Interpreter

By Mitul Savani

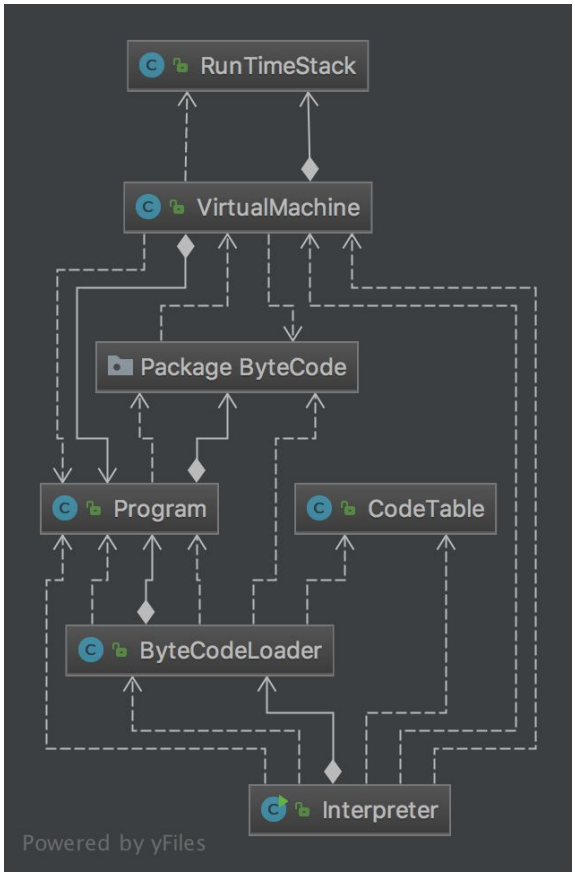
Summary

In this assignment the task was to make a Java Virtual Machine which can interpret the moc language. We were provided with bunch of bytecodes in a class *CodeTable.java* in a form of Hashmap and some skeleton code in class *Interpreter.java, VirtualMachine.java*. My Task was to create a solid code that can read through the each line of the moc language file(fib.x.cod, factorial.x.cod), interpret which bytecode it is, then eventually perform operation.

Getting Started

- To access program with **Fibonacci** file
    - java -jar Interpreter.jar fib.x.cod
- To access program with **Factorial** file
    - java -jar Interpreter.jar factorial.x.cod

Architecture



ByteCodeLoader

This class was very straight forward since there were hint given in it. I read all lines of argument files using code shown below,

```
while((line=byteSource.readLine()) != null)
```

and tokenized each line to get which string belongs to which ByteCode so I called CodeTable.java to fetch more information.

- After getting Bytecodes I added it into data field:program of class *Program.java*.
- Implemented try and catch to handle exceptions

PROGRAM

In this class, I had all the byte codes loaded in data field:*program* from class ByteCodeLoader.java.

I implemented only one function, *resolveAddrs* and the task was to find which lines of the file should be read next along with recording lines number. So, I used a Hash Map to save all labels as key and its line number as a value.

RunTimeStack

In this class I made all the basic methods which were given in the Assignment.pdf file, some of the methods are shown below,

- ```
  public int pop() {
      if(!runTimeStack.isEmpty())
      return runTimeStack.remove(runTimeStack.size()-1);

      return -1;
  }
  ```

- ```
  public int push(int i){
      runTimeStack.add(i);
      return i;
  }
  ```

- ```
  public int peek(){
      if(!runTimeStack.isEmpty(){
          int arg =
  runTimeStack.get(runTimeStack.size()-1);
          return arg;
      }
      return 0;
  }
  ```

Discussion : The above methods are in regard with the stack implementation. Peek() will return the last inserted item, Pop() will remove last item added in the stack, Push() will insert item in the stack
  ```

Now, I will show some of the important methods that I implemented in this file.

```
public void newFrameAt(int offset)
{
    framePointer.push(runTimeStack.size()-offset);
}
```

Discussion: The above method newFrameAt will store/push The offset to the framepointer.

```
public int store(int offset)
{
    runTimeStack.set(framePointer.peek()+offset,
runTimeStack.remove(runTimeStack.size()-1));
      return offset;
}
```

Discussion: The above method store will store the offset in the runTimeStack after removing the last item from the Stack.

Virtual Machine & ByteCode Subclasses

I created Virtual Machine and ByteCode Subclasses at the same time, because I think those files co-relate with each other. So I started hovering over all the subclasses to start initializing arguments by using the following code,

```
arg = args.get(0);
comment: arg is string.
```

```
literal = Integer.parseInt(arg);
```

comment: and if I want to change arg into integer than this is how I did.

I also made 'toString' method of all classes since that was later going to be used by function 'Dump'.

I started executing all the functions in RunTimeStack.java and called it via VirtualMachine.java so that I don’t break the encapsulation.

Difficulties faced

1. Designing the function of the files and its architecture was the major difficult task I faced.
2. Making Subclasses and calling functions without breaking encapsulation.
3. I struggled with placing data into the Hash Map in Program.java, and checking which label belongs to which byte code.
4. Making Dump Code function in RunTimeStack.java.

What I learned:

I learned how to approach and design a project.