CS 284: Special Assignment 1
Due: 1 April, 11:55pm
(No late submissions allowed!)

1 Assignment Policies

Collaboration Policy. Homework will be done individually: each student must hand in their own answers. It is acceptable for students to collaborate in understanding the material but not in solving the problems or programming. Use of the Internet is allowed, but should not include searching for existing solutions.

Under absolutely no circumstances code can be exchanged between students. Excerpts of code presented in class can be used.

Assignments from previous offerings of the course must not be re-used. Violations will be penalized appropriately.

2 Assignment

This assignment consists in implementing a *Stack-Based Machine (SBM)*. A SBM has a stack, a memory consisting of ten memory locations (referred to as m0, m1, . . . , m9) and a program that it is currently executing. We call these programs *mini-bytecode programs*. The instructions in the program may manipulate the stack, the memory or affect control flow. We begin this document with two examples.

2.1 Example 1

The following mini-bytecode program adds 5 and 3.4567, placing the result in memory location m0 and leaving the stack empty. The source code below is provided to you as part of the stub (file `eg1.pgm`).

```
push 5
push 3.4567
add
pop m0
exit
```

A line-by-line description would be as follows. It first pushes 5 onto the stack, then 3.4567 onto the stack, then it pops both numbers and pushes their sum onto the stack, it then stores the result in memory location 0 and, finally, it stops. Here is the state of the SBM after execution of the above program:

```
Pgm   : [push 5.0, push 3.4567, add, pop m0, exit]
Pc    : 6
Stack : []
Memory: [8.4567, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
------------------------------------------------
```

Each of the items in the state may be explained as follows:

- `Pgm` is the program that was executed.
- `Pc` is the *program counter* and tells us what instruction is to be executed next. Instruction 1 is the first instruction, 2 is the second one, and so on. Execution stops at an `exit` instruction.
- `Stack` is the stack. In this example, it is empty.
- `Memory` is the memory. In this examp0le, it has 8.4567 in memory location m0. All other memory locations have 0.0.

2.2 Example 2

The following example computes the factorial of 5. The result is left on top of the stack. The source code below is provided to you as part of the stub (file `eg2.pgm`). If you change the first instruction from `push 5` to `push 10`, then it computes the factorial of 10.

```
push 5
pop m0
push m0
push m0
label l2
dec
jmpz done
pop m0
push m0
mul
push m0
jmp l2
label done
pop m0
exit
```

Then the output of `main` is:

File	Instruction	Code	Comment
`Exit.java`	`exit`	0	Stop execution
`PushLiteral.java`	`push lit`	1	Eg. `push 3.4` pushes 3.4 onto the stack
`PushLocation.java`	`push mi`, $i \in 0..9$	2	Eg. `push m3` pushes the content of memory location m3 onto the stack
`Pop.java`	`pop mi`, $i \in 0..9$	3	Eg. `pop m0` removes the topmost element of the stack and stores it in memory location m0
`Add.java`	`add`	4	Pop two elements from the stack and push their sum.
`Sub.java`	`sub`	5	Pop two elements from the stack and push their difference.
`Mul.java`	`mul`	6	Pop two elements from the stack and push their product.
`Div.java`	`div`	7	Pop two elements from the stack and push their quotient.
`Label.java`	`label name`	8	Declare a label with name `name`
`Jmpz.java`	`jmpz labelName`	9	Jump to `labelName` if the topmost element in the stack is zero, otherwise continues to next instruction.
`Jmp.java`	`jmp labelName`	10	Jumps to `labelName`
`Dec.java`	`dec`	11	Decrement number at top of the stack without removing it.

Figure 1: Instruction Set

```
Pgm    : [push 5.0, pop m0, push m0, push m0, label l2, dec, jmpz done,
          pop m0, push m0, mul, push m0, jmp l2, label done, pop m0, exit]
Pc     : 16
Stack  : [120.0]
Memory: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
------------------------------------------------
```

2.3 The SBM System

The SBM system consists of the Java classes listed below. At then end of this document you will find a UML diagram that supplies more detail:

- `Instruction.java`: Abstract class. This class has a number of subclasses, one for each instruction. Each instruction has a code. The list of all instructions is given in Figure 1.
- `Runtime`: This class is the main one. Your code will go here. The rest of the document describes the code you have to write.

Note: We will not be feeding syntactically incorrect mini-bytecode programs to your solution. For example, you will never get an instruction such as `popp` or `pop m222` or `pop 33`.

3 Implementation

You are requested to implement the following:

1. Operation:

   ```
   private int jumpToLabel(String labelName)
   ```

   It should return the code line where the `labelName` is declared. If the `labelName` is not declared, then it should throw an `IllegalStateException` with the message `"runtime: label not found"`.

2. Operation:

   ```
   private boolean processInstruction(Instruction i)
   ```

   Given an instruction, this method accordingly updates the memory, stack and program counter. It should return false if the current instruction is an exit and true for all other instructions.

3. Operation:

   ```
   public void run()
   ```

   This method executes the program stored in the data field `pgm`. It should first initialize the SBM via the `initialize` method (provided in the stub) and then process each instruction on the program, one by one, until the `exit` instruction is reached. The latter will be indicated by having `processInstruction` return false (see description above).

4 Running the Examples

The examples described at the beginning of the document can be executed by changing the name of the file in the third line of the method below:

```java
public static void main(String[] args) {
        Runtime r = new Runtime();
        r.readFromFile("eg1.pgm");
        r.run();
        System.out.println(r);
    }
```

4.1 Example 3

Source in `eg3.pgm`.

```
pop m0
exit
```

The output of `main` is a `java.util.EmptyStackException` runtime exception because the stack is empty.

4.2 Example 4

Source in `eg4.pgm`.

```
push 5
pop m0
exit
```

Then the output of `main` is:

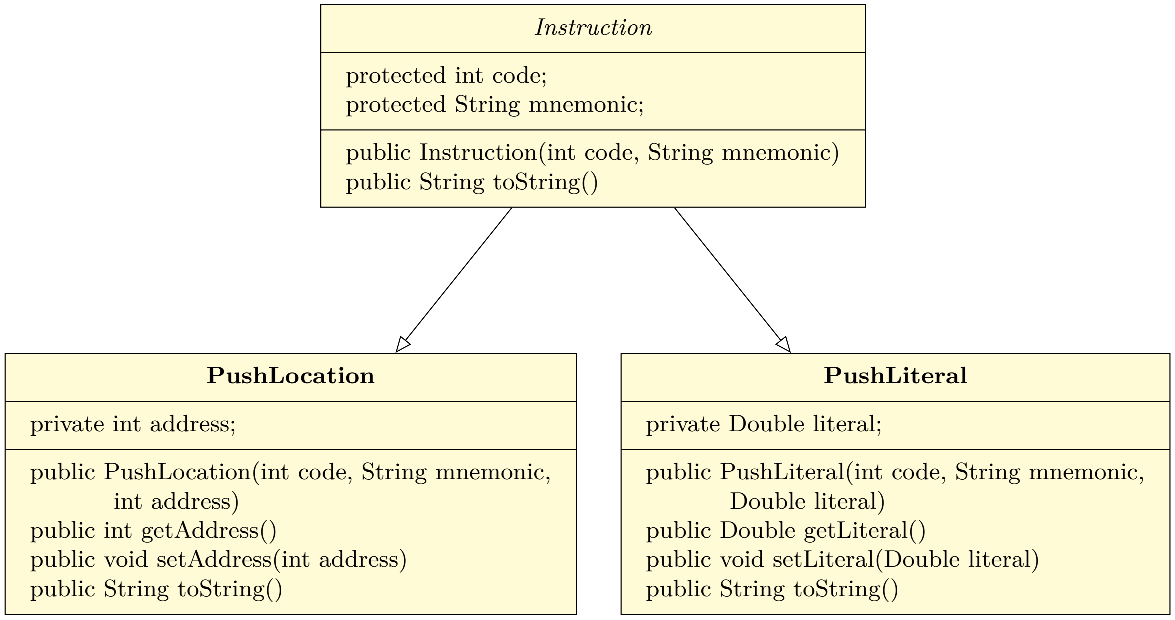
```
Pgm   : [push 5.0, pop m0, exit]
Pc    : 4
Stack : []
Memory: [5.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
-----------------------------------------------
```

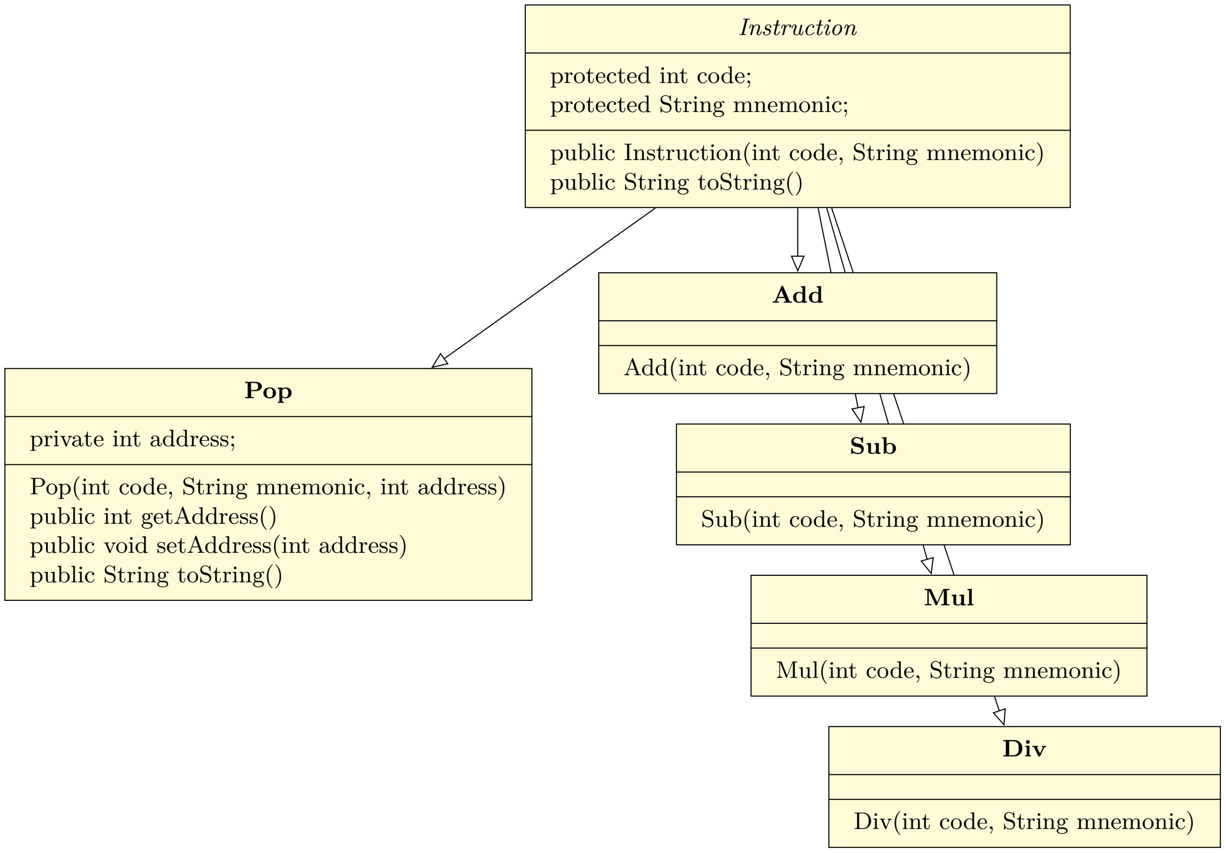
5 Submission instructions

Submit a single file named `sa1.zip` through Canvas that includes all files in stub (which must have been completed according the to instructions above) and a file `RuntimeTest.java` with your test cases. No report is required. Your grade will be determined as follows:

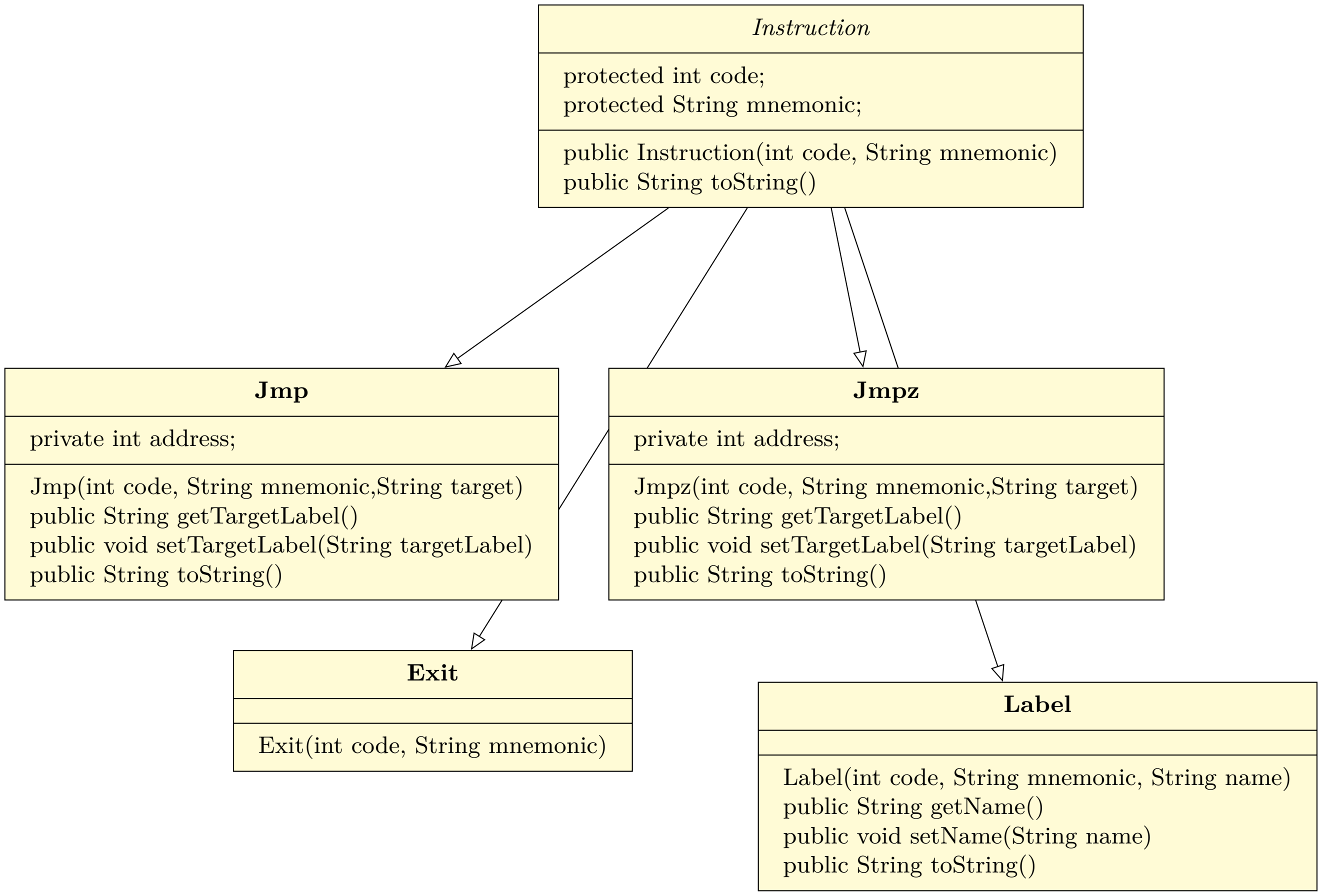
- You will get 0 if your code does not compile.
- The code must implement the following UML diagram precisely.
- We will try to feed erroneous and inconsistent inputs to all methods. All arguments should be checked.
- Partial credit may be given for style, comments and readability.

The UML diagram of the abstract class `Instruction` (italic typeface below indicates it is abstract) together with its subclasses (indicated with the arrow) `PushLiteral`, `PushLocation`, `Pop`, `Add`, `Sub`, `Mul`, `Div` are given below:

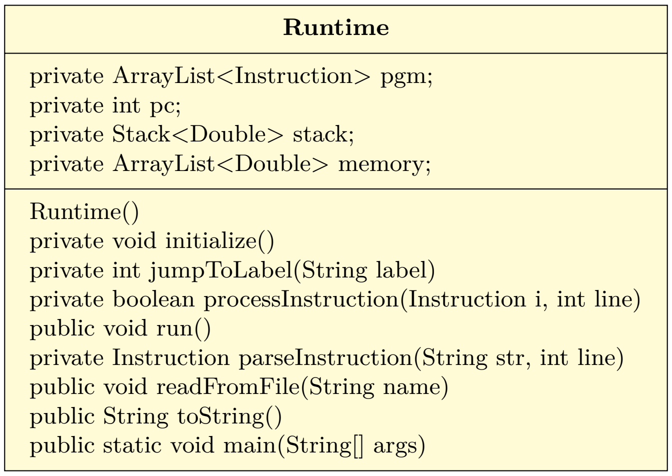

Instruction
protected int code;
protected String mnemonic;
public Instruction(int code, String mnemonic)
public String toString()
PushLocation
private int address;
public PushLocation(int code, String mnemonic, int address)
public int getAddress()
public void setAddress(int address)
public String toString()
PushLiteral
private Double literal;
public PushLiteral(int code, String mnemonic, Double literal)
public Double getLiteral()
public void setLiteral(Double literal)
public String toString()



Instruction
protected int code;
protected String mnemonic;
public Instruction(int code, String mnemonic)
public String toString()
Pop
private int address;
Pop(int code, String mnemonic, int address)
public int getAddress()
public void setAddress(int address)
public String toString()
Add
Add(int code, String mnemonic)
Sub
Sub(int code, String mnemonic)
Mul
Mul(int code, String mnemonic)
Div
Div(int code, String mnemonic)


Instruction
protected int code;
protected String mnemonic;
public Instruction(int code, String mnemonic)
public String toString()
Jmp
private int address;
Jmp(int code, String mnemonic,String target)
public String getTargetLabel()
public void setTargetLabel(String targetLabel)
public String toString()
Jmpz
private int address;
Jmpz(int code, String mnemonic,String target)
public String getTargetLabel()
public void setTargetLabel(String targetLabel)
public String toString()
Exit
Exit(int code, String mnemonic)
Label
Label(int code, String mnemonic, String name)
public String getName()
public void setName(String name)
public String toString()

The class `Runtime` should include the following operations:


Runtime
private ArrayList<Instruction> pgm;
private int pc;
private Stack<Double> stack;
private ArrayList<Double> memory;
Runtime()
private void initialize()
private int jumpToLabel(String label)
private boolean processInstruction(Instruction i, int line)
public void run()
private Instruction parseInstruction(String str, int line)
public void readFromFile(String name)
public String toString()
public static void main(String[] args)