PostScript,

It's just a stack

In this homework you are to write a mini-interpreter for PostScript, the stack-based language that controls printing in almost every modern printer. When you print using a Postscript printer, the print dialogue creates a program written in the Postscript language and sends it to the printer. The printer executes the program, the result being a set of commands that causes the print engine to create the document.

The goal of this homework is

1. to learn about the PostScript language,
2. to gain experience using stacks,
3. to complete an existing partial implementation, and
4. to evaluate arithmetic expressions using a stack-based language.

Specifically, in this homework you implement a subset of Postscript that can evaluate the arithmetic expressions needed to position graphic elements on a page.
(Your code will not be generating pictures, unfortunately...)

This homework is based on Bailey's laboratory: A Stack-Based Language, which you can find in the last pages of this document.

Design Document

- Read Bailey's Lab : A Stack-Based Language.
- Read this document.
- Do the **Explore** section below to understand the language mechanisms.
- Write the design document to complete the project that is described.
    - Do not worry about the code details of parsing the PostScript program, starter files are provided for this functionality. (The API for the starting code is available online...)
    - Draw diagrams of the stack in different states.
      Use sequence of diagrams to show the state of the stack as the computations for a program take place.
      Use different programs to cover all the operations listed below.
    - Develop a sketch of the `Interpreter` class.
      The purpose of this class is to push PostScript tokens onto and pop them out of a `Stack` object, to interpret the result of a PostScript program.
    - Include answers to Bailey's **Thought Questions** 1, 2, and 5.

Your design document is due Friday, November 7th by 5:00 P.M. Bring a physical copy to class and I will make a photocopy right after class or submit a small file on Moodle (as a last resort send me an email) by the 5 o'clock deadline.

Explore

Explore the real program (if you have a MAC see instructions on Moodle) or the Java `DEMO` version provided by following the setup steps.

Setup

Copy the starter files provided on Moodle.

- Create a `ps` folder and incorporate the following files
    1. `Interpreter.java`
    2. `Token.java`
    3. `Reader.java`
    4. `SymbolTable.java`
    5. `InterpreterDEMO.class`
    6. and don't forget to link the external jar `bailey.jar`
- Run the `main` method‡ of the `Interpreter` class and
- Before starting coding Read carefully through the code to make sure you understand the functionality of the classes provided.

Use the `DEMO` boolean to explore how Postscript works: entering commands on the console.

‡You can also run the commented main methods of the other classes to understand each class individually. To do so select the appropriate `.java` file.

Commands

With the real program or our `DEMO` version try the following commands.

- Enter the following program (press Enter for each new line)

```
1 2 add
pstack
```

- Run the "program" again but enter each element (i.e. token) separately and use `pstack` in between each token to check the stack state.
- Try to make-up your own programs so as to understand how the stack-based language computes arithmetic expressions. The basic operations to investigate are

```
pstack              quit
add                 sub
mul                 div
pop
exch                dup
eq                  not
lt                  gt
```

- Run each of the following programs. Push each token one-by-one and periodically check the stack so as to understand how the computations proceed.
    - `1 3 4 mul add pstack`

- `1 3 add 4 mul pstack`
- `10.1 dup mul pstack pop`
- `3 1 exch sub pstack pop`
- `1 2 eq pstack pop`
- `1 1 eq pstack pop`
- ```
  /pi 3.141592653 def
  /radius 1.6 def
  pi radius dup mul mul pstack pop
  ```

For the last program each of the two first lines is a single token: you must enter the full line atomically (that is at once).

If you have done this exploration you are ready to write your design document.

Implementation

Your task is to complete the class `Interpreter`.

- Print a prompt that is similar to the prompt you observed while exploring: it indicates the numbers of elements in a non-empty stack.
- You are required to implement the following operations

  ```
  pstack,  quit,  sub,  pop,  dup,  exch,
  ptable‡, def,   add,  mul,  div,
  eq,      ne,    lt,   gt,   not
  ```

  as well as

  ```
  procedure definitions, procedure calls & if
  ```

  which are described in Bailey's Thought Questions 3 & 4.
- It is recommended that you implement the operations in the order listed above.
- Your program must report meaningful errors when it encounters invalid input. Feel free to use `Assert.condition()` and `Assert.fail()` or print messages as in the `DEMO` to indicate invalid operations. Token already generates some of the errors that you need.

`‡ptable` is a nonstandard PostScript command that prints the symbol table (procedure definition included). It isn't part of the real program, but it is included in our `DEMO` version.

Notes

Remember to **not** include the following

```
import java.util.*;
```

because it will create conflicts between Java API classes and similarly named classes in Bailey's structure package.
Instead import classes such as `Random` and `Scanner` as follows

```
import java.util.Random;
import java.util.Scanner;
```

In addition consider the following notes and suggestions.

- You should only need to modify the `Interpreter` class, nothing else.
```

- Make use of the functionality of the classes you are given to start. Be careful not to spend time developing code that replicates functionality that is already present.
- A "debugging mode" for your program may prove useful to facilitate debugging and testing. The `Interpreter` starting code uses a command-line parameter to enter a debugging mode: the flag `-debug` enter this mode of execution.
- Develop your interpret method incrementally. Get the simple `push`, `pop`, and `pstack` operations working, then move on to the arithmetic operators, then to the definition and usage of symbols and, only when everything else is working, to procedure definitions and calls.
- Decompose your code into small, manageable helper methods as you go.
- Note that PostScript's handling of procedures is a little funny. When a procedure is entered directly from the command line using braces it is a *value*, but when a variable evaluates to a procedure, it is immediately executed.

Submit

Submit on Moodle your `Interpreter.java` and a `readme`.

The readme.txt file should include

- your student information at the top,
- sequences of PostScript **input** you used to test **all the functionality of your interpreter**, with **output** of the program that help to determine its correctness, and
- if you have not fully implemented the interpreter, list the operations that are working and indicate any commented code to be considered for partial credit.

Be sure

- to document your program,
- to include a description of each method with pre and postconditions where appropriate and
- to use comments and descriptive variable names to clarify sections of the code which may not be clear to a reader.

10.5 Laboratory: A Stack-Based Language

Objective. To implement a PostScript-based calculator.

Discussion. In this lab we will investigate a small portion of a stack-based language called PostScript. You will probably recognize that PostScript is a file format often used with printers. In fact, the file you send to your printer is a program that instructs your printer to draw the appropriate output. PostScript is stack-based: integral to the language is an operand stack. Each operation that is executed pops its operands from the stack and pushes on a result. There are other notable examples of stack-based languages, including `forth`, a language commonly used by astronomers to program telescopes. If you have an older Hewlett-Packard calculator, it likely uses a stack-based input mechanism to perform calculations.

We will implement a few of the math operators available in PostScript.

To see how PostScript works, you can run a PostScript simulator. (A good simulator for PostScript is the freely available `ghostscript` utility. It is available from `www.gnu.org`.) If you have a simulator handy, you might try the following example inputs. (To exit a PostScript simulator, type `quit`.)

1. The following program computes $1 + 1$:

   ```
   1 1 add pstack
   ```

   Every item you type in is a *token*. Tokens include numbers, booleans, or symbols. Here, we've typed in two numeric tokens, followed by two symbolic tokens. Each number is pushed on the internal stack of operands. When the `add` token is encountered, it causes PostScript to pop off two values and add them together. The result is pushed back on the stack. (Other mathematical operations include `sub`, `mul`, and `div`.) The `pstack` command causes the entire stack to be printed to the console.

2. Provided the stack contains at least one value, the `pop` operator can be used to remove it. Thus, the following computes 2 and prints nothing:

   ```
   1 1 add pop pstack
   ```

3. The following "program" computes $1 + 3 * 4$:

   ```
   1 3 4 mul add pstack
   ```

   The result computed here, 13, is different than what is computed by the following program:

   ```
   1 3 add 4 mul pstack
   ```

   In the latter case the addition is performed first, computing 16.

4. Some operations simply move values about. You can duplicate values—the following squares the number 10.1:

```
10.1 dup mul pstack pop
```

   The `exch` operator to exchange two values, computing $1 - 3$:

```
3 1 exch sub pstack pop
```

5. Comparison operations compute logical values:

```
1 2 eq pstack pop
```

   tests for equality of 1 and 2, and leaves `false` on the stack. The program

```
1 1 eq pstack pop
```

   yields a value of `true`.

6. Symbols are defined using the `def` operation. To define a symbolic value we specify a "quoted" symbol (preceded by a slash) and the value, all followed by the operator `def`:

```
/pi 3.141592653 def
```

   Once we define a symbol, we can use it in computations:

```
/radius 1.6 def
pi radius dup mul mul pstack pop
```

   computes and prints the area of a circle with radius 1.6. After the pop, the stack is empty.

Procedure. Write a program that simulates the behavior of this small subset of PostScript. To help you accomplish this, we've created three classes that you will find useful:

`Token`

`Reader`

- `Token`. An immutable (constant) object that contains a double, boolean, or symbol. Different constructors allow you to construct different `Token` values. The class also provides methods to determine the type and value of a token.
- `Reader`. A class that allows you to read `Tokens` from an input stream. The typical use of a reader is as follows:

```
Reader r = new Reader();
Token t;
while (r.hasNext())
{
    t = (Token)r.next();
    if (t.isSymbol() && // only if symbol:
        t.getSymbol().equals("quit")) break;
    // process token
}
```

This is actually our first use of an `Iterator`. It always returns an `Object` of type `Token`.

- `SymbolTable`. An object that allows you to keep track of `String`–`Token` associations. Here is an example of how to save and recall the value of π:

`SymbolTable`

```
SymbolTable table = new SymbolTable();
// sometime later:
table.add("pi",new Token(3.141592653));
// sometime even later:
if (table.contains("pi"))
{
    Token token = table.get("pi");
    System.out.println(token.getNumber());
}
```

You should familiarize yourself with these classes before you launch into writing your interpreter.

To complete your project, you should implement the PostScript commands `pstack`, `add`, `sub`, `mul`, `div`, `dup`, `exch`, `eq`, `ne`, `def`, `pop`, `quit`. Also implement the nonstandard PostScript command `ptable` that prints the symbol table.

Thought Questions. Consider the following questions as you complete the lab:

1. If we are performing an `eq` operation, is it necessary to assume that the values on the top of the stack are, say, numbers?

2. The `pstack` operation should print the contents of the operand stack without destroying it. What is the most elegant way of doing this? (There are many choices.)

3. PostScript also has a notion of a *procedure*. A procedure is a series of `Tokens` surrounded by braces (e.g., `{ 2 add }`). The `Token` class reads procedures and stores the procedure's `Tokens` in a `List`. The `Reader` class has a constructor that takes a `List` as a parameter and returns a `Reader` that iteratively returns `Tokens` from its list. Can you augment your PostScript interpreter to handle the definition of functions like `area`, below?

```
/pi 3.141592653 def
/area { dup mul pi mul } def
1.6 area
9 area pstack
quit
```

Such a PostScript program defines a new procedure called `area` that computes πr^2 where r is the value found on the top of the stack when the procedure is called. The result of running this code would be

```
254.469004893
8.042477191680002
```

4. How might you implement the `if` operator? The `if` operator takes a boolean and a token (usually a procedure) and executes the token if the boolean is true. This would allow the definition of the absolute value function (given a less than operator, `lt`):

```
/abs { dup 0 lt { -1 mul } if } def
3 abs
-3 abs
eq pstack
```

   The result is `true`.

5. What does the following do?

```
/count { dup 1 ne { dup 1 sub count } if } def
10 count pstack
```

Notes: