# Comp 401 - Assignment 7: Command Interpreter

**Date Assigned: Thu Oct 1, 2015**

**Completion Date: Fri Oct 9, 2015, 11:55pm**

**Early Submission Date: Wed Oct 7, 2015, 11:55pm**

**First Late Day: Wed Oct 21, 2015, 11:55pm (after break)**

**Second Late Day: Fri Oct 23, 11:55pm (after break)**

In this assignment you will learn how to create and use a table, parse tokens, and use casts and **instanceof**. In this process, you will connect the scanner and graphics parts of your code. You will also gain more experience with sub-classing.

In the previous assignment you refactored your token hierarchy. You will now use inheritance to re-factor your graphics classes, while honoring the IS-A rules. You will define a table using indexed collections. Using this table you will parse two kinds of commands.

You are free to use Vector, ArrayList, and other indexed collections discussed in class. However, you cannot use a Java class that implements a table or map for you. This means you cannot use Hashtable or HashMap.

## Part 1: Refactoring Shape Classes

Refactor your atomic and composite shape classes and interfaces to follow the inheritance rules discussed in class and given in the last assignment.

At the minimum your refactoring should obey the following constraints. If a class or interface has the X and Y properties, then it should be a subtype of some common type whose tag is "Locatable". If a class or interface has X, Y, Width and Height properties then it should be a subtype of a common type whose tag is "BoundedShape". This means you must tag both the common classes and interfaces. Thus both the locatable (bounded shape) class and interface should have the tag "Locatable" ("BoundedShape"). Of course, the bounded shape class and interface should be subtypes of the locatable class and interface.

In PropertyNames and EditablePropertyNames annotations of a subclass, you need to list the inherited properties as well as the properties implemented in that subclass. So if you

just have a single property, APropertyInThisClass, in a sub-class then OE will complain if you simply have the following annotations above the class:

@PropertyNames({"APropertyInThisClass"})

@EditablePropertyNames({"APropertyInThisClass"})

Rather, you will need to include the properties from all of the inherited classes in the annotations:
@PropertyNames({"AncestorProperty1","AncestorProperty2","AncestorPropertyN","APropertyInThisClass"})

@EditablePropertyNames({"AncestorProperty1","AncestorProperty2","AncestorPropertyN","APropertyInThisClass"})

More specific to this project, if you have X, Y and Point properties in a locatable then you need to put those in the annotations for your image shape as well. These annotations indicate not only which properties exist but also order of the properties in the display. So traversing the hierarchy creates ambiguities regarding positioning.

Similarly, the structure pattern annotations should be duplicated. Note that this is just for the annotations - do not duplicate methods from the super class (otherwise you won't meet the constraints of the assignment).

## Part 2: Table

Implement a table, which stores a modifiable collection of (key, value) associations, where keys are strings and values are arbitrary objects. The interface of this type must provide the following methods:

```
public void put (String key, Object val);
public Object get (String key);
```

It can implement additional methods. The operation put (key, value) checks if key is already associated with a value. If it is, it associates key now with value. Otherwise it adds the new association (key, value) to the collection of associations. The put operation does nothing if key or value is null. The method get (key) returns null if key is not bound to any value; otherwise it returns the value to which key is bound. Identify its structure pattern as MapPattern.

*Implementation Hint:* Consider a table to be a composition of a key column and a value column, where each column in an ordered indexed collection of values. As mentioned above, you are free to use arrays, Vector, ArrayList, AStringHistory, AStringDatabase, and AStringSet for one or both of the collections, or define your own collections. If you use ArrayList or Vector, look at the method "indexOf()", which is like the indexOf() method of AStringDatabase and AStringSet. (Naturally, you cannot directly use the string collections for the value column.) The contains() and

set() methods are useful methods for changing the value associated with a key. The table is not very difficult

The implementation of this table should be a few lines of code (it should fit in one or two PPT slides) - I have given this problem as an exam question in the past.

Tag this class and its interface as "*Table*".

## Part 3: Command Interpreter

Create a new class, tagged CommandInterpreter, that is implements a command interpreter. It has a reference to an instance of the BridgeScene and an instance of the BeanScanner, which are given to it as constructor parameters in that order. *Neither of these references should be visible to ObjectEditor*. This means you should either not define public getters for them in this class or make these getters invisible to ObjectEditor using the @Visible (false) annotation.

The command interpreter has one visible editable property, Command, for entering the command to be entered. When this property is set, it uses the scanner bean to get the token collection (which is returned an array and possibly also a history, depending on whether you did extra credit) associated with the string. It then parses the token list, that is, determines the command denoted by the token list, and then interprets the command, that is, executes the command. The command interpreter uses the instanceof operator on the tokens to parse the token list.

The command interpreter should recognize the following three commands for invoking methods in the scene, whose syntax is abstractly defined by the following grammar

<Command> → <Move Command> | <Say Command>
<Say Command> → say-token quoted-string-token
<Move Command> → move-token word-token number-token number-token

Thus, a command is a say command or a move command.

The say command consists of the say token followed by a quoted string token, as shown below:

say "What is your quest?"

This command invokes the say method in the scene. The quoted-string token gives the argument to the say method, indicating what an approached knight or the guard should utter.

A move command consists of move token followed by a word-token and two number tokens, as shown below:

move Arthur 20 30

The word token indicates the name of an avatar in the scene. An avatar name is the same as the name of the corresponding property in BridgeScene (Arthur, Lancelot, Robin, …). The character name should be case insensitive, that is, it should not matter whether a letter in the character is lower or upper case. The two number tokens indicate the distance the avatar should move in the x and y directions, respectively. Interpreting this command means moving the named avatar by the distances indicated by the two number tokens.

In case of the move commands, you will have to map the character name to an avatar object. Use a Table instance to do so. This means that you should not have a nested if statement to do the mapping. You will need to use casts to use this code as the value type is Object rather than an avatar. Define an invisible. The table should be an invisible readonly property, named Table, of the command interpreter. It should be listed using the PropertyNames annotation. The table will be filled before any command is executed.

You can assume that there are no parsing errors, that is, each token list indicates a valid command.

If you do not do any extra credit, the command interpreter will have only one visible property. When you have only one property, the entire ObjectEditor main window is devoted to displaying it and no label is displayed as there is no need for it. As with any Object property, make sure it is initialized to some non-null value as otherwise ObjectEditor will create a non-editable box for it.

Implementation hints:

1. Despite the table, you will have lots of nested if-else's, especially if you implement the extra credit. To make the implementation easier to understand and code, store the next token in a variable before the if-else executions so you do not have to extract the element from the token collection in each if statement.
2. Store the index of the next token and in an instance variable of the class. This way it is accessible to multiple methods in your command interpreter. The index will be initialized to 0 each time the interpreter setter is called, of course. But multiple methods called from the setter can move the index. By having multiple methods, you will increase the modularity of your code and also reduce its size. For example you can have a method to parse a signed integer (if you are doing extra credit). This method will be called both when parsing the x distance and the y distance of the move method.Tag this class as "Command Interpreter". You do not have to tag its interface.

## Extra Credit

1. Support the following version of the move command:

   <Move Command> → move-token word-token <Number> <Number>
   <Number> → number-token | +token number-token | -token number-token

This rule says that the x and y distances are indicated by just a number token, or a + token followed by a number token, or a - token followed by a number token. When you interpret this command take into account the signs to determine the x and y movement. If you support this feature, put an extra tag in the command interpreter: "SignedMove."

2. Detect invalid commands entered by the user. On encountering the first unexpected token or the end of input, abandon interpretation of the command and report the error by indicating the unexpected token/end of input you received and the correct token you expected in its stead. You should display this error message by setting a readonly property of the command interpreter. If you support this feature, put an extra tag in the command interpreter: "ErrorResilient."

## Animating Demoing Main Class

To demonstrate your work, write a main class that creates instances of the ScannerBean, BridgeScene, and CommandInterpreter classes and asks ObjectEditor to display all three objects. Then, as in assignment 4, animate the result of entering different strings. However, this time, you will be assigning each string to the editable property of the parser instead of the scanner. (The parser will of course assign this string to the ScannedStrring property of the ScannerBean). In each animations step, refresh all three objects. Before assigning to the editable property, make sure that you have made one knight approach the bridge by calling the approach method in the simulation from the main class.

If ObjectEditor breaks, print the properties of all three objects after each simulation step.

## Additional Constraints

As mentioned above, do not use a Java implementation of a table.

Be sure to follow the constraints of the previous assignments.

In particular, no magic numbers again, as the penalty for them will be much higher.

## Submission Instructions

- These are the same is in the previous assignment. The TAs will run the main method to see the test cases animate.
- Be sure to follow the conventions for the name and package of the main class.

Good luck!