Computer Science Department
San Francisco State University
CSC 413
Summer
2018

Assignment 2 – The Interpreter

Note that the due date applies ***to the last commit timestamp into the main branch of your repository.***

1. Overview

For this assignment you will be implementing an interpreter for the mock language X. You can think of the mock language X as a simplified version of Java. The interpreter is responsible to processing byte codes that are created from source code. The interpreter and the Virtual Machine (this will be implemented by you as well) will work together to run a program written in Language X. The two sample programs are a recursive version of computing the nth Fibonacci number and recursively finding the factorial of a number.

And as always, if there are any questions please ask them in slack AND in class. This promotes collaborative thinking which is important. NOTE THIS IS AN INDIVIDUAL ASSIGNMENT but you can collaborate with other students.

2. Requirements

1. Implement ALL the ByteCode classes listed in the table on page 5 of this document. Be sure to create the correct abstractions for the bytecodes. It is possible to have multiple abstractions within the set of byte codes classes.

2. Complete the implementation of the following classes
   a. ByteCodeLoader
   b. Program
   c. RuntimeStack
   d. Virtual Machine

The Interpreter and CodeTable class have already been implemented for you. The Interpreter class is the entry point to this project. All projects will be graded using this entry point. The Interpreter class should NOT be changed. Doing so could causes you program to lose points.

3. Make sure that all variables have their correct modifiers. Projects with all members being public will lose points.

4. Make sure not to break encapsulation. Projects that contain objects or classes trying to access members that it should not be allowed to will lose points. For example, if a bytecode needs to access or write to the runtime stack, it should ***NOT*** be allowed to. It needs to request these operations from the virtual machine. Then the virtual machine will carry out the operation.

   It would also be incorrect to make a method in the virtual machine for each byte code, while it would work, this solution will produce a lot of duplicate code AND points will be deducted for this type of solution. Do your best to understand the operations of each bytecode and how they manipulate the data-structures the Virtual Machine maintains, and you will be able to see that same bytecodes operate on these data-structures in a very similar way. Basically, I am asking to code to the virtual machine and not to the byte codes.

3. Submission

When submitting the project please make sure not to modify the file structure. It is ok to only add files to the bytecode folder. ***NO WHERE ELSE.***

Please store the documentation ***PDF*** in the documentation folder given in the repo.

4. The Interpreter

In the following sections you will find coding hints, class requirements, and other information to help you implement and complete the interpret project. Please read these pages a few times before begin coding.

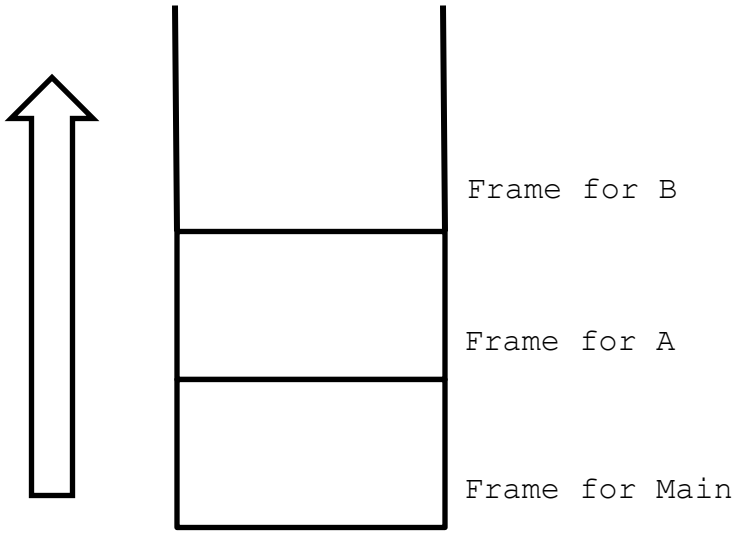
1. Frames (Activation Record) and the Runtime Stack

Frames or activation records are simply the set of variables (actual arguments, local variables, and temporary storage) for each function called during the execution of a program. Given that functions calls and returns happen in a LIFO fashion, we will use a Stack conception to hold and maintain our frames.

For example, if we have the following execution sequence:

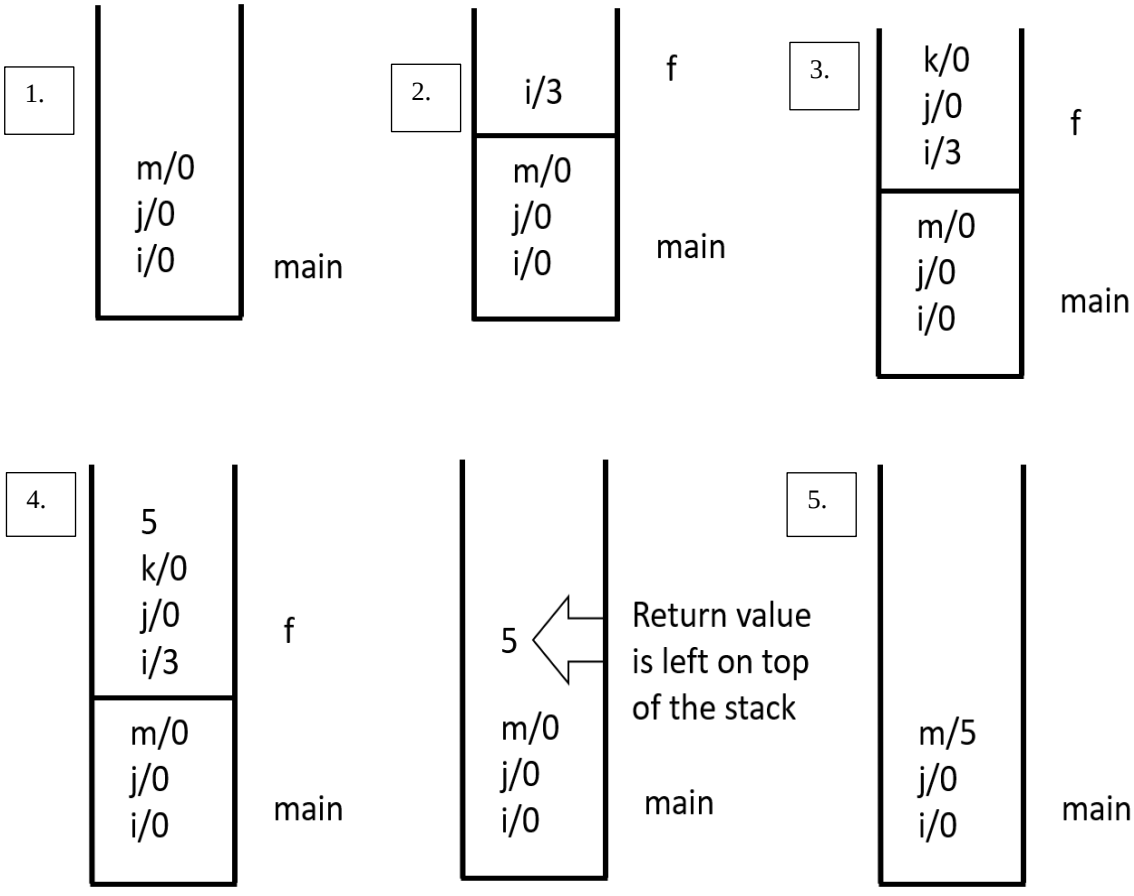
```
Main
        Call A from Main
                Call B from A
```

With the sequence above, we will get the following runtime stack:



Now if we take this concept and apply it to some code written in the Language X we can see a better example of our runtime stack.

```
program {  int i int j
    int f ( int i ) {  2
        int j int k  3
        return i + j + k + 2  3
    }
int m  1
m = f(3)  5
i = write(j+m)
}
```



5. Summary of the X-machine Bytecodes

Bytecode	***Description***	***Example***
HALT	Halt execute of program	HALT
POP	POP n: pop the top n levels of the runtime stack	POP 5 POP 0
FALSEBRANCH	FALSEBRANCH <label> pop the top of the stack; if its false (0) then branch to <label> else execute the next bytecode	FALSEBRANCH xyz<<3>>
GOTO	GOTO <label>	GOTO zyx<<3>>
STORE	STORE N <id> - pop the top of the stack; store the value into the offset n from the start of the frame; <id> is used as a comment, it's the variable name where the data is stored.	STORE 3 i STORE 2
LOAD	LOAD n <id> ; push the value in the slot which is offset n from the start of the frame onto the top of the stack; <id> is used as a comment, it's the variable name where the data is loaded.	LOAD 3 LOAD 2 i
LIT	LIT n – load the literal value n LIT 0 i – this form of Lit was generated to load 0 on the stack to initialize the variable i to the value 0 and reserve space on the runtime stack for i.	LIT 5 LIT 0 i
ARGS	ARGS n ; Used prior to calling a function. n = # of args this instruction is immediately followed by the CALL instruction; the function has n args so ARGS n instructs the interpreter to set up a new frame n down from the top of the runtime stack, so it will include the	ARGS 4 ARGS 0 ARGS 2

	arguments in the new frame for the function,	
CALL	CALL <funcname> - transfer control to the indicated function	CALL f CALL f<<3>>
RETURN	RETURN <funcname>; Return from the current function; <funcname> is used as a comment to indicate the current function, RETURN is generated for intrinsic functions.	RETURN f<<2>> REUTRN
BOP	BOP <binary op> - pop top 2 levels of the stack and perform the indicated operation – operations are + - / * == != <= > >= < | & | and & are logical operators not bitwise operators. Lower level is the first operand: Eg: <second-level> + <top-level>	BOP + BOP – BOP /
READ	READ; Read an integer; prompt the user for input and put the value onto of the stack. Make sure the input is validated.	READ
WRITE	WRITE; Write the value of the top of the stack to output. Leave the value on the top of the stack	WRITE
LABEL	LABEL <label>; target for branches; (see FALSEBRANCH and GOTO)	LABEL xyz<<3>> LABEL Read
DUMP	This bytecode is used to set the stack of dumping in the virtual machine. When dump is on, after the execution of each bytecode, the state of the runtime stack is dumped to the console.	DUMP ON DUMP OFF

1. *Sample compiled source code file*

ByteCodes	Source Code
GOTO start<<1>>	program {
LABEL Read	
READ	
RETURN	
LABEL Write	
LOAD 0 dummyformal	<bodies for read/write
WRITE	functions>
RETURN	
LABEL start<<1>>	
LIT 0 i	int i
LIT 0 j	int j
GOTO continue<<3>>	
LABEL f<<2>>	int f(int i) {
LIT 0 j	int j
LIT 0 k	int k
LOAD 0 i	i + j + k + 2
LOAD 1 j	
BOP +	
LOAD 2 k	
BOP +	
LIT 2	
BOP +	
RETURN f<<2>>	return i + j + k + 2
POP 2	<remove local variables -
LIT 0	j,k>
RETURN f<<2>>	
LABEL continue<<3>>	
LIT 0 m	int m
LIT 3	f(3)
ARGS 1	
CALL f<<2>>	
STORE 2 m	m = f(3)
LOAD 1 j	j + m
LOAD 2 m	
BOP +	
ARGS 1	
CALL Write	write(j+m)
STORE 0 i	i = write(j+m)
POP 3	<remove local variables -
HALT	i,j,m>

2. Execution Trace

ByteCodes	Runtime stack after Executing Bytecode. Stacks grows to the right>
GOTO start<<1>>	
LABEL start<<1>>	
LIT 0 i	[0]
LIT 0 j	[0,0]
GOTO continue<<3>>	[0,0]
LABEL continue<<3>>	[0,0]
LIT 0	[0,0,0]
LIT 3	[0,0,0,3]
ARGS 1	[0,0,0] [3]
CALL f<<2>>	[0,0,0] [3]
LABEL f<<2>>	[0,0,0] [3]
LIT 0	[0,0,0] [3,0]
LIT 0	[0,0,0] [3,0,0]
LOAD 0	[0,0,0] [3,0,0,3]
LOAD 1	[0,0,0] [3,0,0,3,0]
BOP +	[0,0,0] [3,0,0,3]
LOAD 2	[0,0,0] [3,0,0,3,0]
BOP +	[0,0,0] [3,0,0,3]
LIT 2	[0,0,0] [3,0,0,3,2]
BOP +	[0,0,0] [3,0,0,5]
RETURN	[0,0,0,5]
STORE 2	[0,0,5]

The following Bytecodes change the state of the runtime stack in the following way:

Stack Changes:	+1	-1	0
	LIT	BOP	GOTO
	LOAD	STORE	LABEL

Simple Interpreter Schema

1. Load the bytecodes generated by the compiler
2. Execute the codes in the Virtual Machine

class ByteCode is the abstract class which each bytecode (in-) directly extends
e.g. class ReadCode extends ByteCode

6. ByteCodeLoader Class

The bytecode loader class is responsible for loading bytecodes from the source into a data-structure that stores the entire program. We will use an ArrayList to store our bytecodes. This ArrayList will be called program. It will have a class called Program as well. More on this later.

The ByteCodeLoader class will also implement a function that does the following:

1. Reads in the next bytecode from the source file.
2. Build and instance of the class corresponding to the bytecode. For example, if we read LIT 2, we ill create an instance of the LitCode class.
3. Read in any additional arguments for the given bytecode if any exists. Once read, we will pass these arguments to the bytecode's init function.
4. Store the full initialized bytecode instance into the program data-structure.
5. Once all bytecodes are loaded, we will resolve all symbolic addresses

Address resolution will modify the source code in the following way:

The Program class will hold the bytecode program loaded from the file. It will also resolve symbolic addresses in the program. For example, if we have the following program below
0. FALSEBRANCH continue<<6>> 1. LIT 2 2. LIT 2 3. BOP == 4. FALSEBRANCH continue<<9>> 5. LIT 1 6. ARGS 1 7. CALL Write 8. STORE O i 9. LABEL continue<<9>> 10. LABEL continue<<6>>
After address resolution has been completed the source code should look like the following (NOTE you should not modify the original source code file, these changes are made to the Program object):
0. FALSEBRANCH 10 1. LIT 2 2. LIT 2 3. BOP == 4. FALSEBRANCH 9 5. LIT 1 6. ARGS 1 7. CALL Write 8. STORE O i 9. LABEL continue<<9>> 10. LABEL continue<<6>>

File	CodeTable		Program
	Keys	Values	
LIT 1	"LIT"	"LitCode"	<LitCode instance>
LOAD 2	"LOAD"	"LoadCode"	<LoadCode instance>
READ	"READ"	"ReadCode"	<ReadCode instance>

7. CodeTable

The code table class is used by the ByteCodeloader Class. It simply stores a HashMap which allows us to have a mapping between bytecode as they appear in the source and their respective code in the Interpreter program.

A sample entry into this HashMap would be (“HALT”,”HaltCode”), where HALT is the bytecode in source file and HaltCode is the class representing the HALT bytecode.

The code table can be populated though an initialization method. It is ok to hard-code the statements that populate the data in the CodeTable class.

With the CodeTable HashMap correctly populated we can have the ByteCodeLoader loadCode’s function build instances of each bytecode with strings. This is a different approach compared to what we did in Assignment 1 where the objects themselves were in the HashMap. The reason for the different approach is because two bytecodes that are the same type of object CAN have different values. For example,

ARGS 0
ARGS 3

These two bytecodes will both create an ArgsCode class instance, but the objects will contain different values since the bytecode arguments are different.

With this mapping (strings to Class names) we can use Java Reflection to build instances of classes. Java reflection is used to inspect classes, interfaces and their members (methods and data-fields). In other languages (including SQL) this is called introspection.

Therefore, using reflection and our HashMap, we can create new instances in the following way:

```
//The string below is a sample, the value should be read from a file.
String code = “HALT”;
// using the string, retrieve the class name from the CodeTable.
String className = CodeTable.get(code);
// With the class name, retrieve the Class blueprint object.
Class c = Class.forName(“packagename.”+className):
// Note when using the forName function, you need to specify the fully
// qualified class name. This includes the packages the class is
// contained in. The names will be separated by . not / .

// Create an instance for the given class blueprint.
ByteCode bc = (ByteCode) c.getDeclaredConstructor().newInstance();
```

At this point we have an instance of the given bytecode. Although, its reference type is ByteCode. Its actual type will be HaltCode.

The example above uses the no-arg contractor. This is ok since all bytecodes do not need to have any constructors defined. And when classes do not define their own constructors, Java will give you a no-arg constructor for free. Then we will use the respective init function defined in each ByteCode subclass.

The above code gives us the ability to dynamically creates of classes during runtime.

Please note that in older versions of Java we can create and instance using the newInstance method directory with the class object. However, this is not recommended as the Class.newInstance method bypasses exception checking. http://errorprone.info/bugpattern/ClassNewInstance

8. Program Class

The program class will be responsible for storing all the bytecodes read form the source file. We will store bytecodes in an ArrayList which has a designated type of ByteCode. This will ensure only ByteCodes and its subclass can only be added to the ArrayList.

This class should at least contain two instance functions:

- `public ByteCode getCode(int index)`
  - this function returns the ByteCode at a given index.
- `public void resolveAddress()`
  - this function will resolve all symbolic addresses in the program.

Resolving symbolic addresses can be done in many ways. But it is up to you to figure how a clean implementation for mapping the generated labels the compiler users to absolute addresses in the program. An example is given below.

The Program class will hold the bytecode program loaded from the file. It will also resolve symbolic addresses in the program. For example, if we have the following program below
11. FALSEBRANCH continue<<6>> 12. LIT 2 13. LIT 2 14. BOP == 15. FALSEBRANCH continue<<9>> 16. LIT 1 17. ARGS 1 18. CALL Write 19. STORE O i 20. LABEL continue<<9>> 21. LABEL continue<<6>>
After address resolution has been completed the source code should look like the following (NOTE you should not modify the original source code file, these changes are made to the Program object):
11. FALSEBRANCH 10 12. LIT 2 13. LIT 2 14. BOP == 15. FALSEBRANCH 9 16. LIT 1 17. ARGS 1 18. CALL Write 19. STORE O i 20. LABEL continue<<9>> 21. LABEL continue<<6>>

9. CodeTable

Class responsible for storing a HashMap that maps source code ByteCodes to their class representations. The HashMap needs to be private and static. The class will contain to static functions:

- `public static String get(String key)`
  - Given a key which is used to represent a ByteCode is source and return its class name. For example, if key has the value "HALT" its return value is "HaltCode"
- `public static void init()`
  - the init function will create an entry in the HashMap for each byte code listed in the table presented earlier. This table will be used to map bytecode names to their bytecode classes. For example, POP → PopCode.

10. ByteCodeLoader (continued)

The bytecode loader class is responsible for loading bytecodes from the source into a data-structure that stores the entire program. We will use an ArrayList to store our bytecodes. This ArrayList will be called program. It will have a class called Program as well. More on this later.

- `public ByteCodeLoader(String programFile) throws IOException`
  - This constructor will create a new bytecode loader which contains a BufferedReader object. Use the programFile string to initiate the BufferedReader. Reading from the file IS NOT allowed from the constructor.
- `public Program loadCodes()`
  - This function is responsible for loading all bytecodes into the program object. Once all bytecodes have been loaded and initialized, the function then will request that the program object resolve all symbolic addresses before returning. An example of a before and after has been given in the Program section of this document.

11. The Runtime Stack

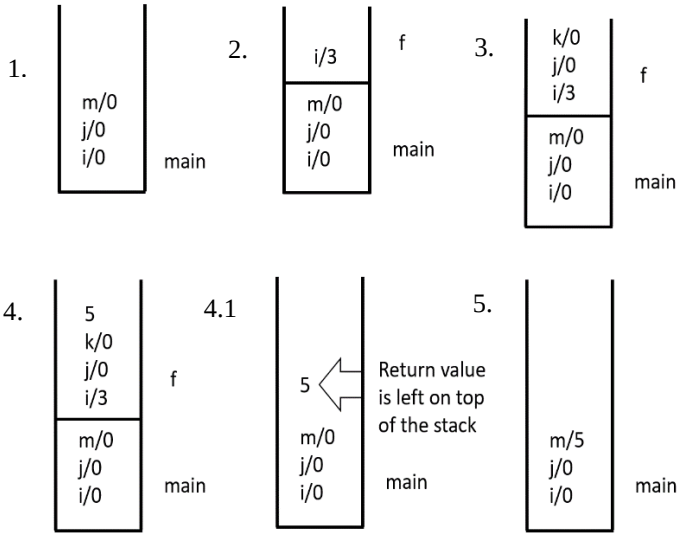
Records and processes the stack of active frames. This class will contain two data structures used to help the VirtualMachine execute the program. It is ***VERY*** important that you do not return any of these data structures. Which means there should be NO getters and setters for these data structures.

The RuntimeStack class will use the following two structures:

1. Stack FramePointer
    a. This stack is used to record the beginning of each activation record(frame) when calling functions.
2. ArrayList<Integer> runStack
    a. This ArrayList is used to represent the runtime stack. It will be an ArrayList because we will need to access ALL locations of the runtime stack.

When interpreting(or trying to understand) the Runtime stack class you should always use BOTH data structures. Both are needed for the correct executing of the program.

Recall from earlier:



Initially the framePointer stack will have 0 for step 1 above. When we call function f at step 2, framePointer stack will have the values 0 and 3 since 0 is the start of main and 3 is the start of f. At step 4.1 we pop the framePointer stack (3 is popped). Then the framePointer Stack will only contain the value 0. This is Mains starting frame index. Note when transitioning from step 4 and 4.1 there is going more work that is needed to be done to clean up each frame before returning from functions.

1. RunTimeStack class

The RunTimeStack class maintains the stack of active frames. When we call a function, we will push a new frame on the attack. When we return from a function we will pop this frame off the stack. Please note in this section we are refereeing to the RuntimeStack class and the runStack data-field contained inside of it.

The runtime stack will contain the following functions (more can be added if encapsulation is not broken.):

- `public RuntimeStack ()`
    - `Construction used to initialize the runtime stack.`
- `Public void dump()`
    - `Void function used to dump the current stack of the RuntimeStack. When printing the runtime stack make sure to include divisions between frames. If a frame is empty, this must be shown as well.`

- public int peek()
  - returns the top of the stack without removing the item.
- public int pop()
  - removes an item from the top of the stack and returns it.
- public int push(int i)
  - used to add an item to the top of the RuntimeStack. Item added is also returned.
- public void newFrameAt(int offset)
  - creates a new frame in the RuntimeStack class. The parameter offset used to denote how many slots down from the top of RuntimeStack for starting a new frame.
- public void popFrame()
  - we pop the top frame when we return from a function. Before popping, the function's return value is at the top of the stack, so we'll save the value, the pop the top frame and then push the return value back to the stack. It is assumed return values are at the top of the stack.
- public int store(int offset)
  - Used to store values into variables. Store will pop the top value of the stack and replace the value at the given offset in the current frame. The value stored is returned.
- public int load(int offset)
  - Used to load variables onto the RuntimeStack from a given offset within the current frame. This means we will go to the offset in the current frame, copy the value and push it to the top of the stack. No values should be removed with loads.
- public Integer push(Integer val)
  - Used to load liters onto the RuntimeStack. For example, LIT 5 or LIT 0 will call push with val being 5 or val being 0.

12. The Virtual Machine

Class used for executing the given program. The VM is the controller of this program. All operations need to go through this class.

It will contain the following data fields:

- RunTimeStack runStack
- int pc
  - the program counter (current bytecode being executed).
- Stack returnAddrs
  - Used to store return addresses for each called function(excluding main)
- Boolean isRunning
  - Used to determine whether the VM should be executing bytecodes.
- Program program
  - Reference to the program object where all bytecodes are stored.

The VM will contain many functions. These will not be listed. The reason for this is that you are expected to abstract the operations needed by the bytecodes and create clean and concise functions. Do your best to limit the amount of duplicate code. Points will be taken away from VMs that contain a lot of duplicate code (or simply creating a function for each byte code). Also, limit the amount of ByteCode logic that appears in the VM. For example, if you are executing the ReadCode bytecode, the processing of reading is not done by the VM, its done by the ReadCode class and then the ReadCode class request the VM to push the read value onto the RuntimeStack.

The returnAddrs stack stores the bytdecode index(PC) that the virtual machine should execute when the current function exits. Each time a function is entered, the PC should be pushed onto the returnAddrs stack. When a function exits the PC should be restored to the value that is popped from the top of the returnAddrs Stack.

One important function in the VM is execute program. A sample base function is given:

```
public void executeProgram(){
	pc = 0;
	runStack = new RunTimeStack();
	returnAddrs = new Stack<Integer>();
	isRunning = true;
	while(isRunning){
		ByteCode code = program.getCode(pc);
		code.execute(this);
		//runStack.dump(); // Used to dump runstack state.
		pc++;
	}
}
```

Note that we can easily add new bytecodes without affecting the operation of the Virtual Machine. We are using Dynamic Binding to achieve code flexibility, extern ability and readability. This loop should very easy to read and understand.

13. The Interpreter Class

The interpreter class is used as the entry point for this assignment. It should remain unchanged. Changing this file could causes points to be lost. Do your best to not let exceptions propagate to this class.

```
package interpreter;

import java.io.*;

/**
 * <pre>
 *     Interpreter class runs the interpreter:
 *     1. Perform all initializations
 *     2. Load the bytecodes from file
 *     3. Run the virtual machine
 * </pre>
 */
public class Interpreter {

      private ByteCodeLoader bcl;

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

      void run() {
            Program program = bcl.loadCodes();
            VirtualMachine vm = new VirtualMachine(program);
            vm.executeProgram();
      }

      public static void main(String args[]) {

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

14. Coding Hints – Suggested Order to Write Code

Below is an ordered list of how you may start this assignment. Note that this is not the only way. If you have a better way you may use your way.

1. Create ALL ByteCode classes listed in the table shown earlier. The methods of the class can be left empty(as method stubs).
2. Implement ByteCodeLoader
3. Implement Program (more specifically the resolveAddress function).
4. Implement RunTimeStack
5. Implement VirtualMachine
6. Fill in ALL the empty ByteCode classes and their method stubs.

You will assume that the given bytecode source programs (.cod files) you will use for testing are generated correctly and contain NO ERRORS.

You will need to make sure that you protect the RunTimeStack from stack overflow or stack underflow errors. Also make sure no bytecode can pop past any frame boundary.

You should provide as much documentation as you can when implementing The Interpreter. Short, OBVIOUS functions don't need comments. However, you should comment each class describing its function and purpose. Take in consideration that your code is the first level of documentation. How you name things in your code matters. If you find yourself writing a lot of comments, the you are either explain a complex algorithm, which is OK, or your code contains named variable and methods that are not descriptive enough.

DO NOT provide any method that returns any components contained WITHING the VirtualMachine(this is the exact situation that will break encapsulation) – you should request the VM to perform operations on its components. This implies that the VM owns the components and is free to change them as needed without breaking clients' code (for example, suppose I decided to change the name of the variables that holds my runtime stack - if your code had a referenced that variable then your code would break. This is not an unusual situation – you can consider the names of methods in the Java libraries that have been marked deprecated).

The only downside is it might be a bit inefficient. Since I want to impress on everyone important software engineering issues, such as encapsulation benefits, I want to enforce the requirement that you ***DO NOT BREAK*** encapsulation. Consider that the VM calls the individual ByteCodes' execute method and passes itself as a parameter. For the ByteCode to execute, it must invoke 1 more methods in the runStack object. It can this by executing VM.runStack.pop(); however, this DOES break encapsulation. To avoid this, you will need to have a corresponding set of methods within the VM that do nothing more than pass the call to the runStack. For example, a pop function can be implemented in the VM in the following way:

```
public int popRunStack() { return runStack.pop();}
```

Then in a ByteCodes' execute method a pop can be executed as:

```
int temp = VM.popRunStack();
```

Each bytecode class should have fields for its specific arguments. The abstract class ByteCode SHOULD NOT CONTAIN ANY FIELDS(instance variables) THAT RELATE TO ARGUMENTS. This is a design requirement.

It is easier to think in more general terms (i.e. plan for any number of arguments for a bytecode). Note that the ByteCode abstract class should be aware of the peculiarities of any bytecode. That is, some bytecodes might have zero arguments (HALT) or one argument, etc. Consider providing an init function with each bytecode class. After constructing a bytecode instance during the loadCodes phase, you can then call init passing in an ArrayList String of arguments. Each ByteCode object will then interrogate the ArrayList and extract the needed arguments itself. The ByteCode abstract class should not record any arguments for any bytecodes. Each ByteCode concrete class will need instance variables for each of their arguments. There may be a need for additional instance variables as well.

When you read a line from the bytecode file, you should parse the arguments and placed in an ArrayList of Strings. Then passing this ArrayList to the bytecodes init function. Each bytecode is responsible for extracting relevant information from the ArrayList and storing it as private data.

The ByteCode abstract class and its concrete classes must be contained in a package named bytecode.

Any output produced by the WRITE bytecode will be interspersed with the output from dumping (if dumping is turned on). In the WRITE action you should print one number per line. DO NOT print out something like :

Program output : 2

You only need to include the value and that's it. It should just be

2

There is no need to check for division by zero error. Assume that you not have a test case where this occurs.

15. DUMPING PROGRAM STATE

There is one special bytecode that is used to determine whether the VM should dump the current state of the runStack. This bytecode is call DUMP. It will have 2 states ON and OFF. No other form of DUMP will occur in this program or any given program.

DUMP ON is an interpreter command to turn on runtime dumping. This will set an interpret switch that will cause runStack dumping AFTER execution of EACH bytecode. This switch or variable will be stored as a private data-field in the VirtualMachine class.

DUMP OFF will reset the switch to end dumping. Note that DUMP instructions will not be printed.

DO NOT dump program state unless dumping is turned ON.

Consider the following bytecode program:

```
GOTO start<<1>>
LABEL Read
REUTRN
LABEL Write
LOAD 0 dummyformal
WRITE
RETURN
LABEL start<<1>>
LIT 0 i
LIT 0 j
GOTO continue<<3>>
LABEL f<<2>>
LIT 0 j
LIT 0 k
LOAD 0 i
LOAD 0 j
DUMP OFF
BOP +
LOAD 2 k
BOP +
LIT 2
BOP +
RETURN f<<2>>
POP 2
LIT 0 GRATIS-RETURN-VALUE
RETURN f<<2>>
LABEL continue<<3>>
DUMP ON
LIT 0 m
LIT 3
ARGS 1
CALL f<<2>>
DUMP ON
STORE 2 m
DUMP OFF
LOAD 1 j
LOAD 2 m
BOP +
ARGS 1
WRITE
STORE 0 i
POP 3
HALT
```

When dumping is turned on you should print the following information **JUST AFTER** executing the next bytecode:

- Print the bytecode that was just executed (DO NOT PRINT the DUMP bytecodes)
- Print the runtime stack with spaces separating frames (just after the bytecode was executed). Values contained in one frame should be surrounded by [and].
- If dump is not on then DO NOT print the bytecode, nor dump the runtime stack.

Following is an example of the expected printout from the program given above (we only give a portion of the printout as an illustrative example).

```
LIT 0 m    int m
[0,0,0]
LIT 3
[0,0,0,3]
ARGS 1
[0,0,0] [3]
CALL f<<2>>  f(3)
[0,0,0] [3]
LIT 0 j    int j
[0,0,0] [3,0]
LIT 0 k    int k
[0,0,0] [3,0,0]
LOAD 0 i <load i>
[0,0,0] [3,0,0,3]
LOAD 1 j <load j>
[0,0,0] [3,0,0,3,0]
…
STORE 2 m    = 2
[0,0,5]
```

Note:

Following shows the output if dump is on and 0 is at the top of the runtime stack.
RETURN f<<2>> exit f:0 note that 0 is returned from f<<2>>

If Dumping is turned on and we encounter an instruction such as WRITE then output as usual; the value may be printed either before or after dumping information e.g.,

```
LIT 3
[0,0,03]
3
WRITE
[0,0,0,3]
```

The following dumping actions are taken for the indicated bytecodes, other bytecodes do not have any special treatment when being dumped.

LIT 0 <id> int <id> note for simplicity ALWAYS assume lit is an int declaration.
LIT 0 j int j
LOAD c <id> <load id>
LOAD 2 a <load a>
STORE c <id> <id>=<top-of-stack>
e.g. if the has:
[0,1,2,3]
and we execute STORE 1 k
then we will dump:
STORE 1 k k = 3
RETURN <id> exit <base-id>:<value>
<value> is the value being returned form the function
<base-id> is the actual id of the function; e.g. for RETURN f<<2>>, the <base-id> is f

CALL <id> <base-id>(<args>)

e.g. if the stack has

[0,1,2,3]

And we execute CALL f<<3>> after executing ARGS 2 then we will dump

CALL f<<3>> f(2,3)

We will also strip any brackets (< and >); the ARGS bytecode just seen tells us that we have a function with 2 arguments, which are the top 2 levels of the stack – the first arg was pushed first, etc.

DUMPING IMPLEMENTATION NOTES

The Virtual Machine maintains the state of your running program, so it is a good place to have the dump flag. You should not use a stack variable in the ByteCode class.

The dump method should be a part of the RunTimeStack class. This method is called without any arguments. Therefore, there is no way to pass any information about the VM or the bytecode classes into RunTimeStack. As a result, you can't really do much dumping inside RunTimeStack.dump() except for dumping the state of the RunTimeStack itself. Also, ***NO BYTECODE CLASS OR SUBCLASS SHOULD BE CALLING DUMP.***

It is impossible to determine the declared type of a variable by looking at the bytecode file. To simplify matters you should assume whenever the interpreter encounters LIT 0 x (for example), that it is an int. When you are dumping the bytecodes, it is ok to represent the values as int.