Comp 401 - Assignment 3: Tokens and Interfaces

Early Submission Date: Wed Sep 13, 2017 (+5%)

Completion Date: Fri Sep 15, 2017

First Late Date: Tue Sep 19, 2017 (-10%)

Second Late Date: Fri Sep 22, 2017 (-25%)

This assignment builds on the previous assignment. In that assignment, each token string you extracted from the input string (number/word/quoted string) was stored in a String or int, which in turn was printed on the console. Now, you will store information about the tokens in instances of token classes defined by you. For example, you will now have a token class for a number token, whose properties will give information about both the value of the number (e.g. 10) and also the input string entered by the user to represent the number (e.g. 0010). You will create token Bean classes to represent the various kinds of tokens, and also convert token strings to instances of the token classes. The first part is easy, as we give you the readonly and editable properties of these classes. The second part is also relatively easy as you will simply call constructors in these classes to convert token strings to token objects. However, it assumes that you understand properties well. *Do not attempt this assignment until you have done and understand the ClassDualRoles praxes and quiz.*

You will also scan for two additional 1-character tokens: "{" and "}", which we will refer to as the start and end tokens.

All of these classes must implement appropriate interfaces.

Constructors and Pointers	PowerPoint PDF YouTube Mix	Docx PDF Drive			lectures.constructors_pointers Package Git (Constructors and Pointers)

Programming Interfaces	PowerPoint PDF YouTube Mix	Interfaces Chapter Drive		Token Beans and Interfaces Checks File	lectures.interfaces Package Git (Interfaces)

Start and End Tokens

Scan for two additional 1-character tokens: "{" and "}", which we will refer to as the start and end tokens.

Token Classes

For each kind of token string you detect above (number/word/quoted string/plus (extra credit)/ minus (extra credit), start token, end token), create a token class to represent the token. In addition, create two token classes for the start and end tokens. Let us refer to these classes by the kind of the tokens they represent. Thus, you will define Number, Quote, Word, Minus (extra credit), Plus (extra credit), Start and End classes. How you name these classes is up to you. However, you should associate each of these classes (and their interfaces) with a tag that indicates its role, using the tags "Number", "Word", "Quote", "Plus", "Minus", "Start", "End" for number/word/quoted string/plus (extra credit)/ minus(extra credit), start token, end token , respectively. Thus, the number token class should have the tag "Number" declared above its header:

@Tags ({"Number"})

These classes will all be Beans, which means they will define readonly and/or editable properties. If by now you understand Beans, this is a trivial assignment. *Otherwise, you should go back and study the ClassDualRoles lecture/praxes, and do the quiz on it. Without understanding what Beans and properties are, you cannot do this assignment!*

To make your task easy we give you below both the properties and constructors of these Bean classes. To make our grading task easier and to help you create reusable code, we require these properties to have names given by us. In this assignment, you might not use both the setter and the constructor. Both are required to give you practice with setters and constructors.

The number class defines two properties: (1) an editable String property, Input, storing a legal string representation of a number, and (2) a readonly int property, Value, representing the int value of the string. Thus, if the editable property is the string "00200"

the readonly property should be the **int** 200. You can assume that the editable property will always be assigned a legal value. The class should also define a constructor for assigning an initial value to the editable String property.

The word class also defines two properties: (1) an editable String property, Input, storing a legal string representation of a word, and (2) a readonly String property, Value, that is a lower case representation of the string. Thus, if the editable property is the string "MoVE," the readonly String property should be the String "move". Again you can assume that the editable property will always be assigned a legal value. This class also should define a constructor for assigning an initial value to the editable String property.

The other classes define a single editable String property, Input, representing a legal quoted-string (without the quotes), plus, minus, start-token and end-token, respectively. Of course, if you have not done the extra credit part to recognize a sign, then you need not define the plus and minus classes. All instances of each of these classes will have the same state if the setter is never called on them, which is the case in this assignment.

Thus, each of these classes defines an editable property, Input, defining a legal token string associated with the class; and the word and number classes define an additional readonly property, Value, storing an alternative representation of the token string.

The setter for the editable property allows testing of each editable class individually. It is possible to create an instance of the class and assign different values to the editable property (possibly using ObjectEditor) to see the effect on the dependent properties. As you see below, your scanner would never call the setter –it would simply use the constructor to give the property an initial value.

Along with the Tag annotation, you should add both the StructurePatternName, PropertyNames, and EditablePropertyNames annotations for each of these Bean classes based on the annotations given in the previous assignment. For example, the Word class should have the following annotations:

```
@Tags({"Word"})
@StructurePattern(StructurePatternNames.BEAN_PATTERN)
@PropertyNames({"Input", "Value"})
@EditablePropertyNames({"Input"})
```

Again, you will need to import these classes and associate your project with oeall22.jar, as in the previous project. These annotations are used to check if you have defined appropriate getters and setters for the properties. You still need to define these methods – annotations are not replacements of them.

Extended Scanner Bean

Modify your scanner bean class to use these token classes. The setter method of the scanner Bean breaks up its parameter into various token strings as before. For each of these token strings, it does the following. It (a) creates an instance of the corresponding token class; (b) assigns the input token string to the editable String property of the newly created token instance (using the constructor of the token class), and (c) prints (using System.out.println()), on different lines, the

(i) instance itself, and
(ii) all properties of the instances.

If you miss printing the instance or any of the properties, you will get an incorrect output message and zero credit for this part of the assignment.

If you were detecting errors in the previous assignments, then make this method print these also on the console.

As in the previous assignment, the tokens are not stored in the scanner bean. It is possible to create a new instance and (a) not assign it to a variable or (b) assign it to a variable holding another object. Thus, the following is legal:

```
System.out.println (new ABMISpreadsheet());
```

Here, the instantiated class is not assigned to a variable and simply printed.

Similarly, the following is legal:

```
bmi = new ABMISpreadsheet();
bmi = new ABMISpreadsheet();
```

Here, the same variable is being assigned two different objects in succession. After the second one is assigned, the first one is lost and "garbage collected" if no other variable points to it.

In this assignment, you will be creating a new instance, assigning it to a variable, printing it, and then essentially throwing the instance away by ignoring it afterwards and assigning a new instance to the variable that held it.

To illustrate the working of the scanner bean, if an input line is:

MoVe 050 { saY "hi!" }

this line should be assigned to the editable property of the scanner bean class, which will produce output of the form:

<Token ToString>

```
MoVe
move
<Token ToString>
050
50
<Token ToString>
{
<Token ToString>
saY
say
<Token ToString>
hi!
<Token ToString>
}
```

Here, <Token ToString> is a placeholder for some actual string println() produces when it prints an instance of a token class. To illustrate, let's say you have an instance of your word token class stored in a variable called myToken. <Token ToString> is a placeholder for what would print if you did this:

System.out.println(myToken).

You should get something that *looks like this* in the console for <Token ToString>:

mp.tokens.AWordToken@397dea61

This is because in this implementation, the Word token class is named AWordToken and is in the package mp.tokens.

You should not reuse token objects for different token input strings. For example, if you have two different words in the string, you should create two different tokens, which will be printed with different ids, as in:

```
mp.tokens.AWordToken@397dea61
mp.tokens.AWordToken@274bae14
```

As before, you can report errors.

Even if you know how to do so, do not override the toString() method in your token classes. The properties of the token classes provide sufficient information to print the token classes. Overriding will make it harder for us to determine if you are printing token classes by simply viewing the output.

Interfaces for Token Classes and the Scanner Bean Class

All of the token classes you have defined so far have an editable String property, Input, that stores a substring of the scanned string. Define an interface that contains the getter

and setter methods of this property, and make all of the token classes implement this interface. Tag this interface as “Token”:

@Tags({"Token"})

Some of the token classes, of course, have additional properties. Create additional interfaces for these classes to make it so that each public method in a class is declared in some interface implemented by that class. Make these token classes implement these interfaces. In this and later assignment, *an interface should have all of the tags of the classes that implement it. For example, the interface for the Word class should also have the tag “Word”*. This makes grading easier.

For example, the number class will now implement two interfaces: the common interface containing a getter and setter for the editable string property and another containing the getter for the readonly int property. Similarly, the word class will also define two interfaces, a common interface and a specialized one. Each of the other token classes will implement a single interface.

It is possible to cast from interface to another so you can access all properties of an object. Suppose C is a class that implements two interfaces T1 and T2 and C has a no argument constructor. Then the following is legal:

T1 t1 = new C();

T2 t2 = (T2) t1

Now t1 and t2 point to the same object, but have different types.

Similarly define an interface for the scanner bean and make the scanner bean class implement it.

You should tag both the interface and class.

Main Class

As in the previous assignment, the main class now instantiates the scanner Bean class to create a scanner Bean object. As in the previous assignment, it does not directly scan each input line or print the tokens in it. Instead, it simply assigns the input line to the editable property of the Bean object by calling the setter method, which results in the tokens in the line being printed in the manner described above. The termination condition remains the same, an entry of an input line starting with a dot.

Algorithm Hints

The scanner algorithm remains the same. The Scanner Bean is a bit more complicated. Instead of printing a substring of the scanned string directly, it creates an appropriate token Bean instance (passing the substring to the constructor) and prints the instance.

If you implemented the ScanningIterator, the next() method can still return the token string instead of an instance of the common token interface. If you want to return an instance of the token class, then you need to use the operation **instanceof** in the scanner bean to access all of the properties of the token.

Constraints

You should follow all constraints of the previous assignments with the following qualifications.

1. The Scanner Bean should print the tokens – a token should not print itself, which violates the separation of concerns principle – how an object exports its state and how that state is displayed are separate concerns.
2. If you know inheritance, feel free to make use of it in this assignment.
3. You can use the String toLowerCase() method.
4. As mentioned above, every public method of an instantiated class must be in some interface implemented by that class, in this and future assignments.
5. Do not use classes as types of variable or array elements – use interfaces instead. Follow these two constraints in all future assignments also, even if it is not explicitly stated.
6. Be sure to use the appropriate Tag annotation for each Bean class. The tags for classes from previous assignments (such as the scanner ban) remain the same.
7. Follow all style rules required in class material. This means you should not have public (non-final) variables.
8. As mentioned in assignment 1, if you are creating multiple packages, make sure the top level package begins with the expected name (grail) to ensure you do not get spurious illegal import messages. You do not need to do this for the main package, as mentioned in the assignment.
9. As mentioned in Piazza each class and interface declaration should be preceded by the keyword public, which ensures that it is in a separate file.
10. As mentioned above, do not override the toString() method in token classes, even if you know how to so.

11. An interface can be empty, that is, have no methods.

Submission Instructions

- These are the same is in the previous assignment except your document need not contain debugging screen shots. It should contain screenshots of your program running.
- Be sure to follow the conventions for the name and package of the main class so that its full name is: **main.Assignment3.**

Good luck!

Looking Ahead

Command Classes

Create a command token class for each of the following command names: “move”, “say, “rotateLeftArm”, “rotateRightArm”, “repeat”, “define”, “call”, “thread”, “wait”, “proceedall”, “sleep”, “undo”, “redo”. A command token class has the same properties and constructor(s) as the word token class. Thus, you will essentially copy and paste the code of the word token class into each command class, unless you know inheritance.

Recognizing Commands in the Scanner Class

Modify the setter method of the scanner to classify word into commands. If the lowercase representation of a word is equal to one of the command names, then create an instance of the associated command token class instead of the word class. Thus, if the scanned string contains the word “MoVE”, you would create an instance of the token class associated with the command “move”. You should use the String equals() (or equalsIgnoreCase()) method to test for equality of two strings. If a word is not one of the predefined commands, then it should be stored in an instance of the word token class, as in the last assignment.

Redoing the Scanner Properties

The scanner now has an additional property, which is a readonly stored property of type T[], where T is the interface implemented by all tokens (Please do not name this interface as T!). The getter method of this property returns an array of all token objects (instances of your token classes) created while scanning the String property of the scanner. There, should be no empty slots in the array, that is, the length of the array is the number of tokens in the editable scanner property storing the scanned string. You can assume a limit on the number of tokens in a scanned string. This means that you can create a large

array whose elements are copied into the array returned by the getter method of the readonly property.

The setter method of the scanner should no longer print the properties of the token objects it instantiates. Instead it sets the value of the variable holding the readonly property

Please do not submit the looking ahead part.