**Threads and Swing**

The Swing classes in Java are not threadsafe; if you access a Swing object from multiple threads, you run the chance of data corruption, hung GUIs , and other undesirable effects. To deal with this situation, you must make sure that you access Swing objects only from one particular thread.

**Swing Threading Restrictions**

A GUI program has several threads. One of these threads is called **the event dispatching thread**. This thread executes all the event-related callbacks of your program (e.g., the actionPerformed() and keyPressed() methods in our typing test program). Access to all Swing objects must occur from this thread.

The reason for this is that Swing objects have complex inner state that Swing itself does not synchronize access to. A JSlider object, for example, has a single value that indicates the position of the slider. If the user is in the middle of changing the position of the slider, that value may be in an intermediate or indeterminate state; all of that processing occurs on the event-dispatching thread. A second thread that attempts to read the value of the slider cannot read that value directly since by doing so the thread may read the value while the value is in its intermediate state. Therefore, the second thread must arrange for the event-dispatching thread to read the value and pass the value back to the thread.

Note that it’s not enough for our second thread simply to synchronize access to the JSlider object. The internal Swing mechanisms aren’t synchronizing access, so the two threads still simultaneously access the internal state of the slider. Remember that locks are cooperative: if all threads do not attempt to acquire the lock, race conditions can still occur.

It may seem like this restriction is overkill: the value of a JSlider is a single variable and could simply be made volatile. Actually, that's not the case. The value of things within Swing components can be very complex. Many Swing components follow a model-view-controller design pattern, and accessing those components from one thread while the model is being updated on the event-dispatching thread would be very dangerous. Even the simplest of Swing components contain complex state; it's never acceptable to call any of their methods from a thread other than the event-dispatching thread.

Consequently, all calls to Swing objects must be made on the event-dispatching thread. That's the thread that Swing uses internally to change the state of its objects; as long as you make calls to Swing objects from that thread, no race condition can occur. Four exceptions to this rule are:

- Swing objects that have not been displayed can be created and manipulated by any thread. That means you can create your GUI objects in any thread but once they've been displayed, they can be accessed only on the event-dispatching thread. A GUI object is displayed when the show() method of its parent frame is called.

- The **repaint()** method can be called from any thread.

- The **invokeLater()** method can be called from any thread.

- The **invokeAndWait()** method can be called from any thread other than the event-dispatching thread.

**Using invokeLater() and invokeAndWait()**

The invokeLater() and invokeAndWait() methods allow you to define a task and ask the event-processing thread to perform that task. If you have a non-GUI thread that needs to read the value of a slider, for instance, you put the code to read the slider into a Runnable object and pass that Runnable object to the invokeAndWait() method, which returns the value the thread needs to read.

**Which invokeLater( )? Which invokeAndWait( )?**

Java defines the invokeLater() and invokeAndWait() methods in two different classes: javax.swing.SwingUtilities and java.awt.EventQueue. This is due to historical reasons, and you can use whichever class you like. The methods are identical. The invokeLater() method of the SwingUtilities class simply calls the invokeLater() method of the EventQueue class, so they are functionally identical; the same is true of the two invokeAndWait() methods.

For the most part, the invokeAndWait() method looks similar, but it has three important semantic differences. First, the invokeLater() method runs asynchronously at some time in the future. You don't know when it will actually run. On the other hand, the invokeAndWait() method is synchronous: it does not return until its target has completed execution. As a rule of thumb, then, you should use the invokeAndWait() method to read the value of Swing components or to ensure that something is displayed on the screen before you continue program execution. Otherwise, you can use the invokeLater() method.

The second difference is that the invokeAndWait() method cannot itself be called from the event-dispatching thread. The thread running the invokeAndWait() method must wait for the event-dispatching thread to execute some code. No thread, including the event-dispatching thread, can wait for itself to do something else. Consequently, if you execute the invokeAndWait() method from the event-dispatching thread, it throws a java.lang.Error. That causes the event-dispatching thread to exit (unless you've taken the unusual step of catching Error objects in your code); in turn, your entire program becomes disabled.

The third difference is that the invokeAndWait() method can throw an InterruptedException if the thread is interrupted before the event-dispatching thread runs the target, or an InvocationTargetException if the Runnable object throws a runtime exception or error.

If you have code that you want to take effect immediately and that might be called from the event-dispatching thread, you can use the SwingUtilities.isEventDispatchThread() method to check the thread your code is executing on. You can then either call invokeAndWait() (if you're not on the event-dispatching thread) or call the Swing methods directly.