# Java Lab 5: Multithreaded Fractal Explorer

This week we will finish off the Fractal Explorer with one more feature, the ability to draw the current fractal with multiple background threads. This will produce two big improvements: first, the user interface won't freeze while the new fractal is being drawn, and second, if your computer has multiple processors then your drawing will be much faster. Although multithreaded programming can be very challenging, it will be very easy to make this week's changes by leveraging Swing's built-in support for background threads.

So far, whether you have realized this or not, all of our fractal computation and drawing has been performed on Swing's *Event-Dispatch Thread*. This is the thread that handles all Swing events, such as button-clicks, redrawing, and so forth. This is why your user interface freezes when a fractal is being computed; since the computation is being performed on the event-dispatch thread, no events can be processed until the computation is completed.

This week we will change the program to use one or more background threads to compute the fractal. Specifically, the dispatch thread will *not* be used to compute the fractal. Now, if a computation is going to be performed by multiple threads, we need to be able to break it down into multiple independent parts. When drawing fractals, this is very easy to do - we can simply give each thread a single line of the fractal to compute. The harder part is making sure we follow the important Swing constraint that we only interact with Swing components from the event-dispatch thread, but fortunately Swing again provides tools to make this very easy.

It is in fact a very common situation in UI programming, that the user interface must trigger a long-running operation, but the operation should take place in the background so that the UI remains responsive. Web browsers are probably the most widely-used example of this; while a page is being downloaded and rendered, the user must be able to cancel the operation, or click on a link, or do any number of other things as well. To facilitate this kind of interaction, Swing provides the `javax.swing.SwingWorker` class, which makes it very easy to perform a task on a background thread. `SwingWorker` is an abstract class; Swing expects you to extend it, and provide the functionality to perform the background task. The most important methods to implement are:

- `doInBackground()` - this method actually performs the background operation. Swing calls this method on a background thread, not the event-dispatch thread.
- `done()` - this method is called when the background task is completed. It is called on the event-dispatch thread, so this method is allowed to manipulate the user interface.

The `SwingWorker` class has a confusing specification, it is actually `SwingWorker<T, V>`. The type `T` is the type of the value returned by `doInBackground()`, when the entire task is completed. The type `V` is used when a background task returns intermediate values as it runs; these intermediate values would be exposed by the `publish()` and `process()` methods. *It is not always necessary to use either or both of these types;* in these cases, we can simply specify `Object` for the unused type(s).

## Drawing in the Background

You will be working primarily within the `FractalExplorer` class this week. Some of the code will be new, but some of it will actually be refactoring code that you have already written.

- You should create a subclass of `SwingWorker` called `FractalWorker`, that is an inner class of the `FractalExplorer`. This is the easiest way to construct this code, since it will need to access quite a few of the `FractalExplorer`'s internal members. Remember that the `SwingWorker` class is a generic, so you will need to specify the parameters - you can simply specify `Object` for both of them, because we actually won't use them. Therefore you will end up with a declaration like this:

  ```
  private class FractalWorker extends SwingWorker<Object, Object>
  ```

- The `FractalWorker` class will be responsible for computing the color values for a single row of the fractal, so it will need two fields: the integer y-coordinate of the row that will be computed, and an array of `int`s to hold the computed RGB values for each pixel in that row. The constructor should take the y-coordinate as an argument and store it; the constructor won't have to do anything else. (Specifically, you shouldn't allocate the array of `int`s at this point, since it won't be needed until the row is actually computed.)

- Remember that the `doInBackground()` method is called on a background thread, and is responsible for performing the long-running task. Therefore, in your implementation you will need to take some of the code from your earlier "draw fractal" function, and put it into this method. Of course, instead of drawing to the image-display, the loop will need to store each RGB value into the corresponding element of the integer array. You are actually *not allowed* to modify the image-display from this thread, because you would be violating Swing's threading restrictions!

  Instead, allocate the array of integers at the start of this method (it will need to be large enough to hold an entire row of color values), and then store each pixel's color into this array. It should be very easy to adapt the code you wrote before - the only differences are that you will only compute the fractal for the specified row, and that you aren't updating the screen yet.

  Your `doInBackground()` method will need to return something of type `Object`, since that's what the `SwingWorker<T, V>` declaration specifies. Just return `null`!

- The `done()` method is called when the background task is completed, and this method is called from the Swing event-dispatch thread. This means you can modify Swing components to your heart's content. Therefore, in this method, you can simply iterate over the array of row-data, drawing in the pixels that were computed in `doInBackground()`. It really couldn't be much simpler.

  As before, when the line is finished drawing, you will need to tell Swing to redraw the portion of the display that was changed. Since you only changed a single line, it would be overly expensive to tell Swing to redraw the entire display! Therefore, you can use the version of

`JComponent.repaint()` that allows you to specify a region to repaint. Note that the method is a little odd - it has an unused `long` parameter at the front, and you can simply specify 0 for that argument. For the rest, simply specify the row that was drawn - starting at (0, y) and with a size (displaySize, 1).

- Once you have completed your background task class, the next step is to hook it into the fractal drawing process. You already moved some of your code from your "draw fractal" function to the worker class, so now you can change your "draw fractal" function to do this:
  - For every single row in the display, create a separate worker object for that row, and then call `execute()` on the object. This will start up the background thread and run the task in the background!

  ...and that should be all the changes you need to make! Remember that the worker class takes care of generating the row data and then painting the row, so your "draw fractal" function should be very simple now.

Once you have this functionality completed and debugged, you should now have a much faster and more responsive user interface. If you happen to have multiple cores, you will definitely see a substantial improvement.

You will notice one issue with your user interface - if you click on the screen or on a button during a redraw, the program will process it even though it should probably be ignored until the operation is completed. Fortunately this is pretty straightforward to fix.

# Ignoring Events During Redraw

The easiest way to solve the issue of ignoring events during redraw is to keep track of how many rows are remaining to be completed, and ignore or disable user interactions until all rows are drawn. This has to be done very carefully though, or else we will have some very nasty bugs. What we can do is this: add a "rows remaining" field to our Fractal Explorer class, and use it to know when a redraw is completed. *We will only read and write this value from the event dispatch thread so that we never introduce any concurrent accesses.* If we only interact with a resource from a single thread, we won't have any concurrency bugs. Here is what you need to do:

- Create a function `void enableUI(boolean val)` that will enable or disable your interface's buttons and combo-box, based on the specified value. You can use the Swing `setEnabled(boolean)` method to enable or disable these components. Make sure that your method updates the enabled-state of your save button, your reset button, and your combo-box.
- Your "draw fractal" function needs to do two additional things. First, it should call `enableUI(false)` before anything else, to disable all the UI controls during drawing. Second, it should set the "rows remaining" value to the total number of rows to be drawn. Do this *before* you start any of the worker tasks, or else it won't be updated properly!
- In your worker's `done()` method, decrement the "rows remaining" value by 1 as the last step of this operation. Then, if the rows remaining is 0 after being decremented, call `enableUI(true)`.
- Finally, modify your mouse-listener implementation to return immediately if the "rows remaining" value is nonzero. In other words, you will only respond to mouse clicks if there aren't anymore rows to be drawn. (Note that we don't have to make a similar change to the action-event handler because we disable all of those components with the `enableUI()` method.)

Once you have completed these steps, you should have a nice, solid fractal display program that can draw fractals with multiple threads, and that won't allow users to do anything while the rendering process occurs in the background. Pretty neat.

# All Done!

Once your program is operating correctly, and you have properly commented all of your code, submit your work on csman.

---