Lexical Analysis (through Regular Expressions)

In this exercise, you will:

- Break a stream of characters into a sequence of *tokens*.
- Assign a *tag* to each token, denoting its kind (identifier, numeric, operator, etc.).
- Filter out *padding* tokens (whitespaces and comments).
- Display your results nicely in a table.

You will implement a lexical analyzer using the off-the-shelf lexical analyzer generator *JFlex*, in Java. Consult this manual at all time, read the examples presented there and go over the definition of the syntax. In particular, use the directives `%line`, `%column`, `%class`, and `%token`. Be sure to make your code readable using macros instead of long expressions.

The Token Class The purpose of the Lexer is to generate a list of tokens. Define a class called `Token` and provide a method (in some other class) that accepts a `String` argument and produces a collection of tokens (`Collection<Token>`). The `Token` class must store at least the following information:

- `line, column`, the line number and column number where the token occurs in the input file (1-based, that is, first line is 1 and first column is 1);
- `tag`, an identifier for the token (use a `String`);
- `value`, a `String` holding the specific character sequence that was matched for the token (including any delimiters such as the double quotes of string literals).

Tag Values As token tags, the following words should be used:

Token type	E.g.	Tag
Regular identifier	ted	`ID`
Class identifier	Troll	`CLASS_ID`
Integer literal	81	`INTEGER`
String literal	“wisdom”	`STRING`

For other tokens — reserved words, operators, and delimiters — the token's text will also serve as its tag (in which cases the member `tag` will hold the same string as `value`). Refer to the IC Specification for a complete listing of tokens.

Input Format The input is an arbitrary sequence of ASCII characters, stored in a regular file. Note that the end-of-line control sequence is represented by a special ASCII code point, which is sometimes 10 (`\n`, Unix and Mac), sometimes 13 (`\r`, older Macs and Commodore), and sometimes both (nicknamed “CRLF”, usually Windows).

Output Format A table with 4 columns containing the token's text, tag, line and column locations, in respective order. The first line should be the table header, followed by one line per token. Whitespace and comments should be skipped by the analyzer, so in particular they would not be printed.

The table should be printed to the program's standard output stream (`java.lang.System.out`). Any errors are directed to the standard error stream (`java.lang.System.err`).

Here is a nice example (only the first few lines of the output are shown):

```
ted = 42;  /* ultimate answer */
todd = 16 * ted + 9;
```

⇒

```
token    tag        line : column
ted      ID          1      1
=        =           1      5
42       INTEGER     1      7
;        ;           1      9
todd     ID          2      1
=        =           2      6
16       INTEGER     2      8
*        *           2     11
  ⋮
```

To print a token with its info, you need to call the *PrintToken* procedure below.

```
public static void PrintToken(String token, String tag, int line, int column) {
    System.out.println(token+"\t"+tag+"\t"+line+":"+column);
}
```

Command-line Interface You should provide a jar file with pre-configured `Main` in the default package. Invocation will be as simple as:

`java -jar` *bin⟨input-filename⟩*

Important!!

Submitting JFlex files Your source code will be compiled by an automated script. To make sure your lexical definition file is processed correctly, give it the extension `.lex`. You can assume *JFlex* will be executed as follows:

`jflex --nobak` ⟨*your-filename*⟩`.lex`