

***CSC 413 Project Documentation***  
***Spring 2022***

***Dylan Burns***  
***922717234***  
***Section 02***

***<https://github.com/csc413-SFSU-Souza/csc413-p2-Dylan-Burns>***

## Table of Contents

1. Introduction
  1. Project Overview
  2. Technical Overview
  3. Summary of Work Completed
2. Development Environment
3. How to Build/Import your Project
4. How to Run your Project
5. Assumption Made
6. Implementation Discussion
  1. Class Diagram
7. Project Reflection
8. Project Conclusion/Results

## Introduction

### 1.1 Project Overview

The Interpreter project is designed to interpret a custom language named “X”. The Interpreter reads or “interprets” ByteCodes within the code source files (*factorial.x.cod*, *fib.x.cod*) line by line, and executes the relating ByteCode. The Interpreter works alongside the Virtual Machine which is responsible for running the programs. The first program recursively computes the *nth* fibonacci number and the second program recursively computes the *nth* factorial.

### 1.2 Technical Overview

The Interpreter project is composed of predefined classes and classes that the student must complete. There are 3 packages: *bytecodes*, *loaders*, and *virtualmachine*.

The *bytecode* package contains all the required ByteCodes for the interpreter to process and run the program..

The *loaders* package contains 3 classes:

*ByteCodeLoader.java* - loads the ByteCodes by reading from the *.x.cod* files.

*CodeTable.java* - fills a Table with the ByteCodes for reference during initialization and execution.

*InvalidProgramException.java* - an exception thrown when the program is invalid.

The *virtualmachine* package contains 4 classes:

*RuntimeStackIllegalAccess.java* - an exception thrown for calls to the Runtime Stack that are invalid (out of bounds).

*Program.java* - contains an ArrayList of all the ByteCodes

*RunTimeStack.java* - maintains a valid runTimeStack and a valid framePointer Stack

*VirtualMachine.java* - controls program execution

### 1.3 Summary of Work Completed

The work completed on this project includes finishing the incomplete or non-existent classes to build a working version. The following classes were completed:

*bytecode: ArgsCode, BopCode, ByteCode, CallCode, DumpCode, FalseBranchCode, GotoCode, HaltCode, JumpCode, LabelCode, LitCode, LoadCode, PopCode, ReadCode, ReturnCode, StoreCode, and WriteCode.*

*loaders: ByteCodeLoader.*

*virtualmachine: Program, RunTimeStack, VirtualMachine.*

### Development Environment

The Interpreter program executes as expected on *openjdk-17 version 17.0.1* within the IntelliJ IDEA 2021.3.3 (Ultimate Edition)

### How to Build/Import your Project

Step 1: download the zip file from the github repository link provided on the title page.

Step 2: unzip and open the project in IntelliJ IDEA.

Step 3: Edit project configuration by adding the java SDK version 17.

Step 4: Run *Interpreter.java* to compile the project

### How to Run your Project

Step 5: Edit project configuration by adding one of the *.x.cod* files to the CLI arguments to the application. Enter either *factorial.x.cod* or *fib.x.cod* as a CLI argument.

\*\*\*\* If you want to turn DUMP ON \*\*\*\*

open the desired *.x.cod* file and add "DUMP ON" to the first line of the file.

Step 6: Run *Interpreter.java*

## Assumption Made

User is running JRE that supports *openjdk17* and follows the proper import/build/run instructions.

## Implementation Discussion

I chose to create an abstract *ByteCode* class as the parent class of all other *bytecodes* rather than an interface because all *bytecodes* are closely related. For the *bytecodes* that ‘jump’ around the program during execution, I created an abstract class *JumpCode* which extends *ByteCode* to offer additional functionality and avoid repeat code. All functionality of the *bytecodes* are restricted through the Virtual Machine and its methods to maintain encapsulation.

All *bytecode* classes inherit 3 methods:

*init(ArrayList<String> args)* - initializes the corresponding *bytecode* with its appropriate field value.

*execute(VirtualMachine vm)* - executes the task defined for the corresponding *bytecode*

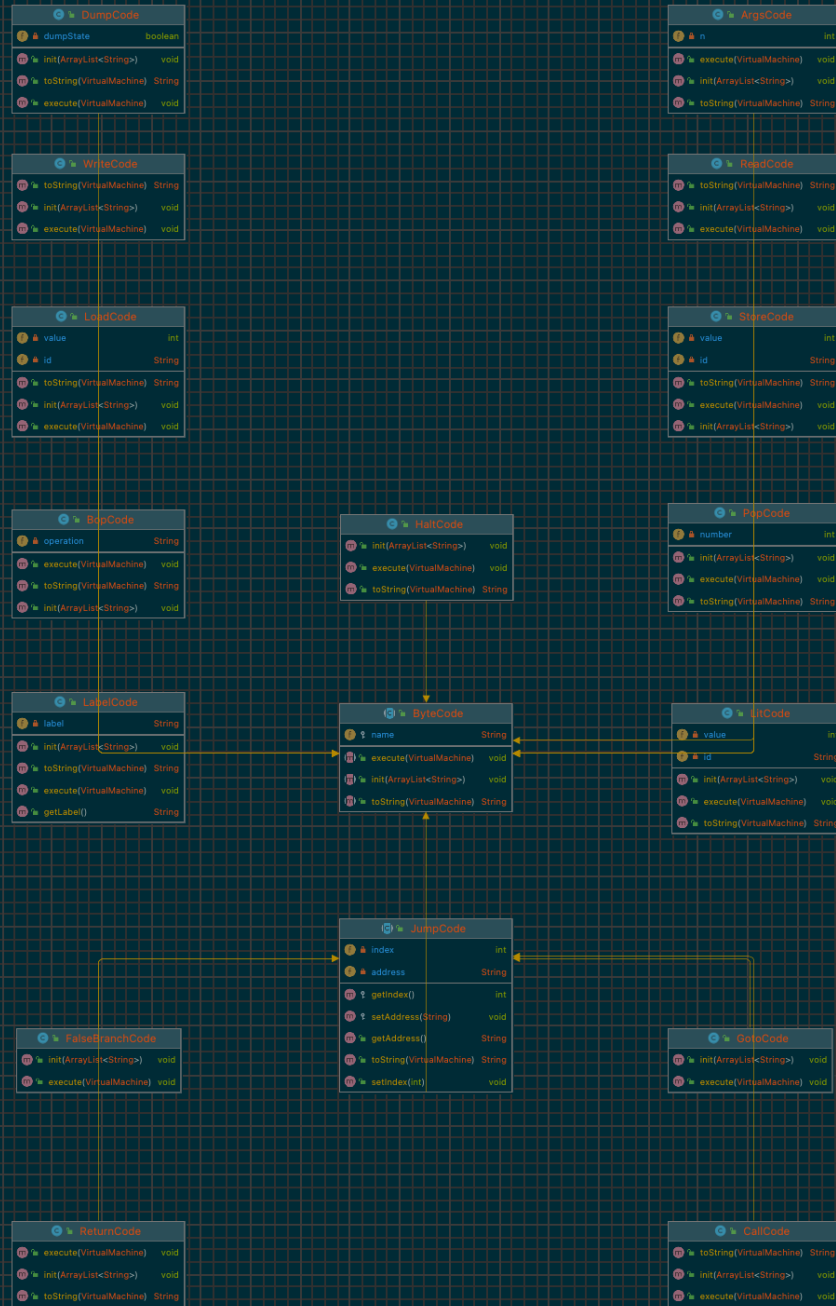
*toString(VirtualMachine vm)* - prints out the *bytecode* (only when dump is on)

For the *RunTimeStack* class I added a *runTimeStack* to access the location of the ByteCodes in the runtime stack. As well as a *framePointer* to hold the initial activation record for function calls. Refer to the *RunTimeStack* class to view its methods and their assigned task.

I did my best to follow the basic OOP principles and guidelines by reducing unnecessary coupling and designing modular code. Each method has a well defined purpose and naming conventions were designed to be simple and easy to understand. Some methods have no comments due to the descriptiveness of the code itself.



## Class Diagram



## Package bytecodes





## Package loaders



### CodeTable



  *codeTable* `HashMap<String, String>`

  *init()* `void`

  *getClassName(String)* `String`

### ByteCodeLoader

  *codeSource* `BufferedReader`

  *loadCodes()* `Program`

### InvalidProgramException

Package virtualmachines

VirtualMachine

returnAddress

Stack<Integer>

program

Program

isRunning

boolean

programCounter

int

runTimeStack

RunTimeStack

dumpState

boolean

executeProgram()

void

dumpFrameRunTimeStack(int)

String

dumpRunTimeStack()

String

halt()

void

popReturnAddress()

int

popFrameRunTimeStack()

void

DumpOn()

void

peekRunTimeStack()

int

getFrameSizeRunTimeStack()

int

pushRunTimeStack(int)

void

storeRunTimeStack(int)

void

setProgramCounter(int)

void

getProgramCounter()

int

DumpOff()

void

loadRunTimeStack(int)

void

pushReturnAddress(int)

void

popRunTimeStack()

int

addFrameAtRunTimeStack(int)

void

RunTimeStack

framePointer

Stack<Integer>

runTimeStack

ArrayList<Integer>

load(int)

int

addFrameAt(int)

void

dump()

String

popFrame()

void

peekFrame()

int

Program

program

ArrayList<ByteCode>

getCode(int)

ByteCode

resolveAddress()

void

addCode(ByteCode)

void

public int popReturnAddress()

☐ interpreter.virtualmachine.

VirtualMachine

getFrameSize()

int

store(int)

int

push(int)

int

getSize()

int

dumpFrame(int)

String

RuntimeStackIllegalAccess

## Project Reflection

This was by far one of the more challenging projects I've worked on over the years. I enjoyed the process of breaking it down into easily digestible chunks. If I had to start from the beginning I would do a few things differently to ease the coding process. First I would spend more time understanding the bigger picture of the program and what its purpose is. I'd (hopefully) ask why it is important to maintain encapsulation within the program and design my code accordingly.

Understanding how the *Program*, *RunTimeStack*, and *VirtualMachine* operated in unison was another challenge for me. Once I understood that the *RunTimeStack* maintains the order of operations, and the *Program* holds the *bytecodes* to be operated on it became much clearer. The *VirtualMachine* works with the two to allow the *Interpreter* to process the *.x.cod* files and determine control and flow during execution.

A few suggestions I'd make with the interpreter.pdf :

1. The pdf states the bytecodes must be contained in a package named *interpreter.ByteCode* but when importing the project from the github repository created from canvas the package is *interpreter.bytecodes*.
2. I think students would benefit from having the coding hints inlined with their corresponding reference.
3. The pdf should include sample output for students to verify their program output is as expected.

Other than those few small adjustments I felt the project was described in depth. Class discussions were also helpful to clear up any confusions.

## Project Conclusion/Results

Aside from the mild carpal tunnel this assignment gave me, I enjoyed the process of understanding each level of abstraction and its associations within the program. I think this project was on the edge of my comfort zone with java so it provided an opportunity for growth.