CSC 413 Interpreter Documentation

Name: Vimean Sam
ID: 915819611
Link to GitHub Repository: https://github.com/csc413-03-sp18/csc413-p3-Vsam326

Project Introduction: In this project, I am writing an interpreter program for a fake "X" language. I am given 2 files of factorial.x.cod and fib.x.cod and I am supposed to load the bytecode contents of the X language from the .x.cod file in order to execute the .x.cod file based on the bytecode by storing each bytecode instructions and their arguments to an arraylist. Using Netbeans IDE 8.2, I load the contents of the .x.cod file and store in an arraylist. While bytecode instructions are loaded in an arraylist, I isolated any arguments that are instances of labelCode class and stored them into a Map based on their indexes in program size (line number) in order to keep track of the code that re-branch program counter. Then, I loaded literal values to the runtime Stack in order to perform binary operations based on the instruction.

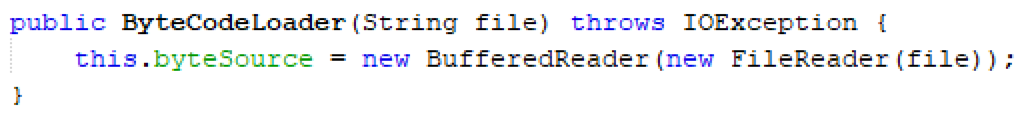
Build Instructions: First, I cloned the project by downloading the project files from GitHub and place it on my desktop. Then, I go to Netbeans and create new project and choose "Java Project from Existing Source" and set my project folder to the directory of where I cloned my files to.

Run Instructions: Then after my project is created, I clicked on a scroll-down button named <default config> and click to customize. Then in a pop-up window, I set my main class to Interpreter.java and working directory to my project location (in this case C:\Users\Vimean Sam\Desktop\csc413-p3-Vsam326). In the argument section, I would type the x file such as factorial.x.cod for fib.x.cod interchangeably depending on my test cases.

Assumptions: While coding this big project, I had to keep an assumption that all the test cases will be valid input such as entering an int instead of char on the READ bytecode. In addition, I assume that the input won't be a large number for factorial.x.cod and fib.x.cod as integer limitation will be exceeded based for factorial and large integer will froze up Fibonacci sequence (long compile time). Also, I considered scenarios that would break my code such as loading and storing from/at an index greater than the top of stack index and ways to protect my stack against the invalid retrieving.

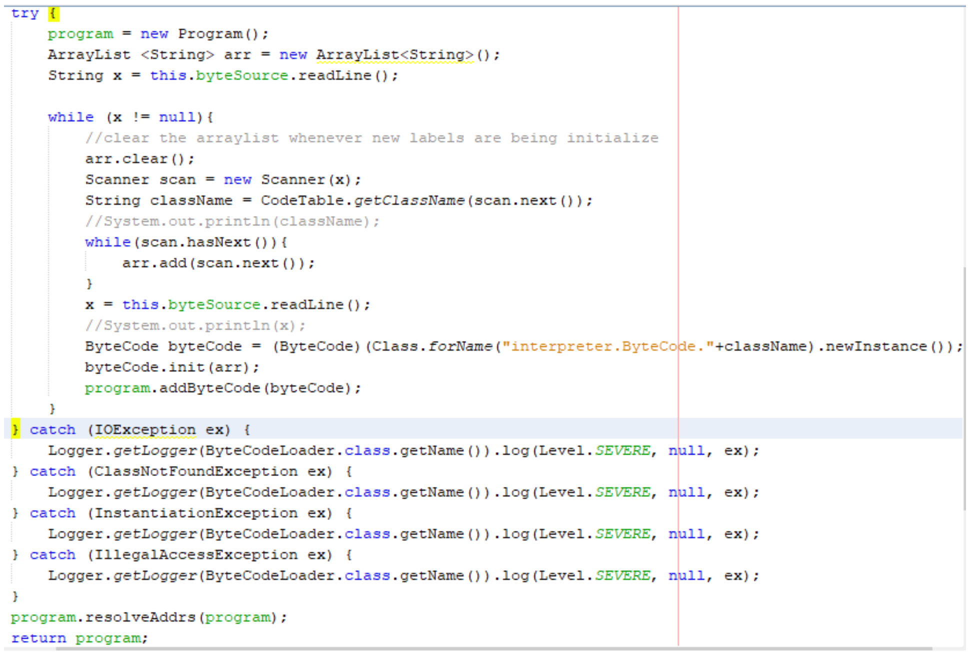
Implementation Discussion:

1. **ByteCodeLoader.java** is my main start to the project. This class will load contents from the .x.cod files by using bufferedreader to read files.



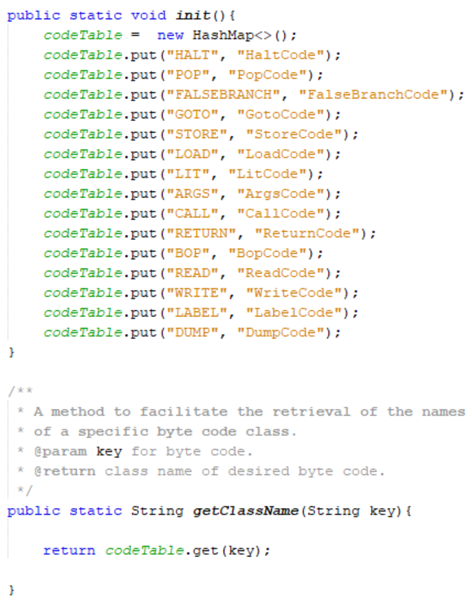
```
public ByteCodeLoader(String file) throws IOException {
    this.byteSource = new BufferedReader(new FileReader(file));
}
```

In loadcode() method, I use a scanner to tokenize each string and break it apart to analyze the class based on **Codetable.java** and its arguments. Then, I would add the arguments to a temporary arraylist and initialize them depending on which bytecode class they belong to. Then, I add the bytecode to program and resolve its address which I'll talk about in the next page.



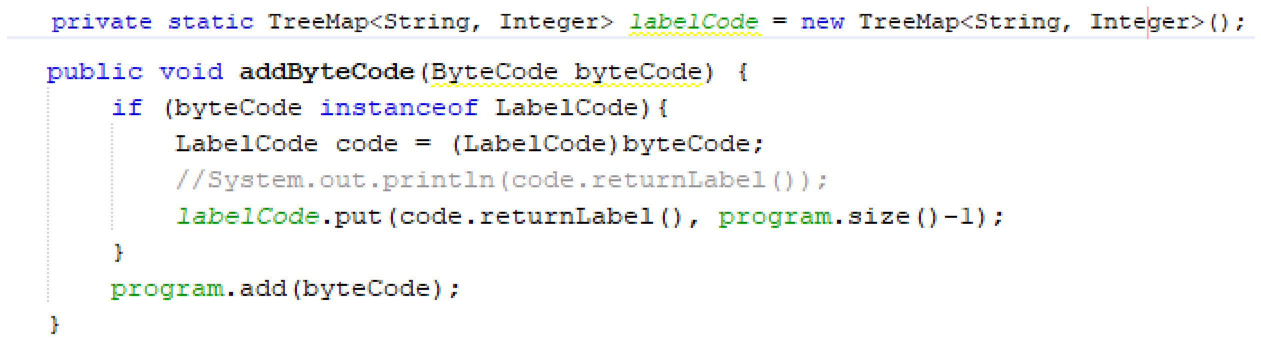
```
try {
    program = new Program();
    ArrayList <String> arr = new ArrayList<String>();
    String x = this.byteSource.readLine();

    while (x != null){
        //clear the arraylist whenever new labels are being initialize
        arr.clear();
        Scanner scan = new Scanner(x);
        String className = CodeTable.getClassName(scan.next());
        //System.out.println(className);
        while(scan.hasNext()){
            arr.add(scan.next());
        }
        x = this.byteSource.readLine();
        //System.out.println(x);
        ByteCode byteCode = (ByteCode)(Class.forName("interpreter.ByteCode."+className).newInstance());
        byteCode.init(arr);
        program.addByteCode(byteCode);
    }
} catch (IOException ex) {
    Logger.getLogger(ByteCodeLoader.class.getName()).log(Level.SEVERE, null, ex);
} catch (ClassNotFoundException ex) {
    Logger.getLogger(ByteCodeLoader.class.getName()).log(Level.SEVERE, null, ex);
} catch (InstantiationException ex) {
    Logger.getLogger(ByteCodeLoader.class.getName()).log(Level.SEVERE, null, ex);
} catch (IllegalAccessException ex) {
    Logger.getLogger(ByteCodeLoader.class.getName()).log(Level.SEVERE, null, ex);
}
program.resolveAddrs(program);
return program;
```

The **codetable.java** class create all the bytecode instructions.



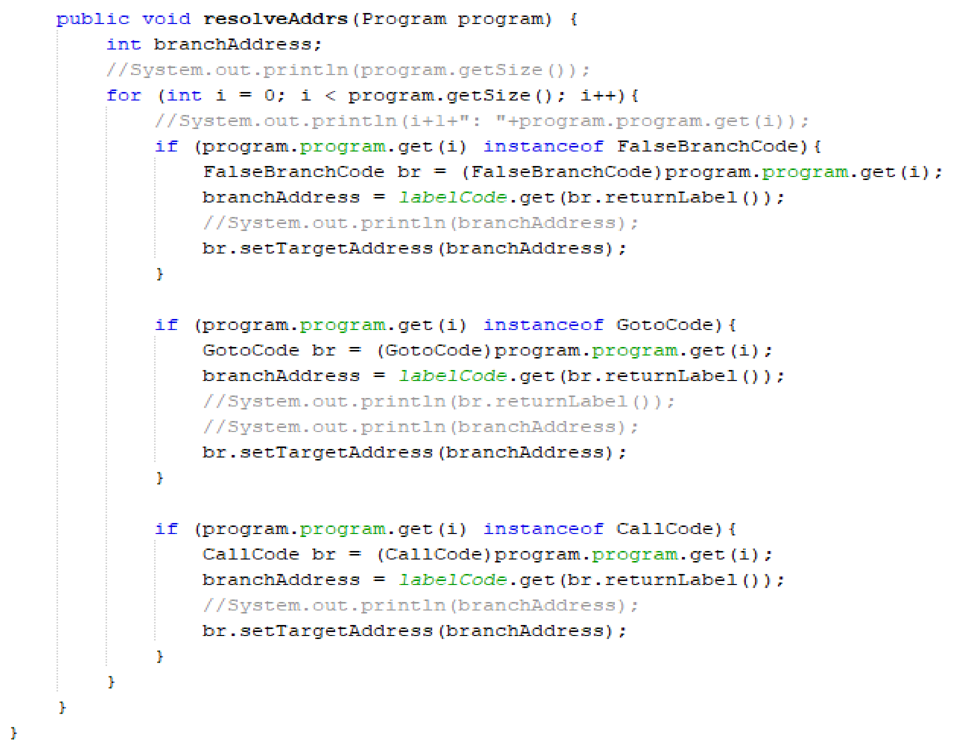
```
public static void init(){
    codeTable =  new HashMap<>();
    codeTable.put("HALT", "HaltCode");
    codeTable.put("POP", "PopCode");
    codeTable.put("FALSEBRANCH", "FalseBranchCode");
    codeTable.put("GOTO", "GotoCode");
    codeTable.put("STORE", "StoreCode");
    codeTable.put("LOAD", "LoadCode");
    codeTable.put("LIT", "LitCode");
    codeTable.put("ARGS", "ArgsCode");
    codeTable.put("CALL", "CallCode");
    codeTable.put("RETURN", "ReturnCode");
    codeTable.put("BOP", "BopCode");
    codeTable.put("READ", "ReadCode");
    codeTable.put("WRITE", "WriteCode");
    codeTable.put("LABEL", "LabelCode");
    codeTable.put("DUMP", "DumpCode");
}

/**
 * A method to facilitate the retrieval of the names
 * of a specific byte code class.
 * @param key for byte code.
 * @return class name of desired byte code.
 */
public static String getClassName(String key){

    return codeTable.get(key);

}
```

2. After bytecode instructions are loaded to program, there are a few branching instructions that I resolved in **Program.java**. So I made a method name addByteCode(ByteCode) which take bytecode instructions as a parameter. In this method, I created a map that keeps track of instances of labelcode and where it occurred using program.size()-1 as an index (line number-1). For example: 0. GOTO start<<1>> label code occurs when program size is 2 and 2-1 is the index of
1. LABEL Read
the occurance which is 1. This method is called through the loop in bytecodeloader's loadcode() method above, so it will run and evaluate each time an instance of labelcode occurs until there's no more instruction to add.



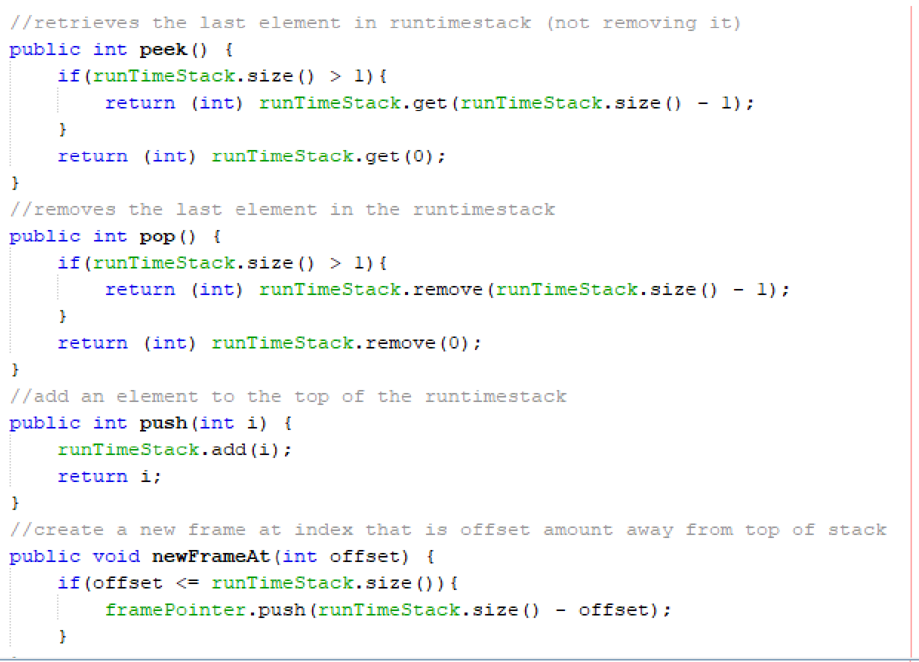
```java
private static TreeMap<String, Integer> labelCode = new TreeMap<String, Integer>();
public void addByteCode(ByteCode byteCode) {
    if (byteCode instanceof LabelCode){
        LabelCode code = (LabelCode)byteCode;
        //System.out.println(code.returnLabel());
        labelCode.put(code.returnLabel(), program.size()-1);
    }
    program.add(byteCode);
}
```

In resolve address, I compared, the three branchcode of FALSEBRANCH, GOTO, and CALL to the labelcode map. So, I looped through program and grabbing instance of FALSEBRANCH, GOTO, and CALL and set its address in interpreter.bytecode classes for later branching of program counter.

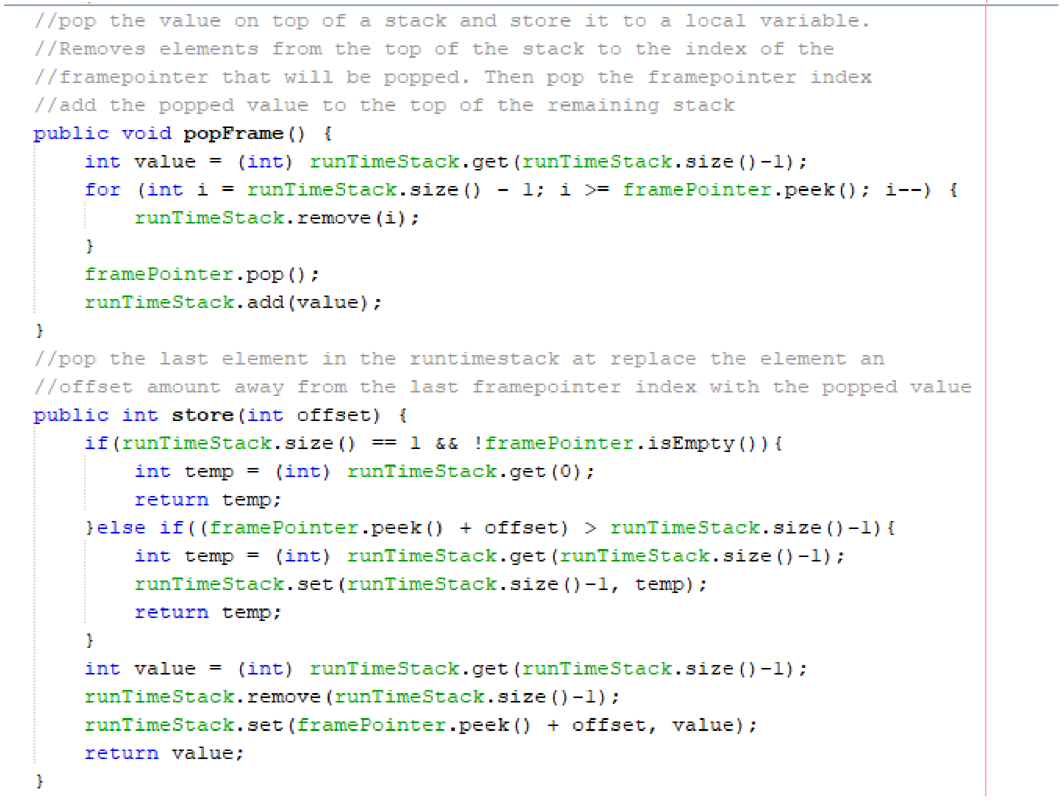


```java
public void resolveAddrs(Program program) {
    int branchAddress;
    //System.out.println(program.getSize());
    for (int i = 0; i < program.getSize(); i++){
        //System.out.println(i+1+": "+program.program.get(i));
        if (program.program.get(i) instanceof FalseBranchCode){
            FalseBranchCode br = (FalseBranchCode)program.program.get(i);
            branchAddress = labelCode.get(br.returnLabel());
            //System.out.println(branchAddress);
            br.setTargetAddress(branchAddress);
        }

        if (program.program.get(i) instanceof GotoCode){
            GotoCode br = (GotoCode)program.program.get(i);
            branchAddress = labelCode.get(br.returnLabel());
            //System.out.println(br.returnLabel());
            //System.out.println(branchAddress);
            br.setTargetAddress(branchAddress);
        }

        if (program.program.get(i) instanceof CallCode){
            CallCode br = (CallCode)program.program.get(i);
            branchAddress = labelCode.get(br.returnLabel());
            //System.out.println(branchAddress);
            br.setTargetAddress(branchAddress);
        }
    }
  }
}
```

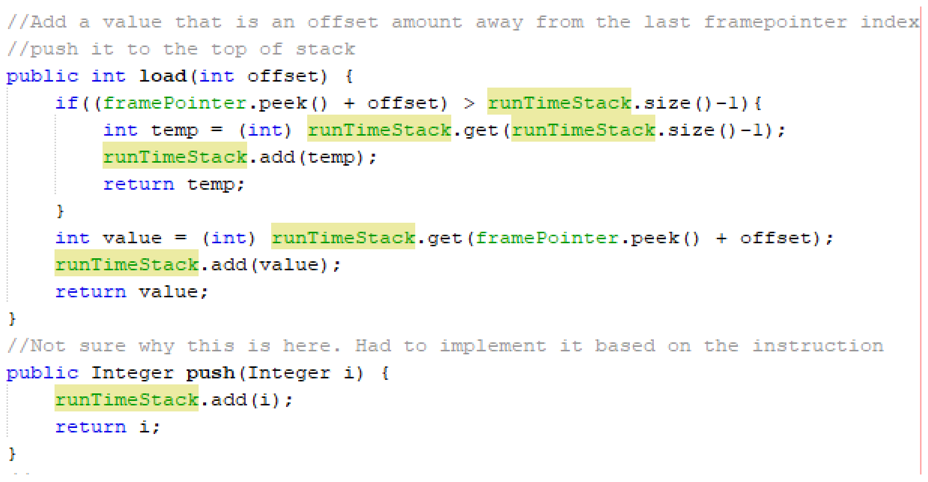
3. **RunTimeStack.java** is the “moving part” of this project as it contains pop, push, peek, load, store, newframe, and popframe that manipulates the stack. In addition, this class contains dump() method that prints out the runTimeStack during each bytecode execution. Its function is explained in comments below:



```java
//retrieves the last element in runtimestack (not removing it)
public int peek() {
    if(runTimeStack.size() > 1){
        return (int) runTimeStack.get(runTimeStack.size() - 1);
    }
    return (int) runTimeStack.get(0);
}
//removes the last element in the runtimestack
public int pop() {
    if(runTimeStack.size() > 1){
        return (int) runTimeStack.remove(runTimeStack.size() - 1);
    }
    return (int) runTimeStack.remove(0);
}
//add an element to the top of the runtimestack
public int push(int i) {
    runTimeStack.add(i);
    return i;
}
//create a new frame at index that is offset amount away from top of stack
public void newFrameAt(int offset) {
    if(offset <= runTimeStack.size()){
        framePointer.push(runTimeStack.size() - offset);
    }
```

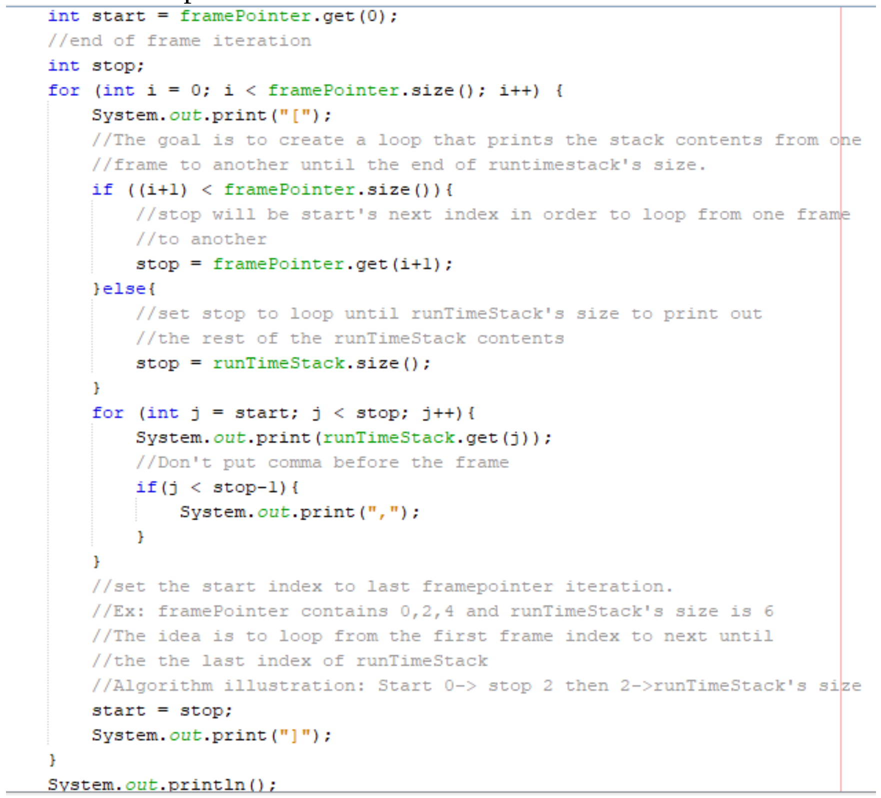


```java
//pop the value on top of a stack and store it to a local variable.
//Removes elements from the top of the stack to the index of the
//framepointer that will be popped. Then pop the framepointer index
//add the popped value to the top of the remaining stack
public void popFrame() {
    int value = (int) runTimeStack.get(runTimeStack.size()-1);
    for (int i = runTimeStack.size() - 1; i >= framePointer.peek(); i--) {
        runTimeStack.remove(i);
    }
    framePointer.pop();
    runTimeStack.add(value);
}
//pop the last element in the runtimestack at replace the element an
//offset amount away from the last framepointer index with the popped value
public int store(int offset) {
    if(runTimeStack.size() == 1 && !framePointer.isEmpty()){
        int temp = (int) runTimeStack.get(0);
        return temp;
    }else if((framePointer.peek() + offset) > runTimeStack.size()-1){
        int temp = (int) runTimeStack.get(runTimeStack.size()-1);
        runTimeStack.set(runTimeStack.size()-1, temp);
        return temp;
    }
    int value = (int) runTimeStack.get(runTimeStack.size()-1);
    runTimeStack.remove(runTimeStack.size()-1);
    runTimeStack.set(framePointer.peek() + offset, value);
    return value;
}
```



```java
//Add a value that is an offset amount away from the last framepointer index
//push it to the top of stack
public int load(int offset) {
    if((framePointer.peek() + offset) > runTimeStack.size()-1){
        int temp = (int) runTimeStack.get(runTimeStack.size()-1);
        runTimeStack.add(temp);
        return temp;
    }
    int value = (int) runTimeStack.get(framePointer.peek() + offset);
    runTimeStack.add(value);
    return value;
}
//Not sure why this is here. Had to implement it based on the instruction
public Integer push(Integer i) {
    runTimeStack.add(i);
    return i;
}
```

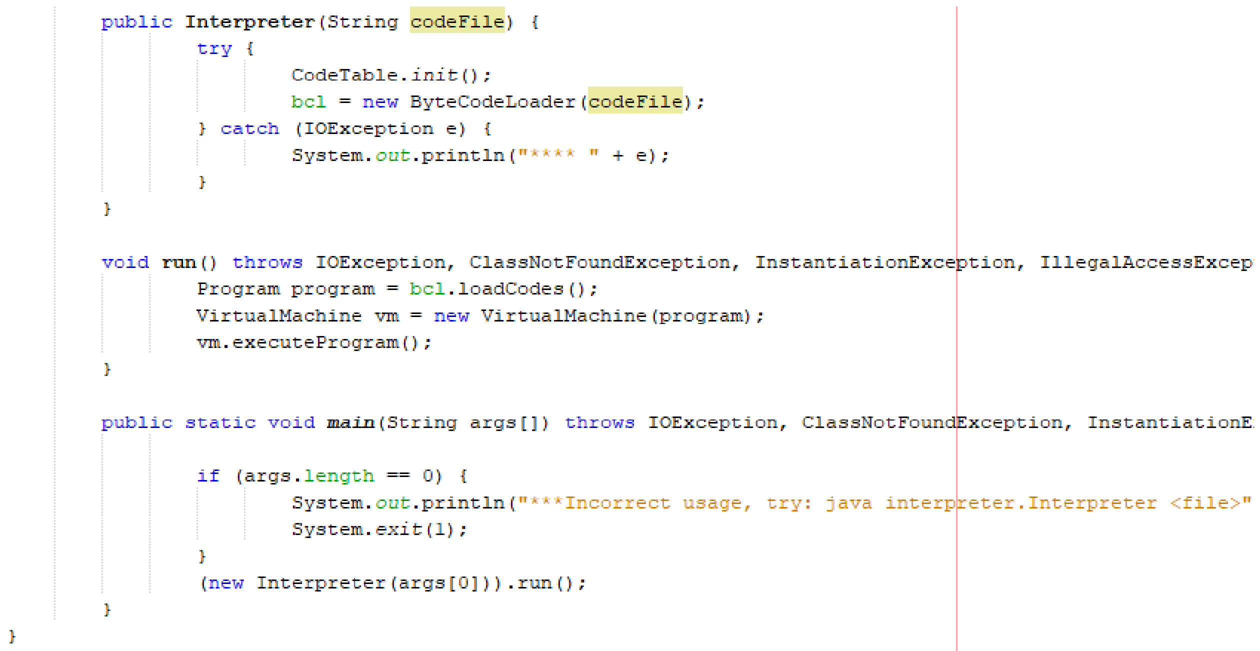
Code for dump method execution:



```
int start = framePointer.get(0);
//end of frame iteration
int stop;
for (int i = 0; i < framePointer.size(); i++) {
    System.out.print("[");
    //The goal is to create a loop that prints the stack contents from one
    //frame to another until the end of runtimestack's size.
    if ((i+1) < framePointer.size()){
        //stop will be start's next index in order to loop from one frame
        //to another
        stop = framePointer.get(i+1);
    }else{
        //set stop to loop until runTimeStack's size to print out
        //the rest of the runTimeStack contents
        stop = runTimeStack.size();
    }
    for (int j = start; j < stop; j++){
        System.out.print(runTimeStack.get(j));
        //Don't put comma before the frame
        if(j < stop-1){
            System.out.print(",");
        }
    }
    //set the start index to last framepointer iteration.
    //Ex: framePointer contains 0,2,4 and runTimeStack's size is 6
    //The idea is to loop from the first frame index to next until
    //the the last index of runTimeStack
    //Algorithm illustration: Start 0-> stop 2 then 2->runTimeStack's size
    start = stop;
    System.out.print("]");
}
System.out.println();
```

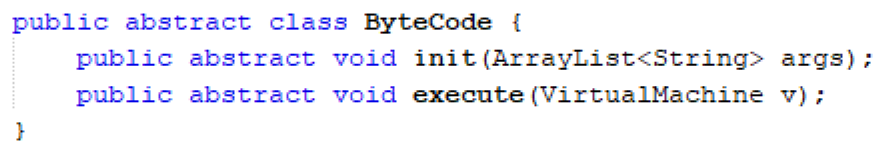
4. **VirtualMachine.java** executes the bytecode instructions and dump method that prints the bytecode and runtimeStack. ExecuteProgram() method is given in the instruction pdf but I added an if statement to dump (print) the bytecode and runtimeStack based on the condition (ON/OFF) of the newly added global variable dumpFlag. In addition, this class contains getters/setters for private variables runtimeStack, framePointer, returnAddr and program counter (pc) to perform execution such as pop, peek, push, etc..for the stacks, retrieve and modify pc based on branch codes, and detect the flag/condition for DUMP bytecode.

5. **Interpreter.java** is the main class for this project. This class initializes codetable, loads bytecode from filename and execute the virtual machine.



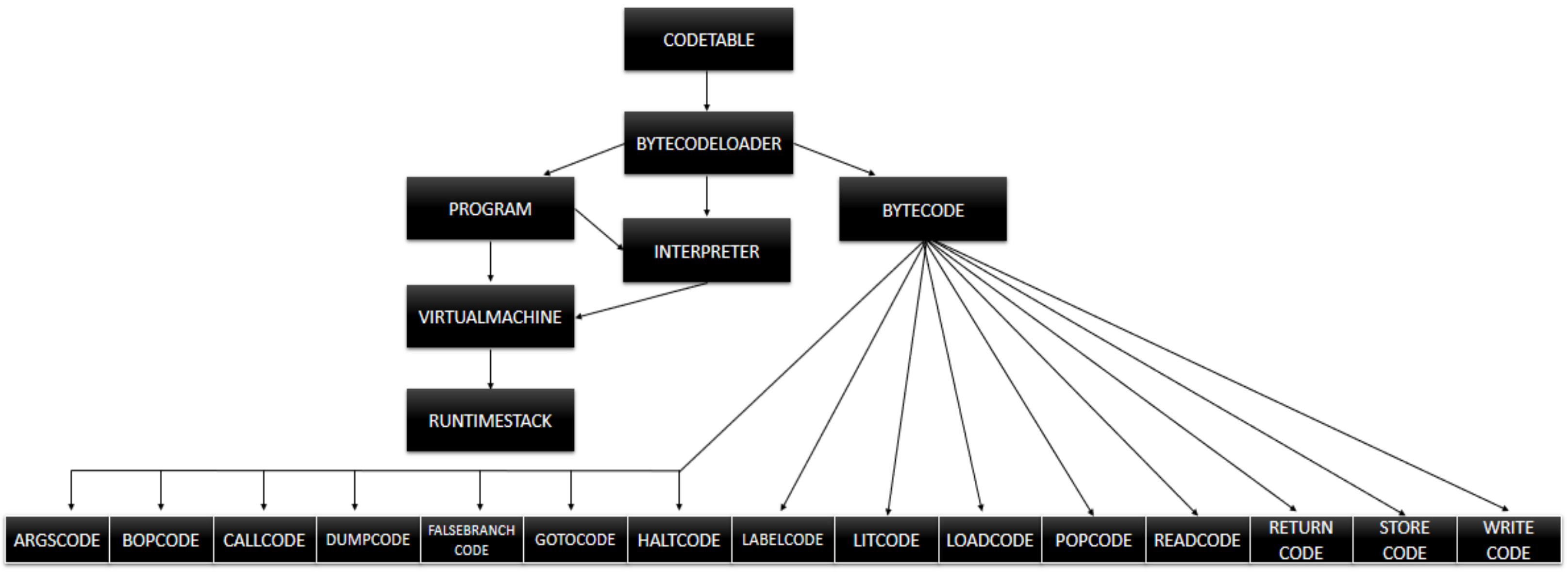
```
    public Interpreter(String codeFile) {
        try {
            CodeTable.init();
            bcl = new ByteCodeLoader(codeFile);
        } catch (IOException e) {
            System.out.println("**** " + e);
        }
    }

    void run() throws IOException, ClassNotFoundException, InstantiationException, IllegalAccessExcep
        Program program = bcl.loadCodes();
        VirtualMachine vm = new VirtualMachine(program);
        vm.executeProgram();
    }

    public static void main(String args[]) throws IOException, ClassNotFoundException, InstantiationE

        if (args.length == 0) {
            System.out.println("***Incorrect usage, try: java interpreter.Interpreter <file>"
            System.exit(1);
        }
        (new Interpreter(args[0])).run();
    }
}
```

method. In addition subclasses of ByteCode is in this package and they contain init(arraylist <String> args) to store their arguments and execute(VirtualMachine v) to perform what each bytecode instruction.



```
public abstract class ByteCode {
    public abstract void init(ArrayList<String> args);
    public abstract void execute(VirtualMachine v);
}
```

Class Diagrams:



Results and Conclusion: This interpreter is arguably the toughest project I've ever completed. There are many things going on and the stacks were constantly changing which was difficult to track errors. The hardest part of the project is understanding the purpose of Interpreter and the bytecodes that contribute to "moving parts". After reading the instructions pdf numerous and going to office hours to receive help, I got a clear picture of the project goal and I ended up with a functional interpreter that runs any valid .x.cod files. At the end, I learned a lot from this project and I realize that there are numerous things going on behind the scene each time I compile/run my program.