Stacks – Image Enhancer

Purpose

Although Java has a Stack class already available, it is good to implement some data structures ourselves using more basic classes, in order to understand in detail how the data can be represented and how the data structure's associated methods really work. The stack is relatively simple, and it affords a good starting case.

This assignment involves working with some starter code that implements an actual desktop/laptop application program. Thus, we get to put the stack data structure to use in a realistic setting.

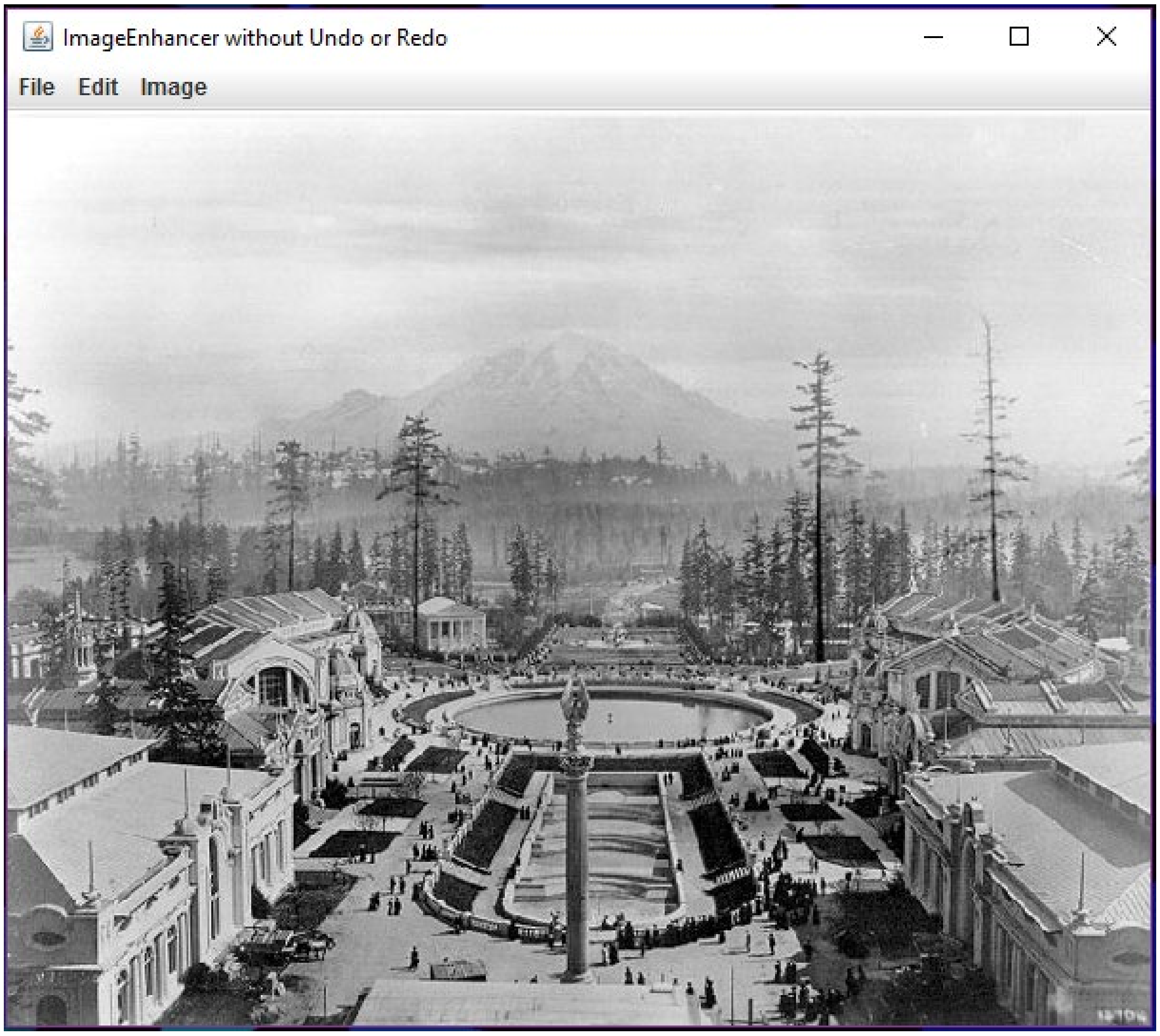
Description

In this assignment you'll add an Undo feature and a Redo feature to a Java application for image enhancement. These features should make use of stacks. Parts of the challenge will be the implementation of the stack operations and integrating it into the application.

How to Get started

Start with the "Image Enhancer" program ImageEnhancer.java. Create a Java project and put a copy of Rainier Vista image (from the 1909 Alaska-Yukon-Pacific Exposition) into the main project folder. Make sure you can compile and run this program. You operate it by choosing image transformations from the Image menu. The program will apply the selected operation to the image and show the result as the new "working" image. When you select a second transformation, it is applied to the working image, rather than the original image. Many different sequences can be tried. However, there is no way to undo an operation in this program (other than restarting the program, which is not very cool).

Here's a screen shot of the ImageEnhancer program that you'll start with.



Basic Specifications

Create another project, ImageEnhancerWithUndoAndRedo. Put a copy of the starter code in this project but rename the file and public class to be **ImageEnhancerWithUndoAndRedo**. Also put another copy of the Rainier Vista image in this folder, so the new version will work, too.

Change the title on the new application's window to be "Image Enhancer WITH Undo AND Redo by " followed by your full name. For example, it might be something like "Image Enhancer WITH Undo AND Redo by Mary-Lou Jones".

Create a new class, in a separate file named **BufferedImageStack.java**. This class must implement a stack of BufferedImage objects, and you must not import the built-in Stack class, but rather you will utilize an array to hold the elements of your buffered image stack. When a new instance of your BufferedImageStack class is created, the array should be given a size of 2.

Comment each method in this file. You are encouraged, but not required, to use standard JavaDoc conventions for this. Also provide a comment at the top of this file that gives the name of the file, your name, and an explanation for the purpose of this file as part of the whole application. The methods to provide are the following:

- **push**(BufferedImage someBufferedImage): enters the buffered image onto the stack and returns nothing. If this would exceed the capacity of the array, then a new array should be allocated having double the size of the old array, and the old array's elements copied to the new array.
- **pop**(): throws an exception if the stack is empty; otherwise returns the top buffered image, removing it from the stack. The exception should be an instance of java.util.EmptyStackException. In this assignment, you are not required to ever replace a large array by a smaller array when the number of stack elements decreases because of pop operations.
- **isEmpty**(): returns true if there are no items in the stack; false otherwise.
- **get**(int index): returns the buffered image at the position given by the index. (This is not commonly available in a stack, but it facilitates the testing by our autograder.) If the index is out of range, the method should throw an IndexOutOfBounds exception. (Note: get(0) gets the bottom element of the stack -- the one that was pushed in first, but not yet popped out.)
- **getSize**(): returns the number elements currently in the stack.
- **getArraySize**(): returns the current size of the array being used to hold the stack elements. Like the get operation, this is not a normal stack operation, but may be used by the grading software to assure compliance with the specifications.

Make use of your new BufferedImageStack class to implement the Undo feature in your application. If the user has not yet applied any operations to the image, or has undone all the operations performed, then the Undo menu item must be disabled. When disabled, it will be displayed as grayed out. The starter code sets this up for you by default.

The user should be able to undo all the operations that have been applied so far, to go all the way back to the original image. Thus, your applications will have multiple levels of Undo. Whenever, there is an operation that can be undone, the Undo menu item must be enabled.

Once your Undo feature is working, figure out how to provide functionality for the Redo feature. The Redo menu item should only be enabled and do something if an Undo operation has just been performed.

There should also be multiple levels of Redo. However, if the user performs an Undo action and then applies some other operation to the image, there is a question of whether any remaining Redo options should persist or whether they should be deleted. For this assignment, assume they should be deleted.

Whenever an Undo command or a Redo command is successfully performed, your application should print out a message (using System.out.println) that explains clearly how many elements are currently in the Undo stack and how many are currently in the Redo stack. Code to do this is provided in the starter code but is commented out. After you have implemented your undo and redo functionality, you should uncomment this code.

Grading Rubric

In this assignment, you can earn 100 points as follows.

- Window title change correctly handled: 5
- BufferedImageStack.java file and class exist and all specified methods work correctly: 40. (push, pop, isEmpty, get, getSize, getArraySize). If any of these are missing or incorrect, subtract 8 for each one missing or incorrect, but net points for this cannot go negative.
- Two instances of your BufferedImageStack class are created in your application: one for Undo and one for Redo, and these are clear in the code: 5.
- 40 points for correct implementation of functionality.
- `Undo works properly from an undoable state.`
- `Redo works properly from a readoable state.`
- `(15 points deducted from the 40 for each of these two`
- `features unimplemented or not working properly.)`
-
- `Undo item grayed out and disabled when stack is empty.`
- `Undo item not grayed out and enabled when stack is not empty.`
- `Redo item grayed out and disabled when no operation is available to redo.`
- `Redo item not grayed out and enabled when some operation IS available to redo.`
- `(5 points deducted from the 40 for each of these four`
- `features unimplemented or not working properly,`
- `subject to cutoff at 0.)`
- 10 points for writing very clear readable code with clear comments in English for each new class, method and function, and static variable. Any other variables should have names suggestive of their meaning, with comments unless the variable is a loop variable (i.e., int i) or a size or length (int n). JavaDoc format is fine but not required. The comments at the top of each file, described earlier, also contribute to these points.

Note: Some points (max 20) will be deducted if the program is particularly slow or it flashes multiple images on the screen when an Undo or Redo is performed.

Dealing with BufferedImage Objects

The actual image processing content in this assignment is optional material -- not required. For those interested, here is some information about the image processing.

For this assignment, the only type of image processing object that you really need to pay attention to is the BufferedImage class. When you look at the source code ImageEnhancer.java, you can see calls to its constructor. For example, there's this code:

```
biWorking = new BufferedImage(width, height, BufferedImage.TYPE_INT_RGB);
gWorking = biWorking.getGraphics();
gWorking.drawImage(biTemp, 0, 0, null);
```

This creates a new buffered image object for an image with width *width* and height *height*. That means *width* columns and *height* rows. The type of image is RGB meaning that each pixel will have three numbers associated with it: one for red, one for green, and one for blue. The buffered image is assigned to a variable biWorking, which is the working buffered image ... the one that the user sees on the screen most of the time.

The two lines of code that follow that constructor call above do something important: they put image data into the buffered image. This is done in two steps. First, a "graphics" object is obtained by calling the getGraphics() method of the BufferedImage object biWorking. The graphics object supports "painting" into the pixels of the buffered image, and one of the painting methods is drawImage. By invoking the drawImage method of the graphics object gWorking, the pixel values from another image, here biTemp, get written into the pixels of biWorking. The contents of biTemp are read from a file.

You will need to arrange for BufferedImage objects to be created and saved on your buffered image stack. You will need to arrange for BufferedImage objects to be created (typically as copies of biWorking) and given pixel data (using the same approach involving *drawImage*described above).

Comments on the Use of AWT and Swing

The ImageEnhancer application uses a graphical user interface ("GUI"). It is implemented using a combination of Java AWT (Abstract Windowing Toolkit) and Java Swing (another library of components provided by the Java Runtime Environment).

The important thing to understand for this assignment is that user actions are handled primarily by one method, called actionPerformed. In order to add more functionality to the program, you should first add any needed components such as menus and menu items, and then add more cases to the body of the actionPerformed method. The method is called when the user clicks the mouse, etc., and information about the user-generated event is passed in as the ActionEvent object, bound to the variable e. The body of that method consists of some tests to find out what the user acted on (e.g., selecting from one of the menus), and then some code that performs the requested action.

Miscellaneous Advice

Remember that a stack uses the LIFO access pattern: Last-In, First-Out. (Some say FILO -- First-In, Last Out). Understand what a buffered image is and how to create one and copy pixel data into it.

Think carefully first and then code. Drawing a graph or diagram on paper can help a lot when thinking about Undo and Redo.