An RPN Calculator

Problem description

Reverse Polish Notation (RPN) is a way of specifying arithmetic expressions that never requires parentheses or a conventional set of rules about order of operations. It was, and still is, used in some calculators most commonly those produced by Hewlett-Packard, and is available as an option in the OS X Calculator app. In this assignment you'll implement a basic calculator that takes expressions written in RPN and evaluates them, as well as providing a few bits of extra functionality.

The fundamental idea of RPN is that the input is a sequence of tokens, where each token is either a number or some sort of operator. Numbers are kept in a stack, and if the next token of the input is a number it is simply pushed onto the stack. If the token is an operator, then that operator is applied to numbers that are popped off the stack, and the resulting value is then pushed back on to the stack.

A simple example

Suppose that the expression:

```
3 4 * 5 2 + -
```

is to be evaluated. The evaluation proceeds as follows (in the **`Stack`** column, the top of the stack is on the right).

Input	Stack	Next operation
`3 4 * 5 2 + -`		Push `3`
`4 * 5 2 + -`	`3`	Push `4`
`* 5 2 + -`	`3 4`	Evaluate `3*4` and push the result
`5 2 + -`	`12`	Push `5`
`2 + -`	`12 5`	Push `2`
`+ -`	`12 5 2`	Evaluate `5+2` and push the result
`-`	`12 7`	Evaluate `12-7` and push the result
	`5`	

Detailed specification

Provide a class **`RPNApp`** which reads lines of input from **`System.in`** and processes each line as an expression to be evaluated in RPN form.

Basic operators

Each line is to be treated as a sequence of white-space separated tokens. Each token is one of:

Number A string that can be recognised as a **`long`** value.

Binary operator One of: **`+`**, **`*`**, **`-`**, **`/`**,or **`%`**.

The result of processing a line is written to **`System.out`**. This result is either a string representation of the stack[1] or an error message of the form **`Error:`** followed by a description of the error. The error types and descriptions are as follows:

Error type	Description
Unrecognised token	**`bad token '<t>'`** (where **`<t>`** is the token that caused the error)
Too few operands	**`too few operands`**
Division by 0	**`division by 0`**
Remainder by 0	**`remainder by 0`**

Examples

Input	Output
`3 4 * 5 2 + -`	`[5]`
`1 2 3 4 5 6`	`[1, 2, 3, 4, 5, 6]`
`1 fred`	`Error: bad token 'fred'`
`1 2 + +`	`Error: too few operands`
`1 2 2 - /`	`Error: division by 0`

After an error, processing should continue with the next line of the input.

Repeat operators

For each of the binary operators, add a repeating form indicated by "!" (i.e., **`+!`**, **`*!`**, **`-!`**, **`/!`**, **`%!`**). The interpretation of this operator is that the binary operator should be repeatedly applied until the stack contains exactly one item.

Input	Output
`1 2 3 4 +!`	`[10]`
`1 2 3 4 *!`	`[24]`
`1 2 3 4 -!`	`[-2]`

If the stack is empty, then it is a **`too few operands`** error to apply one of these, but if the stack contains only one item they are allowed (but have no effect).

[1] There is no requirement that the stack contain only one value at this point, so, e.g., **`1 2 3 +`** is valid input, producing output **`[1, 5]`**. The **`toString`** method of the **`java.util.Stack`** class should be used to format the output.

Special operators

Add the following extensions (new types of allowed tokens) to the **`RPNApp`**:

- Add an operator, **`d`** which simply duplicates the top item of the stack, and an operator **`o`** which outputs the top item of the stack (without removing it) followed by a single space, but not a newline.
- Add a new binary operator, **`c`**, which stands for "copy". If the top element of the stack is y and the second from the top is x, then the result of applying **`c`** should be to remove both from the stack and then push y copies of x onto the stack, like this:

Input	Output
`1 3 c`	`[1, 1, 1]`
`2 4 c *!`	`[16]`

  The **`c`** operation can trigger a **`too few operands`** error if the stack contains fewer than two items, and also a new type of error **`negative copy`**, if the top number on the stack is less than zero (copying 0 times *is* allowed).

- A new operator **`r`** which stands for "roll" or "rotate". If the top element of the stack is k and the k preceding lower elements (from bottom to top) are x_1, x_2, ..., x_k, then the result of applying **`r`** should be to rotate the top element x_k down $k-1$ positions in the stack, like this:

Input	Output
`1 2 3 4 4 r`	`[4, 1, 2, 3]`
`1 2 3 4 3 r`	`[1, 4, 2, 3]`
`1 2 3 4 2 r`	`[1, 2, 4, 3]`

  The **`r`** operation can trigger a **`too few operands`** error if the stack contains too few items for the requested roll, and also a new type of error **`negative roll`**, if the top number on the stack is less than zero (rolling 0 items *is* allowed, but, like rolling 1 item, has no effect).

Adding parentheses

Since parentheses are not required when using RPN let's add a different operation which makes use of them.

The operation is that when a '**`(`**' is encountered, all commands up to the matching '**`)`**' are repeated k times where k is the number on top of the stack when the '**`(`**' is encountered (0 times if $k <= 0$).

Input	Output
`1 3 ( 2 * )`	`[8]`
`1 1 1 6 ( + d 3 r )`	`[21, 13, 21]`
`1 1 10 ( d 3 r + o )`	`2 3 5 8 13 21 34 55 89 144 [89, 144]`

A new type of error, **`unmatched parentheses`**, should be triggered if the parentheses are not equally matched. In the basic form only one pair of parentheses will be present in the input. For an extra challenge see if you can handle multiple pairs of parentheses which may or may not be nested.

Input	Output
`1 3 ( d 2 ( 1 + ) * )`	`[255]`

Group work and Submission

For this assignment we require you to work in teams of three people. You may select your own group and inform us of your choice by 4pm Wednesday April 14^{th}.

Send an email to iain.hewson@otago.ac.nz letting us know the name and University user code of each student in your group. Any students who don't select their own group will be assigned to one by us and informed via email. Groups will be assigned in a pseudo-random manner. Students will be teamed up others who have completed a similar amount of internal assessment.

By week 8 of the semester, you will be able to check your assignment against some basic tests by running the command:

```
asgn-check
```

Your assignment code should be in the package **`week10`**. You will be able to submit your assignment using the command:

```
asgn-submit
```

Any assignments submitted after the due date and time will lose marks at a rate of 10% per day late.

All students in each group should submit their work (which should be identical) using **`asgn-submit`**. When submitting your assignment you will be asked to rate the contribution of you and your team mates.

Do *not* work with any other student who is not in your group, or ask the demonstrators for help with the assignment. However, feel free to discuss any questions you may have with Iain Hewson.

All submissions will be checked for similarity.

Marking

This assignment is worth 14% of your final mark for Cosc 241. It is possible to get full marks. In order to do this you must write correct, well commented code which meets

the specifications.

Marks are awarded for your program based on both implementation (9%) and style (5%). It should be noted however that it is very bad to style to have an implementation that doesn't work. Partial marks will be given for a partially complete implementation.

In order to maximise your marks please take note of the following points:

- Your code should compile without errors or warnings.
- Your program should use good Java layout (use the **`checkstyle`** tool to check for layout problems).
- Make sure each file is clearly commented.
- Most of your comments should be in your method headers. A method header should include:
  - A description of what the method does.
  - A description of all the parameters passed to it.
  - A description of the return value if there is one.
  - Any special notes.

Part of this assignment involves you clarifying exactly what your program is required to do. Don't make assumptions, only to find out that they were incorrect when your assignment gets marked.

If you have any questions about this assignment, or the way it will be assessed, please see Iain or send an email to iain.hewson@otago.ac.nz.