CSC 413 Project Documentation

Fall 2018

Steve Tu

918460002

CSC413.02

https://github.com/csc413-01-summer2019/csc413-p2-steve1316

Table of Contents

1 Introduction

1.1 Project Overview:

Assignment 02 features an interpreter for files containing a series of bytecodes that the interpreter would then need to convert into an executable format via Java. The Program would then resolve the addresses of the bytecodes that jump to their specified labels. After that, the Virtual Machine would then execute the bytecodes using the method of encapsulation to hide the functionality of the bytecodes' execution. Through all of this, it replicates how the actual runtime stack works in terms of storing and removing values from the stack in Java.

1.2 Technical Overview:

First, the bytecodes are read line by line inside the ByteCodeLoader by grabbing the name of the bytecode and referencing it to the appropriate class name inside the CodeTable. After creating the bytecode object along with any arguments it may have had, it would then send the bytecode to the Program. The Program would then replace any addresses that the JumpCodes targets with the appropriate PC integer so the Virtual Machine knows where to jump to next. The Virtual Machine, with the completed Program object with its resolved addresses, now goes through each bytecode and executes them. It does not know what the bytecode is so it just calls the inherited method, execute(), and by abstraction the workflow would know exactly which bytecode to execute. The bytecodes' execute() method would then call upon the Virtual Machine object to execute the relevant operations utilizing the RunTimeStack methods. Additionally, the runtime and frame stacks are used to keep track of integers that the bytecodes utilize and it is meant to replicate the workings of the actual runtime stack of Java. If a new frame is created using the bytecode Args, it would then keep track of the values that belong to their associated frame in order to reflect how scopes and the functionality of methods work in Java.

1.3 Summary of Work Completed:

I first started by following the steps in the Coding Hints section of the instructional PDF for this assignment. After creating all the bytecode classes and making sure the abstraction was clear, I implemented ByteCodeLoader by using and tweaking the provided code under the CodeTable section of the PDF in order to capture the name of the bytecode and making a class from it. I would then make a new bytecode object out of the class and then I would send whatever given arguments I had to the bytecode and then finally resolved the jump addresses into a brand-new Program object. Next, I implemented the resolveAddrs method in order to fix all of the jump addresses so that the Virtual Machine would know where to jump to next when it encounters a JumpCode bytecode. I utilized a HashMap in order to store all of the Labels in a for loop and then used another for loop to replace each designated address of JumpCode bytecodes with their correct addresss to jump to. Then, I moved on to RunTimeStack and I implemented all of the functionality necessary like peek, push, pop, etc. in regards to operations done to the runtime and frame stacks and also the creation and destruction of frames as dictated by the bytecodes. For the Virtual Machine, I just simply used the code provided under the Virtual Machine section of the PDF while also tweaking it to make sure it can tell whether or not the dump state is on in order to execute dumping. After that, I implemented the relevant methods in order to call upon the RunTimeStack operations so that the bytecodes can utilize its methods in their execute() methods. After all that is done, I went back and finished up the empty methods in the bytecode classes, making sure that the appropriate bytecodes inherited from ByteCode or JumpCode, and that their toString() methods worked properly for dumping. When all of that was done, I went back to the RunTimeStack and implemented dumping functionality.

2 Development Environment:

I used Java SE Development Kit 12.0.1 and IntelliJ IDEA.

3 How to Build/Import your Project:

First, I downloaded the GitHub repo onto my Desktop. After that, I opened up IntelliJ and selected the “Import Project” option. I then browsed to my Desktop folder and selected the entire repo folder such that the root of the project was this folder and not the interpreter folder inside it. I then made sure “Create project from existing sources” was selected and made sure that the Source file was the correct path so that the root was the entire repo folder itself. After that, I made sure that the project SDK used was “jdk-12.0.1” on my C drive and then click “Finish”. Now I have my completed project in front of me in IntelliJ and I want to build it. I go up to the “Build” menu at the top of the IntelliJ window and from the dropdown menu, selected “Build Project”.

4 How to Run your Project:

Now that the project has been built and I want to debug it, I hovered my mouse over the “Edit Configurations” option of the Debugger and selected it. I clicked on the “+” icon at the top left of the screen that popped up and selected “Application”. After entering in a new name, I go down to the “Main Class:” section and selected the “interpreter.Interpreter” so that the program would start from the Interpreter class. Then I moved to the “Program arguments” and entered in the name of the tester that I want to debug this program with. These tester files have a “.x.cod” extension to the names so I entered in the one that I want. Clicking “OK” would finalize this tester so now I can run and/or debug this program however I want.

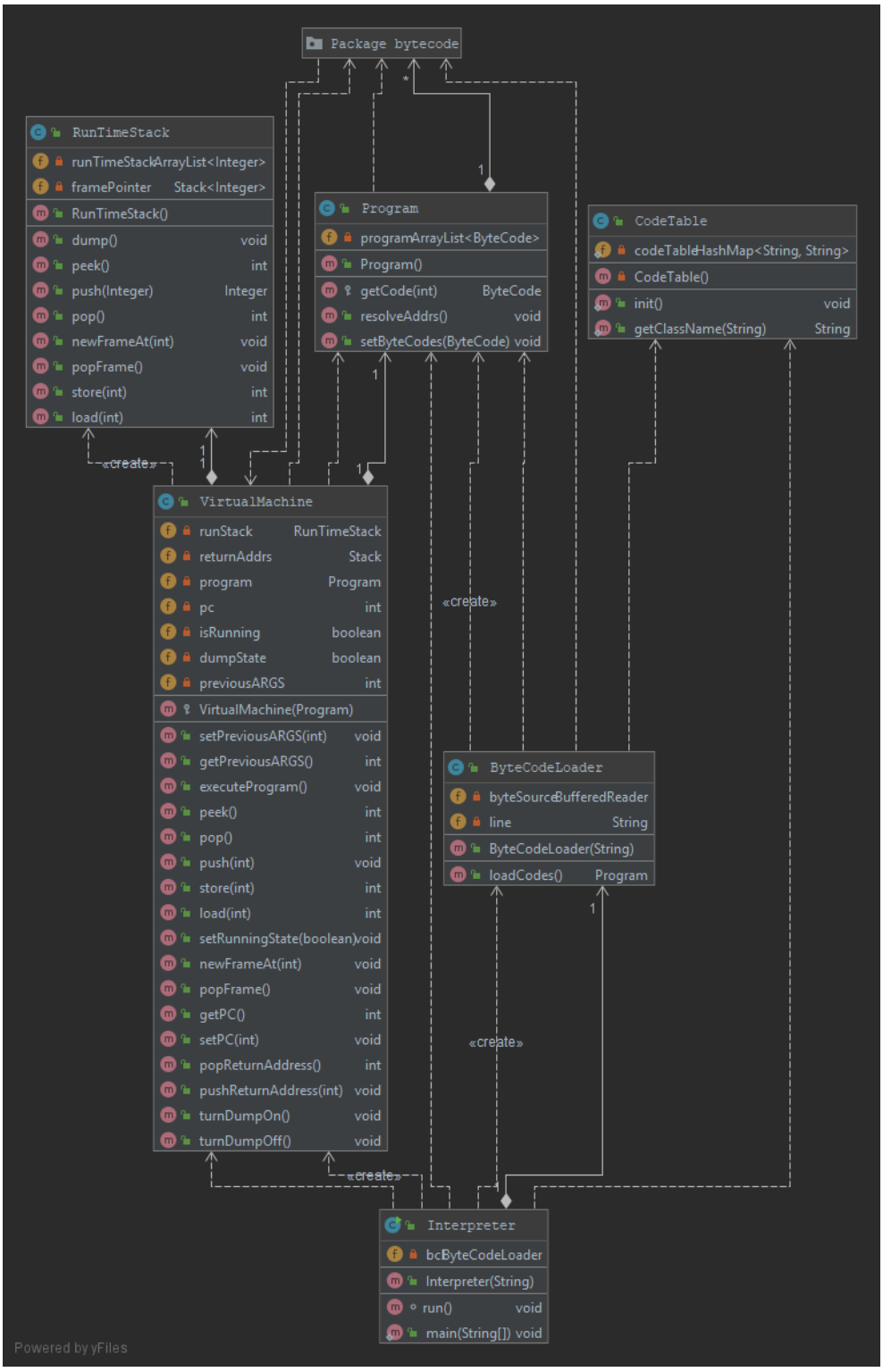
5 Assumption Made:

I know that the grader would only utilize bytecodes that would not be purposefully misspelled and that the data types that we would be handling were integers. We are, however, expected to encounter bytecodes that would intentionally try to destroy our runtime stack and/or disrupt the workflow of our interpreter program, like “POP 66”. We would need to handle these in a manner that does not propagate any errors back up to the Interpreter level. One assumption that I made was that I was free to use the code snippets provided in the instructional PDF as

6 Implementation Discussion:

For the ByteCodeLoader, I used the code snippet from the PDF as a base and tweaked the variables to make sure that they worked. I also put it all in a try-catch block in order to prevent an error. The arraylist created inside the loadCodes() is used to hold any arguments that the bytecodes read in the file came with so that I can initialize them while creating the bytecode object. For the Program's resolveAddrs(), I created a HashMap to store each Label along with their associated line number read from the file. The first for loop will go through each bytecode stored in the program object and if the bytecode was a LabelCode, store its label string into the HashMap along with its line number integer. After that, the second for loop will again go through each bytecode and if the bytecode is an instance of JumpCode like GotoCode, store its label string and use that string as a key in order to get the value or line number from the HashMap and store that into the targetAddress integer. The integer will then be sent to the bytecode in order to set it so the Virtual Machine knows where to jump to next. For the RunTimeStack, dumping is done via loops. If there is only one frame, print the runtime stack using its toString(). If there is more than one frame, I used iterators to go through the runtime and frame stacks in order to print out each element in the frame and then when I reach the end of the current frame and if there is more, close off the brackets and continue printing. The peek() just copies the value at the top of the runtime stack and returns it. The push() just simply adds the value to the top of the stack. The pop() will save the value at the top of the stack before removing it and I have checks in place to see if there is only one element left in the stack and if the stack is already empty. The newFrameAt() pushes the position of the first element of the new frame onto the frame stack. The popFrame() saves the value at the top of the stack and I used a for loop to remove every element in the current frame before finally popping the frame from the frame pointer stack. After that I put back the saved value back onto the runtime stack. The store() checks if the runtime stack is empty or not and then saves the value at the top of the stack and then removes it. It will then go to the position from the beginning of the current frame plus the offset provided by the bytecode and then replace the value there with the saved one from earlier. The load() will check if the runtime stack is empty or not and then saves the value at the beginning of the current frame plus the offset provided by the bytecode. It will then place it on top of the runtime stack.

6.1 Class Diagram



7 Project Reflection:

This was obvious, but this was a more significant assignment that I ever had to do so far. The huge PDF provided was daunting at first and the number of bytecodes that I had to implement was baffling. But as I started to get more familiarized with the flow of the interpreter and how everything comes together utilizing each other, I began to start working on the project in small chunks so that I can keep a relatively clear picture on the end goal of the particular class/method. I never had much experience using StringTokenizer, HashMap, and Iterators but they made my life so much easier once I learned more about them and how to use them. At the end, I'm feel a little bit more confident about Java and coding in general now that I'm done with it all. I was able to successfully complete it on time with a lot of debugging needed to make sure the functionality of the RunTimeStack was correct and that I was dumping the bytecodes in the correct format. I got a lot of insight from Slack and asking my fellow classmates on how best to tackle each section of this assignment.

8 Project Conclusion/Results:

In conclusion, I was able to finish this assignment and was able to successfully pass the tests given to us in the repo and from Slack. I believe that I completed all the requirements needed from us in regards to the implementation of the bytecode classes, ByteCodeLoader and the creation of new bytecodes from the read file, Program with its resolving addresses capability, RunTimeStack's manipulation of the runtime and frame stacks, and the Virtual Machine executing the whole program. I also believe that I maintained encapsulation such that bytecodes never directly access the data inside the stacks and that dumping only interacts with the runtime and frame stacks and not the bytecodes themselves.