THREAD STATES

During its lifecycle a thread can move through the following states:

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BORN</td>
<td>The thread object has been built but its run method has yet to be started</td>
</tr>
<tr>
<td>RUNNING</td>
<td>The run method has been started and is running</td>
</tr>
<tr>
<td>DEAD</td>
<td>The run method has stopped running</td>
</tr>
<tr>
<td>SLEEPING</td>
<td>Thread suspends itself for a certain amount of delay time</td>
</tr>
<tr>
<td>WAITING</td>
<td>Thread is suspended pending notification or for a delay time</td>
</tr>
</tbody>
</table>

In the original version of Java, methods of the Thread class were provided to transition a thread through these states.
Methods **suspend**, **resume** and **stop** have been deprecated because they are unsafe. I show you how to safely control a thread’s state in the topic *Safe Transitioning of Thread States*.

**A Demonstration of Thread States**

Let’s construct a Java program that demonstrates transitioning a thread through these five states. For our thread, we use the **Counter** class that is explained on pages 3 and 4 of the topic *Java Threads*. A **Counter** object displays a ticking counter as shown to the right. The class definition is reprinted below.

```
import javax.swing.*;
import java.awt.*;
import javax.swing.border.*;

public class Counter extends Thread {
    private int counter = 0;    // counter to increment
    private int delay;          // sleep time
    private JTextField display; // display for counter
    // fancy JPanel to display, made public so another
    // object can add it to a containment hierarchy
    public JPanel displayPanel;

    public void run() {  
        try // needed for call to Thread.sleep (see below)
        {
            while ( true ) // do forever
            {
                counter++;    // increment counter & display
                display.setText( Integer.toString( counter ) );
                Thread.sleep( delay ); // sleep
                // sleep can throw InterruptedException
            }
        }
    }
}
```

A Demonstration of Thread States

Let’s construct a Java program that demonstrates transitioning a thread through these five states. For our thread, we use the **Counter** class that is explained on pages 3 and 4 of the topic *Java Threads*. A **Counter** object displays a ticking counter as shown to the right. The class definition is reprinted below.
private final int DELAY_RANGE = 2000;

public Counter(String name) {
    super(name); // call superclass's constructor
    // initialize this thread's delay to a random
    // integer between a half second and a second
    delay = (int)(Math.random() * DELAY_RANGE) + 500;
    // initialize the display JTextField
    display = new JTextField(4);
    display.setHorizontalAlignment(JTextField.RIGHT);
    display.setEditable(false);
    display.setFont(new Font("Courier New", Font.BOLD, 72));
    display.setBackground(new Color(255, 255, 153));
    display.setForeground(new Color(0, 0, 255));
    // create the fancy JPanel
    displayPanel = new JPanel();
    displayPanel.setBorder(new TitledBorder(new EtchedBorder(), name));
    // create box with delay label and value
    Box delayBox = Box.createVerticalBox();
    delayBox.add(new JLabel("Delay"));
    delayBox.add(Box.createVerticalStrut(5));
    delayBox.add(new JLabel(Integer.toString(delay)));
    // add the JTextField and JLabel to the JPanel
    displayPanel.add(display);
    displayPanel.add(delayBox);
} // end constructor
To control the counter, we create the class `ControlPanel` shown below. A `ControlPanel` object is a `JPanel` object containing four buttons arranged in a $2 \times 2$ grid (see the picture at right). Its constructor (lines 29–42) creates the button listener, creates the buttons and builds the grid.

The class constructor creates the buttons by calling method `makeButton` (see lines 38–41) whose declaration is on lines 14–27. `makeButton` is a utility that factors the code for making a button, adding it to the grid and registering its listener.

`ControlPanel` has a private inner class `StateListener` (lines 44–75) that handles user actions on the buttons. Per the event handling model, `StateListener` implements the `ActionListener` interface by implementing its `actionPerformed` method, which uses a cascading if-else statement to determine which button the user has pressed. In response, `actionPerformed` calls the appropriate `Thread` method to transition the counter into the requested state.

```java
import javax.swing.*;
import java.awt.event.*;
import java.awt.*;

public class ControlPanel extends JPanel
{
    // control buttons
    private JButton startBtn, pauseBtn, resumeBtn, quitBtn;
    // Counter thread
    private Counter counter;
    // Listener to the buttons
    private StateListener listener;

    public JButton makeButton
    ( String text, Color hue, boolean isOn )
    // Make a button with given label, color and enabled.
    // Add it to the non-local 'btnGrid'.
    // Register the non-local 'listener' as its listener.
    {
        JButton btn = new JButton( text );
```
btn.setFont( new Font( "Dialog", Font.BOLD, 32 ) );
btn.setBackground( hue );
btn.setEnabled( isOn );
btn.addActionListener( listener );
this.add( btn ); // add to grid
return btn;

public ControlPanel( Counter c )
{
    // save Counter thread to instance variable
    counter = c;
    // change panel layout
    setLayout( new GridLayout( 2, 2, 2, 2 ) );
    // build listener object
    listener = new StateListener( );
    // make buttons
    startBtn = makeButton( "START", Color.GREEN, true );
    pauseBtn = makeButton( "PAUSE", Color.RED, false );
    resumeBtn = makeButton( "RESUME", Color.GREEN, false );
    quitBtn = makeButton( "QUIT", Color.YELLOW, false );
}

private class StateListener implements ActionListener
{
    public void actionPerformed( ActionEvent e )
    {
        // handle user clicking the buttons
        if ( e.getActionCommand( ) == "START" )
        {
            counter.start( ); // start counter
            startBtn.setEnabled( false );
            pauseBtn.setEnabled( true );
            quitBtn.setEnabled( true );
        }
        else if ( e.getActionCommand( ) == "PAUSE" )
        {
            counter.suspend( ); // suspend counter
            pauseBtn.setEnabled( false );
            resumeBtn.setEnabled( true );
        }
    }
}
else if ( e.getActionCommand( ) == "RESUME" )
{
    counter.resume( ); // resume counter
    pauseBtn.setEnabled( true );
    resumeBtn.setEnabled( false );
}
else if ( e.getActionCommand( ) == "QUIT" )
{
    counter.stop( ); // stop counter
    pauseBtn.setEnabled( false );
    resumeBtn.setEnabled( false );
    quitBtn.setEnabled( false );
}
} // end actionPerformed
} // end StateListener
}

Class **UnsafeThreadStates** (shown below) is an applet that combines the counter and the control panel into an executable Java program. It builds the counter at line 14 and the control panel at line 16, passing the counter to the control panel’s constructor. The counter’s display and the control panel are added to the applet’s containment hierarchy at lines 18 and 19.
When the applet executes line 14, the counter is placed into the BORN state. The GUI appears as:

![GUI with buttons START, PAUSE, RESUME, QUIT](image1)

Only the START button is enabled. The user clicks it to transition the counter to the RUNNING state. This is implemented by the listener’s `actionPerformed` method calling `counter.start` (see `ControlPanel.java` line 50).

![Screen showing counter at 14](image2)

The user transitions the counter between the RUNNING and WAITING states by alternate clicks of PAUSE and RESUME. These are implemented by the listener’s `actionPerformed` method calling `counter.suspend` and `counter.resume` (see `ControlPanel.java` lines 57 and 63).
The user transitions the counter to the DEAD state by clicking QUIT; implemented by the listener’s `actionPerformed` method calling `counter.stop` (see `ControlPanel.java` line 69).

A thread can only be transitioned to the SLEEPING state by itself.

**Safety Issues**
Method `stop` has been deprecated because it can leave objects in an inconsistent state. When a thread is stopped, it automatically releases all the intrinsic locks that it holds. If the thread held a lock so that it could atomically update a shared object, then this could potentially leave the update only partially done.

Methods `suspend` and `resume` can cause deadlock. Suppose thread A is suspended when it holds a lock on some object O; A is waiting for thread B to resume it. One problem arises in that no other thread can access O while A is suspended. A worse problem arises if B tries to synchronize on O before resuming A. This causes B to be suspended by the JVM pending release of the lock. The result: A and B are both suspended, the lock on O will never be released, deadlock has occurred.

Because of these safety issues, use of these three methods is not recommended for commercial-grade software.