**Threading with Swing**

You may have written a GUI-enabled application using a framework such as Swing, and never even given a thought to threads. And for some types of application, you pretty much don't have to. If everything in your application happens directly in response to a button click or other Swing event, and your application never "does things with threads" (it never sleeps, spawns another thread or process, "does something in the background" etc), then be it by good design or good fortune, it may be behaving correctly. But there are times, particularly if your application*does* want to "do something with threads", when you need to understand how threading works with respect to the GUI system.

As you'll know if you've programmed with GUI systems such as Swing, everything generally happens in response to **events**. You define an**event handler**, which is effectively the code you would like to be called when something happens. For example, we can define a button like this:

JButton b = new JButton("Click me");

b.addActionListener(new ActionListener() {

**public void actionPerformed(ActionEvent e) {**

**// Code will be performed when button clicked**

**}**

});

To our program, the call to our method just "comes out of nowhere". But in reality, it must come from *somewhere*. That is: there must be some particular *thread* that's calling into our method. Well, there is, and it's quite a well-defined thread.

**The event dispatch thread**

The thread making calls to our event handlers is the **event dispatch thread**. This is a special thread that the GUI system sets up for performing UI tasks. Essentially, all user interface code will be executed by this special thread. Having a single designated thread handling the entire UI avoids a lot of issues that would occur if we tried to allow, say, different event handlers to be called by arbitrary threads.

In most Swing applications, if you've not thought about it and done anything special to use other threads, then practically all of your application will probably happen in the event dispatch thread. What you might not have thought about is whether it's all happening *correctly*...

**Proper threading behaviour of GUI applications**

There are essentially two rules of thumb that you need to remember:

* always **manipulate** your user interface **from the event dispatch thread** (with one or two safe exceptions);
* **never block or delay** the event dispatch thread— in other words, *never* call methods such as Thread.sleep(), Object.wait(),Condition.await() inside an event handler.

In more detail, these two guidelines have a few implications:

* all tasks that we perform **inside an event handler** should be **instantaneous**; we should *not* perform a long-running task (such as making a database query) or call [Thread.sleep()](http://www.javamex.com/tutorials/threads/sleep.shtml) or [Object.wait()](http://www.javamex.com/tutorials/synchronization_wait_notify.shtml) or similar calls directly from an event handler;
* to perform such **long-running tasks**, we need to arrnage for them to happen **in another thread**, e.g. by [starting a new thread](http://www.javamex.com/tutorials/threads/thread_runnable_construction.shtml) specially for the task;
* we need to be wary of the more "subtle" library calls that might cause our thread to wait: if we use one of the concurrency utilities such as a [ThreadPoolExecutor](http://www.javamex.com/tutorials/threads/ThreadPoolExecutor.shtml), we need to make sure we use a variant that won't ever block waiting for room in the queue; other slightly more obvious no-nos are awaiting a latch, joining another thread, calling Future.get()... essentially *anything* that "waits" for something cannot be called inside an event handler;
* if we're in our other thread and need to update the UI (e.g. to report progress to the user), we generally need to arrange for that update code— and *only* the update code— to happen in the event dispatch thread;
* by "manipulating the UI", we mean calling methods on or changing the state of any Swing components but also modifying **any objects they *depend* on** such as table models, cell renderers etc; **firing events** must also happen in the event dispatch thread.

**How do you make something happen in the event dispatch thread?**

So far, we've glibly said that in certain cases we must run something on the event dispatch thread. But how do we actually do that? On the next page, we look at the special method [SwingUtilities.invokeLater()](http://www.javamex.com/tutorials/threads/invokelater.shtml) which provides us with this functionality.

**Threading with Swing (ctd):  
SwingUtilities.invokeLater()**

In our introduction to [threading with Swing](http://www.javamex.com/tutorials/threads/swing_ui.shtml), we said that any updates to the user interface must happen on the **event dispatch thread**. So from any other thread— in practice, that means code that *isn't* called directly from an event handler— we must specifically arrange for our GUI update code, and generally *only* that code, to be called on the event dispatch thread.

So, supposing we have a button that launches a series of database queries. We dutifully start up a new thread so that our queries won't block the user interface:

JButton b = new JButton("Run query");

b.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent e) {

Thread queryThread = new Thread() {

public void run() {

**runQueries();**

}

};

**queryThread.start();**

}

});

That was the easy bit. But now, from our query thread, we want to update a progress bar or some other component showing the current progress to the user. How can we do this if we're no longer in the event dispatch thread? Well, the SwingUtilities class, which provides various useful little calls, includes a method called invokeLater(). This method allows us to post a "job" to Swing, which it will then run on the event dispatch thread at its next convenience. So here is how to use SwingUtilities.invokeLater() from out runQueries method:

// Called from non-UI thread

private void runQueries() {

for (int i = 0; i < noQueries; i++) {

runDatabaseQuery(i);

updateProgress(i);

}

}

private void updateProgress(final int queryNo) {

**SwingUtilities.invokeLater(new Runnable() {**

public void run() {

**// Here, we can safely update the GUI**

**// because we'll be called from the**

**// event dispatch thread**

**statusLabel.setText("Query: " + queryNo);**

}

});

}

Here, statusLabel would be a JLabel or JTextField or something of that ilk— it doesn't matter terribly much. The point is: whatever GUI component it is, we must make sure that the code to update it is inside a call to invokeLater().

There's a bit of awkward syntax that we've glossed over, but which it's important to get used to for Swing programming generally. Essentially, we use an **[anonumous inner class](http://www.javamex.com/tutorials/swing/anonymous_inner_classes.shtml)** to define our "job"— more specifically, an implementation of the Runnable interface. Anonymous inner classes are a bit of syntactic shortcut. We could also have written something like:

class UpdateJob implements Runnable {

private final String progress;

UpdateJob(String progress) {

this.progress = progress;

}

public void run() {

statusLabel.setText(progress);

}

}

...

Runnable task = new UpdateJob("Query: " + i);

SwingUtilities.invokeLater(task);

But usually, it's a bit tedious to have to write a separate class definition for every pattern of update job. (Note that either way, they still *compile*to a different class.)

**Application startup code**

There's one place where it's very easy to forget that we need SwingUtilities.invokeLater(), and that's on application startup. Our applications main() method will always be called by a special "main" thread that the VM starts up for us. And this main thread is *not* the event dispatch thread! So:

The **code that *initialises* our GUI must also take place in an invokeLater()**.

So our initial main() method should look something like this:

public class MyApplication extends JFrame {

public static void main(String[] args) {

**SwingUtilities.invokeLater(new Runnable() {**

public void run() {

MyApplication app = new MyApplication();

app.setVisible(true);

}

});

}

private MyApplication() {

// create UI here: add buttons, actions etc

}

}