

# Threads and Swing

- Most post-initialization GUI work naturally occurs in the event-dispatching thread. Once the GUI is visible, most programs are driven by events such as button actions or mouse clicks, which are always handled in the event-dispatching thread
- You need to be aware of how to schedule work in the event-dispatch thread
  - If your program creates threads to perform tasks that affect the GUI
  - If it manipulates the already-visible GUI in response to anything but a standard event, then you need to be concerned about thread usage
- It is not just a matter of updating graphics
  - What if you were getting the text of a JTextField, if you are not being thread safe, you could catch it after a user ordered a change to it (in an actionPerformed method, for example), but before the value was actually set in the JTextField, giving you corrupted data.

# When to use multiple threads with Swing

- Programs that must perform a lengthy initialization operation before they can be used.
  - This kind of program should generally show some GUI while the initialization is occurring, and then update or change the GUI.
  - The initialization should not occur in the event-dispatching thread; otherwise, repainting and event dispatch would stop.
  - After initialization the GUI update/change should occur in the event-dispatching thread, for thread-safety reasons.
- Programs whose GUI must be updated as the result of nonstandard events
  - For example, suppose a server program can get requests from other programs that might be running on different machines. These requests can come at any time, and they result in one of the server's methods being invoked in some possibly unknown thread.
  - That method can safely update the GUI by executing the GUI-update code in the event-dispatching thread.

# Swing Utilities for dealing with Threads

- The `SwingUtilities` class provides two methods to help you run code in the event-dispatching thread:
  - `invokeLater`
    - Requests that some code be executed in the event-dispatching thread. This method returns immediately, without waiting for the code to execute.
  - You can call `invokeLater` from any thread to request the event-dispatching thread to run certain code. You must put this code in the `run` method of a `Runnable` object and specify the `Runnable` object as the argument to `invokeLater`. The `invokeLater` method returns immediately, without waiting for the event-dispatching thread to execute the code.

```
Runnable updateAComponent = new Runnable() {  
    public void run() { component.doSomething(); }  
};  
SwingUtilities.invokeLater(updateAComponent);
```

# Swing Utilities for Thread continued

- `invokeAndWait` - Acts like `invokeLater`, except that this method blocks until the `Runnable` object completes executing. As a rule, you should use `invokeLater` rather than this method.
  - *`invokeAndWait` must be called outside of the event-dispatch thread, or it will throw an exception*
- Whenever possible, you should use `invokeLater` instead of `invokeAndWait`. If you use `invokeAndWait`, make sure that the thread that calls `invokeAndWait` does not hold any locks that other threads might need while the call is occurring.

```
void showHelloThereDialog() throws Exception {
    Runnable showModalDialog = new Runnable() {
        public void run() {
            JOptionPane.showMessageDialog(myMainFrame,
                                        "Hello There");
        }
    };
    SwingUtilities.invokeLater(showModalDialog);
}
```

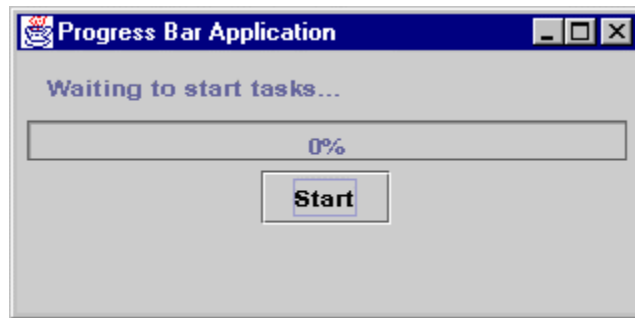
# When to use invokeLater

- For example, a thread that needs access to GUI state, such as the contents of a pair of text fields, might have the following code:

```
void printTextField() throws Exception {
    final String[] myStrings = new String[2];
    Runnable getTextFieldText = new Runnable() {
        public void run() {
            myStrings[0] = textField0.getText();
            myStrings[1] = textField1.getText();
        }
    };
    SwingUtilities.invokeLaterAndWait(getTextFieldText);
    System.out.println(myStrings[0] + " " + myStrings[1]);
}
```

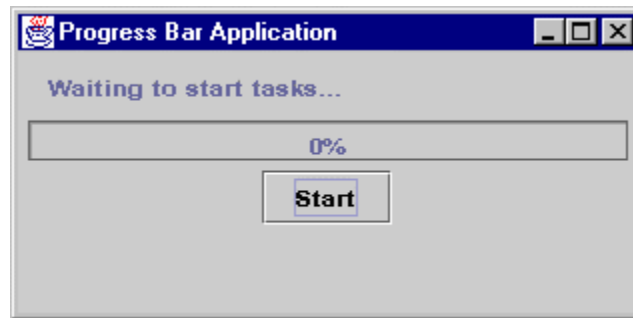
# Code Example, ProgressBar.java

- Most simple example, several problems.
  - AWT thread blocks
  - Uses repaint() for update



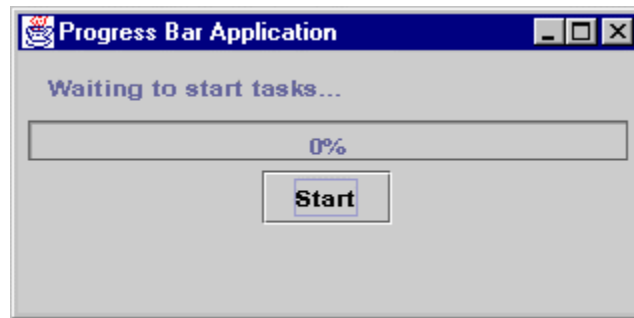
# Code Example, ProgressBar1.java

- Now uses `paintImmediately`
- AWT thread still blocked



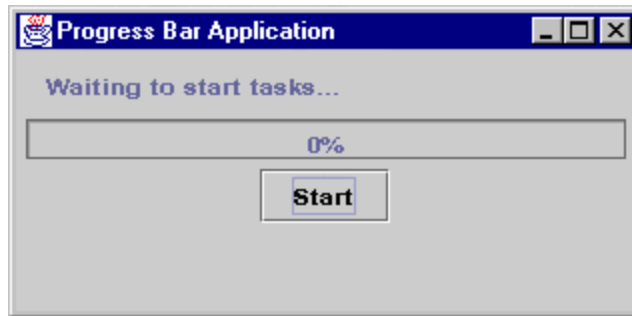
# Code Example, ProgressBar2.java

- Moves animation to a new thread
- Still uses `paintImmediately()` and `setValue()` in new thread



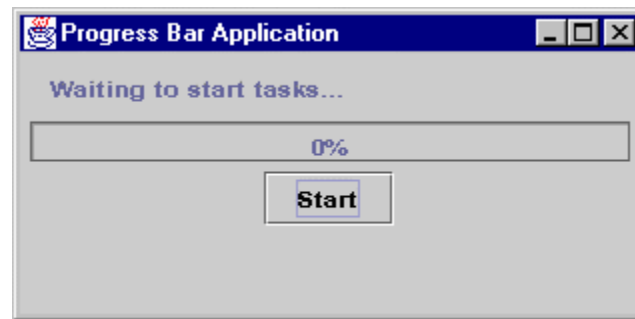
# Code Example, ProgressBar3.java

- Now uses `invokeLater`
- But `println()` statement make it appear to work, if we remove them and reuse `Rectangle` objects, we see a new problem
- `iCtr` value is not thread safe, the thread was placed in the queue 20 times before they were cycled. Since `iCtr` was changed to 20, all 20 threads used `iCtr=20`.
- Could use a `Thread.sleep()`...but...



# Code Example, ProgressBar4.java

- Now uses `invokeAndWait!`



# ImageIO Example

- ImageReader class of javax.imageio package supports a IOReadProgressListener
  - imageStarted
  - imageComplete
  - imageProgress
  - Also support loading and monitoring of thumbnails
- ImageWriter exists as well
- Deals with BufferedImage
- ImageProgressBar.java
  - Uses ImageReader and IOReadProgressListener
  - Updates from listener cause progress bar to be updated
  - Image is loaded in separate thread



# Thread Gotcha's

- There is no guarantee that two threads will have a consistent view of any given variable, unless (a) the threads synchronize on the same some object at some stage, or (b) the variable is declared volatile.
- Using volatile forces the thread to retrieve the value from main memory before it is read, and to set it immediately after it is written, ensuring that

# Animation

- There are several ways to animate a GUI
- Thread.sleep() issues
  - On most systems, a sleep less than 100 msec is fairly inaccurate
  - On one Windows XP system, min sleep time is 14 msec
  - On a similar Linux system, min sleep time was 7 msec
- Timer Resolution
  - Win 95/98 – 55 ms
  - Win 2K/XP – 10-15 ms
  - Linux, Solaris, MacOS – 1 ms

# java.swing.Timer

- java.swing.Timer class
  - Since a timer generates an `ActionEvent`, they can be used to update displays in a thread safe manner.
  - Places each call to `actionPerformed` on the event-dispatch thread
  - Useful in very simple animation
  - Have the Timer fire an event every N seconds, use that event to update your GUI, since all Event driven actions are performed in the event-dispatch thread
  - Tests show that Timer fires no quicker than every N seconds, but not exactly every N seconds (some extra)
  - Only useful for 20 fps or less animation, large overhead for

# java.util.Timer

- Instead of scheduling calls to `actionPerformed()`, the `run()` method of a `TimerTask` is invoked
- Run at:
  - Fixed rate
    - Scheduled relative to scheduled time of original task (doesn't drift)
  - Fixed period after a previous task
    - Similar to `javax.swing.Timer`
- May obtain up to 100 FPS using this timer
- Drawback is that details of the timer and sleeping operations are out of reach of programmer

# Animation

- In “Killer Game Programming” by Andrew Davison, the following algorithm is proposed

```
public void run() {  
    while (running) {  
        animationUpdate();  
        animationRender();  
        paintScreen();  
        sleep();  
    }  
}
```

# Animation explained

- Sleep
  - Frees up the CPU for other tasks
  - Controls refresh rate
- Why not repaint()?
  - Lose control of when animation occurs
  - Event coalescence
  - Can't measure how long it takes (it merely requests the event to occur)
- PaintScreen() uses “active rendering”
  - We are in charge of painting to screen
  - Done outside of normal AWT thread
  - Use getGraphics()
- AnimationPane.java
  - Template to use for animation
  - Detailed control of animation rate (FPS)

# SwingWorker

- New in JDK 1.6
  - Actually has been around as an add on for quite a while
- An abstract class to perform lengthy GUI-interacting tasks in a dedicated thread.
  - When writing a multi-threaded application using Swing, there are two constraints to keep in mind: (refer to How to Use Threads for more details):
    - Time-consuming tasks should not be run on the Event Dispatch Thread. Otherwise the application becomes unresponsive.
    - Swing components should be accessed on the Event Dispatch Thread only.
- These constraints mean that a GUI application with time intensive computing needs at least two threads:
  - a thread to perform the lengthy task
  - the Event Dispatch Thread (EDT) for all GUI-related activities.
- This involves inter-thread communication which can be tricky to implement.

# SwingWorker Continued

- SwingWorker is designed for situations where you need to have a long running task run in a background thread and provide updates to the UI either when done, or while processing.
- Subclasses of SwingWorker must implement the `doInBackground()` method to perform the background computation.
- Workflow
  - Current thread: The `execute()` method is called on this thread. It schedules SwingWorker for the execution on a worker thread and returns immediately. One can wait for the SwingWorker to complete using the `get` methods.
  - Worker thread: The `doInBackground()` method is called on this thread. This is where all background activities should happen. To notify `PropertyChangeListeners` about bound properties changes use the `firePropertyChange` and `getPropertyChangeSupport()` methods. By default there are two bound properties available: `state` and `progress`.
  - Event Dispatch Thread: All Swing related activities occur on this thread. SwingWorker invokes the `process` and `done()` methods and notifies any `PropertyChangeListeners` on this thread.
- Often, the Current thread is the Event Dispatch Thread.

# Example SwingWorker 1

- The following example illustrates the simplest use case. Some processing is done in the background and when done you update a Swing component.
- Say we want to find the "Meaning of Life" and display the result in a JLabel.

```
final JLabel label;
class MeaningOfLifeFinder extends SwingWorker<String, Object> {
    @Override
    public String doInBackground() {
        return findTheMeaningOfLife();
    }

    @Override
    protected void done() {
        try {
            label.setText(get());
        } catch (Exception ignore) {
        }
    }
}
(new MeaningOfLifeFinder()).execute();
```

## Example SwingWorker 2

- The next example is useful in situations where you wish to process data as it is ready on the Event Dispatch Thread.
- Now we want to find the first N prime numbers and display the results in a JTextArea. While this is computing, we want to update our progress in a JProgressBar. Finally, we also want to print the prime numbers to System.out.
- SwingWorker2.java