In this assignment, you will make one or more objects observables to allow actions in the user object to update the view of the collection of avatars. In addition, you will use abstract methods. Furthermore, you will extend components of previous projects to see the benefits of the style principles you have followed so far.

Mapping multiple words to a single user command

In assignment 7, you created a separate token class for each of the words: *join, leave, move, message, undo,* and *redo*. It was not clear what you benefitted from doing so as it was possible to create a single class for all of these tokens and use the `toString()` and `equals()` method, instead of `instanceof`, to distinguish between them. One reason for creating a separate token class is efficiency – `instanceof` is faster than `equals()`. If we must determine multiple times if a word is a particular user command, it is overall more efficient to use `instanceof`. A more compelling reason is information hiding and extensibility. The parser in the user-object of assignment 8 does not know the mapping between words and user commands. Thus, you can change this mapping by changing only the scanner.

To demonstrate this extensibility, change the scanner of assignment 7 to map at least two words to one of the token classes. For instance, map both the words *mv* and *move* to the token class you created for the word *move*. As a result, the user can now enter:

```
move 50
```

or

```
mv 50
```

to move an avatar 50 units. If your assignment 8 is implemented properly, supporting alternate words for the same token class will not involve any changes to the user-object.

It is up to you to determine which command you map to multiple words, and what the additional word is.
**Uniting the two avatars and abstract methods**

In assignment 5, you created two kinds of avatars defined by two different interfaces. However, in subsequent assignments you used only one of these avatars. In particular, in assignment 6, your collection class consisted only of the complex avatars. Similarly, in the user-object of assignment 8, you instantiated only the complex avatar.

In this assignment, allow your avatar collection to consist of both kinds of avatars. To do so, you will have to make the interfaces of these avatars extend a common interface. You will have to change the avatar collection to work in terms of the common interface. You have to be careful about what methods you put into this interface. If you put too few methods, the avatar collection will not be able to execute certain methods on the avatar. If you put too many, you will not be able to unite both avatars.

Next, replace the user-object class of assignment 8 with two user-object classes, one that creates the complex avatar and another that creates the simple avatar. Ideally, these two classes should reuse as much code as possible by being subclasses of a single abstract class with one (or more) abstract method(s). To test this part of the assignment, you will write two different main classes, which will instantiate the two different user-object classes.

**Observables**

Currently, you must do explicit refreshes to see the results of a user command. Use your new knowledge of observables, property change events, and ObjectEditor behavior to automatically update the avatar collection window *in response to execution of the move command* in the user-object window. Try to make sure that the code you write to announce property change events is shared by as many classes as possible. You can read ahead in the PPT slides to see how notifications are sent to ObjectEditor or wait until the Mar 31 lecture.

**Extra Credit**

If you created a session object in the previous assignment, allow it to interactively choose the kind of avatar that is associated with a user object.

*Use the information in the PPT slides about observers of variable sized collections to automatically refresh the avatar collection window in response to execution of the message, join, and leave commands.*

**Constraints**

You can use java.util.ArrayList or java.util.Vector to store the observers. Look up the Java API to see how they work – they are very similar to the collections we have seen so far.
Submission Instructions

1. Submit a print out of your code at the start of class on the submission date together with screen shots showing your code working in various cases, and a document identifying how you support various style and functionality features.

2. Upload the assignment directory in blackboard. In general, for all assignments, you should do so by midnight of the day the assignment is due. But do not change the code after you submit it in class.