ITI 1121. Introduction to Computing II
Winter 2016

Assignment 3
(Last modified on March 9, 2016)

Deadline: March 25th, 2016, 11:59 pm

[PDF]

Learning objectives

- Create new **Exception** types,
- The **clone** method and the **Cloneable** interface.
- Using a stack to implement Undo and Redo
- Serializing objects

Source files

- All source files

Background information

We are going to improve our "circle the dot" game. To avoid complications, you are going to start from the version provided, which might be slightly different from the one you have already implemented.

Our overall goal is simple: we want to add an undo/redo feature to the game. The player should be able to "undo" all the moves, all the way back to the initial state of the game. After going through a series of "undo", the player will be able to "redo" some or all of the undone moves.

This feature, which might seem quite involved at first, will turn out to be quite simple to do thanks to the Model-View-Controller design we have been using. Using a couple of stacks, we will be able to record the various states of the game (that is, the model in the MVC design), and the only thing that we will need to do is store and restores these states as required. For this, we will use one of the feature of Java: object **cloning**, which will provide copies of our objects.

Another feature to add to the game (optionally) is the ability to save it, in order to close the application and then be able to resume the game exactly at the point it was left. In order to implement this, we will use another feature of Java: objects **serialization**. This will give us an easy way to store in a file the state of the game, and to read it back later. To make this simple, we will automatically save the state of the game if the user clicks the button "Quit" midway through a game. When a game is started, we will automatically look for a saved game and if one is found, it will be loaded instead of starting a fresh game (the saved game will be discarded at this point).

Overall, in this assignment, very few new lines of code are required: we will rely heavily on stacks and Java built-in features to achieve our goals.

Shallow copy versus Deep copy

As you know, objects have variables which are either a primitive type, or a reference type. Primitive variables hold a value from one of the language primitive type, while reference variables hold a reference (the address) of another object (including arrays, which are objects in Java).

If you are copying the current state of an object, in order to obtain a duplicate object, you will create a copy of each of the variables. By doing so, the value of each instance primitive variable will be duplicated (thus, modifying one of these values in one of the copy will not modify the value on the other copy). However, with reference variableS, what will be copied is the actual reference, the address of the object that this variable is pointing at. Consequently, the reference variables in both the original object and the duplicated object will point at the same address, and the reference variables will refer to the same objects. This is known as a **shallow** copy: you indeed have two objects, but they share all the objects pointed at by their instance reference variables. The Figure figure1 provides an example: the object referenced by variable **b** is a shallow copy of the object referenced by variable **a**: it has its own copies of the instances variables, but the references variables **title** and **time** are referencing the same objects.

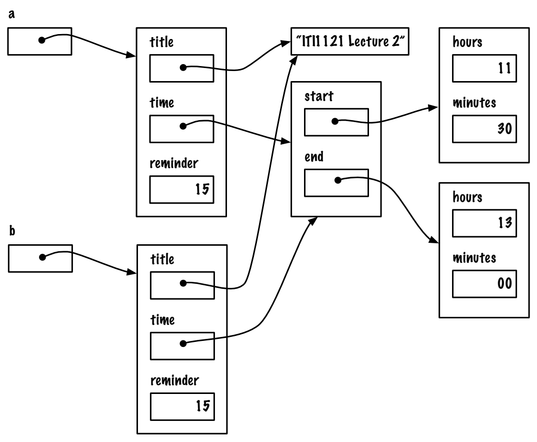

Figure 1: A example of a shallow copy of objects.

Often, a shallow copy is not adequate: what is required is a so-called **deep** copy. A deep copy differs from a shallow copy in that objects referenced by reference variable must also be recursively duplicated, in such a way that when the initial object is (deep) copied, the copy does not share any reference with the initial object. The Figure figure2 provides an example: this time, the object referenced by variable **b** is a deep copy of the object referenced by variable **a**: now, the references variables **title** and **time** are referencing different objects. Note that, in turn, the objects referenced by the variable **time** have also been deep-copied. The entire set of objects reachable from **a** have been duplicated.

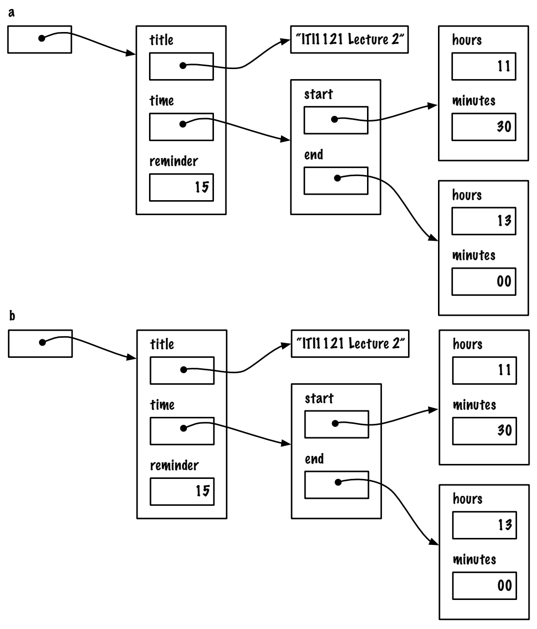

Figure 2: A example of a deep copy of objects.

You can read more about shallow versus deep copy on wikipedia, for example.

1 Exceptions (10 marks)

You have been given an implementation of a stack and of a queue based on linked elements. These implementations do not include precondition checks and do not throw any exceptions. This first question is about correcting this.

1.1 Defining new exception types (5 marks)

You must create two new **unchecked** exception types, respectively called **EmptyStackException** and **EmptyQueueException**.

1.2 Checking preconditions (5 marks)

You must modify the classes **LinkedStack** and **LinkedQueue** to verify the preconditions in each relevant method, and throw the correct exception when required. For example (but there are other cases to handle as well), **pushing** or **enqueuing** a **null** reference into the stack or the queue is not allowed. If this happens, a **NullPointerException** must be thrown.

2 Cloning the model (30 marks)

One of the methods of the class **Object** is the method **clone**:

```
protected Object clone()
                throws CloneNotSupportedException
```

Because clone is a method of the class **Object**, every Java class inherits that method. In a nutshell, the **clone** method returns a shallow copy of the object, provided that the object supports cloning. In many cases, a shallow copy will not be sufficient. Class can provide a more “specialized” implementation by overriding the method **clone**. Typically, the goal is to provide a deep copy, and also possibly avoid copying some of the variables that are not needed in the copy.

Calling **clone** on an object that doesn’t support cloning will generate a **CloneNotSupportedException** exception. A class signifies that it is ready to be cloned simply by implementing the interface cloneable.

In order to obtain a deep-copy, you need to:

1. Implement the interface **cloneable**;
2. Override the method **clone**;
3. Call the **clone** method of the superclass;
4. Recursively clone objects referenced by the reference variables.

Unlike the interfaces we have seen so far, cloneable doesn’t have any method. In a sense, for a class, implementing cloneable is simply a statement that the objects of that class can be cloned.

The goal here is to ensure that objects of the class **GameModel** can be cloned. You must ensure that your cloned copy of the GameModel object has its own copy of the following objects:

- currentDot
- model

Modify the class **GameModel** and possible other classes as required, so that instances of GameModel can effectively be cloned.

The fallowing resources might be useful:

- https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone- -

3 Implementing an “Undo” feature (30 marks)

Now that we can clone GameModel, we can use a Stack to easily add an “undo” feature to our game. What can be “undone” here are the player’s moves, and the corresponding next move from the game (that is, if you undo the last move, the game must go back to the state it was in just before the player’s last move).

Using the stack implementation of question 1, you must implement this feature by cloning the model as required and use the stack appropriately.

You will see that you will need to be able to **restore** a gameModel object to a state that has been previously cloned, so you need to add the corresponding method(s) as well.

In your user interface, you will need to add an “undo” button, to use your feature. Make sure that the button is only enabled when “undo” is indeed possible.

4 Implementing a “Redo” feature (20 marks)

Having done “undo”, it should be relatively straightforward to add a “redo” feature. There is only a little bit of logic to add, to figure out when “redo” should be possible, and when it should not.

Here too, you need to add a new button, and make sure that the button is only enabled when “redo” is possible.

5 BONUS: Saving and Restoring the game (10 points)

Java provides another feature: the ability to **serialize** an object, often to store its state on a file. Much like the case of cloning, a class indicates that it supports serialization by implementing the interface serializable. Unlike cloning, serialization does an automatic deep copy of the object being serialized.

The feature you must implement is the following: when the player exits the game by clicking on the “Quit” button while the game is still going on, you must automatically serialize the current instance of the GameModel object inside a file named “savedGame.ser”, located in the current game directory. When the game starts your program should check if the file “savedGame.ser” is available. If so, it should use it to restore the previously saved game state, and then delete the file. If the file is not present, the game starts as before[1].

Rules and regulation (10 marks)

Follow all the directives available on the assignment directives web page, and submit your assignment through the on-line submission system Blackboard Learn.

You must preferably do the assignment in teams of two, but you can also do the assignment individually. Pay attention to the directives and answer all the following questions.

You must use the provided template classes.

Files

You must hand in a zip file containing the following files.

- A text file README.txt which contains the names of the two partners for the assignments, their student ids, section, and a short description of the assignment (one or two lines).
- The source code of **all** your classes
- The corresponding JavaDoc doc directory.
- StudentInfo.java, properly completed and properly called from your main.

As usual, we should be able to compile your code by simply extracting your zip file and running “javac CircleTheDot.java” from the extracted directory.

Last Modified: March 9, 2016

[1]When a previously saved state is restored, the size of the game is the size of the restored game. If a different size was passed on as parameter when the game was restarted, that value is ignored.