Computer Science Department
San Francisco State University
CSC 413
Fall 2017

Assignment 3 – The Interpreter

Due Date

Saturday, October 21st, 2017 @ MIDNIGHT

Monday, October 23rd, 2017 @ MIDNIGHT FOR 75% credit.

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

Overview

For this assignment you will be implementing an interpreter for the mock language X. Some code has been given to you to get you started.

Also attached to this assignment doc is the pages for explaining the components of the interpreter. The information in the pages should be sufficient to complete the assignment.

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.

Submission

You are required to submit your source code via the GitHub assignment, and documentation in PDF format as described in Documentation Guidelines (ilearn link) into the documentation folder.

Requirements

1. Implement ALL the ByteCodes classes listed in the table on page {}. Be sure the abstract similarities between the classes and create the correct parent class.
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 grading using this entry point.

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 be allowed to. It needs to request these operations from the virtual machine. Then the virtual machine will carry out the operation.

   It is ill advised to take the naive approach and 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.

Notes about the attached pages.

- **Some of the information in the attached pages may be out of date. BUT not much. For example, the pages call to use a vector for the runtime stack. Vectors in java, while they still exist are deprecated and new projects should not use them. You may use an ArrayList in its place.**

XII. The Interpreter

Frames (Activation Records) and the Runtime stack

The set of variables (actual arguments, locals, temps) for each function are stored in the ***activation record*** or ***frame***..

Since functions are called/returned from in a ***LIFO*** fashion we use a **stack** to hold the frames - the ***runtime stack***

If we have the calling sequence:

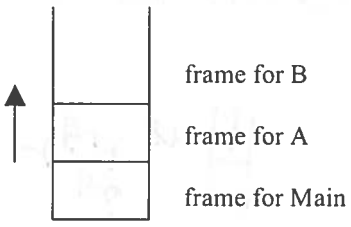
Main program

...

call A (called from Main)

call B (called from A)

Runtime Stack:



Frames: arguments + local variables + temporary storage

```
program { int i      int j
   int f(int i)   {      (2)
         int j      int k      (3)
         return i + j + k  + 2  (4) the sum will be on the stack just prior to return
   }
int m          (1)
m = f(3)       (5)
i = write(j+m)
}
```

offset starts from the current frame.

1:
m/0
j/0
i/0

main

2:
i/3	f
m/0	
j/0	
i/0	

main

3:
k/0	
j/0	
i/3	f
m/0	
j/0	
i/0	

main

4:
5	
k/0	
j/0	
i/3	f
m/0	
j/0	
i/0	

main

4':
5	← return value is left on top of old frame
m/0	
j/0	
i/0	

main

3
4

43 pop 3) - pop 4

5:
m/5
j/0
i/0

main

Summary of the X-machine Bytecodes

Bytecode	Description	Examples
HALT	*halt* execution	HALT
POP	*POP n*: Pop top *n* levels of runtime stack	POP 5 POP 0
FALSEBRANCH	*FALSEBRANCH <label>* - pop the top of the stack; if it's *false* (0) then branch to <label> else execute the next bytecode	FALSEBRANCH xyz<<3>>
GOTO	*GOTO <label>*	GOTO xyz<<3>>
STORE	*STORE n <id>* - pop the top of the stack; store 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 2 i
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 from which the data is loaded	LOAD 3 j
LIT	*LIT n* - load the *literal value n* *LIT 0 i* – this form of the Lit was generated to load 0 on the stack in order to initialize the variable *i* to 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 = #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*, so it will include the arguments	ARGS 4
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>> RETURN
BOP	*bop <binary op>* - *pop top 2 levels* of the stack and perform indicated operation – operations are + - / * == != <= > >= < | & *| and & are logical operators, not bit operators* lower level is the *first operand*: e.g. *<second-level> + <top-level>*	BOP +
READ	*READ* ; Read an integer; prompt the user for input; put the value just read on top of the stack	READ
WRITE	*WRITE* ; Write the value on top of the stack to output; *leave the value on top of the stack*	WRITE
LABEL	*LABEL <label>;* target for branches; (see *FALSEBRANCH, GOTO*)	LABEL xyz<<3>> LABEL Read

two abstract methods. { execute. init.

Bytecodes	*Source Program*
GOTO start<<1>>	program {
LABEL Read	
READ	
RETURN	
LABEL Write	
LOAD 0 dummyFormal	<bodies for read/write functions>
WRITE	
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 - j,k>
LIT 0 GRATIS-RETURN-VALUE	
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 - i,j,m>
HALT	

Execution Trace

Bytecodes executed	*Runtime Stack after Executing Bytecode*
	<stack grows to the right>
GOTO start<<1>>	--
LABEL start<<1>>	
LIT 0	[0]
LIT 0	[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] -- one arg loaded on top of stack
CALL f<<2>>	[0,0,0] [3] -- new frame for *f*
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 from offset 0 in current frame
LOAD 1	[0,0,0] [3,0,0,3,0]
BOP +	[0,0,0] [3,0,0,3]
	-- add top 2 levels; replace with sum
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]
	-- move top to start of frame; reset frame
STORE 2	[0,0,5]

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*

The ByteCodeLoader (BCL)

- Reads in the next bytecode
- Builds an instance of the class corresponding to the bytecode - e.g. if we read *LIT 2* then we build a new *LitCode* instance
- The bytecode class instance is added to the ***Program***
- After all bytecodes are loaded, the symbolic addresses are resolved - e.g.

```
* The Program class will hold the bytecode program loaded from file
* It will also resolve symbolic addresses in the program - e.g.
* if we have the following bytecode program
*
* 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 0 i
* 9. LABEL continue<<9>>
* 10. LABEL continue<<6>>
* <br>
* The addresses in the program above will be resolved to:<br>
*
* 0. FALSEBRANCH 10
* 1. LIT 2
* 2. LIT 2
* 3. BOP ==
* 4. FALSEBRANCH 9
* 5. LIT 1
* 6. ARGS 1
* 7. CALL <address of Write>
* 8. STORE 0 i
* 9. LABEL continue<<9>>
* 10. LABEL continue<<6>>
```

e.g.

File	**CodeTable**		**Program**
	Keys	***Values***	
LIT 1	"LIT"	"LitCode"	<LitCode instance>
LOAD 2	"LOAD"	"LoadCode"	<LoadCode instance>
READ	"HALT"	"HaltCode"	<ReadCode instance>
	...	...	
	Strings	*Strings*	

CodeTable

- Used by BCL
- Contains a *HashMap* with keys being the bytecode strings (e.g. "HALT","POP")

  and values being the name of the corresponding class (e.g. "HaltCode","PopCode")

```
* ByteCodeLoader class will load the bytecodes from the file into the Virtual Machine
* When the bytecodes are loaded we'll get the string for the bytecode and then we'll get
 * the bytecode class for that string from the
* Codetable to construct an instance of the bytecode; e.g.
* if the bytecode loader reads the line: <br>
* HALT <br>
* then we'll build an instance of the class HaltCode and store that instance in the
* Program object;
```

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

The following statements are NOT part of the CodeTable class – they are included here for illustrative purposes

```
String code = "HALT";

String codeClass = CodeTable.get(code);
ByteCode bytecode =

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

// Note that the newInstance method requires an accessible no-argument constructor for
// the class; by accessible constructor we mean that the constructor should be e.g.
public
```

Class.forName("interpreter.HaltCode") will return the class *interpreter.HaltCode*

(Class.forName("interpreter.HaltCode").newInstance()); will build a *new instance* of the class *interpreter.HaltCode*

Note the ability to *dynamically create instances* (we don't know which classes will be used *statically*). **This dynamic nature is extremely important** in the context of e-commerce with dynamically produced content/dynamic web pages/plugins added to e.g. Netscape ==> just ***provide an interface*** for the plugins and they are useable.

Javadoc Documentation of Selected Interpreter Classes

The documentation is obtained by changing directory to the root directory of the system:
d:cd d:\
d:javadoc interpreter -- this creates the documentation for the interpreter package

Class interpreter.Program

```
java.lang.Object
   |
   +----interpreter.Program
```

public class **Program**

extends Object

```
  The Program class will hold the bytecode program loaded from
file
  It will also resolve symbolic addresses in the program - e.g.
  if we have the following bytecode program:
  1. FALSEBRANCH continue<<6>>
  2. LIT 2
  3. LIT 2
  4. BOP ==
  5. FALSEBRANCH continue<<9>>
  6. LIT 1
  7. ARGS 1
  8. CALL Write
  9. STORE 0 i
  10. LABEL continue<<9>>
  11. LABEL continue<<6>>

  The addresses in the program above will be resolved to:
  1. FALSEBRANCH 11
  2. LIT 2
  3. LIT 2
  4. BOP ==
  5. FALSEBRANCH 10
  6. LIT 1
  7. ARGS 1
  8. CALL <address of Write>
  9. STORE 0 i
  10. LABEL continue<<9>>
  11. LABEL continue<<6>>
```

Class interpreter.CodeTable

```
java.lang.Object
   |
   +----interpreter.CodeTable
```

public class **CodeTable**

extends Object

get

```
public static String get(String code)
```

init

```
public static void init()
```

init each entry with the name of the corresponding class for the code; when the bytecodes are loaded we'll get the string for the bytecode and then we'll get the bytecode class for that string from the Codetable to construct an instance of the bytecode; e.g. if the bytecode loader reads the line:
HALT
then we'll build an instance of the class HaltCode and store that instance in the Program object; If we add more bytecodes then we'll need another class for the bytecode to indicate how to interpret it and we'll have to add the code to the table below

Class interpreter.ByteCodeLoader

```
java.lang.Object
   |
   +----interpreter.ByteCodeLoader
```

public class **ByteCodeLoader**

extends *Object*

ByteCodeLoader class will load the bytecodes from the file into the Virtual Machine
When the bytecodes are loaded we'll get the string for the bytecode and then we'll get the bytecode class for that string from the Codetable to construct an instance of the bytecode; e.g. if the bytecode loader reads the line:
HALT
then we'll build an instance of the class HaltCode and store that instance in the Program object;

ByteCodeLoader

```
public ByteCodeLoader(String programFile) throws IOException
```

loadCodes

Load all the codes from file; request Program to resolve any branch addresses

from symbols to their address in code memory, e.g.

```
1. GOTO addr              GOTO 5
2. LOAD 3                 LOAD 3
3. LIT 2         ==>      LIT 2
4. STORE 4                STORE 4
5. LABEL addr             LABEL addr
```

```
Return the bytecode program in an appropriate data structure for
processing by the VirtualMachine

Note that just prior to returning, this method will request the
constructed Program to resolve all addresses
```

The Runtime Stack

Records and processes the stack of active frames

Uses:

Stack framePointers -- used to record *prior frame pointers* when calling

-- functions

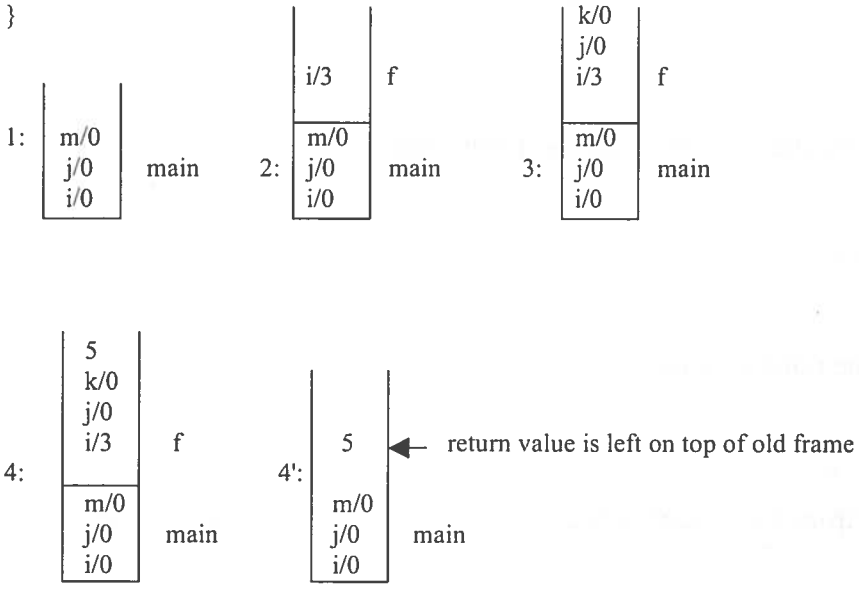
~~*Vector*~~ *runStack* -- the runtime stack; it's a *Vector* rather than a *Stack*

Arraylist

-- since we need access to ALL locations in the current

-- frame, not just the top location

Recall the simulation with the runtime stack presented earlier:



Initially the *framePointers* will have 0 for step 1 above; when we call the function *f* at step 2 *framePointers* will have 03 since 0 is the start of the main frame and 3 is the start of the frame for *f;* At step 4' we pop the *framePointers stack (3 is popped)*;

framePointers will only have 0 on the top

Class interpreter.RunTimeStack

```
java.lang.Object
   |
   +----interpreter.RunTimeStack
```

public class **RunTimeStack**

The RunTimeStack class maintains the stack of active frames; when we call a function we'll push a new frame on the stack; when we return from a function we'll pop the top frame

RunTimeStack

```
public RunTimeStack()
```

dump

```
public void dump()
```

Dump the RunTimeStack information for debugging

peek

```
public int peek()
```

Returns:

the top item on the runtime stack

pop

```
public int pop()
```

pop the top item from the runtime stack

Returns:

that item

push

```
public int push(int i)
```

Parameters:

i - push this item on the runtime stack

Returns:

the item just pushed

newFrameAt

```
public void newFrameAt(int offset)
```

start new frame

Parameters:

offset - indicates the number of slots down from the top of RunTimeStack for starting the new frame

popFrame

```
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, pop the top frame and then push the return value

store

```
public int store(int offset)
```

Used to store into variables

load

```
public int load(int offset)
```

Used to load variables onto the stack

push

```
public Integer push(Integer i)
```

Used to load literals onto the stack - e.g. for lit 5 we'll call push with 5

The Virtual Machine

RunTimeStack runStack

int pc	-- the program counter
Stack returnAddrs	-- push/pop when call/return functions
boolean isRunning	-- true while the VM is running
Program program	-- bytecode program

```
public void executeProgram() {
   pc = 0;
   runStack = new RunTimeStack();
   returnAddrs = new Stack();
   isRunning = true;
   while (isRunning) {
      ByteCode code = program.getCode(pc);
      code.execute(this);
      // runStack.dump();   // check that the operation is correct
      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, extendability and readability (note how easy it is to read and understand this code).

The *returnAddrs* stack stores the bytecode index (PC) that the virtual machine should execute when the current function exits. Each time a function is entered, the PC should be pushed on to 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.

Interpreter.java

```
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 {

        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();
        }
}
```

Coding Hints - Suggested Order to Write Code

1. Bytecode stubs - just include empty method bodies
2. CodeTable
3. Bytecode Loader
4. Program (code resolveAddress method)
5. RunTimeStack
6. Interpreter (given here)
7. Virtual Machine with dump method
8. Fill in Bytecode stubs

Additional Requirements

1. You should be able to execute your program by typing:
Java interpreter <bytecode file>
e.g.,
java interpreter factorial.x.cod

You will need to create a jar file named interpreter.jar to submit via email to the grader (see Appendix A).
You can assume that the bytecode programs you use for testing are generated correctly, and therefore should not contain any errors. You *should check* for stack overflow/underflow errors, or popping past a frame boundary.

2. You should provide as much documentation as is necessary to clearly explain your code. Short, OBVIOUS methods don't need comments, however you should comment every class to describe its function. Take into consideration that the first level of documentation is your code - it should be clear with appropriately named variables, etc. If you need to elaborate on algorithms that are not apparent, then include Javadoc comments for those sections of code.

It is also a good idea to follow the standard Java coding conventions that are recommended by Sun. Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

http://java.sun.com/docs/codeconv

3. DO NOT provide a method that returns components contained WITHIN the VM (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 (e.g., suppose I decide to change the name of the variable that holds my runtime stack - if your code had 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 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 or more methods in the runStack. It can do this by executing *VM.runStack.pop()*; however, this does break encapsulation. To avoid this, you'll need to have a corresponding set of methods within the VM that do nothing more than pass the call to the runStack. e.g., you would want to define a VM method

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

called by, e.g.,

int temp = VM.popRunStack();

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

It's easier to think in more general terms (i.e. plan for any number of args for a bytecode). Note that the Bytecode abstract class should not be aware of the peculiarities of any particular bytecode. That is, some bytecodes might have zero args (HALT), or one arg, etc. Consider providing an"init" method with each bytecode class. After constructing the bytecode (e.g. LoadCode) then you can call its"init" method and give it a vector of String args. Each bytecode object will interrogate the vector and extract relevant args for itself. The Bytecode class SHOULD NOT record the args for any bytecodes. Each concrete bytecode class will have instance variable(s) to record its args.

When you read a line from the bytecode file, you should parse each argument placing them into a vector. Each bytecode takes responsibility for extracting relevant information from the vector and storing it as private data.

5. The Bytecodes should be contained in a subpackage of the interpreter package (*interpreter.bytecode*)

6. 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 only print one number PER LINE. DO NOT output something like:

program output: 2

Only output the actual number on the line by itself.

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

DUMPING PROGRAM STATE

You should include 1 additional bytecode: DUMP
The formats are:
DUMP ON
and
DUMP OFF

DUMP ON is an interpreter command to turn on runtime dumping. This will set an interpreter switch that will cause dumping ***AFTER*** execution of each bytecode.

DUMP OFF will reset the switch to end dumping.
Note that the 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
READ
RETURN
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 1 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
CALL 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)
- **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]
LABEL f<<2>>
[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     m = 5
[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>>*

Notes:
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 the dumping information e.g.,
LIT 3
0003
3
WRITE
0003
The following dumping actions are taken for the indicated bytecodes, other bytecodes do not have special treatment when being dumped (see example above):

LIT 0 <id> int <id> note: for simplicity ALWAYS assume this *lit* is an *int* declaration
  e.g. LIT 0 j int j
LOAD c <id> <load <id>>
  e.g. LOAD 2 a <load a>
STORE c <id> <id> = <top-of-stack>
  e.g. if the stack has:
  0123
  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 from 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:
  0123
  and we execute CALL f<<3>> after executing ARGS 2 then we will dump
  CALL f<<3>> f(2,3)
    we strip out any brackets (<, >); the ARGS bytecode just seen tells us that we have a function with 2 args, which are on the top 2 levels of the stack - the first arg was pushed first, etc.

Dumping Implementation Notes

1. 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 static variable in the *ByteCode* class.

The *dump* method should be 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 contents of *RunTimeStack* itself.

2. 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 value as an int.

Be sure to use Javadoc comments within your code (as is done in the code found herein); you will not turn in the result of running Javadoc since the source programs will be graded.

Reading:

Core Java 2: Chapter 12