CSC 413 Project Documentation

Fall 2018

Ibraheem Chaudry

917227459

CSC 413.01

https://github.com/csc413-01-summer2019/csc413-p2-ichaudry.git

Table of Contents

1 Introduction

1.1 Project Overview

The project involved building an interpreter for a programming language 'X'. The language X consists of several bytecodes that can be used to write programs and the job of the interpreter is to compile the program and perform the necessary bytecode executions.

1.2 Technical Overview

The interpreter is an advanced application and involves numerous abstractions and encapsulated classes for feasible operation.

1. Byte Code classes:
    a. Handle the initialization and execution of all bytecodes
2. Program class:
    a. Stores all the Bytecodes read from file and resolves addresses of branch codes
3. Byte Code Loader class:
    a. Reads file and loads bytecodes in program arraylist
4. Runtime Stack class:
    a. Maintains the runtime stack and the frame pointer stack.
5. Virtual Machine class:
    a. Drives the program and allows the bytecodes to interact with runtime stack while maintaining encapsulation.

1.3 Summary of Work Completed

The interpreter is fully functional and both factorial.x, fib.x are generating the correct outputs.

The dump function is also operational.

2 Development Environment

a) Version of Java used: JDK 12

b) IDE used: IntelliJ IDEA community version by JetBrains

3 How to Build/Import your Project

1. Create a directory/New Folder in a desired location on your computer using a name of your choice i.e. "Calculator".
2. Copy and paste the github repository url that is listed on the front page of this document.
3. This step can vary slightly depending on whether you are using a windows computer or linux/Mac
    a. If you are **using windows:**
        i. First make sure that you have **Git Bash** installed.
            1. If you don't have Git Bash install it from this link https://git-scm.com/downloads
            2. Proceed to next step
        ii. Open a **Git Bash Terminal.** You can do this by pressing the widows key on your keyboard and typing Git Bash and clicking on the application. **(if you have an**

older version of windows you can go to the location you installed it and run it from there.)

iii. From your terminal go to the directory you created in step 1. You can do this by typing this command “cd /path/to/directory” for example “cd /users/documents/calculator”

iv. Once inside the directory type the command “git clone **paste url copied in step 2”** and then hit enter. This will clone the repository to your folder.

b. If you are **using Linux/Mac**”

i. Open a terminal on your computer

ii. From your terminal go to the directory you created in step 1. You can do this by typing this command “cd /path/to/directory” for example “cd /users/documents/calculator”

iii. Once inside the directory type the command “git clone **paste url copied in step 2”** and then hit enter. This will clone the repository to your folder.

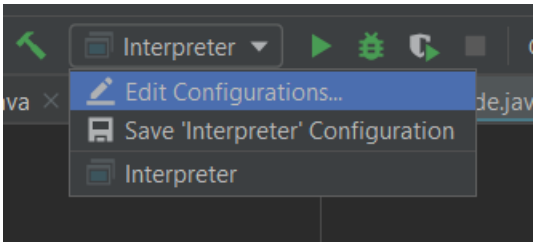
1. If the terminal gives a prompt saying git is not installed you can install git by using the command your terminal provides with the prompt and then try above step again to clone.

4. After you have cloned the repository you need to open IntelliJ IDEA IDE. If you don’t have this software you can download the free community version from here and run it: https://www.jetbrains.com/idea/download/#section=windows
5. Click import Project

6. Locate the directory where you cloned the repository. **SELECT THE MAIN REPO FOLDER AND THEN SELECT IMPORT. THIS IS IMPORTANT AS THE REPO NEEDS TO BE THE ROOT FOR CORRECT OPERATION.**
7. You will be taken through a setup process
    a. Click next to “Create project from existing resources”
    b. Click next after choosing a name and location for project
    c. When asked to choose JDK for project make sure to select the latest one. If you don’t have JDK set up do that first and start the import process again.
    d. Hit next on all the prompts that appear next and you should end up with your project successfully opening in IntelliJ.

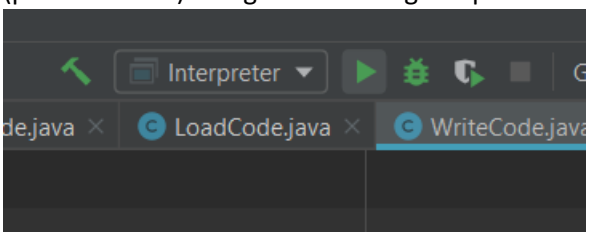
4 How to Run your Project

- When the project is opened in Intellij look in the top drop-down menu that says “interpreter” and then click “Edit Configurations”. (Picture below)



- In the window that opens up select program arguments and type the name of file you’re want to run. **The file has to be in the repo. In this case the file could be ‘factorial.x.cod’ or ‘fiib.x.cod’.** Press finish

- You should now be good to run the interpreter to read your ‘.X’ program. Hit the run button (picture below) and give it an integer input in console to compute.



5 Assumption Made

- Not calculating factorial above 12
- Only takes positive numbers as input
- Only takes integers as input

6 Implementation Discussion

1. Byte Code classes:
    a. All byte codes are represented by their own classes which **extend the abstract bytecode class**
        i. All these classes contain **three important abstract functions** that perform all operations necessary for the bytecode.
            1. Init function: Initializes a bytecode as required

2. Execute function: Performs bytecode specific execution
3. ToString function: This function returns a String that is used for dumping to the console when dump is on.

b. There is another abstract class **specifically for bytecodes that are responsible for branching: False Branch code, Go to code, Call code**. This branch code abstract class also extends the main bytecode abstract class, so the three abstract function mentioned above are present in all the branch codes as well.
   i. An abstraction for branch codes is needed for the resolve address function that is discussed below in the **program** class section.
   ii. This class has variables to store **label** and **address**.

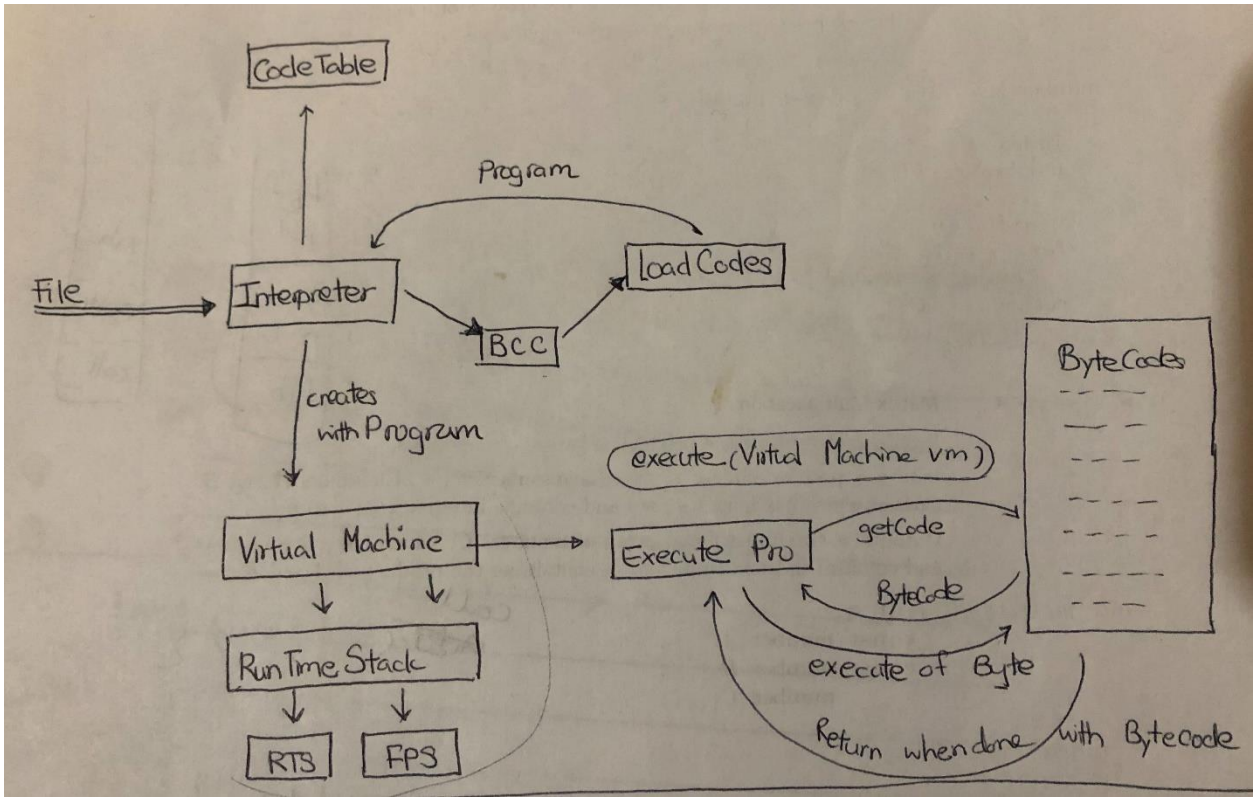
2. Program class:
   a. This class contains the **arraylist** that will store all the byte codes read from a file.
   b. There is also present a hash map which maps labels and addresses. **i.e. at what location in the program array list is a specific label is present.**
   c. The most important function is the resolveAddress function which **sets the addresses of all branch code** classes by using the labels they should branch to and pulling the corresponding address out of the hash map. (**This is the function mentioned above in 1b)**
3. Byte Code Loader class:
   a. This class contains the Load Codes function which reads a '.X' file line by line and passes the arguments as a **LIST** to the bytecode classes **init** function.
      i. A technique called **reflection** is used to make distinct instances of bytecodes
      ii. The bytecodes are then pushed in to the **program arraylist**
      iii. If a label code is encountered its label is stored as a key in the **hash map** from the Program class and the value is the address of the label. **(refer to 2b)**
      iv. After the whole file is read the resolve address function is called. (**refer to 2b)**
4. Runtime Stack class:
   a. This class is important for **maintaining a runtime stack** for when our program is running and keep a track of function frames using a **frame pointer stack**.
   b. The runtime stack is an arraylist while the frame pointer is a stack which has a default 0 inside it to keep track of the main function call.
      i. Both these data structures are private and cannot be accessed directly outside of the runtime stack class.
      ii. To manipulate these data structures many functions are written in the class for example pop, peek etc.
      iii. These functions can be called by the **virtual machine class** that creates an instance of the runtime stack when the program starts running.
5. Virtual Machine class:
   a. The virtual machine has the driver function "executeProgram" that is responsible for running the program.

i. This function iterates through the program arraylist and calls the execute function for each bytecode.

b. The virtual machine creates an instance of the runtime stack and has functions to access the functions in the runtime stack.
   i. These functions in the virtual machine are used by the byte code classes in the execute function to do operations like pop, peek, push etc

At no point can the bytecode class directly access the runtime Stack and its contents. Instead the virtual machine acts as a middle man and handles the interaction between bytecodes and runtime stack. This is necessary for encapsulation to hold and the program to be secure.

6.1 Class Diagram



7 Project Reflection

This project was probably the most hardcore program I have ever written and really was a great opportunity to brush up my skills of writing a dynamic application while keeping security in mind by paying close attention to encapsulation.

The use of reflection to create distinct instances of byte codes was the best thing I got out of this project and it got my mind thinking in all sorts of creative directions as to such methods can be practically applied when creating other applications.

8 Project Conclusion/Results

This project was completed successfully, and all tests were passed. **No bugs detected.**