Homework 3: Answers to Questions

Jade Pearl

CMSC 335-7381 Object-Oriented and Concurrent Programming

Professor Andrew Seely

**Question 1**:

For Swing components:

* JButton generates an ActionEvent which activates when the user clicks on a button in the GUI.
* JTextField generates an ActionEvent when the user types in the field and presses return.
* JComboBox generates ActionEvents when the user selects an item within the drop-down menu/combo box in the GUI.

All of these components would likely need an ActionListener to respond to these events generated by them.

**Question 2**:

Out of the interfaces JTable implements ([TableModelListener](https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/event/TableModelListener.html" \o "interface in javax.swing.event), [Scrollable](https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/Scrollable.html), [TableColumnModelListener](https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/event/TableColumnModelListener.html), [ListSelectionListener](https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/event/ListSelectionListener.html), [CellEditorListener](https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/event/CellEditorListener.html), [Accessible](https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/accessibility/Accessible.html), [RowSorterListener](https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/event/RowSorterListener.html)), only two of them are uniquely implemented from JTable and not implemented by the parent classes of JTable. These interfaces are Scrollable and Accessible.

The methods required by these interfaces implemented by JTable are:

* Required by Scrollable
  + Dimension getPreferredScrollableViewportSize()
  + Int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)
  + Boolean getScrollableTracksViewportHeight()
  + Boolean getScrollableTracksViewportWidth()
  + Int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)
* Required by Accessible
  + AccessibleContext getAccessibleContext()

**Question 3**:

Here are the differences between some of the layout managers in JavaFX:

* FlowPane
  + The nodes are consecutively placed and wrap within the boundaries set in the pane. They can flow vertically or horizontally depending on specification. Vertical flow wraps by height and horizontal wraps by width of the pane.
  + Components automatically reflow accordingly when the container is resized.
* GridPane
  + Components are placed in a grid-like structure of rows and columns.
  + When the container is resized, the components in each “cell” may resize to fill the available space or remain fixed to a size of specified by the programmer.
* AnchorPane
  + The components can be anchored to the top, bottom, left, right, or center of the pane. Components can be anchored in multiple places and anchors can contain more than one component.
  + When the container resized, the nodes maintain their position relative to where they are anchored.
* TilePane
  + Similar to FlowPane, but the nodes are in a grid where each tile or cell is the same size.
  + When the container is resized, the TilePane components adjust their positions and/or sizes to fit the available extra space based on the orientation of the layout (either vertical or horizontal).
* BorderPane
  + Operates on 5 regions: top, bottom, left, right, and center where components can be placed.
  + When the container is resized, by default the center region typically expands to occupy the extra available space while the other regions remain their specified size or adjust based on additional constraints set by the programmer.

**Question 4**:

1. In accordance with the given philosophers problem, this points out a potential issue we computer scientists know as a deadlock. A deadlock would occur when each philosopher picks up their left chopstick and infinitely waiting to pick up their right chopstick because the right chopstick is held by the philosopher’s neighbor.

There is nothing initially wrong with thinking for a while. But the problem becomes a deadlock issue when the philosophers have to pick up the left and then the right chopstick and do so simultaneously. No one is able to eat for a while or return the chopsticks before returning to thinking. They will no longer think and they will eventually starve because all of them are simultaneously waiting to pick up the other chopstick.

1. One possible solution to this may be to add a condition where the philosophers can only eat for a while if they can obtain both chopsticks simultaneously rather than pick one up at a time. If they cannot pick up both chopsticks, they return to thinking. The timing of each philosopher’s thinking would differ, so they would go for the chopsticks at different times. This ensures that a philosopher who has finished thinking at a certain time will pick up both chopsticks unless one is already in use.
2. This simultaneous pickup solution is not entirely starvation free. It may cause a philosopher to wait long periods of time before they can eat if at all depending on how long it takes them to think compared to the length of time it takes for their neighbors to think then eat.

This could be aided by adding a priority factor (sharing is caring or make it fair!). If a philosopher fails to obtain both chopsticks and eat, they can be given priority to eat the next time around. This can also be seen as a turn-taking approach because we can make the other philosophers prioritize a neighbor that has not eaten yet. For example, one philosopher can be marked as a lead who always eats first and waits until their neighbors have eaten again and this loops around the table in a similar manner.

**Question 5**:

The methods (and some characteristics) a class implementing the java.util.concurrent.locks.Lock interface are

* Void lock()
  + Mutual exclusion/blocking: if the lock is not acquirable, the thread becomes disabled and dormant until the lock is available.
* Void lockInterruptibly()
  + Has the same mutual exclusion and blocking as the lock() method.
  + Interruptible: can be interrupted if a thread is waiting for the lock.
* Condition newCondition()
  + Supports conditions that can be implemented by Lock to change the condition of a thread in the program.
* Boolean tryLock()
  + Automatically tries using the lock if available but will keep waiting rather than be blocked.
* Boolean tryLock(long time, TimeUnit unit)
  + Interruptible in a similar way that lockInterruptibly is interruptible while a thread is waiting for the lock.
* Void unlock()
  + Releases the lock.

**Question 6**:

When the JVM encounters a synchronized directive…

It may create synchronized blocks or methods for exclusive access to parts of the code. This includes acquiring and releasing object or class locks, thread blocking and unblocking at different times when trying to obtain locks, threads gaining exclusive access to parts of code when the required lock is obtained. The JVM contributes to concurrency security by preventing the concurrent use of sections by multiple threads, instead allowing one at a time to access.

**Question 7**:

The Lock interface allows more flexibility than synchronization because it allows the use of different locks for various purposes while synchronization allows only for certain locks or methods to be used/access at one time. Locks use lock() and unlock() methods and can be implemented across methods but you cannot do this with synchronization. Acquiring and releasing of locks is impossible when using synchronization as well.

**Resources**:

*Javanotes 6.0, Section 6.6 -- Basic Components*. (n.d.). Math.hws.edu. Retrieved November 30, 2023, from <https://math.hws.edu/eck/cs124/javanotes6/c6/s6.html>

*Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification*. (n.d.). Docs.oracle.com. Retrieved November 27, 2023, from <https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/concurrent/locks/Lock.html>

*Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification*. (n.d.). Docs.oracle.com. Retrieved November 25, 2023, from <https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/accessibility/Accessible.html>

*Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification*. (n.d.). Docs.oracle.com. Retrieved November 25, 2023, from <https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/JTable.html#method-summary>

*Java® Platform, Standard Edition & Java Development Kit Version 21 API Specification*. (n.d.). Docs.oracle.com. Retrieved November 25, 2023, from <https://docs.oracle.com/en/java/javase/21/docs/api/java.desktop/javax/swing/Scrollable.html>

*Lock framework vs Thread synchronization in Java*. (2019, February 27). GeeksforGeeks. <https://www.geeksforgeeks.org/lock-framework-vs-thread-synchronization-in-java/>

Venners, B. (1997, July 1). *How the Java virtual machine performs thread synchronization*.

InfoWorld. <https://www.infoworld.com/article/2076971/how-the-java-virtual-machine-performs-thread-synchronization.html#:~:text=Synchronized%20methods>

*Working With Layouts in JavaFX: Using Built-in Layout Panes | JavaFX 2 Tutorials and*

*Documentation*. (n.d.). Docs.oracle.com. Retrieved November 28, 2023, from <https://docs.oracle.com/javafx/2/layout/builtin_layouts.htm#CHDGHCDG>