Name
Brian Casipit

Github Repository
https://github.com/motiveg/interpreter-for-software-development

---

Table of contents

<u>Overview</u>

Introduction
This program is an interpreter for an x-language. From a general point of view, this program:
- translates compiled .x.cod files into commands
- stores the commands into a list
- executes the commands via a "virtual machine

Summary of technical work
Some code is already provided, including the Interpreter class, CodeTable class, some of ByteCodeLoader, some of Program, and some of VirtualMachine. Every method and ByteCode described in the assignment needs to be implemented and defined. Any additional fields, methods, and classes are to be created as needed.

ByteCodeLoader: The most important function that needs to be defined here is loadCodes(), which reads the .x.cod source file and translates each line into commands, called ByteCodes, which are stored into an instance of Program.

Program: The most important function that needs to be defined here is resolveAddrs(), which stores return addresses inside of ByteCodes that need to jump to labels.
VirtualMachine: This class needs additional methods to communicate between other classes and needs a way to identify when to dump. In order to maintain encapsulation, we can call methods in this class that have instantiations.

RunTimeStack: Every stack method needs to be implemented here. Some additional methods may be helpful.

ByteCode classes: All ByteCode classes need to be created and defined from scratch. Each ByteCode class should have its own private fields and override an init() and execute() method.

Execution and development environment
This assignment was completed using Netbeans IDE 8.2. The Interpreter class has the main method, so we run the program through Interpreter. In order to pass compiled .x.cod files, the working directory needs to be specified for the project and we need to input the file name as an argument (if the file is in a different location, the full path or subdirectories need to be included in the argument).

Scope of work
One of the main focuses of this assignment is to practice core programming principles, especially encapsulation. We need to keep related fields and methods together in classes. Any fields or methods that don't belong (or that can be further organized) should be put into different classes, related classes, or subclasses. For this assignment, we shouldn't be able to modify or use methods in classes that aren't supposed to have access. The VirtualMachine class is the central class (or one of the central classes) in this assignment. If any classes should be able to use methods from other classes, it would be VirtualMachine. Fields however, should only be modifiable within their respective classes, not by other classes.

Compilation and execution

- Install the latest JDK and JRE for your Operating System:
https://docs.oracle.com/javase/7/docs/webnotes/install/

- Navigate to the directory with the .java source files
```
cd <source>
```

- Compile .java source files into .class files. No destination compiles files in the same directory
```
javac <all source files> [optional destination directory]
ex: javac *.java
```

- Create the .jar file with all the .class files. Make sure you navigate to the directory where the .class files are located with cd
```
jar cf <name>.jar <all class files>
ex: jar cf Interpreter.jar *.class
```

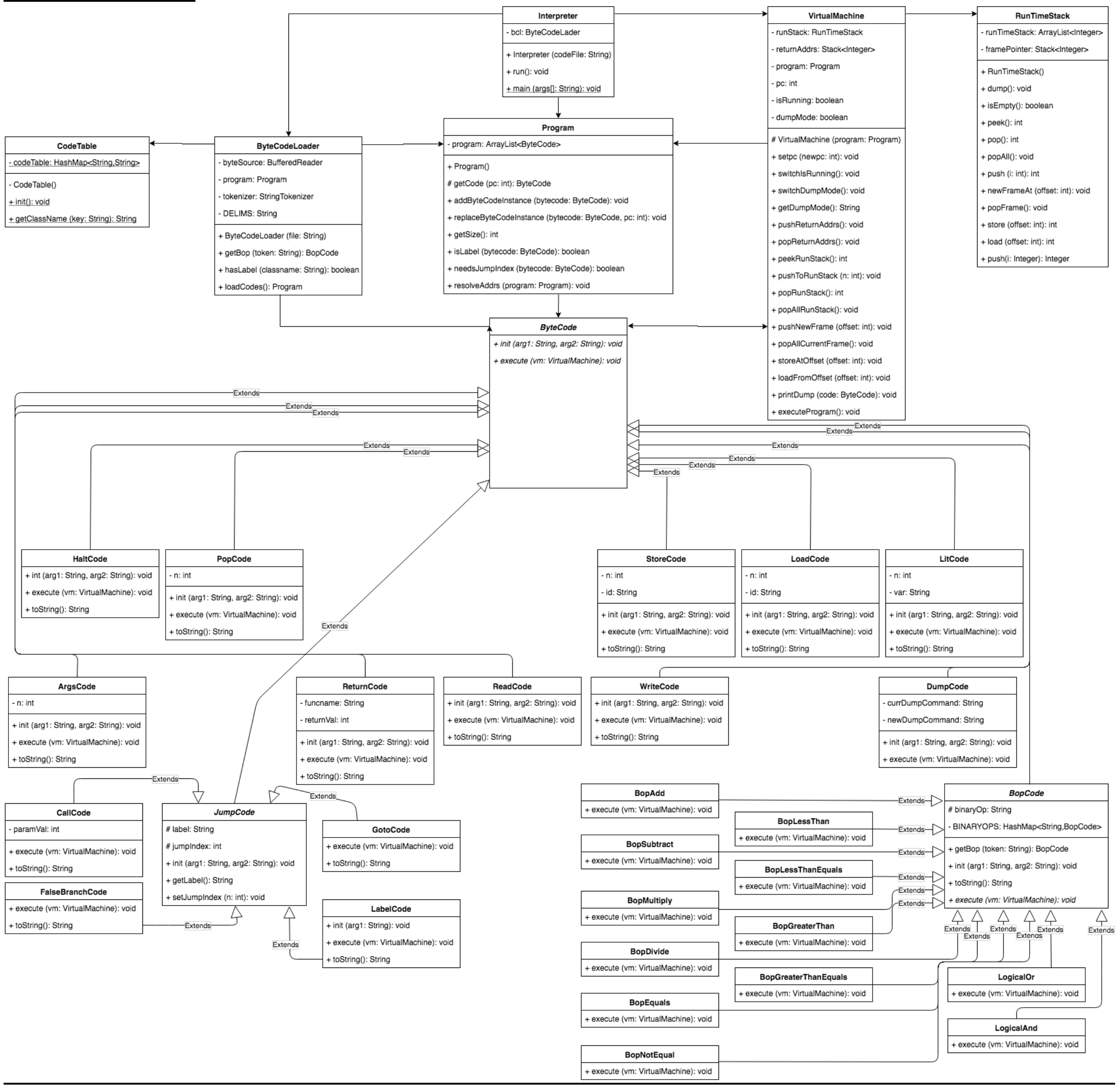
- Run the .jar file and specify the .class file that includes the main method. Also specify the file to read in (which should be a <name>.x.cod file). Make sure you have the correct file path for the .jar file and the .class file with the main method.
```
java -cp <name>.jar <class with main method, including path if needed> <nameof.x.cod file>
ex: java -cp Interpreter.jar interpreter.Interpreter factorial.x.cod
```

Assumptions
In order for the program to work as expected, we need to provide it a correctly compiled file. It should also be assumed that the file is compiled from source code that uses correct logic. If the .x.cod files aren't correctly written/ordered, the output will obviously not produce expected results.

It should also be assumed that for the two compiled files provided, the user will provide valid input. Normal expected input for these two files are integers 0 and greater (but not too large where integer overflow might occur).

Lastly, we're not expected to account for "divide by zero" errors.

Implementation


Interpreter
- bcl: ByteCodeLader
+ Interpreter (codeFile: String)
+ run(): void
+ main (args[]: String): void
CodeTable
- codeTable: HashMap<String,String>
- CodeTable()
+ init(): void
+ getClassName (key: String): String
ByteCodeLoader
- byteSource: BufferedReader
- program: Program
- tokenizer: StringTokenizer
- DELIMS: String
+ ByteCodeLoader (file: String)
+ getBop (token: String): BopCode
+ hasLabel (classname: String): boolean
+ loadCodes(): Program
Program
- program: ArrayList<ByteCode>
+ Program()
# getCode (pc: int): ByteCode
+ addByteCodeInstance (bytecode: ByteCode): void
+ replaceByteCodeInstance (bytecode: ByteCode, pc: int): void
+ getSize(): int
+ isLabel (bytecode: ByteCode): boolean
+ needsJumpIndex (bytecode: ByteCode): boolean
+ resolveAddrs (program: Program): void
VirtualMachine
- runStack: RunTimeStack
- returnAddrs: Stack<Integer>
- program: Program
- pc: int
- isRunning: boolean
- dumpMode: boolean
# VirtualMachine (program: Program)
+ setpc (newpc: int): void
+ switchIsRunning(): void
+ switchDumpMode(): void
+ getDumpMode(): String
+ pushReturnAddrs(): void
+ popReturnAddrs(): void
+ peekRunStack(): int
+ pushToRunStack (n: int): void
+ popRunStack(): int
+ popAllRunStack(): void
+ pushNewFrame (offset: int): void
+ popAllCurrentFrame(): void
+ storeAtOffset (offset: int): void
+ loadFromOffset (offset: int): void
+ printDump (code: ByteCode): void
+ executeProgram(): void
RunTimeStack
- runTimeStack: ArrayList<Integer>
- framePointer: Stack<Integer>
+ RunTimeStack()
+ dump(): void
+ isEmpty(): boolean
+ peek(): int
+ pop(): int
+ popAll(): void
+ push (i: int): int
+ newFrameAt (offset: int): void
+ popFrame(): void
+ store (offset: int): int
+ load (offset: int): int
+ push(i: Integer): Integer
ByteCode
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
Extends
HaltCode
+ int (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
PopCode
- n: int
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
StoreCode
- n: int
- id: String
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
LoadCode
- n: int
- id: String
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
LitCode
- n: int
- var: String
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
ArgsCode
- n: int
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
ReturnCode
- funcname: String
- returnVal: int
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
ReadCode
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
WriteCode
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
DumpCode
- currDumpCommand: String
- newDumpCommand: String
+ init (arg1: String, arg2: String): void
+ execute (vm: VirtualMachine): void
CallCode
- paramVal: int
+ execute (vm: VirtualMachine): void
+ toString(): String
FalseBranchCode
+ execute (vm: VirtualMachine): void
+ toString(): String
JumpCode
# label: String
# jumpIndex: int
+ init (arg1: String, arg2: String): void
+ getLabel(): String
+ setJumpIndex (n: int): void
GotoCode
+ execute (vm: VirtualMachine): void
+ toString(): String
LabelCode
+ init (arg1: String): void
+ execute (vm: VirtualMachine): void
+ toString(): String
BopAdd
+ execute (vm: VirtualMachine): void
BopLessThan
+ execute (vm: VirtualMachine): void
BopSubtract
+ execute (vm: VirtualMachine): void
BopLessThanEquals
+ execute (vm: VirtualMachine): void
BopMultiply
+ execute (vm: VirtualMachine): void
BopGreaterThan
+ execute (vm: VirtualMachine): void
BopDivide
+ execute (vm: VirtualMachine): void
BopGreaterThanEquals
+ execute (vm: VirtualMachine): void
BopEquals
+ execute (vm: VirtualMachine): void
BopNotEqual
+ execute (vm: VirtualMachine): void
BopCode
# binaryOp: String
- BINARYOPS: HashMap<String,BopCode>
+ getBop (token: String): BopCode
+ init (arg1: String, arg2: String): void
+ toString(): String
+ execute (vm: VirtualMachine): void
LogicalOr
+ execute (vm: VirtualMachine): void
LogicalAnd
+ execute (vm: VirtualMachine): void

Code organization

In order to organize and make it easier to implement changes, we can make an abstract ByteCode class and an abstract BopCode class. Their subclasses will inherit related fields and methods to make it easier to run, call, and instantiate them. This also helps with encapsulation.

Results and conclusions

Based on the two .x.cod files provided, this interpreter works as expected. At the moment, the provided files only use simple operations and non-negative integer values. Inputting any unexpected values will probably result in errors or unexpected behavior. In order to further improve the interpreter, unexpected input needs to be protected against to ensure predictable behavior.