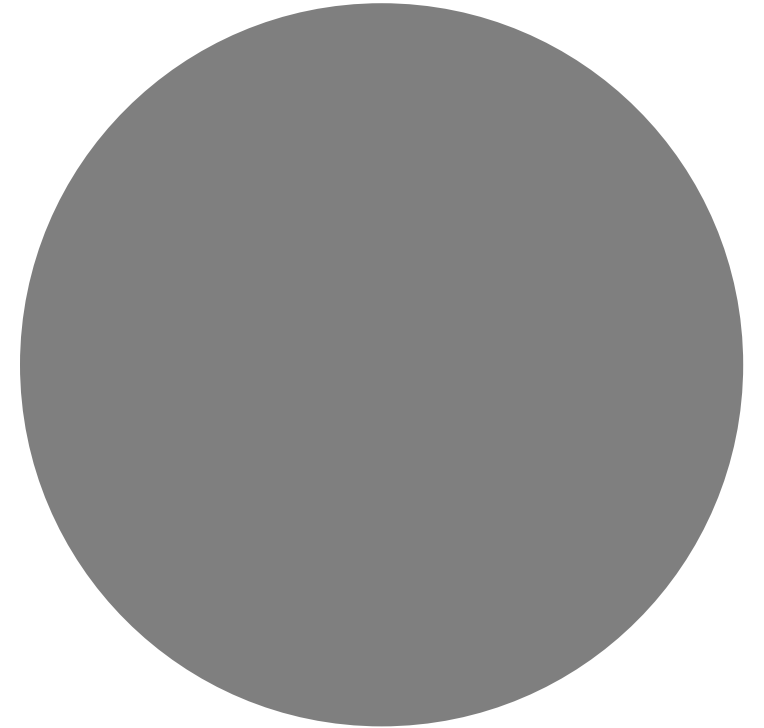


CSC 416/565: DESIGN AND CONSTRUCTION OF COMPILERS

West Chester University
Dr. Richard Burns
Fall 2023





The Scanner / Lexer

Programming Assignment #2 Introduction

Role of These Slides

- These slides are to *supplement* the `README.md` project description in the GH repo starter code, as well as the lecture, and the introduction of **FLEX** in our course textbook
- Consult all resources and ask questions on discord
- See these slides for a visual presentation of the Maven infrastructure

JavaCC

- We'll be creating a **Lexical Specification** for our **Ram23** programming language and will be automatically generating a scanner (and eventually a parser, too) using the [JavaCC](#) tool
 - **JavaCC** is based on Flex
- **JavaCC** Resources:
 - [JavaCC Tutorial](#)
 - [JavaCC and Maven](#)
- You don't have to manually download **JavaCC** as I already included it in our Maven `pom.xml` project specification

JavaCC Specification

- Starts with an optional list of options followed by a Java compilation unit enclosed between `PARSER_BEGIN(name)` and `PARSER_END(name)`
 - This will be the name of the generated parser
- Next is regular expression productions for the lexical specification
 - `TOKEN` is used to specify that the matched string should be transformed into a token
 - `SKIP` is used to specify that the matched string should be thrown away
- In the scanner, a single grammar production (w/ the Kleene closure) is specified to allow the generated scanner to recognize the tokens

```

PARSER_BEGIN(MyParser)
    class MyParser {}
PARSER_END(MyParser)

```

/ For the regular expressions on the right, the token on the left will be returned: */*

```

TOKEN : {
    < IF: "if" >
    | < #DIGIT: ["0"-"9"] >
    | < ID: ["a"-"z"] (["a"-"z"] | <DIGIT>)* >
    | < NUM: (<DIGIT>)+ >
    | < REAL: ( (<DIGIT>)+ "." (<DIGIT>)* ) |
        ( (<DIGIT>)* "." (<DIGIT>)+ ) >
}

```

/ The regular expressions here will be skipped during lexical analysis: */*

```

SKIP : {
    < "--" (["a"-"z"])* ("\\n" | "\\r" | "\\r\\n") >
    | " "
    | "\\t"
    | "\\n"
}

```

/ If we have a substring that does not match any of the regular expressions in TOKEN or SKIP, JavaCC will automatically throw an error. */*

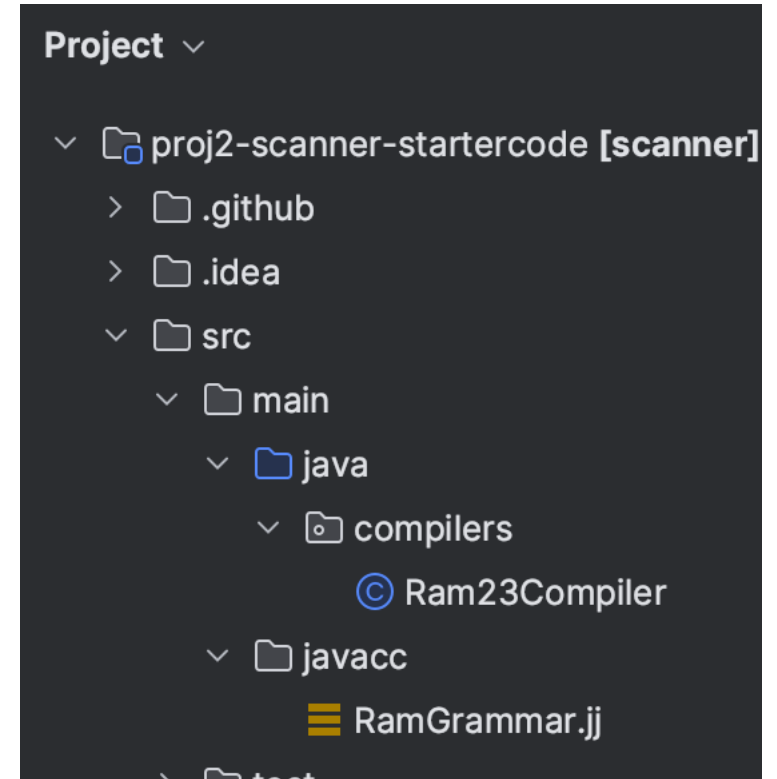
```

void Start() :
{ }
{ ( <IF> | <ID> | <NUM> | <REAL> )* }

```

RamGrammar.jj

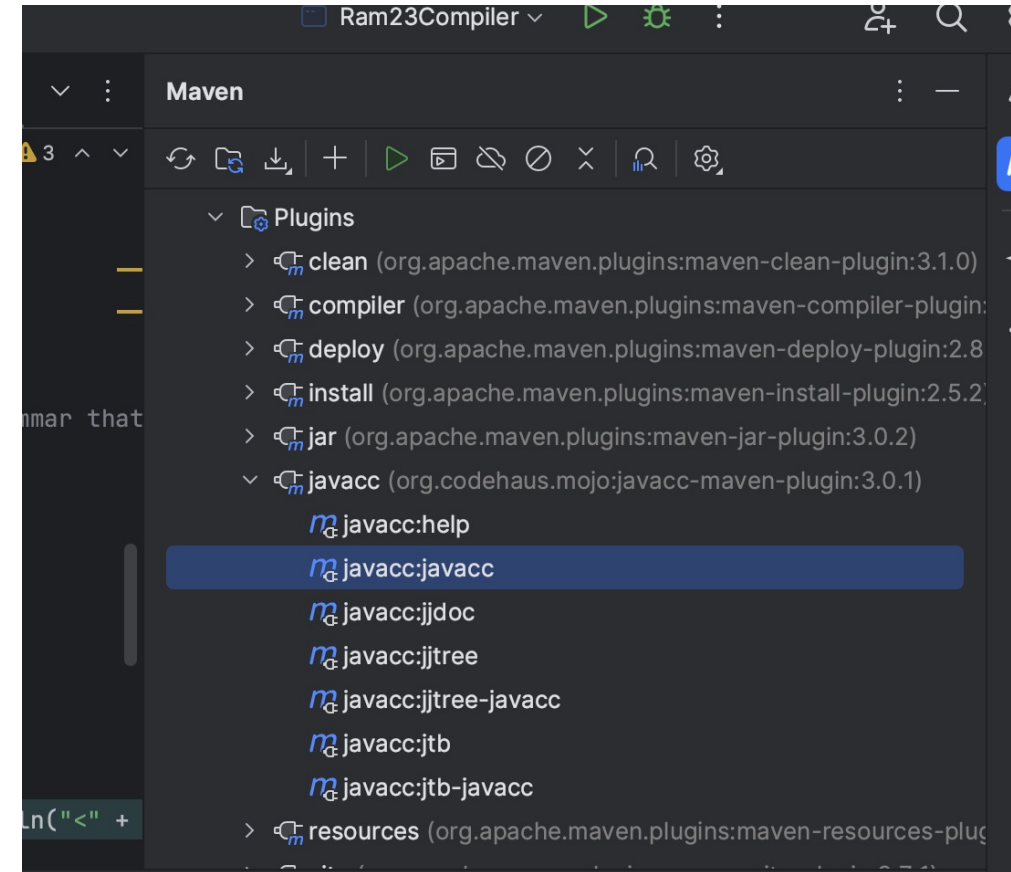
- Located in src/main/javacc/
- **This is the only file that you will be editing!**

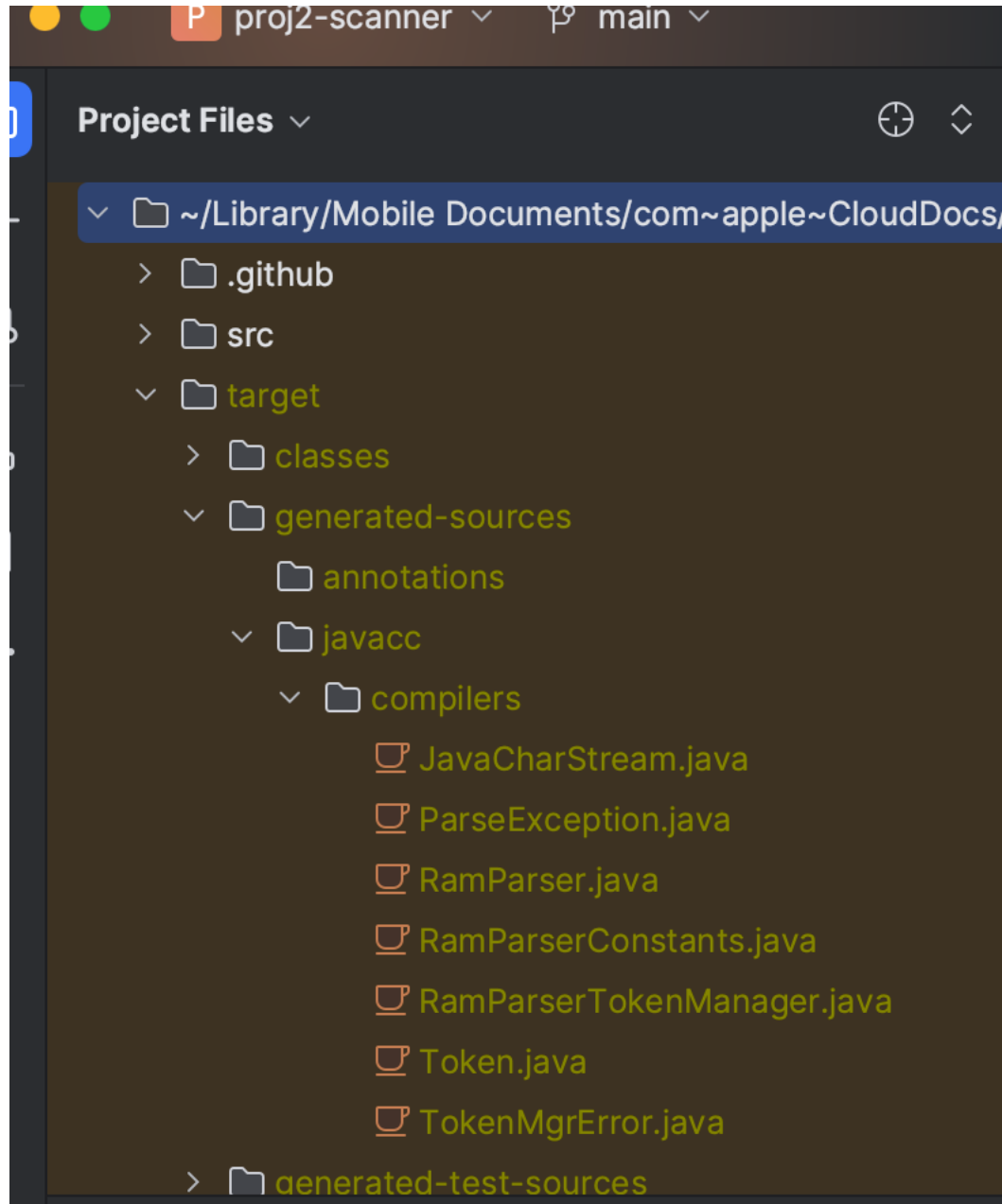


```
44      }*/
45
46
47      /* For the regular expressions on the right, the token on the left
48      will be returned */
49      TOKEN : {
50
51          < IF: "if" >
52          | < ELSE: "else" >
53          | < LPAREN: "(" >
54
55      }
56
57
58
59      // The following is a simple grammar that will allow you
60      // to test the generated lexer.
61      void Goal() :
62      {}
63      {
64
65          /*
66           ( RamToken() )* <EOF>
67          */
68
69          ( RamToken() {...} )*
```

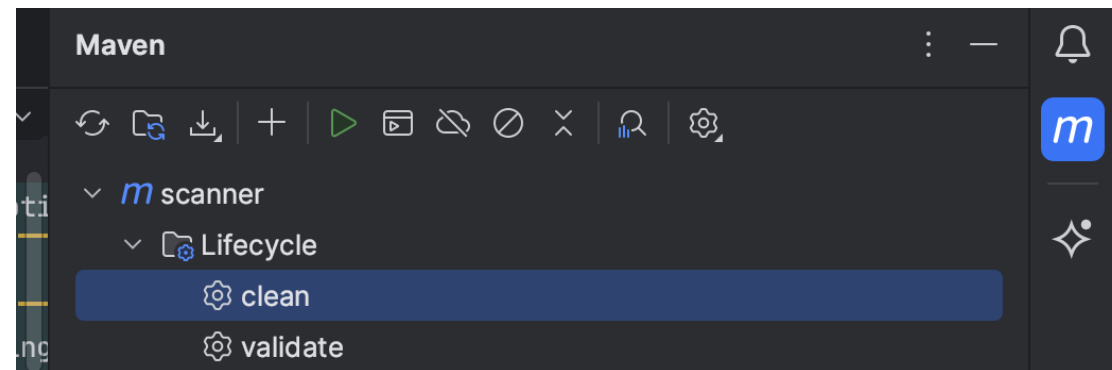
Next Step

- Use **JavaCC** to generate a scanner using the lexical specification that you entered in `RamGrammar.jj`
- Run the `javacc:javacc` Maven goal
- This will create *generated* .java files that will be saved into `target/generated-sources/javacc/compilers`





- If you peek at these files, you can tell how they were machine generated and are tough to read.
- Do not edit these files.
- You can delete them with by running the `clean` Maven lifecycle.



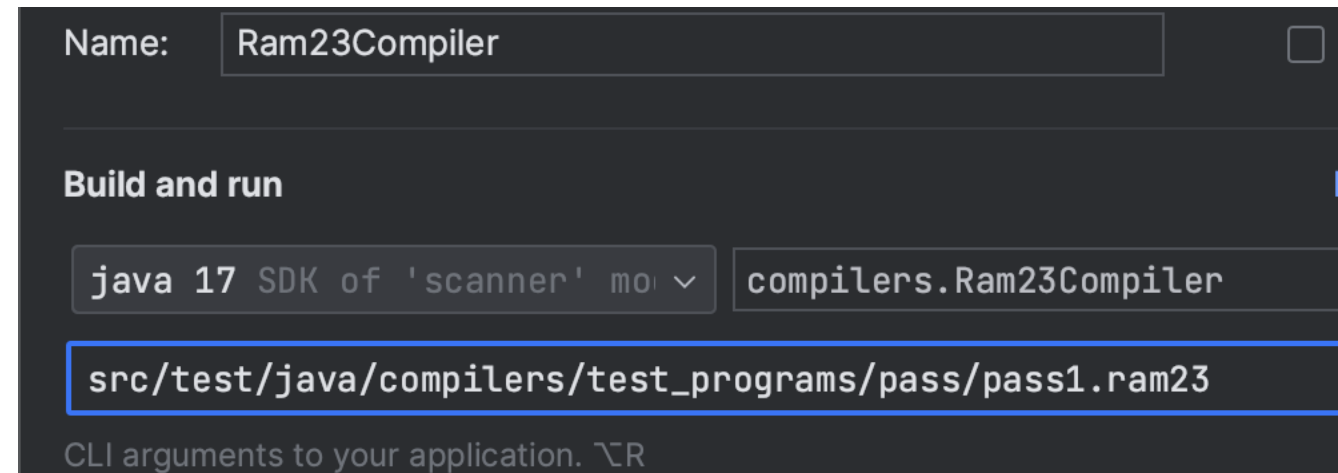
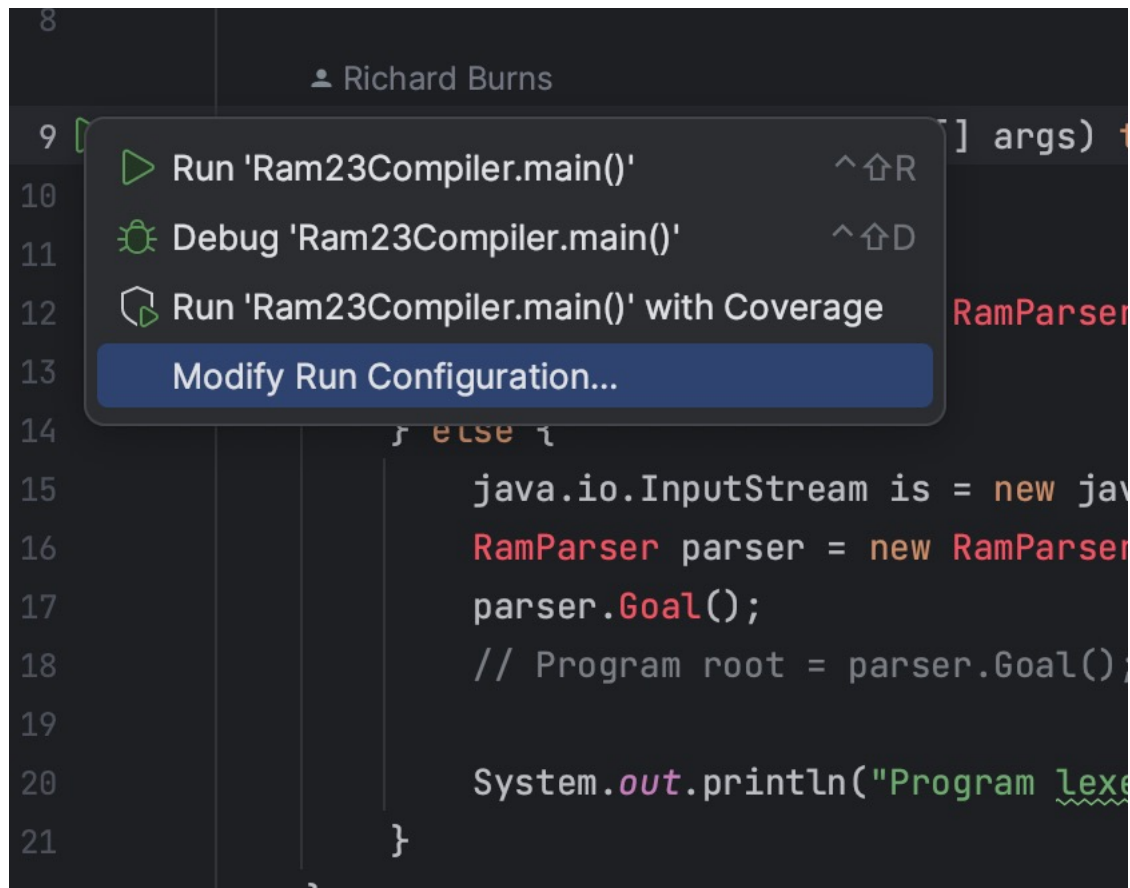
Test your generated scanner

- See `src/main/java/compilers/Ram23Compiler.java`

```
8
9  ▶ @ public static void main(String[] args) throws ParseException, TokenMgrError, File
10
11      if (args.length == 0) {
12          RamParser parser = new RamParser( System.in );
13          parser.Goal();
14      } else {
15          java.io.InputStream is = new java.io.FileInputStream(new java.io.File(args[0]));
16          RamParser parser = new RamParser( is );
17          parser.Goal();
18          // Program root = parser.Goal();
19
20          System.out.println("Program lexed successfully");
21      }
```

Set the arg of a file to use to test the scanner

- Modify the run configuration and enter the name of a `.ram23` file to pass in



What the Completed Scanner Should Output:

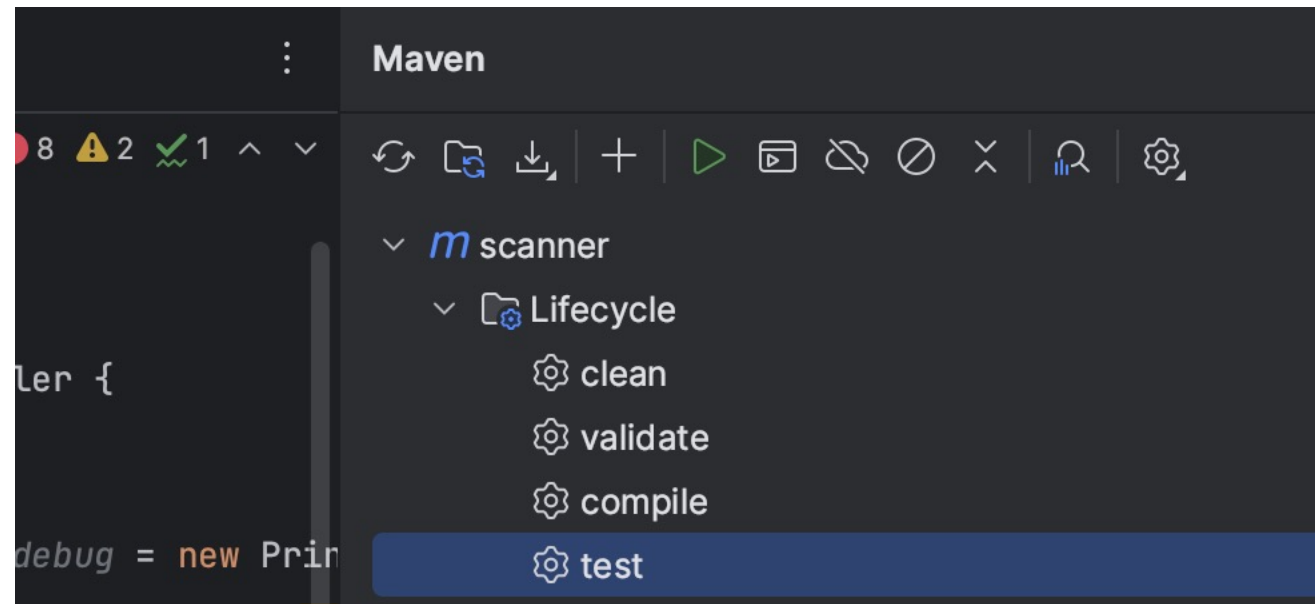
```
/Library/Java/JavaVirtualMachines/jdk-17.jdk/Contents/Home/bin
Kind: 35 line 1, column 1 - line 1, column 5 : class
Kind: 55 line 1, column 7 - line 1, column 11 : Test1
Kind: 13 line 1, column 13 - line 1, column 13 : {
Kind: 45 line 2, column 5 - line 2, column 10 : public
Kind: 47 line 2, column 12 - line 2, column 17 : static
Kind: 53 line 2, column 19 - line 2, column 22 : void
Kind: 43 line 2, column 24 - line 2, column 27 : main
Scanner > src > main > java > compilers > © Ram23Compiler
```

Notice the location of the test programs

- There are 5 `ram23` programs that should **PASS** and 4 `ram23` programs that should **FAIL**
- See `test/java/compilers/test_programs`

Run all 9 tests

- The Maven `test` lifecycle will run all 9 test cases, automatically calling **JUnit** and the `test/java/compilers/PassTest` and `FailTest` classes



Individual Tests in JUnit

- Your GH Actions when you push your repo will run these 9 tests individually. You don't have to look at this code, but this is what `PassTestSingle.java` and `FailTestSingle.java` is managing.