|  |
| --- |
| http://www2.sys-con.com/itsg/virtualcd/java/images/top.jpg |
|

|  |  |  |
| --- | --- | --- |
| http://www2.sys-con.com/itsg/virtualcd/java/images/side.jpg | |  | | --- | | Swing, Threads, and Events  For a More Responsive GUI  *I normally spend a few hours every week reading the discussion forums at the Java Developer Network (http://forums.java.sun.com) and find that one of the most common problems is Swing and how it works in a multithreaded environment. I've helped a number of developers in this area when they find that their application stops responding, behaves erratically, or, the most common problem, runs very slowly.*  The goal of this article is to explain some of the difficulties developers face when they're new to the Swing API and hopefully help you avoid some of the pitfalls and eliminate potential bugs that are difficult to remove once the application has been completed.  **How Swing Works** Unlike other graphical APIs, Swing runs off a single thread model. This single thread processes both the actual drawing of the GUI as well as any events that are received from the user interface. This means that the developer must pay special attention to the handling of anything related to the GUI as well as how long it takes to process an event.  Failure to take this into account can result in the application feeling sluggish or appearing to be locked up. Even worse, the application could accidentally throw an exception in the event thread, causing the entire application to behave in a strange manner.  **Single Thread** The entire Swing API is designed to run in one thread. If that single thread is blocked or slowed down, the entire GUI will slow down or stop responding. To avoid this, the application needs to do most of its work in a separate thread and interact only with the event thread when it's processing an event or when it needs to update the GUI.  Fortunately, most of the Swing API is already designed to do this as long as the application follows the basic design of the API. It's not difficult to avoid some of the pitfalls, but this becomes more difficult when dealing with events.  When a Swing application receives an event, all the objects that have registered themselves as listeners of that event receive it. For example, when a JButton is clicked, it creates an ActionEvent that's passed to each of the listeners that subscribe to that event. The method actionPerformed is then called sequentially on each listener to that event. There's no guarantee that listeners will be called in any order, but it is guaranteed they will all receive it. This situation is illustrated in Figure 1. The JButton fires off an action event to each of its listeners. Listener1 will receive the event and completely handle it before Listener2 or 3 will even know the event occurred. Listener2 will then receive the event and complete its handling of the event before the event is handed off to Listener3.  http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/fig1.jpg  One way a problem can be created is when a listener behaves poorly and does not return in a reasonable amount of time; for example, when a listener is making a call that takes an unusual amount of time (perhaps loading a file from disk, accessing a database, etc., and that code is placed either directly or indirectly in the event thread).  For example, [Listing 1](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s1) has an actionPerformed method that will take a long time to return. If we run this example, when the JButton is clicked the GUI won't respond until the wait method has completed. This is because our example is running in the event thread and the entire application is on hold waiting for this code to finish.  **Not Thread-Safe** Another common situation is updating the GUI from a thread other than the Event thread. This situation is exactly the opposite of the one listed earlier. [Listing 2](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s2) shows a basic situation where we have more than one thread in the application and the wrong thread is attempting to update the GUI. While this code may look perfectly harmless (and in fact, in a simple situation like this, probably is harmless), it's a recipe for disaster.  Since Swing is designed as a single threaded model, the API is not synchronized and therefore not thread-safe. This means we can have a nasty collision in the GUI if we attempt to update the elements from any thread other than the event thread. Doing so can cause data corruption, exceptions to be thrown, and a host of other nasty problems that are extremely difficult to debug. To avoid this problem, always update the GUI via the event thread.  **The Event Thread** If we can't work in the event thread, how does the application do anything? Whenever we have a situation where we know that an event is going to take a long time to be processed, place it in a worker thread.  [Listing 3](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s3) is a rewrite of Listing 1, so instead of blocking the event thread until the waiter method completes, it constructs an anonymous inner class that extends java.lang.Thread, which will call the waiter method for us. It then starts this anonymous thread and returns. In this version of the code, the event will no longer wait until the waiter method has completed, but will return as soon as the thread is initialized and started. This results in a much faster response to the event and the GUI will be "snappier."  If we have a situation where the code that is being called by the event is going to be called frequently from multiple locations, we don't want to rewrite the anonymous class into every location. One way to avoid this duplication of code is to create an inner class that can be instantiated whenever it's needed without duplication. As can be seen in [Listing 4](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s4) , the event instantiates our WorkerThread and starts it. It can then instantiate another copy of this thread from any other location in the class as needed.  Another common situation is where we have a piece of code that is going to take a long time to execute and will only be called from a worker thread. In this situation it may be more advantageous to move the entire method into the worker thread as shown in [Listing 5](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s5) . This allows us to completely separate the logic from the GUI. This worker thread can either be in a separate class file or defined as an inner class, as in the example.  Each of these three examples will allow us to return the event thread in a timely manner and process the actual work in a separate thread, thereby allowing the GUI to continue receiving events, etc.  **Multiple Threads** As applications become more complex, they inevitably require multiple threads to run smoothly. This is in direct contradiction to the design of the Swing API. To get around this disparity Sun has built a couple of methods into the AWT and Swing APIs to allow you to use multiple threads with the single thread model. The basic concept behind them is to have all your GUI-related instructions run in the event thread. To do this, you need to insert them into the event queue. The following two methods are available as static methods in the javax.swing.SwingUtilities class and the java.awt.EventQueue.  *invokeLater(Runnable r)* causes r.run() to be executed asynchronously on the AWT event dispatching thread. This thread will be processed in turn once it has reached the top of the event queue. Even if this method is called from the event dispatching thread, it will still be deferred based on the items in the queue.  [Listing 6](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s6) shows an example of how to use this method. It should be noted that if an exception occurs in the Runnable object placed in the event queue, it will cause the event queue thread to fail as opposed to the thread that placed it in the queue. If there is a chance of an exception being thrown, it should be caught and handled as opposed to letting it kill the event thread.  *invokeAndWait(Runnable r)* causes r.run() to be executed synchronously on the AWT event dispatching thread. This method call will block until all pending events have been completed and the run() method of r has been executed.  [Listing 7](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s7) shows an example of how this method works. As can be seen from the example, it throws two types of exceptions. InterruptedException is thrown if the current thread is interrupted. The second exception, InvocationTargetException, is a wrapper exception that will be thrown if our Runnable throws an exception. The actual exception thrown by the Runnable will be contained inside the InvocationTargetException.  By utilizing these two methods, it's possible to have a multithreaded application interact either asynchronously or synchronously with the Event Queue and thereby adhere to the single thread design.  **Swing Worker** Sun has developed a utility class that performs the function listed earlier as well as adding some additional features that can be useful. SwingWorker is a class you extend off of that allows you to place your time-intensive code in a separate thread that gives you quite a bit of flexibility and control. While this class may not be useful in all situations, it's a lot more useful than constantly re-creating the functionality it provides.  The SwingWorker class is not part of the standard Java API and must be downloaded from Sun's Web site separately and compiled. There is a link to it in the references section at the end of the article.  [Listing 8](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s8) starts off by extending SwingWorker. The only method that we must define is construct(). As can be seen in the example, we have created a loop inside the construct method that will sleep for 100 milliseconds each iteration. Our JFrame instantiates a button that we have attached an ActionListener to. When the button is pushed, our ActionListener creates an instance of our worker class and calls its start method. The start method executes the run method of the parent class SwingWorker that, among other things, calls our construct method. If you explore the SwingWorker class, you'll see that inside the run method our construct method blocks the thread until it's complete.  The SwingWorker class provides us with some additional functionality above and beyond separating our work from the Event Thread. First, the construct method returns an object that we can then retrieve from the worker once the thread has completed. If we attempt to retrieve the value before our worker thread has completed, the get will block until the thread is done.  When the SwingWorker has completed, it will call the finished method that does nothing unless we extend it. The nice thing about the finished method is that it is executed in the event thread as opposed to the worker thread, thus we can execute GUI-related code inside it. As can be seen in our example, we show a message dialog letting the user know we have completed the task.  Now what if we want to interrupt our worker before it has completed its task? The SwingWorker class also has functionality built in to handle this as well. When we want to stop the worker we call the SwingWorker.interrupt() method that will stop the thread dead in its tracks. Note that the finished method will not be called if we interrupt the thread.  **Multi-Threaded GUI Construction** This topic is a bit on the controversial side of Swing programming. The reason behind it is simple. It's easy to create a JFC application that behaves erratically by using threads in the construction. However, that being said, it is possible to construct a JFC application by using threads in the construction.  Why would anyone do this? If we have an application that has several panels, a couple of toolboxes, and a few other complex widgets, then the start-up time can be horrific. A perfect example of this situation is one of my favorite editors, jEdit. In my opinion this is one of the finest editors out there but it suffers from a common problem: it takes forever to get to a working state. Once the application is running it performs beautifully, but be prepared to wait for it to get to that state.  While not every application can benefit from what I am about to describe, it can speed up the initialization of an application and thus give the user the impression of a snappy response. The negatives of this approach are twofold:   * If care is not taken in managing the construction of the GUI, the results can be unpredictable at best. * Your application could easily consume all of the available CPU on the user's machine until initialization is complete. If we succeed in reducing the time to almost nothing, this is not a major issue; if the initialization still takes a while, it's possible it will cause the user's OS to respond poorly.   Considering all this, multithreading the initialization routines of a GUI can speed up its start time significantly. [Listing 9](http://www2.sys-con.com/itsg/virtualcd/java/archives/0812/zarra/index.html#s9) shows a stripped down example of a multithreaded GUI initialization.  In this example, the threading is occurring in the JPanels that will be going into the JFrame. Each of these panels, upon construction, immediately increments the static counter in the JFrame. This counter lets the JFrame know when all the threads have completed. After incrementing the counter, each JPanel then fires up a thread passing itself as the Runnable object. Finally, the constructor starts the thread that it just initiated and then returns. This allows the JFrame to initialize all four of the JPanels very quickly and, in a real-world situation, all of these JPanels would be busy initializing themselves while the JFrame is off doing other work (initializing listeners, etc.).  When the JFrame is ready to actually display it checks and waits for the JPanels to finish their own initialization. In each of the JPanel's run methods, the final method call is to decrement the counter. When all the JPanels have finished, the counter will be back down to zero and the JFrame then knows that it can pack everything and display itself.  While this example is simplistic, it does show us some rules around creating a multithreaded initializa-tion routine:   * ***Make sure none of the objects that are threaded have to interact with each other:*** Otherwise you may have an object trying to access a resource that has not been initialized yet (i.e., trying to listen to an object that has not been constructed yet). * ***Make certain that all the threads have completed before you validate the GUI:*** Calling setVisible(true), pack(), and validate() will all validate the GUI and line everything up. Calling this prematurely could cause your GUI to behave inappropriately. * ***Launching threads within threads is probably not a good idea:*** Unless you are very careful with the threads, this is likely the shortest path to ruin. Avoid doing this unless you are sure about your thread interactions.   **Conclusion** While the examples above cause our application to be more complicated, they also allow the GUI to be more responsive to the user. In the end these tips have to be weighed against the complexity of the GUI you are writing. If your application has a very simple GUI with a small chance of improper interactions or collisions, chances are you don't have to worry too much. However, as an application gets more complex (as they invariably do), threading becomes more important. Making a habit of watching thread interactions in your GUI can save you from a massive rewrite later when the application has outgrown itself.  **Acknowledgment** A special thanks to Jon Svede for all of his hard work reviewing and commenting on the drafts of this article.  **References**   Using a Swing Worker Thread: <http://java.sun.com/products/jfc/tsc/articles/threads/threads2.html>   Threads and Swing: <http://java.sun.com/docs/books/tutorial/uiswing/overview/threads.html>   High Performance GUIs with the JFC/Swing API: <http://developer.java.sun.com/developer/community/chat/JavaLive/2002/jl0423.html>  **Author Bio** Marcus S. Zarra is the director of technology at Extraquest Corporation in Denver, Colorado. He began developing professionally in 1985 for school systems and then for legal firms in Phoenix, Arizona. Marcus has been a Java developer since 1996, working in the banking, telecommunications, and insurance industries. [mzarra@extraquest.com](mailto:mzarra@extraquest.com)  http://www2.sys-con.com/itsg/virtualcd/java/images/SourceCode.jpg  **"Swing, Threads, and Events"  Vol. 8, Issue 12, p. 39**    **Listing 1**  public void actionPerformed(  ActionEvent evt) {  //Calling a method that takes a long  //time to return  waiter();  }  private void waiter() {  //I am a method that takes a long  //time to return  try {  Thread.sleep(60000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  **Listing 2**  public ExampleClass extends JFrame  implements Runnable {  private JTextField example1;  private JTextField example2;  public ExampleClass() {  getContentPane().setLayout(  new FlowLayout());  example1 = new JTextField(  "Field 1");  example2 = new JTextField(  "Field 2");  getContentPane().add(example1);  getContentPane().add(example2);  Thread t = new Thread(this);  t.start();  }    public void run() {  //I am not the event thread,  //I should not be updating the GUI  example1.setText("Updated");  }  }  **Listing 3**  public void actionPerformed(  ActionEvent evt) {  Thread t = new Thread() {  public void run() {  waiter();  }  };  t.start();  }  private void waiter() {  //I am a method that takes a long  //time to return  try {  Thread.sleep(60000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  **Listing 4**  class WorkerThread extends Thread {  public void run() {  waiter();  }  }  public void actionPerformed(    ActionEvent evt) {  WorkerThread wt = new WorkerThread();  wt.start();  }  private void waiter() {  //I am a method that takes a long  //time to return  try {  Thread.sleep(60000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  **Listing 5**  class WorkerThread extends Thread {  public void run() {  //I am a method that takes a long  //time to return  try {  Thread.sleep(60000);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  public void actionPerformed(  ActionEvent evt) {  WorkerThread wt = new WorkerThread();  wt.start();  }  **Listing 6**  Runnable example = new Runnable() {  public void run() {  System.out.println(  "Example running from " +  Thread.currentThread());  }  };  SwingUtilities.invokeLater(example);  System.out.println("This probably will "  +"be executed before the example "  +"thread");  **Listing 7**  Runnable example = new Runnable() {  public void run() {  System.out.println(  "Example running from " +  Thread.currentThread());  }  };  try {  SwingUtilities.  invokeAndWait(doHelloWorld);  } catch (InterruptedException e) {  e.printStackTrace();  } catch (InvocationTargetException e) {  e.printStackTrace();  //Wrapper for an exception that was  //thrown from our example  }  **Listing 8**  public class MyWorker  extends SwingWorker {  JFrame parent;    public MyWorker(JFrame p) {  parent = p;  }    public Object construct() {  for (int i = 0; i < 10; i++) {  try {  Thread.sleep(100);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  return null;  }  public void finished() {  JOptionPane.showMessageDialog(  parent, "I'm finished!");  }  }  public class MyFrame extends JFrame  implements ActionListener {  private JButton \_pushMe;  private JButton \_stopMe;  private MyWorker myWorker    public MyFrame() {  JPanel buttons = new JPanel(  new FlowLayout());  \_pushMe = new JButton("Push Me");  \_stopMe = new JButton("Stop Me");  \_pushMe.addActionListener(this);  \_stopMe.addActionListener(this);  buttons.add(\_pushMe);  buttons.add(\_stopMe);  getContentPane().add(buttons,  BorderLayout.NORTH);  }    public void actionPerformed(  ActionEvent evt) {  if (evt.getSource() == \_pushMe) {  myWorker = new MyWorker();  myWorker.start();  } else {  if (myWorker != null) {  myWorker.interrupt();  }  }  }  }  **Listing 9**  public class ThreadedFrame extends  JFrame {  private static int initCounter = 0;    private JTabbedPane \_tabbedPane;    public ThreadedFrame() {  getContentPane().setLayout(  new BorderLayout());  \_tabbedPane = new JTabbedPane();  getContentPane().add(\_tabbedPane,  BorderLayout.CENTER);  \_tabbedPane.addTab("Tab One",  new TabOne());  \_tabbedPane.addTab("Tab Two",  new TabTwo());  \_tabbedPane.addTabe("Tab Three",  new TabThree());  \_tabbedPane.addTab("Tab Four",  new TabFour());    while (initCounter > 0) {  try {  Thread.sleep(50);  } catch (InterruptedException e) {  e.printStackTrace();  }  }    pack();  setVisible(true);  }  public synchronized static final void  incCounter() {  initCounter++;  }  public synchronized static final void  decCounter() {  initCounter--;  }  }  public class TabOne extends JPanel  implements Runnable {  public TabOne() {  ThreadedFrame.incCounter();  Thread t = new Thread(this);  t.start();  }  public void run() {  //Initialize the GUI code here  ThreadedFrame.decCounter();  }  }  public class TabTwo extends JPanel  implements Runnable {    public TabTwo() {  ThreadedFrame.incCounter();  Thread t = new Thread(this);  t.start();  }    public void run() {  //Initialize the GUI code here  ThreadedFrame.decCounter();  }  }  public class TabThree extends JPanel  implements Runnable {    public TabThree() {  ThreadedFrame.incCounter();  Thread t = new Thread(this);  t.start();  }    public void run() {  //Initialize the GUI code here  ThreadedFrame.decCounter();  }  }  public class TabFour extends JPanel  implements Runnable {  public TabFour() {  ThreadedFrame.incCounter();  Thread t = new Thread(this);  t.start();  }    public void run() {  //Initialize the GUI code here  ThreadedFrame.decCounter();  }  }  All Rights Reserved  Copyright ©  2004 SYS-CON Media, Inc.    E-mail: [info@sys-con.com](mailto:info@sys-con.com)  Java and Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. SYS-CON Publications, Inc. is independent of Sun Microsystems, Inc. | |