**Homework 3**

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

* **JButton**

A JButton can generate an action listener or a mouse event. Since buttons are normally used in response to the user clicking on the button an action listener returns an actionPerformed message to the action listeners of that component. A JButton can also use an Item event which tells when an item has been selected or deselected and a Key Event for using keyboard strokes with the component. For example, if we wanted to make a login button that opened the next page we could create a JButton and an event for it like the following:

JButton button1 = new JButton(“Login”);

button1.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent evt) {

//…Action to do when button is clicked

}

});

* **JTextField**

The JTextField is often used to display a value, or to get a value from the user. Although JTextFields can implement ActionListener no action event is necessary to get the data from the user as it’s built into the class. However, for more difficult features you might want to use an ActionListener with the JTestField. For example, say you want to have an item that is added to a list after the user inputs the item and presses the enter key. This would be done as follows:

defaultListModel model = new defaultListModel();

JTextField input = new JTextField(“Type your input here and then press enter”);

JList model = new new JList (model);

input.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent evt) {

model.addElement(input.getText());

}

});

* **JComboBox**

A ComboBox is like the above components but it lets the user choose from several options. It can also be used with an Action Event to get the selected item and then do some action. For example, the snippe below shows how one could use the JComboBox to select a book and then display a corresponding image of that book.

String[] books = new String[] { “Lord Of The Rings”, “Redwall”, “Sound and the Fury”, “Absolute Java”};

JComboBox <String> selection = new JComboBox(books);

selection.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent evt) {

String value = (String) selection.getSelectedItem();

int digit = Integer.valueOf(value.replaceAll(“component N°","").trim());

String imageName = "image" + digit + ".png";

}

});

**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?**

Interfaces implemented by JTable are:

ImageObserver, MenuContainer, Serializable, EventListener, Accessible,

CellEditorListener, ListSelectionListener, RowSorterListener,

TableColumnModelListener, TableModelListener, Scrollable

Required Methods are:

CellEditorListener:

editingStopped(ChangeEvent e)

editingCanceled(ChangeEvent e)

RowSorterListener:

sorterChanged(RowSorterEvent e)

Accessible:

getAccessibleContext()

ListSelectionListener:

valueChanged(ListSelectionEvent e)

TableColumnModelListener:

columnAdded(TableColumnModelEvent e)

columnSelectionChanged(ListSelectionEvent e)

columnMarginChanged(ChangeEvent e)

columnMoved(TableColumnModelEvent e)

columnRemoved(TableColumnModelEvent e)

Scrollable:

getScrollableTracksViewportHeight()

getScrollableTracksViewportWidth()

getScrollableBlockIncrement(Rectangle visibleRect, int orientation, int direction)

getScrollableUnitIncrement(Rectangle visibleRect, int orientation, int direction)

getPreferredScrollableViewportSize()

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

1. **FlowLayout**

FlowLayout puts each component in a row in that components preferred size. If the container is too small for the component, then it’ll put the new component on a new row. The FlowLayout is easiest to use when the components need to be placed in a row fashion from right to left and once that row is filled it’ll make a new row and start there again. The components also take up just as much space as they need (has they have preferred size here)

1. **GridLayout**

GridLayout like the name implies places components in a grid. Each grid cell has equal size and the component can fill as much of the cell as necessary. When the GridLayout is resized, the cells will grow evenly with the size of the layout, so they take up all available space. The components stay in the cells and it will fit one component per cell.

1. **BorderLayout**

BorderLayout places everything in five regions. There is an East, West, North, South and Center region. When the layout resizes The Center region gets all the new available space and horizontal and vertical gaps can get added in between each region. The BorderLayout seems good for UI that are focused in the center of the window.

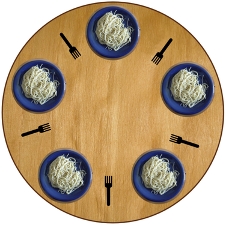
1. **GridBagLayout**

The GridBagLayout is a very versatile layout. The components are places in rows and columns and the sizes can vary and a single component can also take up more than one row or column. The resizing of the components is based on weights the program can assign to the individual components in the layout, so the layout can be programed to resize all the components automatically, and evenly if one wants to. It aligns the components vertically or horizontally based on the baseline.

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

If the components used are not affected by the containers size, font, look and feel, or language then it might be fine to use absolute positioning. However, for most cases not using a layout manager can cause problems when the container is resized. When you use absolute positioning the programmer is required to enter the size and position of each component so when the top-level container is resized the components often to not resize with the container (Therefore if you have a 20x20 button it will stay 20x20 no matter how big the container gets) This is why absolute positioning should only be used in certain cases where it is necessary.

**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**

It would be wrong for all the philosophers to do this at once because when the philosophers go to pick up the right chopstick at the same time gridlock will occur. This is because once each philosopher picks up the left chopstick on #2 all the chopsticks will be in a philosopher’s hand leaving none for a left chopstick on #3.

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

Deadlock can be avoided if the chopsticks are monitored. So, let’s say the philosophers have a mutex, in this case a waiter, that makes sure a philosopher can not pick unless the waiter give him permission. Basically, the waiter puts a lock on the chopsticks and the philosopher must ask the waiter for chopsticks when they get hungry. This way if two chopsticks are not available then the philosopher must wait until two are free otherwise the philosopher gets chopsticks and can start eating.

**c. Is your solution starvation free? Literally!**

No, this solution is not starvation free. Although it will lessen deadlock a philosopher might wait a long time for chopsticks to be available and thus become ‘starving’. To handle starvation, we would need to add a few more things to the last part. We could add that a chopstick is dirty (already used by that philosopher) or it can be clean (not yet used by that philosopher) This way when a chopstick is requested by another philosopher he can look and see if the chopstick is dirty or clean, if it is clean he keeps it, and if it is dirt he gives it away. This way there is a priority over who has eaten and who is 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?**

Methods the class must implement are:

***lock():*** Acquires the lock. If the current thread becomes disable until the lock is acquired.

***lockInterruptibly():*** Acquires a lock unless the thread is interrupted. If the lock is not available the thread becomes dormant until the lock is acquired by the current thread, or some other thread interrupts the current one and supports interruption of lock acquisition.

***tryLock():*** Acquires the lock only if the lock is free for acquisition. Returns true if lock is acquired and false if not. Ensures the lock acquired was unlocked.

***tryLock(long time, TimeUnit unit):*** Acquires the lock if it becomes free during the given waiting time and the thread has not been interrupted. If lock is acquired method returns true, otherwise it returns false and the disables the thread until the lock is acquired, another thread interrupts the current thread, or the specified waiting time elapses.

***unlock():*** Releases the lock

***newCondition():*** Returns a new condition instance that is bound to the current Lock instance. The lock must be held by the current thread before waiting on the condition.

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

The synchronized keyword works on the object (instance) level, so any thread trying to access that instance will need a lock to access it. In all cases the synchronized directive helps make sure the thread is safe and makes sure only one thread can access the synchronized block or method at a time. So, although this is the same what is being lock changes depending on what is synchronized. For a synchronized block the variable or object is explicitly provided. For example, for the code:

synchronized (foo) { …. }

the variable foo is what is being synchronized in this case.

class myClass {

synchronized void method1 {

}

For the synchronized method above then the lock is to access the ‘this’ variable of the method, however if it is static method then the lock is placed on the Class object instead and thus change what the lock is needed to access.

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

When the JMV encounters a wait() call the current thread releases control of the lock and only awakens when a call to notiy() or notifyAll() is called and then the thread must wait until it re-obtains ownership of the lock monitor to continue execution.

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

A call to wait() is only legal if the current thread is the owner objects monitor (lock) this can happen when a synchronized instance method has been executed of the given object, or a synchronized block was executed on the given object, or by executing a synchronized static method on the Class.

**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. |