Comp 401 - Assignment 1: Writing a Number Scanner

Early Submission Date: Wed Aug 30, 2017 (+5%)

Completion Date: Fri Sep 1, 2017

First Late Date: Tue Sep 5, 2017 (-10%)

Second Late Date: Fri Sep 8, 2017 (-25%)

In this assignment, you will revise your programming skills by doing an assignment involving the use of many of the concepts that are a pre-requisite for this course, which include loops, and methods (procedures). In addition, you will learn how to scan a string. Listed below is material you should look at before the assignment. The last item addresses assignment-specific concepts. The earlier constitute background material.

JDK Download	PowerPoint PDF			-	-
Eclipse Install & Use	PowerPoint PDF Installing JDK on Mac	PDF		-	-
Checkstyle with UNC Checks : Install and Use	PowerPoint PDF			-	-
Runtime local checks: Install and Use	PowerPoint PDF			-	-
Relevant Java Syntax	PowerPoint PDF	PDF		-	lectures.java_basic_overview Package Git (Basic Overview)
Scanning	PowerPoint PDF YouTube Mix	Docx PDF Drive	Scanning Visualization	Number Scanner Checks File	lectures.scanning Package Git (Scanning) -

Scanning Input Lines for Numbers

Write a Java program that processes input lines until the user enters a terminating line beginning with the character '.'. Each of the other lines is expected to be a string consisting of digits and space characters. Each digit sequence is a token detected by the scanner and can be converted into a number. The program scans the string and prints the tokens (digit sequences in the string). It also prints the sum and product of the numbers denoted by the tokens on the line. It should print each token and the sum and product on a different line.

It does this for each input line other than the terminating line.

Thus, the tokens produced by your scanner denote numbers. Later your program will recognize other kinds of tokens such as words and quote characters.

You can assume that there:

1. is exactly one space character after each token (including the last one).
2. is at least one token in each input line.
3. a line consists of only spaces and digits.

The following example illustrates the nature of the expected program behavior for two input lines and a terminating line, in italics:

```
String?
2 3 0100
Tokens:
2
3
0100
Sum: 105
Product: 600
String?
4 6 20
Tokens:
4
6
20
Sum: 30
Product: 480
String?
.
Goodbye!
```

You are free to choose the prompts for user input ("String?" in the example above) and the labels (e.g. "Tokens:" in the example above) you print for the various aspects of the output. You are free to add other additional lines. In other words, as long as you put each expected output string on a separate line, you can create any user-interface you like.

Though in this example we have only three tokens on each line, in general, input lines can have a variable number of tokens. You can assume at least one token on each line.

As your string consists of only space and digits, you do not have to worry about negative numbers. Thus, '-' is an illegal character.

You do not have to do any error checking. Thus, your program can terminate abnormally if the user does not follow these constraints.

You can concatenate a value with another string using the + operation as in:

```
String labelledSum = "Sum:" + 105;
System.out.println ("The string is: " + labelledSum);
```

The End-Space Problem

As mentioned above, you can assume that there is exactly one space character after each token, *including the last one.* Requiring a space after the last token makes the implementation easier, and thus saves you effort as the programmer. However, as an end-user testing the program, this assumption is unreasonable as it makes you enter an extra space. There are several ways to address this problem:

a) Check if the last character entered by the user is a space and if not, add a space to the input string (using the + String operation) before submitting it to the scanner.
b) Do the variable space extra credit which allows a variable number of spaces at the end of each token.
c) Our test cases make the last space character optional. Therefore, you can assume that this space is not entered by the user, and pass the string directly to the scanner. If the test cases fail, use step (a).

Program Structure

Implement *and call* the following **public** static methods in your main class):

1. processInput(): It takes no argument and reads input lines in a loop (or recursively), passing each non terminating line to scanString() as the argument. It does not return a value. Its loop (or recursive call sequence) exits when a terminating line is detected.
2. scanString(): It takes a String argument, finds the numbers in the String, and produces the Console output associated with the string. It uses indexOf() to perform its task. It returns no value. This is the most difficult part of the

assignment – see the sections implementation hints and possible mistakes if you have trouble.

3. indexOf(): It takes as arguments a String s, a char ch, an int fromIndex (in this order). It returns the index (position) of the first occurrence of the ch in s that is greater than or equal to fromIndex, or -1 if the ch does not occur. Thus, the value returned is either -1 or k, where fromIndex <= k < s.length(), and aString.charAt(k) == ch. You are of course free to rename the arguments.

We require this program structure to increase program modularity and give you practice with defining and calling methods. To help us automatically find these methods, we dictate the names of these methods. Later we will see the concept of tags to give you a bit more flexibility.

You will need to implement the methods in the *reverse of the order presented above*, that is, in the order: indexOf(), scanString(), and processInput(), as indexOf() is needed in scanString(), which is needed in processInput().

You can, of course, write the scanner without these methods, but we require them. *We will not answer questions asking us if you need to define and use these (and the extra credit) functions unless you give a specific reason why you think these functions make your code less modular.*

Look at AModularConsoleReadingUpperCasePrinter in the lectures.scanning package of JavaTeaching on how static methods are called and defined.

You are free to create additional methods, public or non-public.

For those of you have used the **new** operator, there is no need to instantiate a class or prefix the name of a method with an object expression. If you have defined a class in which a method in the class calls another method in the same class, the situation is very similar here except that we are defining static methods and the keyword **this** makes no sense.

Variable Spaces (Extra Credit)

Allow a token to be succeeded or preceded by a variable number of blanks as in the input string "   2   3   0100   ". You should still be able to handle a blank after each token, as our test cases will assume that.

To implement this feature you should implement and call a method:

indexOfNot():

that takes the same arguments as indexOf() and returns the index of the first character that is *not* equal to ch,

Error Recovery (Extra Credit)

Do not terminate the program after encountering the first illegal (unexpected) character. Print the illegal character and continue scanning assuming the character had not been input. How you recover from or report the errors is up to you – for example, you can break up 123.23 into two legal numbers, 123 and 23, or a single token, 12323, or simply discard it by not converting it into a number. Similarly you can report errors as they occur or report them together. *We will not answer questions about nature of error reporting* as we have very flexible grading of this feature.

Scanning Iterator (Extra Credit)

If you know Generics and the Java Iterator interface (or are willing to study them on your own), make your scanner a separate class, called ScanningIterator, that implements Iterator<String>. This class should be in the package mp.scanner. Again, we dictate the name and package of the class so we can automatically find it.

This class will be responsible only for detecting tokens. ScanningIterator will have a constructor that takes the string to be scanned as an argument. Its next() method will return the next token() in the string and hasNext() will return false if there is no next token, and true otherwise.

Your main class will change in a few ways to use the iterator. The indexOf() method (and if you are doing variable spaces, indexOfNot()) will now be declared and called in ScanningIterator. It must still be static and public. The main class will instantiate ScanningIterator and call the next() method in the iterator until hasNext() returns false.

If you do this extra credit, the tester will take some points off from your regular credit for not implementing the index methods, but it will give you these points for having them in the iterator. In addition, it will give you points for doing the iterator. So the points you get for this extra credit will more than make up for the points you lose.

Constraints

1. *Public classes and main method and constructors:* All classes should be public, as well as all constructors (which you will study later) and main methods in them. Otherwise LocalChecks cannot see them.
2. *Do all scanning yourself*: Scanning is so important that Java has predefined functions that make the writing of this program trivial, which will be prohibited. These functions include certain functions of the String class (such as indexOf() and split()) and the Scanner class (such as nextInt() and next()). Therefore, we will place constraints on what aspects of Java you can use to prevent the use of these functions. You can use any of the Java features given in the section on scanning and "Relevant Java Syntax." In addition, you can and should use the Character.isDigit() (for extra credit and later assignments), Scanner.nextLine(), substring(), charAt(), length() and the Integer.parseInt() functions. Character.isDigit() is like Character.isUppercase() except that it tells us whether a

character is a digit rather than whether it is an uppercase letter. substring(), charAt(), and length(), applicable to any string, as explained in the class material. Integer.parseInt() takes a string consisting of digits and converts into an integer. Thus, Integer.parseInt("100") returns the integer 100. It assumes that the number represented by the string is small enough to be stored as an int. Your solution should make the same assumption. You will the Scanner class to read lines; make sure you use only the nextLine() method. You essentially have to make sure you do not use any function that automatically breaks the line into tokens- which is the task your code has to perform. *We will not answer questions asking us if you can use a certain class or method unless you reproduce some part of this assignment that makes the constraints ambiguous*

3. *No static variables:* Do not use static variables in your main class unless you are doing a recursive processInput() (see common mistakes).
4. *Single Scanner instantiation after calling main*: You should execute " new Scanner(System.in)" once after main is called. This means it should be executed outside the input reading loop in a loop-based solution. See the section on "Executing new Scanner (System.in) for local runtime checks" if you are doing a recursive processInput() or need more details on Scanner instantiation. Java allows multiple Scanner instantiations to be done at any time when input is submitted interactively but not when it is submitted through a testing program. You will get a NoSuchElementException from automatic testing if you do not follow these rules:

```
Scanned string222 444 666
222 444 666 Sum:
1332
Product:
65646288
Execution exception caused by invocation exception caused by:
java.util.NoSuchElementException: No line found
at java.util.Scanner.nextLine(Unknown Source)
at main.Assignment1.processInput(Assignment1.java:14)
```

5. *No star import*: Do not use a "star" import such as import java.util.*, as that is bad programming and will include many banned imports.
6. *Avoid "obscure" code:* Follow the programming practices and principles you have learned so far on how to write elegant code. In particular, do not write obscure code, but if you do, explain it through comments; and do not comment for the sake of commenting. A program without comments is acceptable as long as you think it would be clear to the graders. You should follow the case conventions.
7. *Main class long name= main.Assignment1:*To make it possible for the TAs and the grader program to find your main class, put it in a package called main, and call it Assignment1. In general, for assignment N, name the main class as AssignmentN, and put it in the main package, depicted hierarchically below:

```
project AssignmentN;
        package main;
```

```
          public class AssignmentN;
          other classes in
     other packages
```

8. *No System.exit():* The local checks program calls your main multiple times, once for each input it tests. So if you call System.exit(), the local checks UI will terminate after the first test, and you will not see the output. In later assignments, the TAs' grader will also call some of your methods directly. So if you call System.exit() the entire grading session for all students will terminate. So please do not call System.exit() in your programs, just call return.
9. Do not concatenate all (valid) tokens in a string or put them in an array before printing - print each token as you find it; there is no need to store it.

If we have not specified some aspects of the extra credit or regular work, you are free to choose your own interpretation and *please do not ask for clarification as there will be none to give.* Ask questions about these requirements only if they are not clear and tell us what is ambiguous about them.

Testing

You should test not only the input/output behavior but also each individual method you have implemented such as indexOf() and scanString().Your main method can create Strings pass them to these methods to test them. Here is an example of how you could do this:

```
if  (indexOf("12 21", '2', 2)== 3) {

        System.out.println (“index of succeeded”);

} else {

        System.out.println (“index of failed”);

}
```

Similarly, you can call scanString(“12 21”) to see if it gives the right output. It will be good to implement separate methods such as testIndexOf() and testScanString(), called by main, to do the testing. So your main will first call these methods and then processInput(). Your screenshots will show both the output of these test methods and processInput(). You have great freedom in how you test your code and if you indeed test individual methods. Testing of individual methods is mainly to help you debug your program and understand our test cases. You should definitely show console input/output.

Possible Hotswap Error Because of Input Loop

This program reads input in a loop until the user enters a special “end of input” line. While testing your program, you may not actually enter this special line, and start editing the previously run program while the program is waiting for the next input. This will result in a “hot swap failed” error from Eclipse. To prevent this from happening, see the Eclipse install slides on how to terminate a program.

Possible Exceptions You Might Get

`NumberFormatException,` as in:

```
java.lang.NumberFormatException: For input string: ""

at
java.lang.NumberFormatException.forInputString(NumberFormatException.java:65
)

at java.lang.Integer.parseInt(Integer.java:592)

at java.lang.Integer.parseInt(Integer.java:615)

at main.Assignment1.scanString(Assignment1.java:44)
```

Hover on parseInt() to see what it does. This means you have called parseInt() with a string that cannot be translated into a string. In this example, the top line of the stack trace (stack of method calls that lead to this problem) tells you that parseString() was asked to convert the empty string, “” to a number. This means, you did not tokenize correctly as only integer tokens should be converted to integers.

`StringIndexOutOfBoundsException,` as in:

```
ava.lang.StringIndexOutOfBoundsException: String index out of range: 7

        at java.lang.String.charAt(String.java:658)

        at main.Assignment1.indexOf(Assignment1.java:21)
```

Hover on charAt() to see what it does. This means you are asking charAt() to access a character in a string at a position that does not exist in the string. The top line tells you the index you tried. In this example, the string is of length <= 7.

Executing new Scanner (System.in) for local runtime checks

For local runtime checks to work, there are constraints on where you call new Scanner (System.in) and how many times you call it.

1. It should be called a single time.
2. It should be called after main is called.

So if you are not doing a recursive processInput, declare the Scanner variable in processInput before the loop is called as in:

```
public static void processInput() {
      Scanner scanner = new Scanner (System.in);
     //rest of processInput
}
```

If you are doing a recursive processInput, you cannot use the solution above as each call will create a new instance of Scanner. So in this case make it a static variable but initialize it in main() as in:

```
static Scanner scanner;
public static void main(String[] args) {
     scanner = new Scanner (System.in);
     // rest of main
}
```

A static variable is accessible from each call to processInput.

This is only for local runtime checks to work. Our grader will simulate your environment, so you will not lose points for not following these constraints as long as you are sure about your code.

Implementation Hints

Read this only if you have trouble developing your own solution. The small print is encouraging you to first think of the problem on your own.

As in the class example, you should define a variable that keeps track of the index of the start of each token. In the class example, the size of the token was constant (1) and there were no spaces in between tokens. This means that the startIndex of a token was always one more than the startIndex of the previous token. Now the tokens are of variable size. This means that in changing the startIndex, you must take into account the end of the variable-length token and spaces in between tokens. Given a start index, the end of the token can be computed using the indexOf() function As in the class example, make sure the startIndex does not go beyond the length of the string. After each token is processed, the start index is changed to go past the token, and becomes the fromIndex of indexOf(). You do not need an array (or array list) to store the tokens – you can keep a simple running sum and product. This in turn implies that the loop in scanString(), unlike traditional for loops, does not increment the start index by one or even a constant amount. The loop for scanString() can be tricky. If you have trouble, first write code for extracting the first, second and third token from the input using indexOf() and test with a string that has at least three tokens such as "12 21 34 "); Now write to do this with an infinite loop:

```
while (true) {
  …
|
```

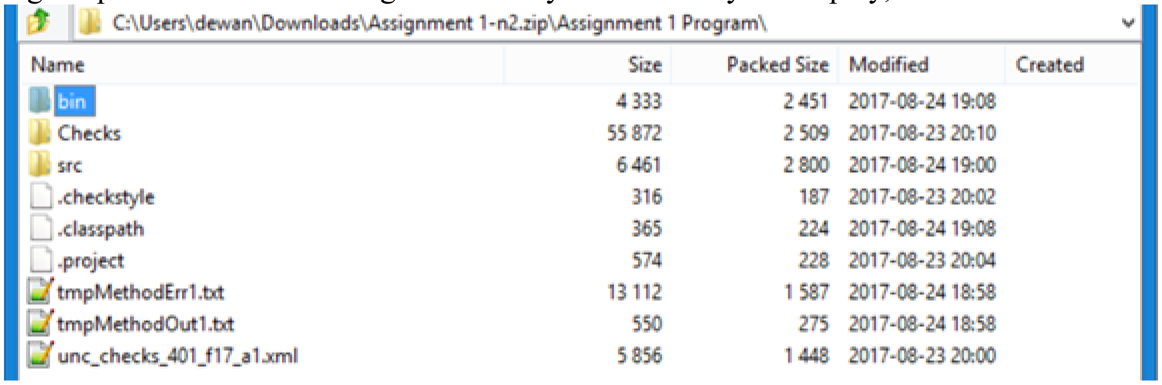
Finally, put an if statement in the loop that breaks out when there are no more tokens. This condition will test the start or end of the last token processed.

Submission Instructions and Feedback

The submission instructions for this and future assignments:

1. Upload the assignment *entire* project directory in Sakai. It should include the source and binary folders, the "RunTests" files, the "Checks" folder, ".checkstyle"

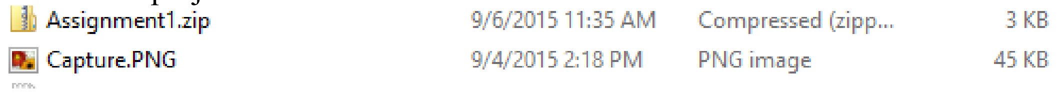
file, and any other file you have put in the project folder. You will need to zip it to upload it. In Eclipse, you can determine the folder in which a project is stored by selecting the project, right-click→Properties→Resource. Here is an example of a legal zip file. The files starting with a . may not show in your display, that is fine:



C:\Users\dewan\Downloads\Assignment 1-n2.zip\Assignment 1 Program\

Name	Size	Packed Size	Modified	Created
bin	4 333	2 451	2017-08-24 19:08	
Checks	55 872	2 509	2017-08-23 20:10	
src	6 461	2 800	2017-08-24 19:00	
.checkstyle	316	187	2017-08-23 20:02	
.classpath	365	224	2017-08-24 19:08	
.project	574	228	2017-08-23 20:04	
tmpMethodErr1.txt	13 112	1 587	2017-08-24 18:58	
tmpMethodOut1.txt	550	275	2017-08-24 18:58	
unc_checks_401_f17_a1.xml	5 856	1 448	2017-08-23 20:00	

2. You are also responsible for demonstrating to us that that these features do work correctly. You should do so by submitting electronic static screen shots of the console window (on Windows, the Snipping tool is a great way to capture screens). The screenshots should show that you have implemented all the regular and extra credit features in the assignment. These should be sufficient to tell the grader that the program works - they should not have to run the program to verify this. If you do not show evidence that a feature works, we will assume that it does not. ~~Put these in a folder and upload it in Sakai.~~ Provide as many screenshots as you need to show your work - one may be enough if you have good error recovery. *Each screenshot should be uploaded as a separate file.* The files should not be in the project folder. Here is an example submission folder, which contains the zipped assignment project and one screen shot. If you have more than one screenshot, then you would have multiple image files. You can submit multiple files to Sakai. *Do not put two zip files in your submission, as we then do not know which the project is.* We place no constraint on the name of the zip file that contains the project.



Assignment1.zip	9/6/2015 11:35 AM	Compressed (zipp...	3 KB
Capture.PNG	9/4/2015 2:18 PM	PNG image	45 KB

3. Make your main classes public - otherwise the grader cannot access them
4. You can submit multiple times. The last submission before the TA assignment download is the one that counts, and assignments will not be downloaded before the regular submission deadline.

Once your assignment is graded, to get feedback, go to Sakai and scroll to the bottom. You should see a bunch of attachments from the instructors – the most important is feedback.txt. Those should be your feedback. If you need more come see one of us.

How to ask get clarification on assignment requirements

Since we are using a natural language to specify requirements, and you have different backgrounds, it is understandable that you will ask for clarifications, which can help us better explain the requirements.

To help us understand your question and improve the assignment, please reproduce the wording, verbatim, in some requirement that is causing confusion and tell us why you think it is ambiguous. Also make this a public question, do not send a private message (or email) to instructors.

This means the following are not valid clarifications:

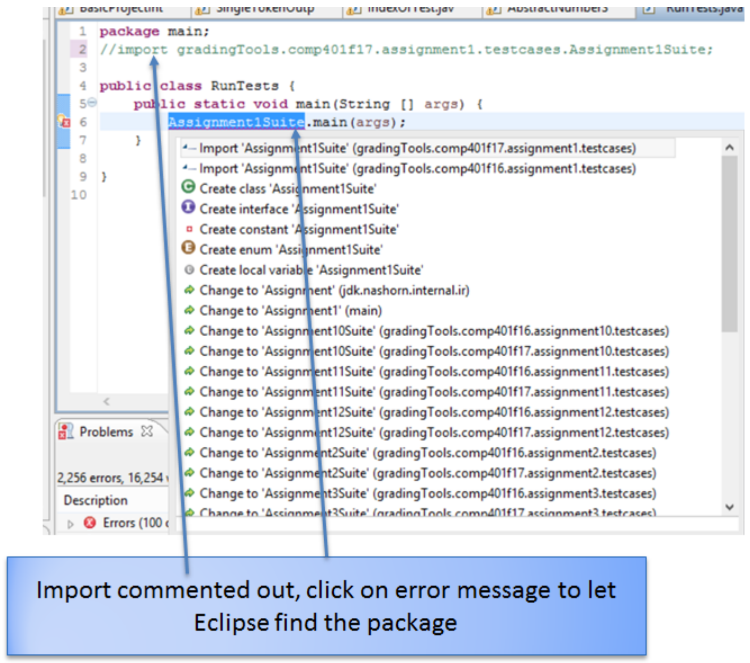
> I understand the requirement but have got the right output without following it, are you serious about the requirement.
>
> I do not think I followed the requirement ..., I hope that is ok.
>
> Is it ok to do the following .....

None of these follow the format I just mentioned above.

If you do not follow this format, we may simply mark the question as resolved or delete it, which is an indication to you that you need to rephrase or make it public.

Finding Packages

To use (method 1 of) LocalChecks, you need to know the package name of the appropriate package of the appropriate suite. When you know the class name, you can use Eclipse's error correction facilities to determine the appropriate package, as shown below:


Problems
2,256 errors, 16,254
Description
Errors (100
Import commented out, click on error message to let Eclipse find the package

```java
package main;
//import gradingTools.comp401f17.assignment1.testcases.Assignment1Suite;

public class RunTests {
    public static void main(String [] args) {
        Assignment1Suite.main(args);
    }

}
```

```
Import 'Assignment1Suite' (gradingTools.comp401f17.assignment1.testcases)
Import 'Assignment1Suite' (gradingTools.comp401f16.assignment1.testcases)
Create class 'Assignment1Suite'
Create interface 'Assignment1Suite'
Create constant 'Assignment1Suite'
Create enum 'Assignment1Suite'
Create local variable 'Assignment1Suite'
Change to 'Assignment' (jdk.nashorn.internal.ir)
Change to 'Assignment1' (main)
Change to 'Assignment10Suite' (gradingTools.comp401f16.assignment10.testcases)
Change to 'Assignment10Suite' (gradingTools.comp401f17.assignment10.testcases)
Change to 'Assignment11Suite' (gradingTools.comp401f16.assignment11.testcases)
Change to 'Assignment11Suite' (gradingTools.comp401f17.assignment11.testcases)
Change to 'Assignment12Suite' (gradingTools.comp401f16.assignment12.testcases)
Change to 'Assignment12Suite' (gradingTools.comp401f17.assignment12.testcases)
Change to 'Assignment2Suite' (gradingTools.comp401f16.assignment2.testcases)
Change to 'Assignment2Suite' (gradingTools.comp401f17.assignment2.testcases)
Change to 'Assignment3Suite' (gradingTools.comp401f16.assignment3.testcases)
Change to 'Assignment3Suite' (gradingTools.comp401f17.assignment3.testcases)
```

Understanding Checks

The error reporting can be much improved and please give us feedback on how this can be done.

Here is what you do when a check fails and you do not understand why.

UNCCheckstyle:
Post the message to Piazza.

LocalChecks:

Stop the program and re-run LocalChecks.
Do not execute Test All
Uncollapse all test suites.
Double click on one of the failed tests.
Look at the message in the explorer window for that test.
Look at the console output
If you do not understand the messages quickly, post the output to Piazza.
Contact an instructor if asked or you do not get an answer soon.
If you submit the assignment with the error, there are two cases:
   a) the test was wrong, so you get the credit.
   b) your program was wrong, in which case you do not get credit.
Repeat this process of rerunning the program and double clicking for each failed test case.

If you do this just before a deadline, there is a good chance you will not get a reply in time.

Localchecks score vs final grade

The score output by localchecks is not your final grade.

Your final grade is a function of:

> How you submitted the assignment. If you do not submit it correctly, our grader cannot unzip it correctly and even run localchecks on it. For instance if you submit two zip fliles, the grader does not know which zip to unzip for the project.
>
> The checkstyle output, which warns you about constraints that cannot be checked by localchecks.
>
> The human grading of your code and output (such as checking screenshots and source code for obscure code.

So make sure you follow all requirements and use as many tools as possible (LocalChecks, CheckStyle and GradingServer) to validate them. Let us know of issues with these tools that are unreported. *The final grade is a function of what you submit to Sakai, not the tools you use to validate them.*

Good luck with your first 401 program!

Understanding Assignment Final Grade

Along with your assignment score, you will get a series of feedback attachments. Scroll down in Sakai to see them. Send a private or public message in Piazza if you do not understand or have an issue with the grading. **When in doubt, ask!**

Possible Mistakes You Might Make

Not making the methods we expect such as main and indexO public. LocalChecks cannot find such methods.

Not making the classes we expect such as main.Assignment1 public. LocalChecks cannot fins such classes.

Comparing the entire line with "." using equals() (which I did not mention in the assignment) or using indexOf() to find a '.', instead of comparing the first character with '.' which give different results.

Parsing the terminating line into an integer , thereby throwing an exception.

Continuing to ask for input even after the terminating line has been recognized and not parsed.

Not implementing your indexOf() function.

Using the banned predefined indexOf() function of String.

Printing the number (200) rather than the token (0200)

Defining but not calling indexOf() and other required methods.

Instantiating the Scanner object once when in a static variable declaration – this instantiation is done before main is called.

Instantiating the Scanner object multiple times in a loop or in a static variable declaration.

Using localchecks but not checkstyle to ensure you follow all constraints.

Adding all (valid) tokens in a string or array before printing.

The loop in scanString using an index variable that goes from beginning to end of string, and this variable is not accessed inside the loop - the loop condition should use some variable that is accessed in the loop such as end or start of last token (depending on how your loop is structured).

Class name begins with lowercase letter - Eclipse will yell at you if a class name does not start with an uppercase letter.

Package name begins with uppercase letter - Eclipse will yell again if the package name does not start with lowercase.

Assignment submitted as part of JavaTeaching - create an independent project.
Submitting an RAR or some other archive. We need a zip.