Version Number:

1.00 (July 28, 2020)

Assignment 4

Land Registry, Part 4

To be submitted online not later than Friday, August 7, 2020, 11:59 p.m.
(However, submissions will be accepted up to Sunday, August 9th, without penalty.)

Description:

In this lab you'll add a graphical front end (GUI) to the existing Land Registry code. Along the way, you'll demonstrate your understanding of the following course learning requirements (CLRs), as stated in the *CST8284—Object Oriented Programming (Java)* course outline:

1. Produce code that has been tested and executes reliably (CLR VIII)
2. Debug program problems using manual methods and computerized tools in an appropriate manner. (CLR X)
3. Introduce Swing for building a Graphical User Interfaces (GUI) using Java Foundation Classes (CLR XII)

Worth
4.5%
of your total mark

Assignment 4

Land Registry, Part 4

Program Description

In this assignment you'll build a graphical front end to your existing Land Registry; at the completion of this assignment your program should be fully graphical. That is, there should be no console-driven input or output: everything is entered or displayed via the GUI interface to the user.

Marks will be deducted for any text output that appears in the Eclipse console during program execution, including as the result of thrown exceptions. Furthermore, all input must be entered via a graphical control only (e.g. button, check box, text field entry, etc.).

This is not to suggest that your finished application will have a production-quality look to it, with all of the buttons nicely aligned and each frame resizable to fit any screen size. Given the time constraints, you don't want to waste the few hours remaining in the semester getting these 'look-and-feel' issues just right. However, your code should be operationally robust, able to catch bad input and correct the user with suitable dialog, and enabling and disabling parts of the user interface when appropriate.

I. Download the Assignment4 project and copy your existing classes to it.

a) Download the CST8284_20S_Assignment4.zip file and unzip the project in Eclipse, just as you have done with earlier labs. The project file has a cst8284.asgmt4.landRegistry package, and this includes a new class, RegViewGUI, which will control all graphical input and output. This class will replace RegView, which should be deleted from your code prior to submission. But to begin, copy all your existing classes to the new package, including RegView.

b) There is no UML diagram for this assignment. So you are free to add new methods to your existing code, provided they follow the general guidelines established in Assignments 1, 2, and 3. And as well, your code must follow the best practices guidelines outlined earlier in the semester. There is no sample output provided—the output is graphical, after all. But you are required to use and display the dialogs indicated below as part of the execution of your code; you cannot include them with your submission and then ignore them during actual program execution.

Additionally, there is no starter code for this assignment, since you should reasonably be able to take your code from Assignment 2 and adapt it to this assignment, even without fully functioning exceptions in Assignment 3.

Most importantly, you must heed the warning just given in the introduction: your code must behave the way a normal windows application behaves, and it must be robust enough to handle bad input in the textboxes, or prevent input altogether when it is inappropriate.

II. Add the following new features to your Land Registry

The main purpose of this assignment is to wrap a graphical interface around your existing `RegView` code. Since each student builds their code slightly differently, it isn't possible at this stage to give specific instructions that will work for everyone. But in general terms, you'll need to do the following to implement your existing Land Registry code in GUI form. How you actually do this, and the specific decisions you make, will depend on your implementation of the code up to this point.

In the following steps, you'll gradually remove code from the existing `RegView` class and transplant it to `RegViewGUI`. At the end of this process, you should be able to copy any remaining code in RegView over to RegViewGUI, *and then delete RegView entirely*.

Reminder:
You must include code to sort the properties ArrayList in this assignment. See the instructions at the end of this document.

a) First, in RegLauncher, replace the existing main() method with the following

```
public static void main(String[] args) {
   javax.swing.SwingUtilities.invokeLater
   (new Runnable() {
      public void run() {
         new RegView().launch();
      }
   });
}
```

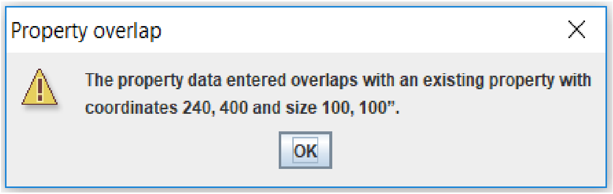
where the run()method contains the code that normally launches your program. The above code loads your program into a separate thread, permitting other programs to run in parallel with your GUI app. (The subject of threading will be taken up next semester in CST8288.)

The starting point for GUI execution in this application is RegViewGUI.launchDialog(). If you want to gradually transfer your code over from RegView to RegViewGUI (a fairly safe procedure which lets you test each code modification you make as you go) you can call RegViewGUI.launchDialog() from within RegView.launch(). Alternately, you can call RegViewGUI.launchDialog() directly from main(), inside the run() method, and operate on the GUI code directly; the choice is up to you. The guidelines in this document mostly follow the first of the two options.

Note:
RegViewGUI is provided as a framework for you to use and develop in this assignment. Feel free to modify it in any way you feel appropriate to achieve the requirements of this assignment.

b) In RegView, you can delete executeMenuItem() and displayMenu()— along with the fixed constants used to display the menu— since the buttons in the RegViewGUI dialog now replace the menu choices; clicking on a button has the same effect as selecting a menu item. At the same time, you can get rid of the Scanner import statement, and the new Scanner declaration in your code; *you won't be needing these any more*.

c) Whenever a BadLandRegistryException is thrown, it should cause a JOptionPane dialog to be displayed, like the one shown below. Note that the exception's getHeader() method is used to display the header string in the menu, while getMessage() provides the message displayed in the body of the dialog.


Property overlap

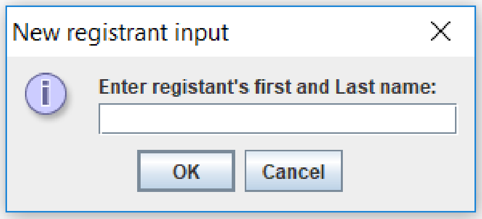
The property data entered overlaps with an existing property with coordinates 240, 400 and size 100, 100".

For information on using JOptionPane to create dialog windows, the web site

```
https://docs.oracle.com/javase/tutori
al/uiswing/components/dialog.html
```

is a good place to start.

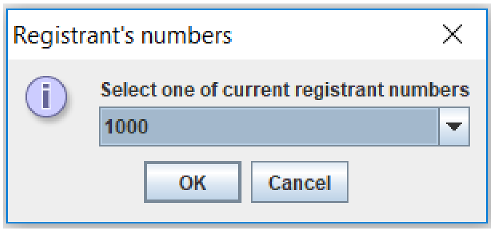
d) You'll also need to use a JOptionPane, like the one shown below, to add a new Registrant. (See the same URL listed above.)


New registrant input

Enter registant's first and Last name:

This dialog should be invoked whenever the 'Add New Registrant' button is clicked on in the RegViewGUI. But now, your code will need to call addNewRegistrant(), causing a new Registrant object to be added to the registrants ArrayList. As before, this triggers the generation of a new, unique regNum—the code in RegControl should not need to change if you've followed the instructions in Assignments 1, 2, and 3 correctly. But now, when once you've added the new Registrant to the ArrayList, `RegViewGUI`'s regNum combo box will need to be updated as well; more on this shortly.

e) A third JOptionPane (not shown) will need to be added to allow the user to change the existing registration number associated with a property to a new regNum.



This dialog will be needed when the user selects the 'Change Registration Number' button, which will call your existing `changePropertyRegistrant()` method to change the regNum of each of the properties displayed in the `propertiesPanel`. Again, more on this shortly.

Other dialogs may be needed to confirm that some event has taken place or is about to take place. But see the note at the end of this document about using such dialogs sparingly.

Collectively, your dialogs must be used in such a way that your application operates like any standard Windows app, with everything functioning reliably and predictably, where mouse clicks give logical, intuitive results, and there are NO SURPRISES…we haven't spent the semester building this application just to have it messed up by sloppy code in the final step.

f) `makeNewRegistrantFromUserInput()`, `getResponseToString()`, and `requestRegNum()` may be deleted, along with any references to these methods in your code, since their function is effectively supplanted by `JOptionPane` dialogs and the textfields and buttons of `RegViewGUI`.

(Dialogs simplify the IO process, so once you've excised the console-driven IO elements from `RegView`, you'll find the code in that class is much leaner. And of the Strings currently output by `RegView` will be made obsolete our graphical interface.)

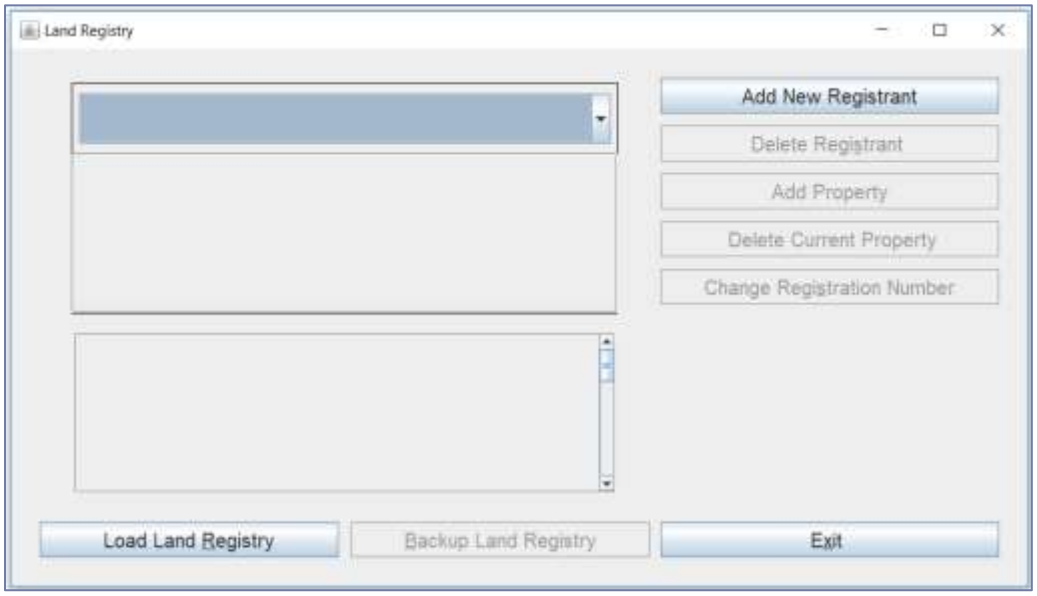
As a final comment before you begin making more direct changes to `RegViewGUI`, it should be noted that there are no general rules for structuring code in graphical applications. This is unfortunate, since GUI code tends to be messy by its very nature. For this reason, it is even more essential than ever to structure your code for readability and ease of maintenance. Take the following three pieces of advice as general guidelines when constructing graphical programs:

1) *Every major dialog window should have its own class. The exception is when pre-built dialogs exist, like the FileChooser in Lab 9 and the JOptionPane dialogs mentioned above.*

2) *Rather than attempt to load all the graphical components of each dialog in a single overlong block of code, break the code into modules such that each layout component gets loaded by a method dedicated to that task. See the code provided in Lab 9 as an example of this, e.g. the use of the getWestPanel() and getCenterPanel() in populating the BorderLayout in the WordSorter application.*

3) *Whenever multiple controls of the same type are loaded more than once into a panel or dialog—examples include a radio button, a check box, or a menu item, which almost never appear by themselves, but in a group—write a method to handle the common features of each component, and reuse this method for each new instance of that component, with each setable feature passed as a parameter to the method. For example, see the code for setting the four buttons in getWestPanel() in Lab 9's WordSorter.java, or the static makeBtn() method of module 10.2 in the slides.*

Since you only need to prove that your GUI front end works correctly and reliably to obtain marks for this assignment, you don't explicitly need to follow these guidelines. However, your coding will go faster, your program will function more



reliably, and you'll spend less time debugging the code, if you modularize your program according to these simple rules.

g) You should be able to run `RegViewGUI` as it is, even with nothing added yet. Whenever the application loads, two buttons will always be enabled: 'Add New Registrant', and 'Exit'—because no other actions are possible initially. (The `ActionListener` code for the Exit button is already provided, so you should be able to click this button to exit the app at any time.)

Start modifying the `RegViewGUI` code by adding the `ActionListener` that causes (1) the 'New registrant input' dialog box to appear (2) returns the Registrants full name, and uses it to instantiate a new `Registrant`, (3) calls `addNewRegistrant()` to load the new `Registrant` into the registrants ArrayList. But you're not done yet; see Section (h) below.

(As mentioned above, you can do all this using a modified `viewAddNewRegistrant()` method and calling it from `RegViewGUI`, or by making an instance of `RegControl` in RegViewGUI and using it to call `addNewRegistrant()`.)

Additionally, if the two Land Registry files exist on startup, you can enable the 'Load Land Registry' button and add an ActionListener to it so the two ArrayLists are loaded from their respective files whenever the 'Load Land Registry' button is clicked.

h) Whenever the registrants ArrayList is loaded or changed, as in (g) above this should cause the registrants combo box in the top left corner of the dialog to be reloaded with the list of current regNums. To see how this would work, replace the line

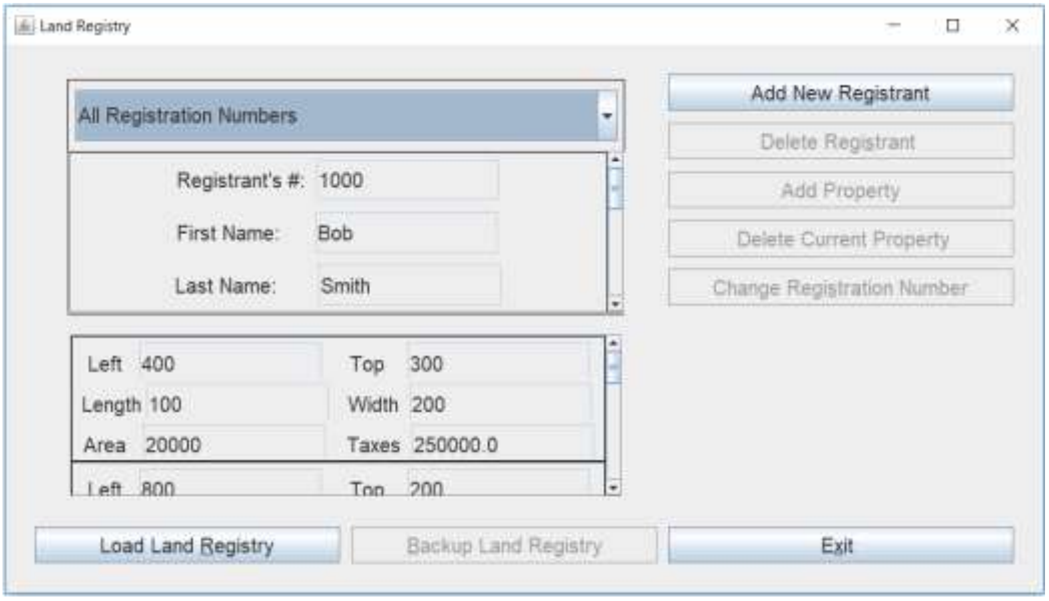
```
mainPanel.add(LoadRegistrantsPanel(null),
registrantConstraints);
```

near the top of RegViewGUI, with

```
mainPanel.add(LoadRegistrantsPanel(getRegTest
ArrayList()),registrantConstraints);
```

Note that `getRegTestArrayList()` returns an ArrayList of `Registrants` that you can use to test your code with—at least, in the early stages of development. Once you've satsified yourself that the sample ArrayList will be loaded in the scroll pane, rewrite the existing code so that any change to registrants ArrayList causes the combo box to be reloaded with the new ArrayList.

Another method, `getPropertyTestArrayList()`, is provided



to test the properties `JScrollPane`. This can be demonstrated by replacing the existing code with

```
mainPanel.add(LoadPropertyPanel(getPropertyTestArrayList()), propertyConstraints);
```

near the top of the program.

Note that the default setting for the combo box is to display *all* the registrants available in the scrollpane, along with *all* the properties—information obtained from `listOfRegistrants()` and `listOfProperties(0)` respectively. This is the situation that currently exists in the `RegViewGUI` code when you plug in the two test ArrayLists. (See the sample output below.)

But once a particular `regNum` is selected from the combo box, the situation changes—and you'll need to provide the code for it at this point. Now, the registrants scroll pane should be populated with the information from the Registrant whose regNum is selected, obtained via `findRegistrant()`. This is passed to `loadRegistrantsPanel()` as an ArrayList consisting of only that one item. At the same time, the `loadPropertyPanel()` method should be passed a list of all properties associated with that regNum, using `listOfProperties(reg.getNum())`—or nothing, if that registrant has no properties associated with it.

In short, selecting a specific `regNum` should trigger the loading of the `Registrant` information *and* all its associated properties. You'll need to modify the existing `RegViewGUI` code to make sure the program behaves accordingly.

Furthermore, any time a specific Registrant appears, the 'Delete Registrant' button should be enabled, so that the current Registrant can be deleted using `deleteRegistrant()`. And don't forget, all properties associated with that `Registrant` must be deleted from the properties ArrayList as well. Furthermore: once a registrant is deleted, that event should update the registrants ArrayList, which should automatically trigger reloading of the `regNum` list in the combo box…which in turn causes the default Registrant and Properties lists to be reloaded—as outlined above.

Also: whenever a `Registrant` is visible in the scroll panel, even if it currently has no property registered, you should be able to add a new property using that `Registrant`. So your code must also enable the 'Add New Property' button whenever a Registrant's number and information appear in the combo box and scroll menu. (See section (j) below for the details on adding an 'Add Property Dialog' to this application.)

Furthermore, any time a Property appears in the list of Properties, the 'Delete Property' and 'Change Registration Number' buttons should be enabled, to allow the user to execute that functionality as well.

And did I mention…anytime there's a registrant in the ArrayList, the 'Backup Land Registry' button should be enabled, to allow the two ArrayLists to be backed up to file.

And of course, the buttons that are enabled by one action—such as a Registrant being loaded—must also be disabled when that situation has ceased or reversed.

So you need to figure out what happens when objects and ArrayLists of objects are loaded and unloaded, and what resources will be available (in the form of what buttons will be enabled, and which disabled) starting from what happens when the ArrayList gets loaded or changed and working through step-by-step what resources need to be available, and when. *This needs to be mapped out in advance of coding*. But it also needs to be done in such a way so as to minimize the code, and hence the complications. Simply adding long lists of code to account for every possibility in each button's ActionListener is a recipe for disaster; the behaviour of your GUI interface needs to be thought out in advance.

To help simplify the business of enabling and disabling buttons, two utility methods are provided with the `RegViewGUI` code:

- `setButtonIn()` takes three parameters: the JPanel containing the buttons (either the `eastBtnPanel` or the `southBtnPanel`), the component number, and whether it is to be enabled or not. The component number for the two panels, along with fixed constants for ON and OFF, are declared at the top of RegViewGUI.

  For example, to disable the 'Delete Registrant' button, you'd write

  ```
  setButtonIn (eastBtnPanel,
     DELETE_REGISTRANT, OFF)
  ```

- Since it is laborious to turn groups of buttons ON or OFF one at a time, a second utility method, turnAllBtnsIn() has been provided to toggle all the buttons to the same state, except for a list of buttons you can exclude, which remain unchanged. For example, to turn all the buttons in the eastBtnPanel ON except for 'Add New Registrant' and 'Delete Registrant', you'd write:

  ```
  turnAllBtnsIn (eastBtnPanel, ON,
     ADD_NEW_REGISTRANT,
     DELETE_REGISTRANT)
  ```

  This should simplify your coding considerably. But remember: you still need to figure out which buttons will be enabled and disabled depending on which Registrant or Property resources are displayed in the GUI at any point in time.

i) Add two additional buttons to the `eastBtnPane` that allow the Properties to be sorted by size (i.e. area) and taxes, from largest to smallest. Of course, these buttons will only need to be enabled when there is more than one Property listed in the `propertiesPanel`. (Note that you'll need to change the `GridLayout` in `loadEastBtnPanel()` from 5 to 7.) The properties scroll pane should be reloaded whenever one of these selections is made.

j) You'll need to construct a special dialog to enter the new Property values, i.e. the length, width, left and top values, using the `regNum` of the registrant whose `regNum`/information currently appears in the combo box/scroll pane. This does not need to be elaborate…and please don't waste time lining up all the labels and text boxes exactly. But remember to cite any sources.

k) There are several ways to close a JFrame when you wish to exit. The recommended method is by calling either `this.dispose()` or `JFrame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE)`; I use the former method in the code provided. Whichever method you use, make sure your two Land Registry files are saved reliably before the application shuts down. As previously noted, using `System.exit(0)` is to be studiously avoided. In this application, using this command almost guarantees that your two Land Registry files will be lost or scrambled as a result.

l) And remember: when you've got your code working correctly, you'll need to remove the `RegView` class from your code: everything should be graphical at this point, so *everything* goes through `RegViewGUI` or Property entry Dialog, described in (j) above.

III. Notes, Suggestions, Tips, and Warnings

a) Students are reminded that:
- You should not need to use code/concepts that lie outside of the ideas presented in this course;
- You *must* cite all sources used in the production of you code according to the information provided in Module00. Failure to do so *will* result in a charge of plagiarism. The one exception to this rule is the information in the course notes themselves;
- Students must be able to explain the execution of their code. If you can't explain its execution, then it is reasonable to question whether you actually wrote the code or not. Partial marks, including a mark of zero and a charge of plagiarism, may be awarded if a student is unable to explain the execution of code that he/she presumably authored.
- You do NOT need to document your new code in RegViewGUI

IV. Submission Guidelines

Your code should be uploaded to Brightspace (via the link posted) in a single zip file created by:

1) In Eclipse, right click on the ***project*** name (`CST8284_20S_Assignment4`);
2) select 'Export' then select General >> Archive File, then click Next;
3) in the Archive File menu make sure *all* of the project subfolders are selected (including `.settings`, `src`, `bin`) and the 'Save in zip format' and 'Create directory structure for files' radio buttons are selected;
4) In the 'To Archive File' window, save your zip file to a location you'll remember. But make certain to name your zip file according to the following format, as outlined in Module 00:

Assignment4_Yourlastname_Yourfirstname.zip

including the underscores and capitals, but with *your* last and first name inserted as indicated. Failure to label your zip file correctly will result in lost marks.

Thing NOT to do:

- Aside from RegView, you cannot delete any of the classes from the earlier assignments. And while you can augment existing classes with methods, you must use all the existing classes for their originally-stated fashion.
- Do not prompt the user with a JOptionPane asking to confirm every decision. The last thing anyone wants after clicking a button that says 'Add New Registrant' is to be confronted with a dialog box that says 'Do you REALLY want to add a new registrant?' or even 'New Registrant entered…Click OK'. If the result of a decision is obvious, there's little need to confirm it with an OK/Cancel button each time. The only time this is truly necessary is when a decision will have irrevocable consequences, as for example when you reload the Land Registry from files, or delete a Registrant. Only then is it essential to prompt the user with a confirmatory dialog.
- As always, do not throw exceptions at the point where the data is entered by the user. You should mostly be throwing your exceptions in the setters, or when new information is to be loaded ArrayList (just before the (bad) data is stored), and catching them when the button to enter the data is clicked.
- If you add new components to a JPanel, rather than use the existing methods provided, be sure to clear the existing components first. Adding new components is like adding new objects to an ArrayList: they accumulate. In GUIs, this can lead to bizarre behaviour, as one component crowds out another, causing panels to stretch, and graphical objects to disappear altogether.

Corrections, Clarifications, and Addenda:

As required; check version updates for clarifications and corrections.

Assignment 4 Marking Guide

Requirement	Mark
File labelled and zipped correctly, as per instructions. The doc folder is supplied in your zip file and displays all the classes used in your project, hypertext-linked as required	**1**
Program loads correctly and will begin execution, without triggering exceptions. Note that failure to execute at this stage will seriously impact the rest of the marks in this assignment	**2**
The program works as described, functioning much as it did in earlier assignments, but with all console input and output replaced by a GUI interface instead. That is, all buttons function as expected, calling up appropriate dialogs, loading text, and exiting from those dialogs when this option is specified; and all TextFields return strings as they did in earlier assignments	**7**
Buttons are enabled and disabled appropriately during execution, so that it is not possible to select a feature which should not be available during normal operation	**7**
Properties buttons sort according to area and taxes	**3**
Add New Property dialog functions as expected	**6**
MINUS: late penalty; failure to cite sources; private information not kept secure through data hiding; diagnostic strings output to the console when only graphical output is allowed; abnormal termination, exceptions thrown under certain circumstances; unusual, abnormal and erratic features displayed during execution; your documentation should not include comments which are not part of the program itself (e.g. TODOs and commented-out code: these are for your purposes only: *if it's not part of the program, I don't need to see it).*	
TOTAL :	**26**