JAVA THREADS

In Java, a process is an executing computer program that has a self-contained execution environment, meaning that it has everything it needs to run, including memory space. A thread is a lightweight process, having an execution environment that exists within that of a larger process. A process has one or more threads, each of which shares the process’s resources, including memory and open files. Two or more threads can run at the same time and, when they do, are called concurrent threads.

Concurrent threads are implemented in one of two ways depending on the computer hardware available. When multiple processors are available, concurrent threads can be executed in parallel, each on its own processor. This is called physical concurrency.

**Example**
Suppose thread #1 computes \( a \times b + c \times d \) and thread #2 computes \( w \times x + y \times z \). With two processors the computation can proceed as follows:

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread 1 (Processor 1)</th>
<th>Thread 2 (Processor 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Compute ( A = a \times b )</td>
<td>Compute ( A = w \times x )</td>
</tr>
<tr>
<td>2</td>
<td>Compute ( B = c \times d )</td>
<td>Compute ( B = y \times z )</td>
</tr>
<tr>
<td>3</td>
<td>Compute ( A + B )</td>
<td>Compute ( A + B )</td>
</tr>
</tbody>
</table>

When only one processor is available, it executes concurrent threads by interleaving the execution of their individual operations, a process called logically concurrency.

**Example**
Suppose thread #1 computes \( a \times b + c \times d \) and thread #2 computes \( w \times x + y \times z \). One processor interleaves the machine operations. One possible interleaving is:

<table>
<thead>
<tr>
<th>Time</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Compute ( A = a \times b )</td>
</tr>
<tr>
<td>2</td>
<td>Compute ( B = w \times x )</td>
</tr>
<tr>
<td>3</td>
<td>Compute ( C = c \times d )</td>
</tr>
<tr>
<td>4</td>
<td>Compute ( D = y \times z )</td>
</tr>
<tr>
<td>5</td>
<td>Compute ( A + C )</td>
</tr>
<tr>
<td>6</td>
<td>Compute ( B + D )</td>
</tr>
</tbody>
</table>
Interleaving is often done at the hardware level by \textit{timeslicing} the concurrent threads. Each thread is given an interval of time, called a \textit{quantum}, to execute on the CPU. When its quantum expires, the thread is suspended, put into a ready queue and the CPU is given a different thread to execute for its quantum. Suspension and resumption of the threads continue in a round-robin fashion so that all threads get a chance to advance their computation.

The behavior of concurrent threads is equivalent regardless of whether they are implemented physically or logically. This means that the choice of physical or logical concurrency can be left to the compiler and operating system without you having to make special allowances for it in your code.

Java provides two mechanisms for creating your own threads. The first is to write a class that extends the API class \texttt{java.lang.Thread}.

\begin{verbatim}
<table>
<thead>
<tr>
<th>Thread Syntax #1</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class \texttt{class name} extends Thread</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>public \texttt{class name} ( String name )</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>super( name );</td>
</tr>
<tr>
<td>\textit{thread initialization code}</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>public void run( )</td>
</tr>
<tr>
<td>{</td>
</tr>
<tr>
<td>\textit{thread execution code}</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>
\end{verbatim}

Place into the constructor any initialization code you need and into method \texttt{run} whatever you want your thread to do. The thread runs in the context of the Java program that creates it.
Example

The following class describes a concurrent thread object that increments a counter, displays it in a `JTextField` and sleeps for some determined amount of time given by the value of the variable `delay`. The `run` method, which is inherited from class `Thread` and morphed appropriately, does this repeatedly. The class constructor initializes the delay variable and builds a display panel (shown at right) in which to show the value of the counter.

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

public class Counter implements Runnable {
    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() {
        // morphed from Thread to do what I want it to.
        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
            }
        } catch( InterruptedException e ) {
            // probably won't happen, but we handle it anyway
            String name = Thread.currentThread().getName();
            System.out.println( name + " halted during run." );
            return;
        }
    }
}
```
private final int DELAY_RANGE = 2000;

public Counter(String name) {
    // initialize this thread's delay to a random integer between 0.5 and 2.5 seconds
    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)));  
    displayPanel.add(display);
    displayPanel.add(delayBox);
}  // end constructor

}  // end run method
To execute the thread, build a thread object from your class and use the inherited method `start` to commence its run.

**Example**
The following Java applet builds a `Counter` object and starts it running.

```java
import java.applet.*;
import java.awt.event.*;
import java.awt.*;

public class CounterThread extends Applet
{
    public void init()
    {
        // build the counter object
        Counter counter = new Counter( "Counter 1" );
        // change applet's layout (default is BorderLayout)
        setLayout( new FlowLayout( ) );
        // add counter's display panel to applet
        add( counter.displayPanel );
        // start thread running
        counter.start( );
    }
}
```

The second mechanism for creating your own threads is to write a class that implements the interface `java.lang.Runnable`, which is also available in the Java API. `Runnable` contains only one method, the unimplemented `run`, which you implement in the new class. The Java API specification recommends using this technique if the `run` method is the only `Thread` method you intend to morph.

**Example**
To write the `Counter` class using the `Runnable` interface, the code is exactly the same as previously listed except for line 5, which must be changed to:

```
public class Counter implements Runnable
```

To execute a thread constructed from the **Runnable** interface:

1. Build the **Runnable** object.
2. Build a **Thread** from the **Runnable** object.
3. Call the thread’s **start** method.

**Example**

```java
import java.applet.*;
import java.awt.event.*;
import java.awt.*;

public class CounterRunnable extends Applet
{
    public void init()
    {
        // build runnable object
        Counter counter = new Counter( "Counter 1" );
        // build the thread object from the runnable
        Thread thread = new Thread( counter );
        // change applet's layout (default is BorderLayout)
        setLayout( new FlowLayout( ) );
        // add counter's display panel to applet
        add( counter.displayPanel );
        // start thread running
        thread.start();
    }
}
```
### Exercises

1. Save the classes `Counter` and `CounterThread` to files. Compile and run them using jGRASP.

2. Modify `Counter` class so that it implements `Runnable`. Save `CounterRunnable` to a file. Compile and run the two classes using jGRASP.

3. Modify your program from exercise 1 so that it builds and runs two counter threads. Display them in a GUI similar to that shown at right.

4. Repeat exercise 3 on your `Counter` class from exercise 2.

5. Modify your program from exercise 1 so that it builds an array of 9 counter threads and displays them in a $3 \times 3$ grid, as shown below.

6. Repeat exercise 5 on your `Counter` class from exercise 2.

7. Modify your program from exercise 3 to add a GO button to the GUI (see the picture to the right). The two counter threads must not start until the user clicks the GO button.

8. Repeat exercise 7 on your `Counter` class from exercise 2.