Goals:

- Implement the MyPL lexical analyzer;
- Practice working with unit tests.

Instructions:

1. Use the GitHub Classroom link (posted in Piazza) to copy the starter code into your own repository. Clone the repository in the directory where you will be working on the assignment.
2. Complete the `nextToken()` function in `Lexer.java`.
3. Ensure your code passes the unit tests provided in `LexerTests.java`. (Note you will want to do steps 2 and 3 iteratively.)
4. Ensure your lexer implementation correctly handles the example files within the `examples` subdirectory. Note that there are two MyPL (.mypl) files and two corresponding output files (.out). Your lexer output should identically match the output files.
5. Create additional unit tests as specified in the TODO comment at the end of `LexerTests.java`.
6. Create a short write up as a **pdf file** named `hw1-writeup.pdf`. For this assignment, your write up should provide a short description of the unit tests you created and any challenges and/or issues you faced in finishing the assignment and how you addressed them. The description of the tests can be short, but should state why you designed the tests the way you did (i.e., justify why the test is non-trivial / interesting, and what it is actually testing).
7. Submit your program by ensuring all of your code and writeup is pushed to your GitHub repo. You can verify that your work has been submitted via the GitHub page for your repo.

Additional Requirements: Note that in addition to items listed below, details will also be discussed in class and in lecture notes.

1. It is fine to implement the `nextToken()` function without breaking it into separate helper functions (i.e., you can have it be one large function). However, if you would like to "modularize" it, you are welcome to. If you do break it out into helper functions, you must explain how you did this in your writeup. Also be sure to comment any helper functions you create.
2. You must implement your `nextToken()` function by reading one character at a time via the `read()` and `peek()` helper functions provided in the `Lexer` class. In addition, to report errors, you must use the `error()` helper function provided by the Lexer class. The `isEOF()` and `isEOL()` functions are also provided to help you check for an EOF (end of file) character and end of line, respectively.
3. Java provides some useful helper functions for checking for specific types of characters. In particular, I used the `Character.isWhitespace()` (check for whitespace, which includes newlines, tabs, and so on), `Character.isDigit()`, `Character.isLetter()`, and `Character.isLetterOrDigit()` functions in my implementation.

4. The full set of token types for MyPL are provided in the `TokenType.java` file. Note that your `nextToken()` function is creating and returning `Token` objects with these listed types.

5. Note that `nextToken()` implements an iterator, i.e., stream-based, model. This means that one call to `nextToken()` returns *only* the next token in the input. The `Lexer` object maintains state, including where it is in the current input (to be able to return the next token in the input, and so on).

6. The `Lexer` class maintains `line` and `column` member variables. These variables are to keep track of the current line and column for building tokens. Your `nextToken()` function will need to update these member variables and also use them to build up new token objects.

7. Each token should have a non-empty lexeme. For tokens with "unimportant" lexemes, you can just use their corresponding symbol. For example, the lexeme for + should be `"+"` and the lexeme for `int` should be `"int"`.

8. The `hw1_hello.out` and `hw1_tokens.out` files within the `examples` subdirectory give examples of what your results for running `./mypl -m LEX` on the files should be. Your program must output the exact same information as what is in these files to be considered correct. Note to check your output against those in the given output (.out) files, you might consider using the UNIX `diff` command-line tool. Many editors also support some type of `diff` command as well.

9. A non-comprehensive set of unit tests are provided in the `LexerTests.java` file. To run these tests, simply use the command `mvn test`. (Note that many IDEs, including VS Code, provide integrated support for JUnit, but using this is not requried.) Your implementation will need to pass all of the unit tests from `LexerTests.java` to be considered correct.

10. Note that an additional (simple) `TokenTests.java` file is also provided. These tests are also run when you issue the command `mvn test`. To run just the lexer tests, use:

    ```
    mvn test -Dtest=LexerTests
    ```

    To run a specific test in `LexerTests`, use:

    ```
    mvn test -Dtest=LexerTests#TestName
    ```

    For example, to run the first test use:

    ```
    mvn test -Dtest=LexerTests#emptyInput
    ```

11. Make sure you add your name to the top of the `Lexer.java` file.

Hints and Tips:

1. The basic layout for `nextToken()` that I used in my implementation is, in order: (1) read all whitespace (checking for EOF); (2) check for EOF; (3) check for single character tokens (e.g., arithmetic operators, punctuation, etc.); (4) check for the trickier symbols that can involve or require two characters (e.g., < vs <=, `!=`, and so on); (5) check for comments (note that we will use COMMENT tokens initially, then ignore them in the parser later); (6) check for string values; (7) check for integer and double values; (8) check for reserved words; and then (9) identifiers. Again, it is much easier to do this incrementally as opposed to all at once and then try to debug.

2. Note that the unit tests provided are not guaranteed to be comprehensive. Just because your program passes the unit tests *does not mean your code is correct!* Also, we may grade your code with additional unit tests than those provided. Feel free to add additional unit tests (more than those asked for) to `LexerTests.java`.

Homework Submission and Grading. Your homework will be graded using the files you have pushed to your GitHub repository. Thus, you must ensure that all of the files needed to compile and run your code have been successfully pushed to your GitHub repo for the assignment. Note that this also includes your homework writeup. This homework assignment is worth a total of 30 points. The points will be allocated according to the following.

1. **Correct and Complete (20 points).** Your homework will be evaluated using a variety of different tests (for most assignments, via unit tests as well as test runs using specific input files). Each failed test will result in a loss of 4 points. If 5 or more tests fail, but some tests pass, 4 points (out of the 20) will be awarded as partial credit. Note that all 20 points may be deducted if your code does not run, large portions of work are missing or incomplete (e.g., stubbed out), and/or the specified techniques, design, or instructions were not followed. Because assignments build on each other, in most cases you will need all tests to pass before moving to the next assignment.

2. **Evidence and Quality of Testing (5 points).** For each assignment, you must provide additional tests that you used to ensure your program works correctly. Note that for most assignments, a specific set of tests will be requested. A score of 0 is given if no additional tests are provided, 1–4 points if the tests are only partially completed (e.g., missing tests) or the tests provided are of low quality, and 5 if the minimum number of tests are provided and are of sufficient quality.

3. **Clean Code (2 points).** In this class, "clean code" refers to consistent and proper code formatting (indentation, white space, new lines), use of appropriate comments throughout the code, no debugging output, no commented out code, meaningful variable names and helper functions (if allowed), and overall well-organized, efficient, and straightforward code that uses standard coding techniques. A score of 0 is given if there are major issues, 1 if there are minor issues, and 2 if the "cleanliness" of the code submitted is satisfactory for the assignment.

4. **Writeup (3 points).** Each assignment will require you to provide a small writeup addressing challenges you faced and how you addressed them as well as an explanation of the tests you developed. Additional items may also be requested depending on the assignment. Homework writeups do not need to be long, and instead, should be clear and concise. A score of 0 is given if no writeup is provided, 1 if parts are missing, and 2 if the writeup is satisfactory.