In this assignment, you will extend previous assignments, getting a significant amount of experience with inheritance. You will extend the token hierarchy and the scanner you wrote to create tokens. You will override the toString() method of Object to display string values represented by tokens. You will define a new collection representing people in the gorge that can hold either an avatar or a knight. Finally, you will refactor code of previous assignments to follow the reuse rules given in class: (1) Ensure that same conceptual operations are associated with same method headers in different types (interfaces/classes). (2) Use inheritance to get rid of method header/body duplication in different types (interfaces/classes). Based on what you know so far, you will not be able to get rid of the code duplication in the collection types.

**Tokens revisited**

1. Refactor the token type hierarchy you created in assignment 4 to follow the inheritance rules.
2. Add to this hierarchy a token class for each of the following commands: *move*, *say*, *passed*, *failed*, *animate*, *undo* and *redo*. For instance, for the “move” command token, you must create a special class representing the move command, and for the “say” word taken, you must create another class representing the say command. If you implemented scrolling of the scene, you can recognize also the *scroll* command.
3. Add a method to a number token that returns its int value. To implement this method, you need to convert a string to a number. The method Integer.parseInt(), which you used earlier, takes a String argument and returns its int value.
4. Add code that allows the toString() method to be called on a token-object to get the input string it represents. In other words, the toString() method of a token-object should convert the token object back to the input string it represents. Thus, the toString() method when called on the token object representing the token “MoVE” should return the string “MoVE” (and not the lowercase representation returned by the StringValue property).
5. You do not have to change the Description and StringValue property of a token. That is, the Description and StringValue property of a user-command can be the same as the Description property of a generic command.
The type hierarchy you end up should allow a programmer to use the `instanceof` operation on a token-object to determine if it is a:

- token or some other object,
- number or command token,
- `move`, `undo`, `say`, `passed`, `failed`, and `redo` command token. It should be possible to distinguish between these tokens using just `instanceof` – without invoking `toString()` or some other method on the token object.

Thus, given a token-object representing the token “MoVE”, a programmer should be able to use the `instanceof` operation on the object to determine that:

1. It is a token,
2. It is a command token,
3. It is the `move` command token.

### Subclassing Scanner Class of Assignment 4

Create a subclass of the stateful scanner class of assignment 4 that instantiates the new token classes representing user commands rather than the old token classes. As it is a subclass of the previous class, it has the same properties, and scans an input string consisting of one or more tokens. The difference is that when it encounters a token that corresponds to a user command, it instantiates an instance of the specialized token class rather than the unspecialized command class. Thus, when it encounters the token “MoVE”, the scanner should create an instance of specialized class representing this command, rather than an instance of the class representing a vanilla command. If you need to change the scanner class (and associated interface) of the previous assignment, that is ok, as long as the original class does not understand the new token classes. The ability to reuse code from the previous scanner is relatively limited at this point – when you study iterators you will find a better way to reuse token processing from the super class.

### Refactor Graphics Classes

Refactor the graphics classes of assignment 5 to follow the two inheritance rules given in class. To do so while following ObjectEditor naming conventions, you may have to define empty classes and/or interfaces. This is not bad style. If you cannot find a way to use the rules to do the refactoring in your code, say so, and the graders will point out how you could have refactored if they can find a way.

### Fallen Avatars

Create a dynamic ordered collection whose elements can either be knight avatars or the guard avatar. This collection represents the fallen avatars, shown in the figure below. The collection should provide a method to add a new element to the end of the collection and another to remove the last element added (Last in First Out). The initial positions of the avatars are determined by the coordinates of the added element. The code that adds elements to the collection determines these coordinates - the add method of the collection does not change these coordinates. In fact, as shown in the figure, there is no relationship between the coordinates of the fallen avatars. The collection *does not* have methods to move its elements.
Ideally, it should not be possible to add to this collection an object that is not a (knight or guard) avatar. You can do so by creating an appropriate super type of the type of the two avatars.

**Aligned Movable Chat History**
Create a subtype of the movable chat history of the last assignment that ensures that its elements are left aligned, that is, have the same X coordinate. Unlike the chat history of the previous assignment, the method to add an element to this collection will change the X coordinate of the added element.

**Movable Avatars with Aligned Chat History**
Modify your avatars with chat history to use aligned movable chat history instead of movable chat history. You do not have to subtype your previous types describing a movable avatar with a chat history, you can modify them directly, though you are welcome to subtype them.

**Movable Knights with Chat History**
Similarly modify your collection representing marching knights with chat history to use movable knight avatars with aligned movable chat history. Again, you are welcome but not required to subclass the types describing the previous version of the collection.
**Aligned and Spaced Movable Knights with Chat History**
Create a subtype of the collection above that disregards the X and Y coordinates of the added knights and ensures that the knights are bottom aligned, that is, the lowest point of each knight has the same Y coordinate, and there is a fixed distance, in the X dimension, between (corresponding parts of) two adjacent knights. This distance is defined by this type.

**New Scene (Optional)**
Replace components of the scene of the previous assignment with the new versions of the two avatar collections and the guard. Also add the fallen knight collection to the scene. You will get no extra credit for this optional part, but by doing this part, you reduce work for the next assignment.

**Extra Credit**
1. Allow a user to enter multiple words (chosen by you) for the same command. For example, the words mv, move, MOVE, Move could all map to the token type representing the move command. It is not necessary to distinguish between the synonyms using the `instanceof` operation. However, the `toString()` value of synonyms should show what the user actually entered.
2. Write a tester for the class representing the aligned chat history to ensure that the alignment constraints are met. Similarly, write a tester for the lined and spaced movable knights with aligned chat history to ensure that the alignment and spacing constraints are met.

**Constraints**
As always, try and follow all style principles you have learned so far in class, and use only the concepts seen in class so far, and specified in earlier assignments. In particular, as mentioned several times above, follow the inheritance rules. The rule that says every class must implement an interface that includes all of its public methods is now modified as every class must directly or indirectly implement one or more interfaces that include all of its public methods. You can not only use the `String equals()` (taught in class) but also the `string equalsIgnoreCase()` method. You can use `instanceof` but should minimize its usage. Ideally, you should not use it all except to test your tokens.