**Homework 3**

1. (10 pts) What events do the following components generate:

* JButton

**A JButton creates a clickable button that is attached to an action listener, that triggers a new segment of code for a specific function. This often accompanied by other Java swing components to gather user input that is used for a new task. (java.awt.event.ActionListener)**

* JTextField

**A JTextField provides a text area with which an input can be given or an output displayed in the form of a string variable. This allows for a wide variety of input/output as long as the variable is parsed to and from a string. It can use a document listener to automatically submit input while typing. (javax.swing.event.DocumentListener)**

* JComboBox

**A JComboBox produces a drop-down menu of items populated by a string array. They are great for providing multiple, but limited input choices. It can use an item listener to implement an event upon index selection. (java.awt.event.ItemListener)**

**All three utilizes action events, action listeners, event listeners, etc.**

2. (10 pts) What methods does JTable implement which are required by the interfaces implemented by the JTable class beyond those interfaces implemented by the various parent classes of JTable?

**The JTable class extends the JComponent class and implements the following interfaces and methods:**

**From the JDK:**

**1) TableModelListener:**

**void tableChanged(TableModelEvent e);**

**2) Scrollable:**

**Dimension getPreferredScrollableViewportSize();**

**int getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction);**

**boolean getScrollableTracksViewportHeight();**

**boolean getScrollableTracksViewportWidth();**

**int getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction);**

**3) TableColumnModelListener:**

**void columnAdded(TableColumnModelEvent e);**

**void columnMarginChanged(ChangeEvent e);**

**void columnMoved(TableColumnModelEvent e);**

**void columnRemoved(TableColumnModelEvent e);**

**void columnSelectionChanged(ListSelectionEvent e);**

**4) ListSelectionListener:**

**void valueChanged(ListSelectionEvent e);**

**5) CellEditorListener:**

**void editingCanceled(ChangeEvent e);**

**void editingStopped(ChangeEvent e);**

**6) Accessible:**

**AccessibleContext getAccessibleContext();**

**7) RowSorterListener:**

**void sorterChanged(RowSorterEvent e);**

**REFERENCE:**

**https://docs.oracle.com/javase/7/docs/api/javax/swing/JTable.html**

3. (10 pts) Address how the differences among these various layout managers, focusing on their behavior as their container is resized:

1. FlowLayout

**This is the default Java swing layout. The components are aligned in a single row from left to right until the components can no longer fit within the container and move to a new row. Upon resizing, components retain size but rearrange to fit on rows.**

1. GridLayout

**This layout displays components in equal size cells aligned in specified number of rows and columns. It also includes optional parameters for horizontal and vertical gaps (spacing) between components. Their size will change with resizing, but their relative position in the container will not.**

1. BorderLayout

**This layout divides the container in five areas: top, bottom, left, right, and center in which components can be placed. All unused space is used in the center area. Like GridLayout, the size of components will change with their size, but their relative position will not.**

1. GridBagLayout

**Similar to GridLayout, but all the components have different sizes and their position is static relative to container. The components resize along with the container proportionally.**

1. Absolute Positioning (explain the basic steps required for this manager as well)

**The components use specific x and y coordinates for placement and do not change position or size upon resizing of the container.**

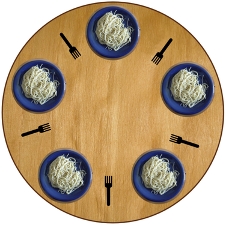
**Basic Steps:**

**- call the method setLayout(null) to set the container’s layout manager to null**

**- call the method setBounds() to specify the position and size of a GUI component**

**- call the method repaint() to get the container/components to repaint themselves**

4. (20 pts) (Ex 1.8.2) The dining philosophers problem was invented by E. W. Dijkstra, a concurrency pioneer, to clarify the notions of deadlock and starvation freedom. Imagine five philosophers who spend their lives just thinking and feasting. They sit around a circular table with five chairs. The table has a big plate of rice. However, there are only five chopsticks (in the original formulation forks) available, as shown in Fig. 1.5. Each philosopher thinks. When he gets hungry, he sits down and picks up the two chopsticks that are closest to him. If a philosopher can pick up both chopsticks, he can eat for a while. After a philosopher finishes eating, he puts down the chopsticks and again starts to think.



a. What is wrong with everybody doing the following - other than that the philosophers never get up from the table?

1. think for a while
2. get left chopstick
3. get right chopstick
4. eat for a while
5. return left chopstick
6. return right chopstick
7. return to 1

**The problem is that when all the philosophers grab the chopstick on the left, all the chopsticks have been taken, yet they still need a chopstick on the right to eat. As a result, none of the philosophers have the two chopsticks necessary for eating. Thus they all are stuck thinking and starving. This repeats if they all put the left chopstick down and grab the one on their right. In order for any philosophers to eat, some will have to wait while others eat.**

b. How can the above be fixed to avoid deadlocks?

**The solution to this problem is to introduce a waiter (monitor) to determine who has access to a pair of forks (locks) at any given time. This gives two philosophers (threads) the opportunity to eat (execute) while the others think (wait). Once the philosophers who were eating are done they then give their forks to the philosophers who were waiting patiently. There does need to be a scheduling system introduced however that ensures all the philosophers get an opportunity to use the chopsticks. For instance, a reservation system could be used that only our philosophers at a time can sit at the table as to avoid competition for the same chopsticks. Meaning after every change of state, one philosopher is dismissed from the table so the outlier can join for a total of four state changes before being switched out again.**

c. Is your solution starvation free? Literally!

**Yes. With the solution stated above, every philosopher gets a guaranteed opportunity to eat and no one goes hungry.**

5. (20 pts) What methods must a class implementing the java.util.concurrent.locks.Lock interface implement? Describe some of the expected characteristics of each of the methods of this interface?

**The locking interface provides exclusive access to a shared resource amongst multiple threads by implementing a lock to limit access to certain threads at certain times.**

**From the JDK:**

**void lock() – Acquires the lock.**

**void lockInterruptibly() – Acquires the lock unless the current thread is uninterrupted.**

**Condition newCOndition() – Returns a new Condition instance that is bound to this Lock instance.**

**Boolean tryLock() – Acquires the lock only if it is free at the time of invocation.**

**Boolean tryLock(long time, TimeUnit unit) – Acquires the lock if it is free within the given waiting time and the current thread has not been interrupted.**

**void unlock() – Releases the lock.**

**REFERENCE:**

**https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html**

6. (10 pts) Explain what the JVM does when it encounters a synchronized directive. Hint: consider carefully what is synchronized.

**When the JVM encounters a synchronized directive, the JVM acquires lock to assist with the coordination between multiple threads. A synchronized directive is used to make a method thread safe, by disallowing concurrent access at any given time. This is done before the execution of the method with which the lock is associated. The thread requesting access while another is synchronized is sent into a wait state.**

7. (10 pts) What happens when the JVM encounters a wait () call?

**The wait() call instructs the current thread to relinquish ownership of the object’s monitor and wait until another thread uses the notify() or notifyAll() calls to re-awaken the object monitor. This is so that the thread that is waiting can resume execution by regaining ownership of the monitor. This an important part of communication between threads that is necessary for synchronization.**

8. (10 pts) Describe the environment in which a wait () call is legal?

**A wait () call is legal when it is used inside a synchronized block. This must accompany the object for which the method is invoked.**

**Grading Rubric:**

|  |  |  |
| --- | --- | --- |
| **Attribute** | **Meets** | **Does not meet** |
| Problem 1 | **10 points** Lists the events associated with each provided component. | **0 points** Does not list the events associated with each provided component. |
| Problem 2 | **10 points** Lists the methods JTable implements.  Lists the methods which are required by the interfaces implemented by the JTable class beyond those interfaces implemented by the various parent classes of JTable. | **0 points** Does not list the methods JTable implements.  Does not list the methods which are required by the interfaces implemented by the JTable class beyond those interfaces implemented by the various parent classes of JTable. |
| Problem 3 | **10 points** Addresses the differences among the various layout managers.  Focuses on their behavior as their container is resized. | **0 points** Does not address the differences among the various layout managers.  Does not focus on their behavior as their container is resized. |
| Problem 4 | **20 points** Explains what is wrong with everybody doing the actions provided.  Explains how the actions be fixed to avoid deadlocks.  Explains if the solution provided is starvation free. | **0 points** Does not explain what is wrong with everybody doing the actions provided.  Does not explain how the actions be fixed to avoid deadlocks.  Does not explain if the solution provided is starvation free. |
| Problem 5 | **20 points** Explains what methods a class implementing the java.util.concurrent.locks.Lock interface must implement.  Describes some of the expected characteristics of each of the methods of this interface. | **0 points** Does not explain what methods a class implementing the java.util.concurrent.locks.Lock interface must implement.  Does not describe some of the expected characteristics of each of the methods of this interface. |
| Problem 6 | **10 points** Explains what the JVM does when it encounters a synchronized directive. | **0 points** Does not explain what the JVM does when it encounters a synchronized directive. |
| Problem 7 | **10 points** Explains what happens when the JVM encounters a wait () call. | **0 points** Does not explain what happens when the JVM encounters a wait () call. |
| Problem 8 | **10 points** Describes the environment in which a wait () call is legal. | **0 points** Does not describe the environment in which a wait () call is legal. |