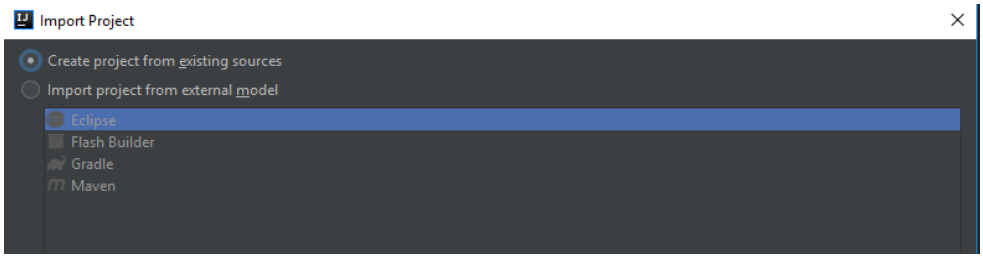
3 How to Build/Import your Project

1) intelliJ IDE download link: https://www.jetbrains.com/idea/
2) GitHub desktop download link: https://desktop.github.com/

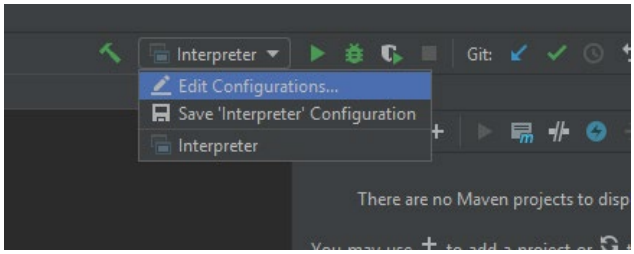
3) Clone Repository through GitHub desktop
    a. Navigate to this link (if you have permissions to view):
        i. https://github.com/csc413-02-fa18/csc413-p2-mecharmor
    b. Click “Clone or download”
    c. Choose “Open in Desktop” (Photo illustration below)



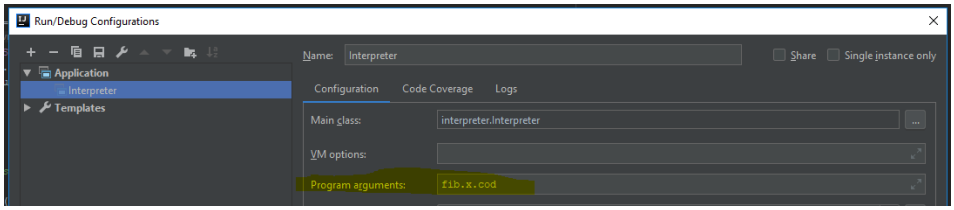
4) Once your repository is cloned you can launch intelliJ and import using “existing sources”.
5) When importing select: “Create project from existing sources” (photo illustration below)



6) Once the project is imported you will need to “edit configuration” on the top right of the screen



7) The “Program Arguments” text box is where you put the file name of the file that needs to be read in.

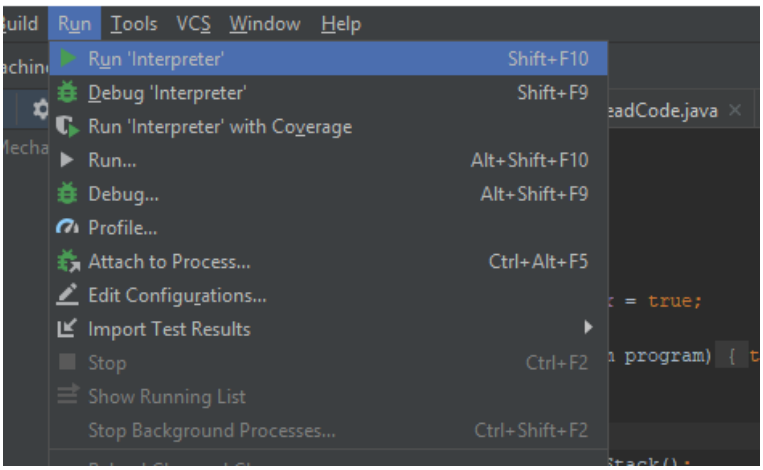


8) After that you are ready to run!

4 How to Run your Project

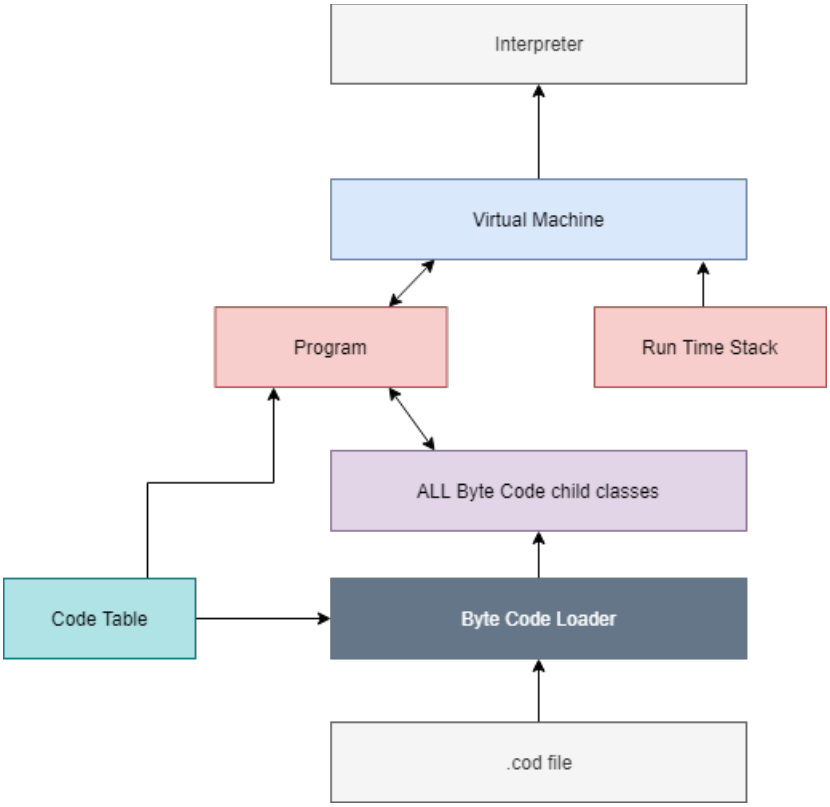
After completing the import on the steps above, the way you can run your project is either by clicking the **big Green Play button on the top right** of intelliJ or by navigating to the top navbar and

choose: “**Run ‘Interpreter’**”. Also you can use the hotkey: “**Shift+F10**”.



5 Assumption Made

When designing this program, I had a difficult time wrapping my head around the flow of the program and which class drives other classes. This is the assumption I made regarding program flow and class relationships:



Program flow & relationships would be: Byte Code Loader reads the .cod file. While reading the file, the Code Table class is used for verification and byte codes are instantiated. The Program class has a global Array List that has direct access to all Byte Codes. The Program class can verify byte codes using the Code Table class. Run Time Stack class does not have a relationship with the child Byte

Codes and its sole purpose is to maintain the run time stack for the Virtual Machine. The virtual machine ties everything together. The VM can access the Array List of Byte Codes through the program class. The VM can manage the run time stack. The VM is ran by the Interpreter.

6 Implementation Discussion

ByteCodeLoader.java

a. loadCodes(void) → Returns a **Program** object
    i. Reads each line of the program file
    ii. Obtains the name of the read bytecode and uses reflection (HashMap in CodeTable.java) to determine it's class name.
    iii. bc.init(arguments) passes an ArrayList to the given bytecode for it to set the required values
    iv. Before loadCodes() returns, it invokes the resolveAddrs() method to set the proper memory reference counter to the byteCodes that jump to labels.
    v. Populated program object is then returned

CodeTable.java

b. validLabelByteCode(String) → returns **Boolean** value
    i. This method is used during the resolveAddrs() phase.
    ii. This is a verifier method to determine if the given byte code needs it's address resolved or not. (Add more byte code names here if future commands need to be implemented)

Interpreter.java

c. This is the Driver class for the whole project.
d. When the main method runs takes in the given argument (which is the text file name) and invokes the run method which instantiates the program and virtual machine.

Program.java

e. resolveAddrs(**void**) → **void** return. Only modifies it's private ArrayList data field (program).
    i. This method uses two data structures to efficiently check for address resolutions.
        1. DefinedFunctions(HashMap) which holds the LABEL names as the 'key' and the 'value' is the memory address.
        2. noMatchFunctionStack (Stack) which holds ALL the valid label byte codes that could not be verified during the first pass of Iteration.
            a. This design allows the definedFunctions HashMap to resolve addresses DURING the first pass and if no match could be found then we assume the future LABEL was not put into the HashMap yet, so push to the noMatchFunctionStack.
    ii. resolveAddrs contains **two** major parts for iteration through our ByteCodes.
        1. Iterating through each byte Code and **resolving DURING** the iteration

2. Once the first pass is complete we will start popping the noMatchFunctionStack and comparing with the now populated definedFunctions HashMap.
3. Note: I threw an exception if no value could be found in the HashMap while popping noMatchFunctionStack because this implies that the given ByteCode program is requesting a label that does not exist.

RunTimeStack.java

f. dump(**void**) → **void** return.
   i. Uses 'StringBuilder' to append to 'sb' for all the values contained within the runTimeStack.java
   ii. Note: A temporary runTimeStack.size() value was pushed to the framePointer stack so my iterator can go through ALL values in the runTimeStack. This assumption was made because the top of the framePointer stack does not mark the top of the runTimeStack.

g. getRunStackAboveTopFrame(**void**) → **ArrayList<Integer>** return
   i. This method is only used by a very few amount of ByteCodes. But essentially what this method does is take ALL the values in the current frame and return an ArrayList of those values. Example: [0,1] → Returned values would be: 0 & 1.

h. Peek(**void**) → **int** return
   i. This peeks the top of the runTimeStack and returns that value

i. Pop(**void**) → **int** return
   i. Pops the top of the runTimeStack as long as the runTimeStack does not pass the current frame.

j. Push(**int** or **Integer**) → **int** or **Integer** return
   i. Two types of this push exist.
      1. One pushes a primitive integer value to the runTimeStack
      2. The other pushes an Object of type Integer to the runTimeStack

k. newFrameAt(**int**) → **void** return
   i. Creates a new frame which is 'offSet' amount index down from the top of the runTimeStack. Note: This value is pushed to the framePointer stack.

l. popFrame(**void**) → **void** return
   i. saves and pops top value off the runTimeStack
   ii. saves and pops top value of the framePointer stack
   iii. While loop that pops everything above the framePointer index. *Ex: [0,1][2,3]. 3 will be saved. 2 will be popped and the new runTimeStack will be: [0,1,3]*

m. Store(**int**) → **int** return
   i. Pops the top of the runTimeStack and inserts that given value at the position of: framePointer.peek() + offset.
   ii. Returns popped value.

n. Load(**int**) → **int** return
   i. Copies the value at runTimeStack index of: framePointer.peek()+offset.
   ii. Adds the copied value to the top of the stack.
   iii. Returns copied value.

VirtualMachine.java

o. executeProgram(**void**)→ **void** return
   i. loops through all values inside the program Array List.
   ii. isRunning Boolean value is modified by the HALT bytecode to stop the program
   iii. dumpRunTImeStack Boolean value allows the executeProgram() method to dump the values inside the runTimeStack
   iv. returnAddrs stores PC values when function calls are made so the RETURN byte code can set the new PC value to returnAddrs.pop().

p. dumpRunStackStatus(**Boolean**) → **void** return
   i. Allows byteCodes to turn ON or OFF dumping.

q. popReturnAddrs(**void**) → **int** return
   i. pops value off of the top of the returnAddrs stack.
   ii. Note: this should only be used by Byte Codes that Jump Back after a function call.

r. pushReturnAddrs(**int**) → **void** return
   i. pushed value onto the top of the returnAddrs stack.
   ii. Note: this should only be used by Byte Codes that invoke function calls.

s. setPC(**int**) → **void** return
   i. sets new Program Counter value

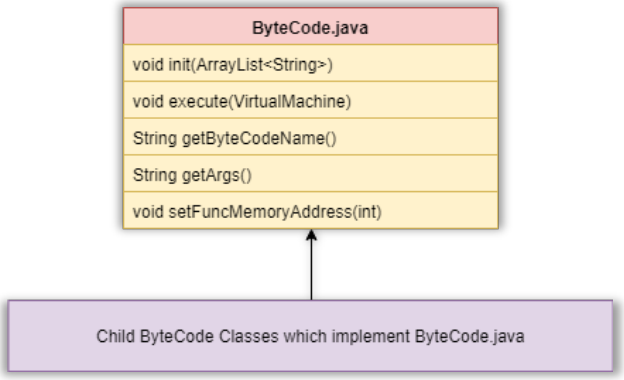
t. getPC(**void**) → **int** return
   i. gets current value of Program Counter

u. setRunningStatus(**Boolean**) → **void** return
   i. Used ONLY by the HALT byte code.
   ii. This method stops the program.

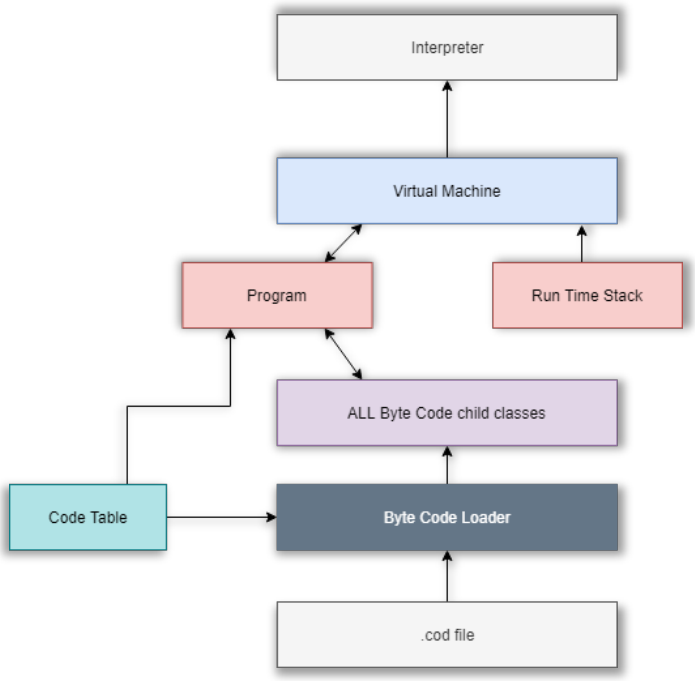
v. All other unmentioned methods contained within virtual machine **invoke** methods inside of **RunTimeStack.java** so please see documentation for **RunTimeStack.java** *for details.*

6.1 Class Diagram

Below is a diagram that illustrates the inheritance of the ByteCode classes:



Class Relationship Diagram: (Explanation in Assumptions area sec. 5)



7 Project Reflection

After spending many hours working on this project. I should have started earlier so I wasn't so slammed in the end for time. The guidelines were tricky to understand and the project itself was easy to make a logic mistake that breaks everything. Those notes aside, I really enjoyed the challenge that this project presented. I had to read the guidelines for the project roughly 8+ times to understand what I needed to do. The class Slack was also extremely helpful in keeping me on track with any logic issues I had.

Debugging became difficult once I finished most of the programming, however debugging became much easier once I enabled the dump() method. Each Byte Code presented its own challenge in regards to proper implementation and it's usage with the Virtual Machine. Additionally, I had to re-read the guidelines multiple times to make sure I was not breaking encapsulation or any other requirements that were noted.

8 Project Conclusion/Results

This project challenged me in my understanding of data structures, program design, and class inheritance. My biggest challenge, at first, was admitting to myself that testing will be limited until every class is implemented. I wrote roughly 1,460 lines of code before I could start testing, this was scary, but with the help of intelliJ debugging I managed to fix any logical errors I made, and the program matched the required output.