Interpreter (50 points)

See requirements in the course reader

Note that program output should look like the following - you should use brackets to group values in a frame and separate frames with a space:

```
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]
```

1. You should be able to execute your program by typing:
   Java -jar interpreter.jar <bytecode file>
   e.g.,
   java -jar interpreter.jar 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 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* 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.

SUBMISSION

Submit the zip file, as mentioned in Appendix A of the Reader

Q. Short overview of Interpreter

Interpreter project overview

So, to start, we have the *ByteCode* class. It's a parent class, abstract, with lots of children.
If you look on in the reader you'll see how many children it will have-- each bytecode inherits
from the *ByteCode* class. For example, you'll have a *LitCode*, *PopCode* and *HaltCode* classes, along with others.

The subclasses of *ByteCode* will have constructors, which according to my notes won't do very much.
Each *ByteCode* will also have an ***init***() method, into which you can pass arguments.
So a *StoreCode* would expect two parameters in its *init*() method-- the value being stored, and
the variable the value belongs to. Some classes can have a variable number of *init*() parameters, e.g. *LitCode*.

You can pick off the parameters for the *init*() method by using a ***StringTokenizer***.

There's also a ***ByteCodeLoader*** class. This reads in the codes from the file and creates
the appropriate *ByteCode* class. For example, if the *ByteCodeLoader* sees..

LIT 2

..in the file, it will build an instance of a *LitCode* and pass in the "2" as an argument
to the *LitCode init*() method.

The *ByteCodeLoader* knows what type of *ByteCode* to create thanks to the *CodeTable* class-- it
stores name/value pairs in a *HashMap*. *ByteCodeLoader* looks up the actual class name using
CodeTable, then uses ***Java reflection*** to create an actual class instance. The reader has more
info on how this should go.

Next up is the *Program* class. This class will hold the *ByteCode* instances that were created
by *ByteCodeLoader*. In the reader there's a line that says "***Program program = bcl.loadCodes()***",
so it's fairly obvious the *ByteCodeLoader* plays a big role here.

The *Program* class also walks through the bytecode program (requested by the *ByteCodeLoader*)
and resolves symbolic addresses-- so that "*GOTO addr*" has the symbolic address (*addr*) resolved
to a ***numeric bytecode address of the target*** to effectively yield e.g. "*GOTO 3*" (if *addr* resolves
to 3 in this case).

We also have the ***RunTimeStack*** class. This class appears to contain
both a *Vector* and a *Stack*.
The *Vector* holds the actual data we're storing, while the *Stack*, named
framePointers, segments the data into *frames*.

If you look at the example in the reader this may make more sense-- the
Vector (named ***runStack***)
holds the actual program values. The entries on "*framePointers*" stack
are indicies into *runStack*,
showing where the frames begin. So if you have data like this in
runStack:

0 1 4 2 9 8 7

And these values in *framePointers*:

0 3

Then you have ***two*** frames, one of which begins at index zero in
runStack, the other of which
starts at offset three in *runStack*. In other words, your two frames
have values of "0 1 4" and "2 9 8 7".

What's left....we have the ***Virtual Machine*** and the *Interpreter*, which
Dr. Levine has completed
for us. The *Virtual Machine* in a nutshell walks through the p*rogram*
and executes the
ByteCodes one by one. When it hits a *HALT ByteCode* it's time to bail.

Q. What classes do I have to create?

For this project you are only given one file to start with (*Interpreter.java*) so you must create several classes on your own. Based on the project description in the reader, you will need to include the following code:

Interpreter.java
This is the file that is given to you and can be downloaded in Interpreter.zip from the class website. You may need to modify this class.

ByteCodeLoader.java
This is the class thatwill be in charge of loading the code from the source file. It also has a method that loads bytecode objects into an instance of the program class. This is done with the help of a Hashtable that matches the bytecode to their equivalent classname.

Program.java
Holds the bytecode objects and has a method to resolve addresses for branch instructions.

RunTimeStack.java
Holds the runtime stack as well as the frame pointers. It has many methods to facilitate looking up values, storing them, and so on.

VirtualMachine.java
The virtual machine executes each byte code that is loaded into the program. It keeps track of the current position in the program; also, it holds a reference to the runtime stack.

ByteCode.java, all ByteCode subclasses
You should create a hierarchy of bytecode classes to handle each individual bytecode that is described in the reader. It is a good idea to think about abstraction, and design your code for extensibility. There should be one concrete bytecode class for each bytecode in the machine language.

CodeTable.java
Holds and initializes the Hashtable that *ByteCodeLoader* uses to create instances of the bytecode classes.

There may be other classes you create in order to implement the Interpreter, however your design must follow the structure provided in the reader (and as described above). For more detailed information about each required class you should consult the reader.

Q. What files should I use to test the Interpreter?

You should test with a variety of cases to make sure your project is working correctly. The provided files *factorial.x.cod* and *fib.x.cod* are good test cases to start with. You can also use *Compiler.java* to create your own test cases.

Q. How do you read input from the keyboard?

You can use the class *BufferedReader* to capture input from the user:

```
import java.io.BufferedReader
import java.io.InputStreamReader

...

try {
     BufferedReader in = new BufferedReader( new InputStreamReader(
System.in ) );
     String line = in.readLine();
} catch( java.io.IOException ex ) {
}
```

Q. How should the Dump bytecode be implemented?

Dumping is set to OFF by default. We need a method to set the dumping state to *ON* or *OFF* when we see *DumpCode*.

The check if dumping is *ON* so that we print out debugger info will be done in another method.

Q. Why are there two push methods in the RunTimeStack class?

The push method is overloaded to accept/return either an int value or an Integer. Having two methods prevents the user from having to convert the value before pushing it on to the stack in some cases.

Q. What should be the initial value of framePointers?

There should be an initial entry value in the *framePointers* stack of 0, representing the start of the *main* function. Each time a function is entered a new entry is pushed on to the *framePointers* stack. When a function exits, the top entry is removed from the *framePointers* stack.

Q. When a Return bytecode is executed, should the program counter jump to the address where the label is?

The RETURN label is just a comment. We shouldn't jump to the address where the label is. We should jump back to the return address which we saved before we executed the CALL <<*function*>>. The label in RETURN bytecode just tells us we have finished the current function.

Q. Is there a difference between RETURN and RETURN <<label>>?

Since we are saving return address on the stack it doesn't really matter if you have function name after RETURN bytecode or not. The program ignores the argument for the RETURN bytecode and just pops return address from return address stack. The only difference lies in how the RETURN bytecode is dumped.

Q. Does the Args bytecode set up a new frame?

Yes. ARGS should set up the new frame - e.g. if prior to ARGS 1 the stack contains 0003 then after executing ARGS 1 the stack should look like 000 3.

Q. Should the load and store methods in RunTimeStack implement the Load and Store bytecodes?

The load and store methods in the *RunTimeStack* implement the functionality of the LOAD and STORE bytecodes. But the bytecodes have a different purpose than the methods. A bytecode will be responsible for executing itself, and printing itself and whatever other common features you have added for your bytecodes. As an example, a STORE code's execute method may call another method to do the possessing which needs to take place (like *vm.store()*, rather than process the store within the *execute()*).

This way if you change how you implement the store method in the *RunTimeStack* (as long as your input and output params haven't changed), neither the STORE bytecode nor the *VirtualMachine* (which executes the bytecode) would need to be changed.

Q. How should the Store bytecode be dumped?

If you're using the *RunTimeStack* method *store(offset)* to implement the STORE bytecode functionality, then you automatically have access to the value just stored, from the return value of this method. You can then save this value for dumping purposes, by having a variable in the STORE bytecode that is set to the value of the variable during the execution method.

Q. What do the first few, automatically-generated lines of every .cod file do (e.g. Read and Write bytecodes without arguments)?

Every X program contains a section of code at the beginning of the file that defines intrinsic Read and Write functions. These methods are generated automatically by the compiler and are required if you want to write out any data from your program.

Q. How do I organize my testing for this project?

```
TESTING

These tests are designed for rapid prototyping and incremental feature
release. As a result, the tests are presented in order, and have
upwards
dependencies (simpleFrame uses the ByteCodes needed in arithmetic). In
keeping with the eXtreme Programming productivity practices, I've
developed
these tests ahead of my feature set and implement to test. Hopefully
this
will be helpful to others as well.
 -----
DEBUGGING
ByteCodes needed: DUMP, LIT, HALT

DUMP ON
LIT 3
LIT 4
DUMP OFF
LIT 5
HALT

Things to look for: The RunTimeStack should exit with three items, 3, 4
and 5.

-----
ARITHMETIC
ByteCodes needed: BOP
arithmetic00.x.cod

DUMP ON
LIT 3
LIT 4
BOP +
LIT 2
BOP -
LIT 6
BOP *
LIT 3
BOP /
HALT
```

Things to look for: The solution should be 10.

-----
arithmetic02.x.cod

```
DUMP ON
LIT 3
LIT 3
BOP ==
LIT 2
BOP <
LIT 2
BOP <=
LIT 2
BOP +
LIT 2
BOP >
LIT 2
BOP +
LIT 2
BOP >=
HALT
```

Things to look for: All of the tests should return true (1).

SIMPLE FRAME OFFSET
ByteCodes needed: LOAD, STORE

Simply loads another 3 on to the stack and replaces the first 3 with a 5.

simpleFrame.x.cod

```
DUMP ON
LIT 3
LOAD 0 i
LIT 5
STORE 0 i
HALT
```

Things to look for: The stack should exit with 2 items, 5 and 3.

BRANCHING, LABEL AND ADDRESS RESOLUTION
ByteCodes needed: GOTO, LABEL, FALSEBRANCH

A simple for loop, with 4 iterations.

labels.x.cod

```
DUMP ON
LIT 4
LABEL start
LIT 1
BOP -
LOAD 0 i
FALSEBRANCH exit
GOTO start
LABEL exit
HALT
```

Things to look for: Watch your vm.pc value carefully, specifically, either
adjust your target branch addresses by -1 to compensate for the pc
incrementation, or move the incrementation. There should be 4 iterations
of the loop, and the stack should exit with 1 item, 0.

FUNCTIONS
ByteCodes needed: CALL, ARGS, RETURN

Test a simple function, increment().

function.x.cod

```
DUMP ON
LIT 7
ARGS 1
CALL increment
HALT
LABEL increment
LIT 7
LIT 1
BOP +
RETURN increment
```

Things to look for: Watch your stack frames, specifically don't forget to
create a new stack frame and get all the arguments on to the new stack.
The stack should exit with 1 item, 8.

NESTED FUNCTIONS
ByteCodes needed: POP

Test a set of nested functions, multiply(), accumulate().

nested.x.cod

```
DUMP ON
LIT 0 doNothing
LIT 0 otherNothing
LIT 3
LIT 4
ARGS 2
CALL multiply
STORE 0 i
POP 1
HALT
LABEL multiply
LOAD 0 i
LOAD 1 j
BOP *
LIT 6
ARGS 2
CALL accumulate
RETURN multiply
LABEL accumulate
LOAD 0 x
LOAD 1 k
BOP +
RETURN accumulate
```

Things to look for: The stack should exit with one item, 18.
There should be no frame offsets (only the original stack should remain).

I/O
ByteCodes needed: READ, WRITE

io.x.cod

```
DUMP ON
READ
LIT 3
BOP *
WRITE
HALT
```

Things to look for: By now, the test infrastructure should have been folded
into your DUMP implementation. Input any valid integer, and see it multiplied
by 3.