Comp 401 - Assignment 3:
Tokens and Interfaces

Date Assigned: Thu Sep 5, 2013
Completion Date: Fri Sep 13, 2013
Early Submission Date: Wed Sep 11, 2013

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, int or some other predefined value (e.g. 10, move), 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). Thus, 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.

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.

<table>
<thead>
<tr>
<th></th>
<th>Constructors and Pointers (8/29, 9/3)</th>
<th>PowerPoint</th>
<th>PDF</th>
<th>State Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>401</td>
<td></td>
<td>PowerPoint</td>
<td>PDF</td>
<td>Interfaces Chapter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Programming Interfaces (9/3, 9/5)</th>
<th>PowerPoint</th>
<th>PDF</th>
<th>Interfaces Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>401,</td>
<td>Token Beans and Interfaces</td>
<td>PowerPoint</td>
<td>PDF</td>
<td>Interfaces Chapter</td>
</tr>
</tbody>
</table>

Part 1: Token Classes
Scan for two additional 1-character tokens: “{” and “}”, which we will refer to as the start and end tokens.
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, plus,
start and end classes. How you name these classes is upto you.

These classes will all be beans, which means they will define readonly and/or editable
properties. To make your task easy we give you below both the properties and constructors of
these classes.

The number class defines two properties: (1) an editable String property storing a legal string
representation of a number, and (2) a readonly int property 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 storing a lega
string representation of a word, and (2) a readonly String property 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 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.

Thus, each of these classes defines an editable property defining a legal token string associated
with the class; and the word and number classes define an additional readonly property 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 a instance of the class and assign different values to the editable property 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.

Part 2: 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 token string to the editable String property of the instance (using the constructor),
and (c) prints (using System.out.println()), on different lines, the (i) instance itself, and (ii) all
properties of the instances. If you were detecting errors in the previous assignments, then the make this method prints these also on the console. As in the previous assignment, the tokens are not stored in the scanner bean.

To illustrate, if an input line is:

```java
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:

```java
<Token ToString>
MoVe
<Token ToString>
move
<Token ToString>
050
50
<Token ToString>
{
<Token ToString>
saY
say
<Token ToString>
"hi!"
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:

```java
System.out.println(myToken);
```

You should get something that looks like this in the console:

```java
tokens.AWordToken@397dea61
```

You should not reuse token objects for different token 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:

```java
tokens.AWordToken@397dea61
tokens.AWordToken@274bae14
```
Part 3: Interfaces for Token Classes and the Scanner Bean Class

All of the token classes you have defined so far have an editable string property that stores a substring of the scanned string. Give the same name to this property in all of the classes, define an interface that contains the getter and setter methods of this property, and make all of the token classes implement this interface.

Some of the token classes, of course, have additional properties. Create additional interfaces for these classes to follow the constraint that each public method of each of these classes is in some interface implemented by that class. Make these token classes implement these interfaces.

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.

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

Part 4: Main Class

As in the previous assignment, the main class now instantiates the Bean scanner class to create a scanner Bean object. 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 consisting of a dot.

Extra Credit: Tag Annotations

Tag each of the scanner-bean, number-token, word-token, quoted-string token, start-token, stopend-token, plus-token, and minus-token classes using the tags: “Scanner Bean,” “Number Token,” “Word Token,” “Quote Token,” “Start Token,” “StopEnd Token,” “Plus Token,” “Minus Token”, respectively. It will be best to copy and paste these strings into your annotations to make sure you do not make errors. The quotes around these strings will probably not paste properly, so you may have to retype them.

The syntax for tagging a class was given in the lecture and is illustrated below for the scanner bean class.

```java
import util.annotations.Tags;
@Tags("Scanner Bean")
public class <Class Name> implements ...
{
 ...
}
```
As you see above, you need to import and add the annotation before the declaration of the tagged class. For this to work, you need have the latest ObjectEditor library (oeall22) linked to your project.

You should not tag the interfaces, and should tag only the classes. Two classes should not have the same tag.

**Constraints**

1. If you know inheritance, feel free to make use of it in this assignment.

2. You can use the String toLowerCase() method.

3. As mentioned above, every public method of an instantiated class must be in some interface implemented by that class. 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.

**Submission Instructions**

- These are the same as in the previous assignment except your document need not contain debugging screen shots.

- Be sure to follow the conventions for the name and package of the main class.

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.