Inner Classes

Overview

In this lab, you will take some existing Swing GUI applications and refactor them to use inner classes.

Source folders

Student project: `StudentInnerClasses`
Solution project: `SolutionInnerClasses`

Roadmap

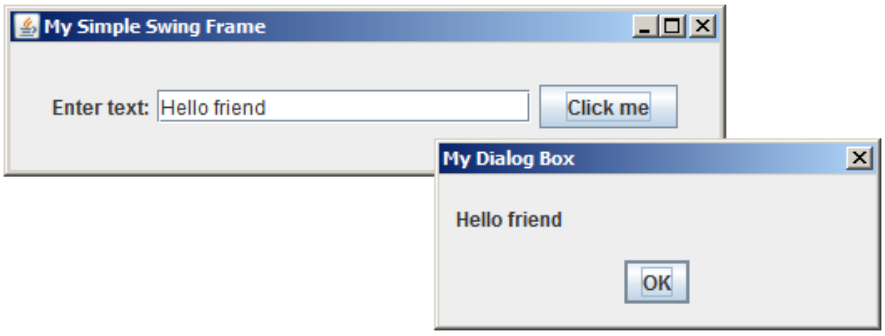
There are 3 exercises in this lab, of which the last exercise is "if time permits". Here is a brief summary of the tasks you will perform in each exercise; more detailed instructions follow later:

1. Defining an anonymous inner class
2. Defining several anonymous inner classes
3. Defining a regular inner class

Exercise 1: Defining an anonymous inner class

In the student project, open `SimpleSwingExample.java`. This application uses the Swing API to create a simple window on the screen. Run the application to see what it looks like:

Enter some text in the text box, and then click the *Click me* button. A message box appears, confirming the text you entered:



Now take a look at the code in `SimpleSwingExample.java`. It's not vital to understand all the details of Swing to complete this lab, but here are some observations to get you started:

- In Swing, a top-level window is represented by a `JFrame` object.
- You can add various components to a `JFrame`, such as a `JLabel`, a `JTextBox`, and a `JButton` in this example. (Technically speaking, you actually add these components to the *content panel* for the `JFrame`, which represents the client area inside the window).
- When the user clicks a `JButton`, an "action event" is generated. To handle this event:
    - Call the button's `addActionListener()` method, passing as a parameter an object of a class that implements the `ActionListener` interface. This interface has a single method named `actionPerformed()`.
    - Implement the `actionPerformed()` method, to handle the button click.

Currently, the `SimpleSwingExample` class implements the `ActionListener` interface directly. This is not the best approach - typically, it is better to place event-handling logic in separate dedicated classes.

Therefore, refactor the `SimpleSwingExample` class so that it doesn't implement `ActionListener` directly. Instead define an anonymous inner class that implements `ActionListener` at the point where it's needed, to handle the button click. There are TODO comments in the code, indicating where you need to make your changes.

Exercise 2: Defining several anonymous inner classes

In the student project, open `DialogBoxExample.java`. This application displays a main frame window that allows the user to display thee different types of dialog box:

Take a look at the code in `DialogBoxExample.java`. The code is similar to the previous example, except that there are now 3 buttons to handle. Note the following salient details in the code:

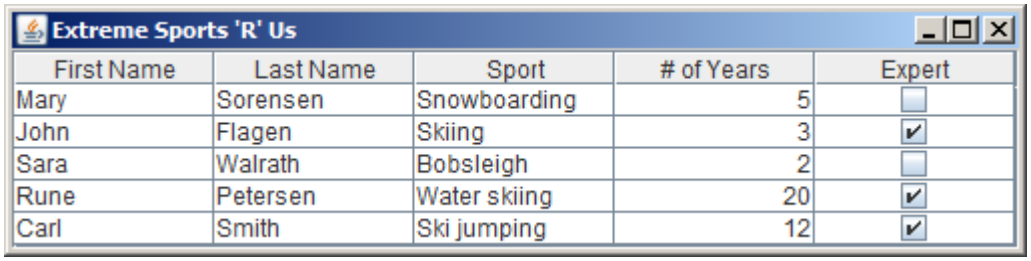
- The `DialogBoxExample` class implements `ActionListener` itself. The class has a single `actionPerformed()` method, which will be invoked to handle button click events for all the buttons.
- The `actionPerformed()` method has to determine which button was actually clicked, so that it can decide what to do.

This situation is clearly unsatisfactory. It is undesirable to have a single implementation of `ActionListener`, because we have three buttons and each requires its own specific handler code.

Therefore, refactor the `DialogBoxExample` class so that it doesn't implement `ActionListener` directly. Instead define three anonymous inner classes, to handle each button click individually. There are TODO comments in the code, indicating where you need to make your changes.

Exercise 3 (If time permits): Defining a regular inner classes

In the student project, open `TableSortDemo.java`. This application displays a frame that contains a `JTable`, which is a Swing data grid component:



First Name	Last Name	Sport	# of Years	Expert
Mary	Sorensen	Snowboarding	5	☐
John	Flagen	Skiing	3	☑
Sara	Walrath	Bobsleigh	2	☐
Rune	Petersen	Water skiing	20	☑
Carl	Smith	Ski jumping	12	☑

Note the following points in the application:

- If you click a column heading, it sorts the table on that column (it toggles between ascending/descending order).
- You can edit the last two columns.

Take a look at the code in `TableSortDemo.java`. Here's a brief summary of the important details (feel free to ask the instructor if you want to learn more about `JTable` in Swing):

- The `TableSortDemo` class extends `JPanel`. In other words, it's a custom panel that decides how to layout its constituent parts.
- The `TableSortDemo` constructor adds a `JTable` to the panel.
- The `JTable` object uses a `MyTableModel` object to define the data, column heading names, and other model-related information for the `JTable`.
- The `MyTableModel` class is currently defined as a top-level class (see the lower half of `TableSortDemo.java`).

Refactor the code, so that `MyTableModel` is defined as an inner/nested class within the `DialogBoxExample` class. You decide which mechanism to use.