Theory: Multithreading in Swing

22 minutes 0 / 5 problems solved Skip this topic Start practicing

511 users solved this topic. Latest completion was 28 minutes ago.

Swing programming is not all about creating windows, labels, and buttons. Creating interfaces is only a part of Swing programmers' job. These interfaces are just a way for the user to interact with the system. You need to understand here that writing GUI application is more like writing two applications. You always keep the **frontend** separately from the **background tasks**. For example, the frontend should not **freeze** when the user tries to click something because there is a background process which is running. That's where you need **multithreading**. This is the heavier side of things. Let's have a quick recap into multithreading before we go into complex topics.

§1. Types of threads used in Swing

There are three types of threads that will be used in Swing Applications:

- **Initial threads:** Initial threads are responsible for completing the initial startup of our GUI. The initial thread gets the GUI up to the point where it is running, and ready for events to fire.
- **The event dispatch thread:** Event dispatch threads handle situations where an event is triggered. Using these threads, we can add code to a queue, where they will be executed. Adding tasks to the queue can be done in two ways: as a response to user action through an `ActionListener` or from outside user action through special methods known as `invokeAndWait` and `invokeLater`.
- **Worker threads:** Worker threads are used to execute background services, specifically services that are time-consuming, or have long execution times. For example, If we wanted to create a process that constantly monitors a server for new messages, a worker thread would be used.

The Swing framework provides you with functionality for each of these thread types. All you have to do is to utilize the options available to you to provide your clients with a pleasant experience. Let's dive into more details on how to use these threads.

§2. Initial Threads

In event-driven programming, execution is based on events caused by user interaction with a GUI. In order for the user to interact with the GUI, they need to be able to view the GUI and all of the components associated with it. The purpose of the initial thread is to get the GUI up and running so that the user can start to interact with it.

There are two methods we can use to create our initial threads, which are `invokeLater` and `invokeAndWait`. The `invokeLater` method schedules our GUI to start in the event scheduler. When the event scheduler is ready, it will start the GUI, and it will be available for the user. In contrast, the `invokeAndWait` method not only schedules our GUI to start in the event scheduler, but also waits until the GUI is started before proceeding with any further code. The `invokeAndWait` method is primarily used for situations where code that requires the GUI to be active runs after the initialization of the GUI.

To see how these functions work, let's take a look at a simple example:

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

    public void run() {
        createAndShowGUI();
    }
});
```

Current topic:

Multithreading in Swing

Topic depends on:

✓ Anonymous classes properties

✗ Swing components

✗ Custom threads

Table of contents:

↑ Multithreading in Swing
§1. Types of threads used in Swing
§2. Initial Threads
§3. Event Dispatch Threads
§4. Worker/Background Threads
§5. Intermediate state
§6. Conclusion
Feedback & Comments

`invokeLater` and `invokeAndWait` are both used in the same way. If you want to use `invokeAndWait`, you can use the same code as above, just replacing the `invokeLater` method name. This code will schedule a `Runnable` object in the event scheduler. When the object is called by the event scheduler, the run method will execute, which in this case, runs a method called `createAndShowGUI;`

§3. Event Dispatch Threads

One issue that occurs when working with multiple threads is unsafe interactions between the threads. If we invoke multiple threads, interactions between them can result in unexpected behavior. The methods that Swing utilizes are not considered to be "thread safe". This means that if we have multiple threads running Swing methods, we can encounter unexpected behaviors between the threads. To help avoid this issue, most code that invokes Swing methods will be run on the event dispatch thread.

Much like our initial threads, we can invoke event threads through the `invokeLater` and `invokeAndWait()` methods. These methods carry the same properties as before, with the only difference being where we execute them from. As an example, consider the following code, which will display a message dialog to the screen.

```
void showDialog() throws Exception {
    Runnable showModalDialog = new Runnable() {
        public void run() {
            JOptionPane.showMessageDialog(myMainFrame,
                                          "Hello World!");
        }
    };
    SwingUtilities.invokeAndWait(showModalDialog);
}
```

When this code is triggered, it will invoke the method to display the message dialog and return once the event dispatching thread has executed the code. This method works well for cases when we need an event to finish before proceeding to trigger other events.

If we wish to trigger an event and return immediately, the `invokeLater` method is the ideal solution. It is important to note that if we execute threads in this way, we need to ensure that they do not interact with each other. This is because the Swing package is not thread-safe. If the threads are to interact with each other, it can lead to unpredicted behavior, such as thread interference and race conditions.

Since we can't guarantee that the Swing methods we are running are thread-safe, it is beneficial to check if we are on the event dispatch thread before we attempt to interact with any event dispatch code. In order to do this, we can use the `isEventDispatchThread` method. This method will return true if the current thread is the event dispatch thread. This allows us to avoid situations where another thread attempts to access an event dispatch thread, further preventing unexpected thread behavior.

This diagram depicts how events, event dispatchers, event listeners, and user interfaces interact with each other:

§4. Worker/Background Threads

The third and last part of threads is by far the most complicated one. Swing introduced Worker threads to manage **long-running background tasks**. Doing **large calculations**, reading a lot of data from the **database**, or **reading long files** are examples for such long-running background tasks. The concept of worker threads is implemented by `javax.swing.SwingWorker` class.

To understand why we place these events on a worker thread, consider if we instead placed large events on the same event dispatch thread as our other events. If a user were to click a long event, it would be added to the event queue. If they then trigger a second event, it would be waiting on the queue until the long event is completed. This would give the appearance that the application has frozen, when in reality, it is just processing a longer event. Even events like closing the window would not work due to them being stuck on the event dispatch queue. To avoid this, we can place the longer event on a worker thread, allowing us to separate lengthy operations and allow the user to still interact with the GUI.

For example, we can create a worker named `SwingWorkerExample` to extend the `javax.swing.SwingWorker` class as follows:

```
final JLabel updateLabel;
class SwingWorkerExample extends SwingWorker<String, Object> {
    @Override
    public String doInBackground() {
        // long running task
        Thread.sleep(500);
        return textToUpdateLabel();
    }


    @Override

    protected void done() {

        try {

            updateLabel.setText(get());

        } catch (Exception ignoreForNow) {

        }

    }

}
```

When we extend the `SwingWorker` class, we will implement the `done` method to describe what happens when the `SwingWorker` has finished executing. We will also implement the `doInBackground` method, which describes what the worker will do when it is executed.

In order to execute our `SwingWorkerExample`, we can add it as an action listener, as shown below.

```
JButton b = new JButton("Execute!");
b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        new SwingWorkerExample().execute();
    }
});
```

In doing this, when we press the button, it will execute our `SwingWorkerExample`, which will start execution of the `doInBackground` method.

§5. Intermediate state

We learned how to successfully manage long-running background tasks, but there's more to it. While obtaining the final result is quite enough in many cases, one of the UX good practices is receiving intermediate feedback from a background process. For instance, while downloading something, the browser displays a progress bar, percentage of loaded bytes, and estimated remaining time. Another familiar example is a long-running search that may show intermediate results as soon as they are found and before the search is completed. Both examples use a mechanism of sharing intermediate results between background thread and event dispatch threads. Swing offers methods for implementation of such interaction:

- `publish(V... chunks)` sends data from a background thread to an event dispatch thread,
- `process(List<V> chunks)` receives sent data in an event dispatch thread.

Here the `V` type is the second parameter of `SwingWorker<T, V>` class.

Let's consider an example and create a part of an application that displays the progress of a background task. The code below shows a window with a progress bar and percentage value:

```java
class ProgressBarTask extends SwingWorker<Integer, Integer> {
    private int counter;
    private final JTextArea textArea;
    private final JProgressBar progressBar;

    ProgressBarTask(JTextArea textArea, JProgressBar progressBar) {
        this.textArea = textArea;
        this.progressBar = progressBar;
    }

    @Override
    public Integer doInBackground() throws Exception {
        while (counter < 100 && !isCancelled()) {
            Thread.sleep(100L);
            publish(counter++);

            setProgress(counter);
        }

        return counter;
    }

    @Override
    protected void process(List<Integer> chunks) {
        int value = chunks.get(0);

        textArea.setText("loading " + value + "%");
        progressBar.setValue(value);
    }
}
```

This class emulates a long-running task by sleeping in each iteration of the while loop. A background thread regularly sends the last value of the `counter` to an event dispatcher thread by invoking `publish` method. The value is handled by an event dispatcher thread in `process` method that updates `textArea`.

The code below launches our example:

```
public class Main {
    public static void main(String[] args) throws Exception {
        JFrame frame = new JFrame();
        frame.setSize(300, 100);
        frame.setLayout(new BorderLayout());
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JLabel label = new JLabel("Long running task");
        final JTextArea textArea = new JTextArea();
        final JProgressBar progressBar = new JProgressBar(0, 100);

        frame.add(label, BorderLayout.PAGE_START);
        frame.add(textArea);
        frame.add(progressBar, BorderLayout.PAGE_END);

        ProgressBarTask task = new ProgressBarTask(textArea, progressBar);
        task.execute();
        task.get();
        System.exit(0);
    }
}
```

§6. Conclusion

Multithreading is an important feature for GUI application development with Swing. There are three types of threads used in Swing applications: initial threads, event-dispatch threads, and worker threads. Using these threads properly is essential for creating GUIs that won't freeze. Worker threads are the most complicated `Thread` type that is very useful for background-running services. Sometimes it is a good idea to get intermediate feedback from such background processes. For that, Swing provides special methods. We hope you enjoyed the multithreading in Swing!

Report a typo

6 users liked this theory. 1 didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion