INTERPRETER

Project introduction and overview:

The project Interpreter takes file '__.x.cod' as input and processes the bytecodes to produce the output. Interpreter implements an abstract class ByteCode which would be inherited by all byte code classes, and VirtualMachine class which would carry out all the operations for bytecodes and RunTimeStack class which would store all the current stack values.

Interpreter loads the bytecodes into an arraylist using ByteCodeLoader class and resolves the address before passing it to VirtualMachine. VirtualMachine has the access to RunTimeStack object which holds the elements in run time stack as well as frame pointer stack, to execute the bytecode operations. VirtualMachine class also has a stack to store the addresses so that the program counter could be restored when it returns from a function. All bytecodes could access the stack indirectly via VirtualMachine.

The Interpreter project was build and executed on NetBeans IDE 8.2

The scope of this project was to implement an interpreter to execute the bytecodes. The Interpreter and CodeTable classes were already implemented and descriptions were given for each bytecode operation. All bytecode classes are implemented including the DUMP bytecode which is used for dumping program state as required. The stacks are protected from bytecodes. The written code has been tested and works as expected for both Fibonacci and Factorial input files provided for testing.

Build instructions for the project:

Import the code files into local folder from GitHub using above given link. In NetBeans IDE, go to Files → New project →Java project using existing sources and select the local folder where project files are stored. Once the new project is created go to Run → Clean and Build project.

Run instructions for the project:

After successful clean and build, go to Run → Set Project Configuration → Customize and provide the input file name in arguments tab. Select project directory as current working directory. After saving the changes, run the code using Run →Run Project. The code will ask for a positive integer value and provide the output.

Assumptions:

- No divide by zero conditions
- Interpreter takes declared type of variables as integer and can hold maximum value upto 2,147,483, 647. Hence factorial can be calculated till input value of 12, after which integer variable overflows (13! = 6.227021x10^9).

Implementation discussions:

Interpreter class:

Interpreter class implements the main method where it takes the input file name from command line argument and store the bytecode objects into an arraylist using ByteCodeLoader class object. The program arraylist then will be passed to VirtualMachine and bytecodes will be processed.

CodeTable Class:

CodeTable class implements hashmap for storing class names for bytecodes. It contains bytecode strings(ex: “POP”, “HALT”) as keys and corresponding class names as values(ex: “PopCode”, “HaltCode”).

Program class:

Program class implements a hashmap to store the line numbers for LABEL bytecodes. The key will be name of the labels and line number of corresponding labels are mapped as values.
The Program class also implements a method to resolve the address of labels where it fetch line numbers from hashmap and store it in corresponding class variables for GOTO, FALSEBRANCH and CALL bytecodes.

ByteCodeLoader Class:

ByteCodeLoader class reads the input file using buffered reader and store the objects of bytecodes in an arraylist program. The incoming line from buffered reader will be tokenized using String.split() method to store the arguments of bytecodes which would be parameters for init functions of bytecodes. The instants of bytecode objects are created dynamically by using *(ByteCode)(Class.forName("interpreter.ByteCode." +codeClass).newInstance())* where codeClass is the class name fetched from CodeTable hashmap. These objects of bytecodes are stored in an arraylist and address are resolved for GOTO, FALSEBRANCH and CALL using resolveAddress() method of Program class.

VirtualMachine Class:

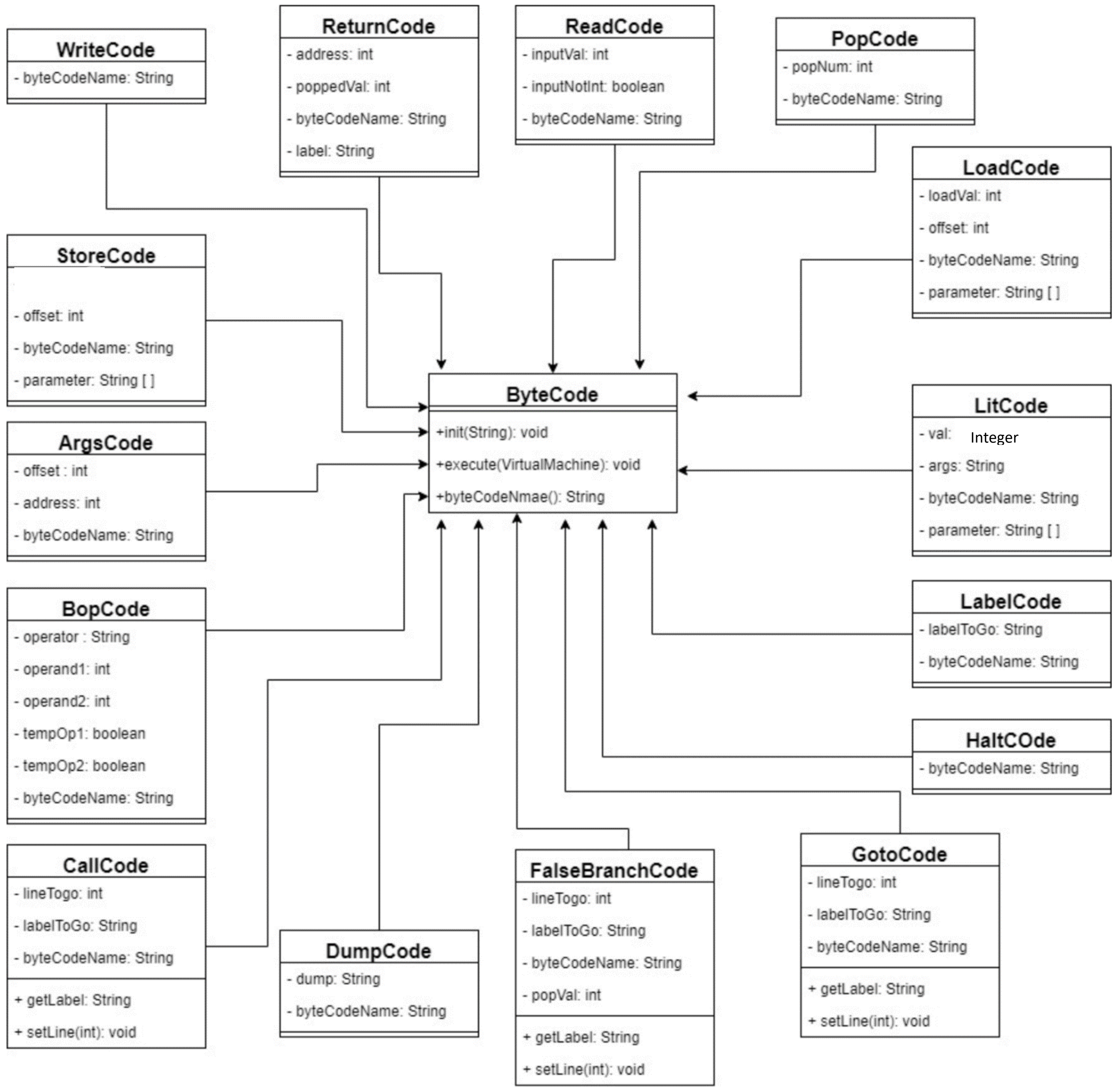
- The arraylist of bytecode objects are passed to the VirtualMachine where PC counter is used to read the arraylist. The object of VirtualMachine itself is passed as a parameter for execute() method of bytecodes using which bytecode operations are performed.
- The VirtualMachine contains a flag - dumpFlag, used for dumping program state. The bytecodes and runTimeStack are printed when dumpFlag is true.
- The VirtualMachine contains object of RunTimeStack using which it can access the stack elements.

- The returnAddress stack is also created in VirtualMachine to store the PC counter value so that address can be restored after returning from function.

Interpreter.ByteCode.ByteCode Class:

- ByteCode Class is an abstract superclass in Interpreter.ByteCode package, inherited by all bytecode classes.
- ByteCode Class has three abstract methods.
    1. Abstract void init(String arg) – stores the parameters of bytecodes into private variables.
    2. Abstract void execute(VirtualMachine vm) – access the stack elements via VirtualMachine object vm and perform the operation.
    3. Abstract String byteCodeName() – returns the byte code name which is used for dumping check(All bytecodes are printed while dumping other than DUMP itself).
- ByteCode class is inherited by 15 sub classes used for each byte codes which overrides the abstract methods of ByteCode Class.
  List of subclasses given below.
    1. ArgsCode – Class for bytecode ARGS n which is used prior to calling a function. It instructs the interpreter to set a new frame n down from top of the stack.
    2. BopCode – Class for bytecode BOP <operator>. It pops the 2 elements from run time stack and performs indicated operations.
    3. CallCode – Class for CALL <function name>. It transfers the program control to the indicated function.
    4. FalseBranchCode – Class for FALSEBRANCH <label>. It pops the run time stack and transfer the control to indicated label if popped value is zero or else continue.
    5. GotoCode – Class for GOTO <label>. It transfers the program control to the indicated label.
    6. HaltCode – Class for HALT. It will halt the execution.
    7. LabelCode – Class for LABEL<label>. Used to target the branches used in GOTO and FALSEBRANCH.
    8. LitCode – Class for LIT n/LIT n <id>. It loads the literal value n on top of the stack.
    9. LoadCode – Class for LOAD n <id>. It pushes the value of the slot which is offset n from start of the current frame into top of the stack.
    10. PopCode – Class for POP n. Used to pop the top n levels of the stack.
    11. ReadCode – Class for Read. Used to prompt the user for input and store the input value on top of the stack.
    12. ReturnCode – Class for RETURN <function name>/ RETURN. Used to return from current function.
    13. StoreCode – Class for STORE n <id>. It pops the top of the stack and store the value into slot which is offset n from start of the current frame.
    14. WriteCode – Class for WRITE. It peeks the top of the stack and display it on the screen.
    15. DumpCode – Class for DUMP. It prints the executed bytecode and print the elements of stack.

ByteCode Class hierarchy:



RunTimeStack Class:

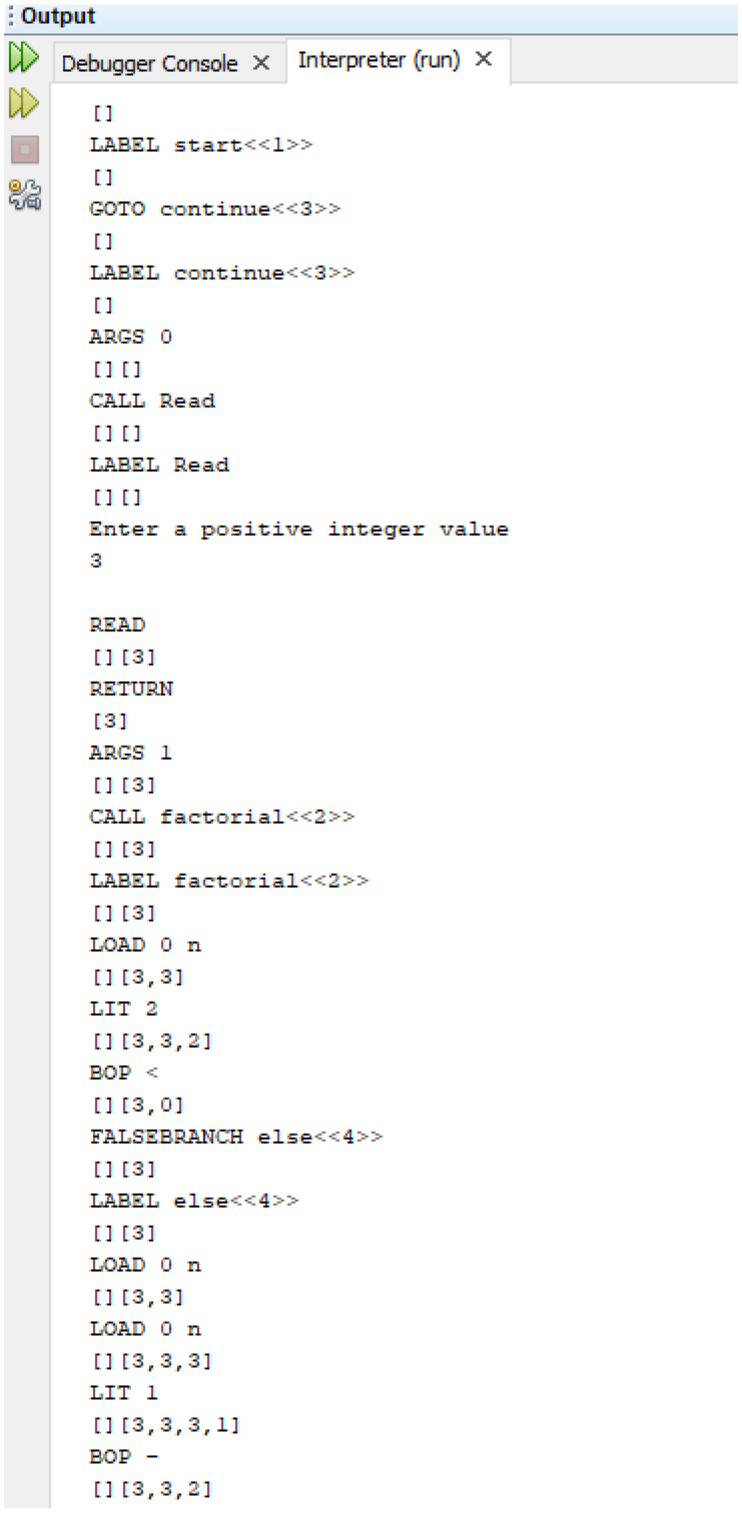
- The RunTimeStack Class implements an arraylist of integer which acts as run time stack for interpreter.
- A stack framePointer is implemented to store indices of frames of arraylist.
- For entry point of project, i.e main method, initial frame pointer is stored as zero.
- RunTimeStack implements following methods to process the stacks.
  1. void dump() – this method is used to print the run time stack elements when dump flag is ON. The stack elements are printed according to the frames. The dumping process is explained below.

2. int peek() – this method is used to return the top value of run time stack.
3. int pop() – this method is used to remove the top of the run time stack and also return its value.
4. int push(int) – this method is used to add the parameter passed to the top of run time stack.
5. Integer push(Integer) – this method is used for loading literal values into run time stack.
6. void newFrame(int) – this method is used to store the indices of new frames when program control is transferred.
7. void popFrame() – this method is used to clear the current frame after function returns the value.
8. int store(int) – this method is used to pop the run time stack and store the value into slot which is offset n from start of the current frame.
9. int load(int) – this method is used to add the value of slot which is offset n from start of the current frame onto top of the run time stack.
10. int getSizeRunTimeStack() – this getter method is used to return the size of the run time stack.

Dumping of program state:

- VirtualMachine contains a private dumpFlag. When the DUMP bytecode is executed, the dumFlag will be set to 'true' using a setter method, setDump(String).
- After executing every bytecode, dumpFlag will be checked and if it is true, the dump() method from RunTimeStack will be called and the run time stack elements will be displayed on the screen framewise.
- Every bytecode classes checks for dumpFlag of VirtualMachine after execution of stack operation and displays the byte code name along with its arguments if dumpFlag is true.

Below screenshot shows the dumping of few bytecodes as an example.



```
[]
LABEL start<<1>>
[]
GOTO continue<<3>>
[]
LABEL continue<<3>>
[]
ARGS 0
[][]
CALL Read
[][]
LABEL Read
[][]
Enter a positive integer value
3

READ
[][3]
RETURN
[3]
ARGS 1
[][3]
CALL factorial<<2>>
[][3]
LABEL factorial<<2>>
[][3]
LOAD 0 n
[][3,3]
LIT 2
[][3,3,2]
BOP <
[][3,0]
FALSEBRANCH else<<4>>
[][3]
LABEL else<<4>>
[][3]
LOAD 0 n
[][3,3]
LOAD 0 n
[][3,3,3]
LIT 1
[][3,3,3,1]
BOP -
[][3,3,2]
```

<u>Results and conclusions:</u>

The Interpreter code is working as per requirement.

Learning outcome:

- Usage of String.split() method.
- Understanding workflow of virtual machine.
- Creating instance of objects dynamically.
- Working of arraylist.

Challenges encountered:

- Tried using StringTokenizer for loading arguments in init(String) methods, but was receiving incorrect tokens with tokenizer. Hence used String.split() method to load the parameters of bytecodes in init(String).