San Francisco State University
CSC 413-03
Spring 2018

Victor Muñoz

https://github.com/csc413-03-sp18/csc413-p3-Vmunoz94

Introduction:

For this assignment we were tasked on implementing an interpreter for the mock language X. Some code was completed for us such as the Interpreter class and the CodeTable class. The ByteCodeLoader, Program, RunTimeStack, and VirtualMachine classes were partially completed for us. The ByteCode class and all its subclasses had to be created and implemented by us. Netbeans IDE was used to make all the necessary changes in the java files that we were given and had to be created.

The first task we had to do was to implement all the bytecodes, which there are a lot of. A list of all the bytecodes that had to be implemented are on page 129 or page 5 on the assignment pdf document. Each bytecode completes a specific task, however there are two methods in every bytecode that do not change. The contents of these methods differ, but the signature does not. That is why there is an abstract ByteCode class which contains abstract methods.

```
public abstract class ByteCode{

        public abstract void init(ArrayList<String> argument);
        public abstract void execute(VirtualMachine virtualMachine);
}
```

Since every bytecode is a subclass of this abstract ByteCode class. That means that every bytecode is forced to implement these abstract methods. The first method is used to get the arguments from each bytecode. The second abstract method is used to execute the bytecode and accomplish its specific task. In total there are 15 bytecode classes that are subclasses of this abstract ByteCode class. With the help of the BytecodeLoader, the bytecodes were keyed to their specific method.

HaltCode: stops execution by setting the private Boolean isRunning in the virtual maching to false

PopCode: Pops the top “n” levels of the runTimeStack, where n is the argument corresponding the POP bytecode. Also need to make sure that the number of times being asked to pop is not greater than the runTimeStack size itself.

FalseBranchCode: Pop the top of the stack; however, if it is zero then branch to <label>. This is done by setting the program counter to the label.

GotoCode: Goes to the label. This is done by setting the program counter to the Label.

StoreCode: Pops the top of the stack and stores the value into the offset n. This method calls the store method in the VirtualMachine class.

LoadCode: Push the value in offset n from the start of the frame onto the top of the stack. This method calls the load method in the VirtualMachine class.

LitCode: Loads the literal value n, where n is the bytecode argument. This calls the push method in the VirtualMachine class.

ArgsCode: Used prior to calling a function. It instructs the Interpreter to set up a new frame n down from the top, where n is the argument. This calls the push method in the VirtualMachine.

CallCode: Transfers control to the given function name. This pushes the current program counter into the returnAddress stack in the VirtualMachine and also sets the program counter to the argument.

ReturnCode: Return code pops the top of the returnAddress stack and sets the program counter to that value. The frame is also popped.

BopCode: Binary operation pops the top two levels of the runTimeStack and performs the operation. Lastly the result is pushed back into the stack. In the case of Boolean operations, false = 0 and true = 1.

ReadCode: Asks user for input and pushes that value into the stack. If the value is incorrect, then ask for another value.

WriteCode: Prints the value on the top of the runTimeStack.

LabelCode: target to go to. If the target's argument is the same as the label's argument then go here. Targets only go to labels.

DumpCode: Sets the DumpMode variable in the VirtualMachine class to either ON or OFF.

Note that some of these bytecodes request the RunTimeStack to do specific tasks; however, this will break encapsulation. Bytecodes should never be allowed to talk to the RuntTimeStack directly, it has to go through the VirtualMachine first. For example, I could have a getter method that retrieves the runTimeStack in the VirtualMachine. This will allow bytecodes to talk to the runTimeStack directly, which is not allowed. In order to not break encapsulation, more methods have been added to the VirtualMachine that request information from the RunTimeStack. In this case, the bytecodes talk to the VirtualMachine and the VirtualMachine talks to the RunTimeStack.

The next task was to implement the ByteCodeLoader class. This method reads the "…".x.cod file one line at a time and saves the line to a string. That string is then tokenized using stringTokenizer which seperates the string into different tokens separated by a space. The very first token will always be the name of the bytecode. Using this first token and the CodeTable class, we can get the corresponding method since the CodeTable contains a hashmap with the keys being the bytecodes and the values being the methods.

```
ByteCode bytecode = (ByteCode) (Class.forName("interpreter.ByteCode."+codeClass).newInstance());
```

This line was used to create a new instance of the ByteCode class. The remaining tokens are then stored in an arrayList. The last thing to do before this method ends to call resolveAddrs in the Program class.

The next task we had to complete was the Program class, specifically the resolveAddrs method. Before implementing the resolveAddrs method, I had to create an add method that passes the bytecodes to the program arrayList. Also, at the very top of the program class, I had to import all the bytecode classes. The resolveAddrs, for me, was the hardest method to implement. Basically, there are certain bytecodes that jump to different locations. Unlike a normal program that executes one line at a time. Bytecodes can jump to a different line; however, they will only jump to a Label bytecode containing the same arguments as the jumping bytecode. There are three bytecodes that can jump to a label code, which are GoTo, FalseBranch, and Call bytecodes. The resolve address method goes through the entire program and checks if the current bytecode is one of the three bytecodes that jump. If it is, then check the entire program again to find a Label bytecode with the same arguments. If there is a corresponding Label bytecode, then set the argument of the jumping bytecode to the position of the Label bytecode.

Next, we had to implement the RunTimeStack. There are 10 methods that had to be created, all but one is described on the assignment pdf.

public void dump(): dumps the RunTimeStack information for debugging. Only the DumpCode sets the dump status. This method prints out the stack depending on the framePointer size.

public int peek(): returns the top of the runTimeStack without removing it

public int pop(): returns the top of the runTimeStack and removes it

public int push(int i): adds i to the top of the runTimeStack and returns it

public void newFrameAt(int offset): start a new frame depending on the offset, which indicates the number of slots down from the top of the runTimeStack for starting a new frame

public void popFrame(): each functions return value is on top of the stack, so we first pop the top of the stack and store the contents into a temporary variable. Next we remove the top frame only. Once the frame is removed, the functions return value, which is stored in some temporary variable is added back to the top of the stack.

public int store(int offset): gets the top value of the stack and stores it onto the some offset in the stack.

public int load (int offset): gets the value at some offset in the stack and loads it to the top of the stack.

public Integer push (Integer i): used to load literals onto the top of the stack.

public int getRunStackSize(): gets the size of the RunTimeStack.

The last task was to implement the VirtualMachine class and the code snippet below was given to us to help complete the executeProgram method.

```
public void executeProgram(){
        pc = 0;
        runStack = new RunTimeStack();
        returnAddrs = new Stack();
        isRunning = true;
        while(isRunning){
                ByteCode code = program.getCode(pc);
                code.execute(this);
                if("ON".equals(dumpMode)){
                        runStack.dump();
                }
        pc++;
        }
}
```

The only other methods that had to be created were setter and getter methods in order for the bytecodes to manipulate the variables in the virtual machine. The other methods are from the RunTimeStack in order to not break encapsulation.

Build instructions:

In order to build this project. First you must clone the repo. Once the repo has been cloned, In Netbeans you need to create a new project and click Java Project with Existing Sources. Once the project has been named and next has been clicked, add the cloned repo folder to the Source Package Folders. Now the project is built, but not yet runnable.

Run instructions:

To run the project, first you must set the project configuration. There are two ways to do this.

The first way is to click on the scroll bar next to the globe at the top of the window usually labeled “<default config>”. Once pressed scroll down to “customize…”, then set the working directory to the cloned repo. And the argument can either be “fib.x.cod” or “factorial.x.cod”. Now the program should be able to run.

The second way is to go to the “run” tab at the very top of the screen. Go to “Set Project Configuration” then click “customize…”. Then from here, set the working directory to the cloned repo. And the argument can either be “fib.x.cod” or “factorial.x.cod”. Now the program should be able to run.

Assumptions:

- All the bytecodes are valid bytecodes.
- All the bytecodes that are going to be used are in the code table.
- Do not assume user will enter correct values
- Do not assume pop will not try to pop more than the frame size

Results:

From this project I learned that it is extremely useful to talk with other people and ask questions. This was the first big project where I was extremely confused at times and if it was not for asking questions, I probably would have stayed confused. I believe this was the only assignment that I have ever had to do, which places a huge emphasis on encapsulation. Teachers have talked about encapsulation but never assigned a project where we had to implement it, so I never really understood the importance of it until now.