Imperial College of Science, Technology and Medicine	University of London
Computer Science (CS) / Software Engineering (SE)	BEng and MEng Examinations Part I
Department of Computing	Integrated Laboratory Course

Laboratory work is a continuously assessed part of the examinations and is a required part of the degree assessment.

Laboratory work must be handed in for marking by the due date.

Late submissions may not be marked.

Exercise: 16	Working: Individual
Title: Calculator Simulation	
Issue date: 1st March 2004	Due date: 8th March 2004
System: Linux	Language: Java

Aim

- To gain experience in the use of abstract data types and the manipulation of stacks.
- To gain experience in *exception handling* using exception classes.

The Problem

- Write a Java main program file **Calculator.java** which simulates a cheap four-function calculator and a support class **TokenStack.java** (using a static implementation) which allows it to manipulate the user input as stacks of tokens (as explained below).
- Write two classes for handling any exceptions in processing the stack or in evaluating the arithmetical expression (eg unrecognised operator) etc. Note: although the pre-condition of the exercise (for the purpose of testing is that any expression provided will be syntactically correct you should still include code to trap some simple exceptions in your **Calculator.java** program. The two classes should be called **StackException.java** and **CalculatorException.java**.
- Your program should prompt for an **expression**, evaluate it, and print the result. An expression (**without** brackets) is defined by:

  $Expression = PartialExpression, \text{“=”};$
  $PartialExpression = Term, \{Operator, Term\};$
  $Term = (non-negative)integer;$
  $Operator = \text{“+”}|\text{“−”}|\text{“*”}|\text{“/”};$

  An example would be “20-375/10*2=”.

- Your program should respect the traditional **operator precedence rules**. That is, * and / bind more tightly than + and - . Adjacent operators of the same binding priority evaluate **from left to right**. So the above expression evaluates (in integer arithmetic) to 20-74, or -54.
- Once you have written your program so that it handles expressions without brackets, you should extend it to handle expressions **with** brackets. These have the same syntax as above, except that the syntax for **Term** is modified to

  $$Term = (non - negative)integer | \text{“(”}, PartialExpression, \text{“)”};$$

- You may assume as a precondition of your whole program that the input expression (whether without or with brackets) is always valid.

Submit by Monday 8th March 2004

What to Do

- Copy the two classes **Token.java** and **TokenReader.java** into your working directory by typing **exercise 16**.
- As these two classes are in the *package* **tokens**, you should create a subdirectory **tokens** and move these two files into the sub-directory.
- Before writing any code, **study the ADT Token.java** and the class **TokenReader** which allow the main program to read in a sequence of *tokens* representing the input expected by the program.
- A Token is a single “piece” of the user input. It can contain either an integer or a character. Integer-valued Tokens from the standard input will always be non-negative, but intermediate results or the final answer might be any integer value. Character-valued Tokens will always be either an arithmetic operator, the terminating equals sign, or (for expressions with brackets) the character ‘(’ or ‘)’. A *Token* is defined, along with encapsulated access methods in the class **Token**.
- The method **readToken** provided by the **TokenReader** class gets *one* syntactically valid **Token** of either kind from the expression on the standard input. As there is only one **TokenReader** object the methods in the class are declared as static. The means that you do not need to create a **TokenReader** object in your main program but you can simply call **readToken** with a statement of the form:

  ```
  Token t;
  t = TokenReader.readToken();
  ```

- Make sure you understand the order of evaluation in an expression.
- Implement **Calculator.java** (the main program, *i.e.* not a member of the **tokens** package) and **TokenStack.java** (which *should* be a member of the **tokens** package.)

- **TokenStack.java** should be an array based, static implementation of a stack of Tokens along with the appropriate access methods, **TokenStack()**, **isEmpty()**,**top()**, **pop()**, **push(Token T)**.
- As the stack is only used by the Calculator you will not need to write an interface for **TokenStack.java**.
- **Calculator.java** should read in an expression, evaluate it and print out the result. It is recommended you maintain two token stacks, a **number stack** and an **operator stack**, containing the operands and operators respectively. You do not need to worry about validation of expressions.
- Your program should print a prompt string before reading in the input and an announcement string before writing out the answer. To help with autotesting, the answer should be the **last** thing output.
- We recommend you get your program working for expressions without brackets first, and then extend it to handle expressions with brackets.
- To handle exceptions you will need to create a class to catch and pass on exceptions. In the case of **StackException.java** it should be a sub-class of **java.lang.RuntimeException**. Details about exception handling can be found in the section of the course notes **The ADT Stack**. In a similar way you should create a class **CalculatorException.java** which should be a sub-class of **java.lang.ArithmeticException**. Both classes should be included in the **tokens** package. Thus there should be **only one** of the six classes *not* in the **tokens** package, namely the **Calculator.java** main program file, in the current directory outside the **tokens** sub-directory.
- The **Token** class uses *assertions* and so you should compile and run your program with assertions enabled, using the commands

  ```
  javac -source 1.4 Calculator.java tokens/*.java
  ```

  and

  ```
  java -ea Calculator
  ```

  respectively.

Unassessed

- How would you develop your program to *validate* expressions for syntax errors?

Submission

- Copy your main program file **Calculator.java** into your sub-directory **tokens** and then *cd* into the sub-directory **tokens** and submit **Calculator.java**, **TokenStack.java**, **StackException.java** and **CalculatorException.java** by typing **submit 16** at your Linux prompt.

Assessment

Without brackets	3
Handling of brackets	2
Design, style, readability	5
Total	10