CSC 413 Project Documentation

Fall 2019

Nimiksha Mahajan

917219230

CSC 413.01

https://github.com/csc413-01-fall2019/csc413-p2-Nimiksha

Table of Contents

1 Introduction

1.1 Project Overview

This project aims to build an interpreter for a mock language X, which is quiet similar to Java. The interpreter class reads code from the source and processes it to give a result. X language uses a group of byte codes such as Load, Store, Read, Write, False Branch, Call etc. and each of these Bytecode classes perform a specific function. The program essentially translates the Bytecodes into Java and then process the commands by implementing the required functions.

In this program, we run the test on 2 .x files, which calculate the Factorial and the Fibonacci series sum. We are given the source code for these 2 programs, and we translate from its Bytecode state to Java and return the result. Additionally, if we turn Dump 'ON', we return the whole stack, which breaks down the process by which we get to the final result.

1.2 Technical Overview

A majority of this program is to create the 15 ByteCode classes which are used to process the .x files. We have a bytecode package which has all the bytecode classes, including the parent ByteCode class and the 15 children classes. all the children classes have the init(), which initializes the variables, the execute(), which take the VirtualMachine object and implements the actual functionality and the toString(), which returns the value dumped from the stack. Bytecode objects are created as the .x.cod files are being read, and these objects are added to the program object.

This program object uses an ArrayList to store all the bytecode objects (the label and addresses are stored in a HashMap though). Also, it is used to resolve the address, and it has the GotoCode, FalseBranchCode and the CallCode. All these bytecode classes need to jump to another label in order to execute the next line of code.

The RunTimeStack class, predominantly uses an array and a stack to carry out its functionality. The Stack is used to store the frame pointer and the ArrayList contains the input values in the process. The class uses these data structures to push a new frame whenever a function is called or to remove the frame after it has been handled.

The VirtualMachine class is the class that actually executed the program by working with the RunTimeStack and the returnAddrs stack. The VirtualMachine also works with the dump() and is responsible for having the processes of the runtime stack (i.e. The various steps in the calculation of Factorial/Fibonacci) being printed to the console.

It is important to note, that throughout the implementation of these functionalities and data structures, try-catch techniques are applied and the exceptions are captured to see if the program is running correctly or not.

To sum it all, the interpreter class starts the program and initializes the code table, from where the values are loaded from byte code to the program object. The Virtual Machine then comes into play and uses the program object to execute to program and return the runtime stack.

1.3 Summary of Work Completed

We were given a skeleton for some of the classes along with some classes (CodeTable and Interpreter) already completed. I completed the implementation for the following classes:

- ByteCodeLoader: The ByteCodeLoader class is responsible for loading bytecodes from the source code file into an ArrayList, and resolving the addresses.

- Program: The program class is mainly responsible for storing all the bytecodes read from the source file into an ArrayList which have a designated type of ByteCode (and its subclass) it can store.

- RunTimeStack: The RunTimeStack class uses frame stack and runtime stack to record and process the stack of active frames (push a new frame when the function is invoked or remove it when it has been processed). This class assists the Virtual machine in executing the program using an ArrayList and a Stack.

- VirtualMachine: The VirtualMachine class is the one responsible for actually executing the Program. All the operations that occur go through the Virtual Machine class, where an instance of the RunTimeStack class and the returnAddrs stack are created, to execute the bytecodes.

- ByteCode Classes – The ByteCode is the parent class and the 15 children classes including ArgsCode, BopCode, CallCode, DumpCode, FalseBranchCode, GoToCode, HaltCode, LabelCode, LitCode, LoadCode, PopCode, ReadCode, ReturnCode, StoreCode and WriteCode. These classes inherit the ByteCode methods and are used by the Program class to carry out the functions.

2 Development Environment

The development environment used for this project was IntelliJ Ultimate Version 2019 2.1, with Project SDK, JAVA version 1.8.0_121.

3 How to Build/Import your Project

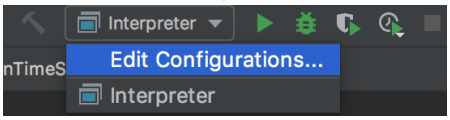
To build this project, clone the GitHub repo https://github.com/csc413-01-fall2019/csc413-p2-Nimiksha

Once the repo is cloned, open IntelliJ, and go to 'Import Project'. Then import the cloned project, and you should be able to see all the classes, and files related to the project.

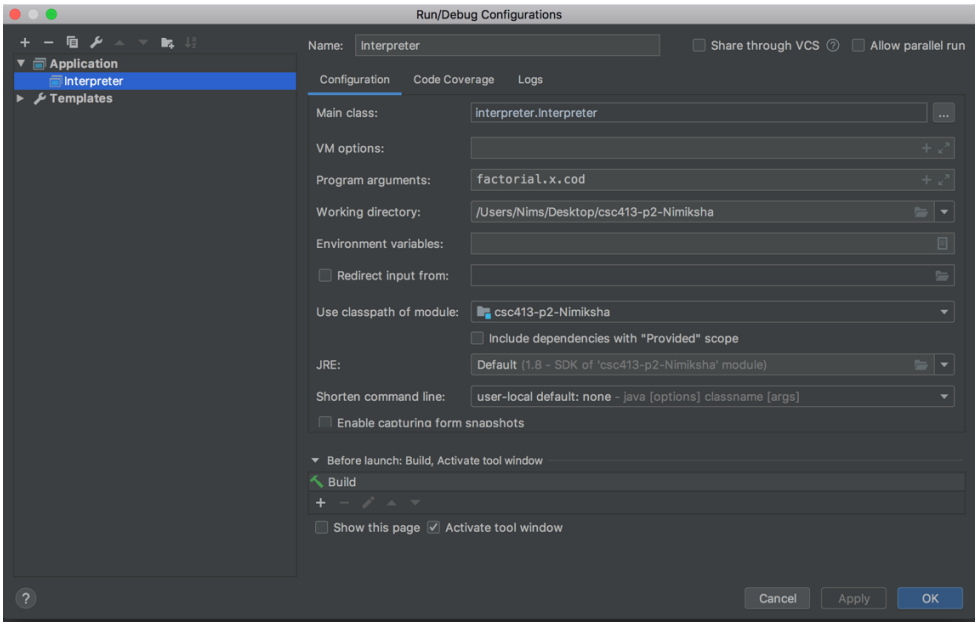
4 How to Run your Project

To run the project, follow the steps:

- Click on Edit Configurations

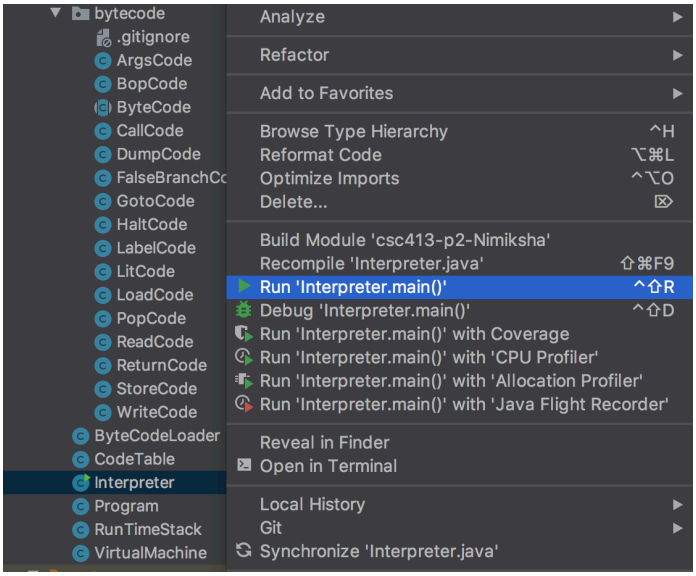


- The run/debug configuration window with open.



- In this window go to the Program Arguments and add the .cod file that you want to run.

- Then click ok and run the Interpreter class.



An important thing to note, while running the program, if you want the entire stack to be printed, Dump must be turned ‘ON’. If dump is off, only the calculations will be shown.

5 Assumption Made

One of the assumptions I made for this project was that there would be 15 bytecode classes, namely ArgsCode, BopCode, CallCode, DumpCode, FalseBranchCode, GoToCode, HaltCode, LabelCode, LitCode, LoadCode, PopCode, ReadCode, ReturnCode, StoreCode and WriteCode (based on the codes given in the CodeTable class).

I also assumed was that I need to made a connection between the ByteCode classes and the VirtualMachine class using the Program class, since the bytecode classes will work with the VirtualMachine class which will further work on the RunTimeStack to perform the requested operations.

Additionally, it was assumed that only the bytecodes listed in the CodeTable class will be required. Also, it was assumed that the code given for the Interpreter class is correct and the driver method in it makes the necessary calls to initialize the program.

6 Implementation Discussion

One of the most important implementations in this project was the implementation for the 15 bytecode classes:

ArgsCode: The ArgsCode class instructs the interpreter to set up a new frame n down from the top of the runtime stack. This will include the arguments in the new frame for the function.

BopCode: The BopCode class pops top 2 levels of the runtime stack and performs the indicated binary/logical operation (+, -, /, *, =, ==, !=, <=, >, >=, <, |, &) on them. The result of these operations is placed on the stack, in place of the operands.

CallCode: The CallCode class saves the return location for the program counter and transfers control to the indicated function.

DumpCode: The DumpCode class sets the state of dumping in the virtual machine. When dump is on, the state of the runtime stack is dumped to the console.

FalseBranchCode: The FalseBranchCode class pops the top of the stack; it branches to the next bytecode if the top of the stack is 0

GoToCode: The GotoCode class sets the program counter to a new location

HaltCode: The HaltCode class halts the execution of a program. It stops the VirtualMachine from running when the program is done

LabelCode: The LabelCode class holds the position of any branches to the label

LitCode: The LitCode class is responsible for loading integers on the Runtime Stack

LoadCode: The LoadCode class pushes the value in the slot which is offset n from the start of the frame onto the top of the stack

PopCode: The PopCode class removes the top element from the Runtime Stack

ReadCode: The ReadCode class reads an integer. It prompts the user to input a value and pushes the value to the stack. It also makes sure that the input is validated.

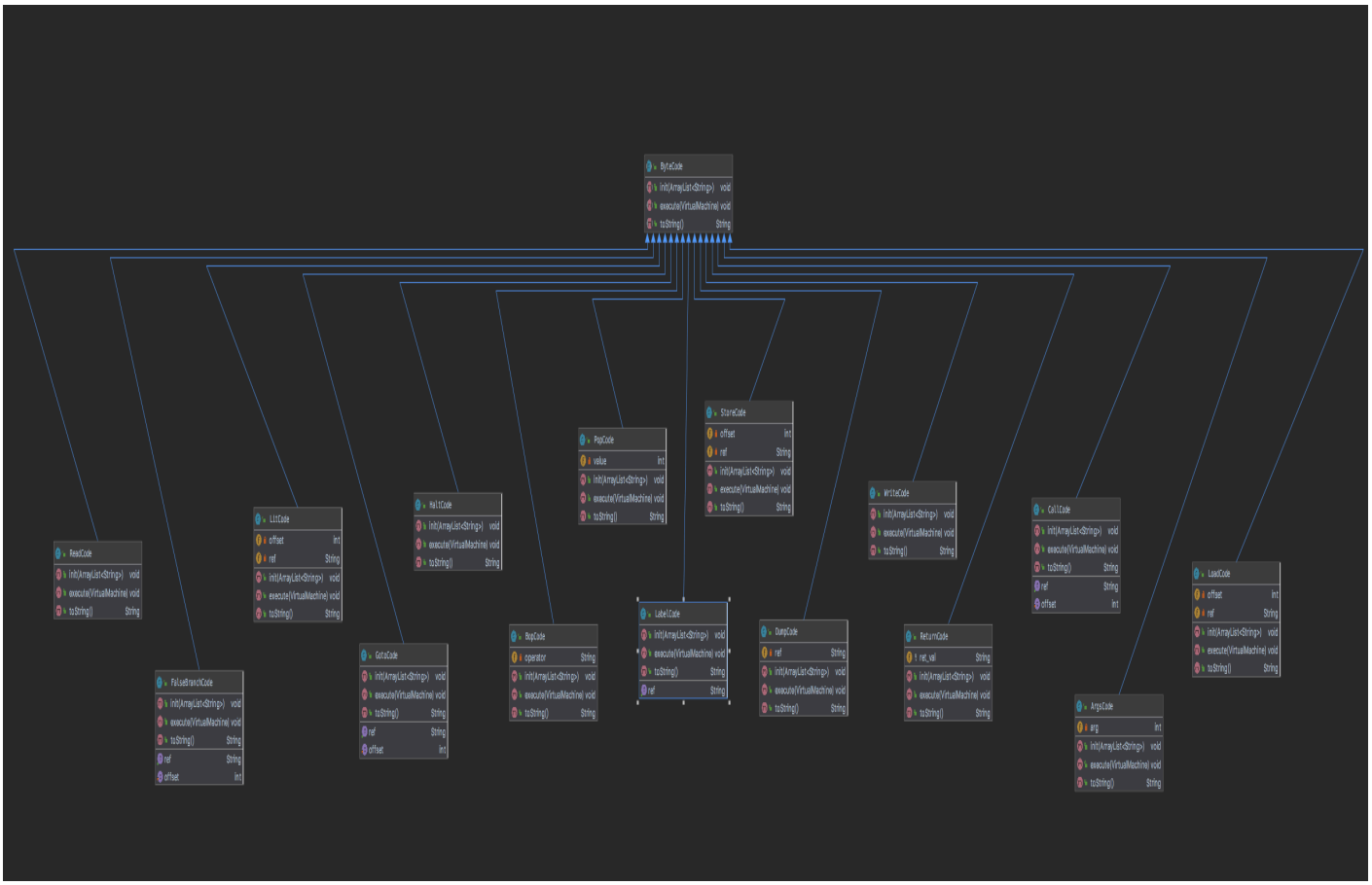
ReturnCode: The ReturnCode class removes the current frame and places the top element of that frame onto the frame that is being returned.

StoreCode: The StoreCode class, based on the current frame pointer and the offset, stores the value into the offset n from the start of the frame

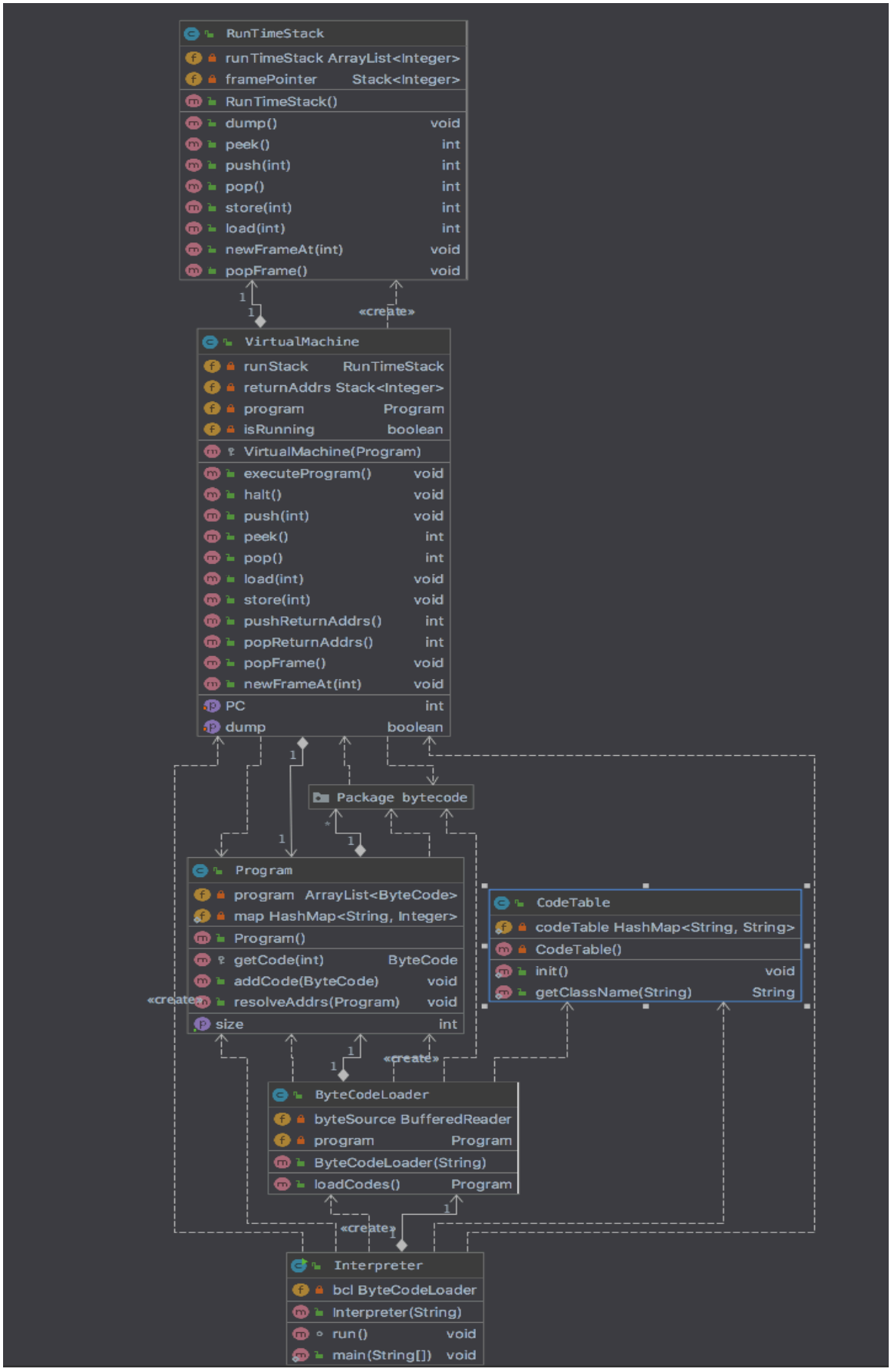
WriteCode: The WriteCode class writes the contents from the top of the stack onto the standard output.

Another important implementation is that for the VirtualMachine Class. The VirtualMachine class activates every bytecode and it executes the Program by making an instance of the RunTimeStack. The VirtualMachine also uses dump, and based on if it is 'ON' or 'OFF' prints the runtime stack.

6.1 Class Diagram



ByteCode class Hierarchy


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
«create»
VirtualMachine
runStack RunTimeStack
returnAddrs Stack<Integer>
program Program
isRunning boolean
VirtualMachine(Program)
executeProgram() void
halt() void
push(int) void
peek() int
pop() int
load(int) void
store(int) void
pushReturnAddrs() int
popReturnAddrs() int
popFrame() void
newFrameAt(int) void
PC int
dump boolean
Package bytecode
Program
program ArrayList<ByteCode>
map HashMap<String, Integer>
Program()
getCode(int) ByteCode
addCode(ByteCode) void
resolveAddrs(Program) void
size int
«create»
CodeTable
codeTable HashMap<String, String>
CodeTable()
init() void
getClassName(String) String
«create»
ByteCodeLoader
byteSource BufferedReader
program Program
ByteCodeLoader(String)
loadCodes() Program
«create»
Interpreter
bcl ByteCodeLoader
Interpreter(String)
run() void
main(String[]) void

7 Project Reflection

As I reflect on the last few weeks that I have worked on this project, I feel like I have learnt a lot. One of the major things I learnt was the attention to detail. After having spent so many days working on the code, when I tried to run it, the code kept crashing. I spent a couple of hour trying to debug, only to find out the error I had was that I was using the wrong case, i.e. I had the GotoCode class as GoToCode. Given Java's case sensitivity, my code kept breaking, and it was a rather silly and frustrating mistake. Another crucial thing I learn from this project was making essential connections between different classes and methods. This project extensively relies of dependence, be it the VirtualMachine's dependence on Program or the Program's dependence on ByteCode classes. I struggles to make those connections, and even now that I have my program, I have to take a minute to make the right connections.

I also learnt the importance of critical Java principles like encapsulation and polymorphism. With the ByteCode classes and the VirtualMachine, it was important to apply these principles, since there correct and smooth functioning depended on the fact the right extensions and connections were made. I have learnt to set the right access modifiers for each class, so that only the important elements are available and the program is secure from unwarranted changes.

Apart from the technical skills, this project taught me the important of really understanding the problem and evaluating it. Reading the 25-page document was hard, but this project forced me to actually pay attention to the details and ask the right questions. I think that is one of my biggest takeaways from this project: understanding the problem and trying to break it down to effectively solve it.

8 Project Conclusion/Results

I was able to write the code for the problem and was able to replicate the sample output for the fib.x.cod and the factorial.x.cod files. The program itself is capable of reading the bytecodes in the source file and executing them. The VirtualMachine class works correctly is able to execute the Program and the RunTimeStack. Also, the dump() works correctly and is able to dump the contents of the runtime stack to the console, to display the desired result. I tried to account for potential errors and make provision to throw Exceptions in order to make sure the code didn't break. I tried to cover as many edge cases as possible, however I'm not too sure how successful I was in doing that.

Overall though, the code works and prints the correct output. I'm glad that I had the opportunity to challenge myself and work on this project.