

9

*Introducing
Concurrency with
Threads*

Self-review Questions

9.1 What is a thread?

A thread is a single path of execution through a program that can be characterized as a sequence of method calls. In a multi-threaded program a number of threads are each following their own distinct paths.

9.2 How are threads created?

A thread is represented by an instance of class `Thread`, so a new thread is started by creating a new `Thread` object.

A new thread can be created in two ways:

- By creating a subclass of `Thread` and overriding the `run` method.
- By creating a class that implements the interface `Runnable` and passing an instance of the class as a parameter to a `Thread` constructor when a new instance of `Thread` is created. `Runnable` requires that an implementing class implements the `run` method.

In both cases a new thread is started by calling the `start` method on the `Thread` object. This will initialize a new thread and make it available for running when its turn comes. The `run` method is called as the first method in the method call sequence of the new thread. The thread will continue running until the `run` method terminates.

9.3 What is the purpose of a critical region?

If two or more threads try to assign a value to the same variable, or call a method on the same object to change its state, there is the possibility that the activities will overlap and corrupt the state of the variable or object. For example, two threads may try to add a value to a data structure at roughly the same time and end up leaving the data structure in undefined state as one thread has only partially updated the data structure when a second thread starts another update. This is often known as a *race condition* as the threads are racing to access a resource.

To avoid these sorts of problem code vulnerable to race conditions is placed inside a critical region. The region guarantees that only one thread can execute the code in the region at any one time, and must leave the region before any other thread can enter. In Java, the principle mechanisms for establishing a critical region are the synchronized block and the synchronized method. Both of these require that a thread obtains a specific object lock before entering the

region. If the object lock is already held by another thread, a new thread cannot enter the region and will be suspended until the lock becomes available. When a thread leaves a synchronized block or method the object lock is released.

9.4 What is thread scheduling?

Thread scheduling is used to control when a thread is running or when it is suspended waiting to run. A thread scheduler typically attempts to give each thread a fair chance to run, avoiding any threads being permanently suspended or not having enough time to do its job properly. A common scheduling strategy is the *round-robin* approach, where the scheduler gives each thread an equal amount of time to run, switching between threads as each uses up its time. This is known as *time slicing*, as the total runtime is sliced up amongst multiple threads.

In practice, thread scheduling gets more complicated than the simple round-robin approach as threads can have different priorities and some threads may be blocked waiting to synchronize with other threads or for access to critical regions.

9.5 How does a thread terminate?

A thread terminates when the `run` method used to start the thread terminates. That is when the thread has completed its task, control returns back to the `run` method in the normal way and the `run` method returns.

A thread will also be forcibly terminated when the entire program is terminated or when an uncaught exception is thrown within a thread causing the `run` method to terminate. In both these cases it is good practice to terminate threads properly by allowing their `run` methods to terminate in a controlled way if the thread state needs to be cleaned up. This avoids the possibility of a thread being terminated and either data being lost or the program not exiting cleanly.

Programming Exercises

9.1 Modify the clock program to display only the time, without the day, date or year.

The clock program displays the date and time by simply creating a new `Date`, which by default is initialized to the current time, and using the `toString` method to get a string representing the date. The string includes the time, day, year, and so on. Displaying just the time can be done by using a method like `substring` to pull out the relevant section of the string returned by `toString`. However, a better approach is to use `SimpleDateFormat` from the Java class libraries as this gives full control over how a date string can be formatted, and avoids depending on the location of characters in a string that using `substring` relies on. The following modified version of the clock program shows how `SimpleDateFormat` is used.

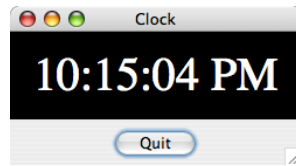
```

import java.awt.Color ;
import java.awt.Font ;
import java.awt.Graphics ;
import java.awt.Graphics2D ;
import java.awt.geom.Rectangle2D ;
import java.util.Date ;
import java.text.SimpleDateFormat;

/**
 * A program to display a simple text clock.
 *
 * <p>The <code>DrawPanel</code> displays the current time at the moment of painting. An associated thread
 * object creates repaint events every second causing the update of the clock.</p>
 *
 * @author Graham Roberts and Russel Winder
 * @version 2006-11-11
 */
public class ClockPanel extends DrawPanel {
    /**
     * Instances of this inner class call the <code>repaint</code> method in the parent object (which is a
     * <code>DrawPanel</code> that is a <code>ClockPanel</code> application) every second.
     *
     * @authors Graham Roberts and Russel Winder
     * @version 2006-11-11
     */
    private class SecondTicker extends Thread {
        public void run () {
            while ( true ) {
                try { sleep ( 1000 ) ; }
                catch ( Exception e ) { }
                repaint () ;
            }
        }
    }
    private final SecondTicker ticker = new SecondTicker () ; { ticker.start () ; }
    public ClockPanel () { }
    public ClockPanel ( final int w , final int h ) { super ( w , h ) ; }
    public void paint ( final Graphics g ) {
        final Graphics2D g2d = (Graphics2D) g ;
        final Rectangle2D background =
            new Rectangle2D.Double ( 0 , 0 ,
                getSize ().getWidth () , getSize ().getHeight () ) ;
        g2d.fill ( background ) ;
        g2d.setFont ( new Font ( "Serif" , Font.PLAIN , 40 ) ) ;
        g2d.setPaint ( Color.white ) ;
        final SimpleDateFormat dateFormat = new SimpleDateFormat ( "hh:mm:ss a" ) ;
        final String time = dateFormat.format ( new Date () ) ;
        g2d.drawString ( time , 20 , 50 ) ;
    }
    public static void main ( final String[] args ) {
        DrawFrame.display ( "Clock" , new ClockPanel ( 250 , 75 ) ) ;
    }
}

```

This program displays:



The string "hh:mm:ss a" is used to control how the time is formatted. There are many other formatting variations possible, see the Javadoc for the details.

9.2 Rewrite class `ClockPanel` to display a graphical representation of a round clock face with hour, minute and second hands.

The following class displays the required clock face. It is capable of displaying a clock face of any reasonable size, scaling the representation depending on the size of the panel. There are two constructors provided, the first taking three arguments specifying the width and height of the panel, and the minimum size of the margin, or space, left around the clock face. The second constructor takes an additional two arguments specifying the colour of the panel background and the colour of the clock face. The default colours are black and white respectively. The clock face is always centred in the middle of the panel.

```
import java.awt.BasicStroke ;
import java.awt.Color ;
import java.awt.Font ;
import java.awt.geom.Rectangle2D ;
import java.awt.geom.Line2D ;
import java.awt.geom.Ellipse2D ;
import java.awt.Graphics ;
import java.awt.Graphics2D ;
import java.awt.RenderingHints ;
import java.util.Calendar ;

/**
 * A program to display an analogue clock face with hour, minute and second hands that is updated every
 * second.
 *
 * @author Graham Roberts and Russel Winder
 * @version 2006-11-11
 */
public class RoundClockFacePanel extends DrawPanel {
    /**
     * Instances of this inner class call the <code>repaint</code> method in the parent object (which is a
     * <code>DrawPanel</code> that is a <code>RoundClockFacePanel</code> application) every second.
     *
     * @authors Graham Roberts and Russel Winder
     * @version 2006-12-03
     */
    private class SecondTicker extends Thread {
        public void run ( ) {
            while ( true ) {
                try { sleep ( 1000 ) ; }
            }
        }
    }
}
```

```

        catch ( Exception e ) { }
        repaint ( ) ;
    }
}
}
private final SecondTicker ticker ;
private final int centerX ;
private final int centerY ;
private final int diameter ;
private final int radius ;
private final int boundaryWidth ;
private final Color backgroundColour ;
private final Color faceColour ;
private final Line2D line ;
private final BasicStroke secondHandStroke ;
private final BasicStroke minuteHandStroke ;
private final BasicStroke hourHandStroke ;
private final BasicStroke tickStroke ;
private final BasicStroke thickTickStroke ;
private final BasicStroke clockBoundary ;
private final Ellipse2D clockFace ;
private final Rectangle2D clockBackground ;
public RoundClockFacePanel ( final int width , final int height , final int margin ,
                             final Color background , final Color face ) {
    super ( width , height ) ;
    centerX = getWidth ( ) / 2 ;
    centerY = getHeight ( ) / 2 ;
    diameter = Math.min ( getWidth ( ) , getHeight ( ) ) - margin ;
    radius = diameter / 2 ;
    boundaryWidth = length ( 0.03 ) ;
    backgroundColour = background ;
    faceColour = face ;
    ticker = new SecondTicker ( ) ;
    ticker.start ( ) ;
    line = new Line2D.Double ( 0 , 0 , 0 , 0 ) ;
    secondHandStroke =
        new BasicStroke ( length ( 0.01 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    minuteHandStroke =
        new BasicStroke ( length ( 0.03 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    hourHandStroke =
        new BasicStroke ( length ( 0.05 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    tickStroke =
        new BasicStroke ( length ( 0.01 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    thickTickStroke =
        new BasicStroke ( length ( 0.02 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    clockFace = new Ellipse2D.Double ( centerX - radius , centerY - radius , diameter , diameter ) ;
    clockBackground = new Rectangle2D.Double ( 0 , 0 , getWidth ( ) , getHeight ( ) ) ;
    clockBoundary = new BasicStroke ( boundaryWidth ) ;
}
public RoundClockFacePanel ( final int width , final int height , final int margin ) {
    this ( width , height , margin , Color.black , Color.white ) ;
}
private void drawAngledLine ( final Graphics2D g2d ,
                              final int x , final int y ,
                              final int startLength , final int endLength ,
                              final BasicStroke stroke , final int position , final int scale ) {

```

```

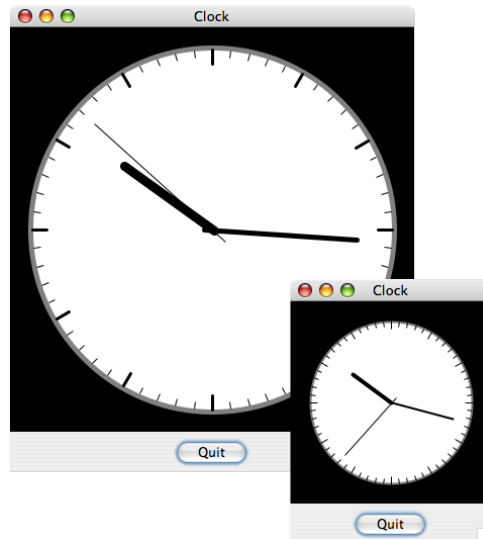
g2d.setPaint ( Color.black );
g2d.setStroke ( stroke );
final double angle = ( Math.PI * ( ( ( position * 360.0 ) / scale ) - 90.0 ) ) / 180.0 ;
line.setLine (
    x + ( startLength * Math.cos ( angle ) ),
    y + ( startLength * Math.sin ( angle ) ),
    x + ( endLength * Math.cos ( angle ) ),
    y + ( endLength * Math.sin ( angle ) )
);
g2d.draw ( line );
}
private int length ( final double scale ) {
    return (int) ( radius * scale );
}
private void drawTicks ( final Graphics2D g2d , final int boundaryWidth ) {
    for ( int tick = 0 ; tick < 60 ; tick++ ) {
        final int end = radius - ( ( ( tick % 5 ) == 0 ) ? length ( 0.09 ) : length ( 0.05 ) );
        final BasicStroke stroke = ( tick % 5 == 0 ) ? thickTickStroke : tickStroke ;
        drawAngledLine ( g2d , centerX , centerY , radius - boundaryWidth + 3 , end , stroke , tick , 60 ) ;
    }
}
private void drawClockFace ( final Graphics2D g2d ) {
    g2d.setPaint ( backgroundColour );
    g2d.fill ( clockBackground );
    g2d.setPaint ( faceColour );
    g2d.fill ( clockFace );
    g2d.setPaint ( Color.gray );
    g2d.setStroke ( clockBoundary );
    g2d.draw ( clockFace );
    drawTicks ( g2d , boundaryWidth );
}
private void drawSecondHand ( final Graphics2D g2d , final Calendar time ) {
    int second = time.get ( Calendar.SECOND );
    drawAngledLine ( g2d , centerX , centerY , length ( 0.88 ) , - length ( 0.1 ) , secondHandStroke , second , 60 ) ;
}
private void drawMinuteHand ( final Graphics2D g2d , final Calendar time ) {
    int minute = ( time.get ( Calendar.MINUTE ) * 10 ) + (int) ( 10 * ( time.get ( Calendar.SECOND ) / 60.0 ) ) ;
    drawAngledLine ( g2d , centerX , centerY , length ( 0.8 ) , - length ( 0.05 ) , minuteHandStroke , minute , 600 ) ;
}
private void drawHourHand ( final Graphics2D g2d , final Calendar time ) {
    int hour = ( time.get ( Calendar.HOUR ) * 10 ) + (int) ( 10 * ( time.get ( Calendar.MINUTE ) / 60.0 ) ) ;
    drawAngledLine ( g2d , centerX , centerY , length ( 0.6 ) , - length ( 0.02 ) , hourHandStroke , hour , 120 ) ;
}
private void drawHands ( final Graphics2D g2d ) {
    final Calendar time = Calendar.getInstance ( ) ;
    drawSecondHand ( g2d , time ) ;
    drawMinuteHand ( g2d , time ) ;
    drawHourHand ( g2d , time ) ;
}
public void paint ( final Graphics g ) {
    final Graphics2D g2d = (Graphics2D) g ;
    g2d.setRenderingHint ( RenderingHints.KEY_ANTIALIASING , RenderingHints.VALUE_ANTIALIAS_ON ) ;
    g2d.setRenderingHint ( RenderingHints.KEY_RENDERING , RenderingHints.VALUE_RENDER_QUALITY ) ;
    drawClockFace ( g2d ) ;
    drawHands ( g2d ) ;
}

```

```
public static void main ( final String[] args ) {  
    DrawFrame.display ( "Clock", new RoundClockFacePanel ( 400 , 400 , 40 ) );  
}  
}
```

As the clock program could potentially be left running for some time the graphics objects used to display the clock face are all created in the main constructor and reused throughout the time the clock is left running. This avoids having to repeatedly create and then discard the `Line`, `BasicStroke` and other graphics objects every second as the clock face is re-drawn.

The image below shows two example clock faces of different sizes, illustrating how the clock face scales to different size panels. When displayed the second hand ticks round at one second intervals, with the minute and hour hands moving as appropriate.



9.3 *Extend the clock program to display five clocks showing Tokyo time, Moscow time, London time, New York time and San Francisco time.*

The main challenge here is to find out how to determine the time in each of the cities relative to the local time on the computer where the program will be run. As always the first step in finding a solution to this kind of problem is to investigate what the standard Java libraries have to offer. This is very worthwhile as it turns out that the libraries offer a complete solution in the form of class `TimeZone` in the package `java.util`.

A `TimeZone` object can be initialised to represent any time zone such as GMT, EST or PST. The object can then be passed to a `Calendar` instance in order to access the time in the particular

zone. A time zone is identified using a set of standard strings such as "GMT", "PST" and "America/New_York" (see <http://mindprod.com/jgloss/timezone.html> for a list of strings used by Java and <http://en.wikipedia.org/wiki/Timezone> for information about time zones in general).

The class below is a modified version of the one from the previous exercise answer. An extra `String` parameter has been added to the constructor to specify the time zone. The only other modification is to add the code to get the `TimeZone` object and call the `setTimeZone` method on the `Calendar` object used to determine the time to display. This is done by these two lines of code in the `drawHands` method:

```
final Calendar time = Calendar.getInstance ( ) ;
time.setTimeZone ( TimeZone.getTimeZone ( location ) ) ;
```

The main method creates the five clocks for each of the cities listed in the question.

```
import java.awt.BasicStroke ;
import java.awt.Color ;
import java.awt.Font ;
import java.awt.geom.Rectangle2D ;
import java.awt.geom.Line2D ;
import java.awt.geom.Ellipse2D ;
import java.awt.Graphics ;
import java.awt.Graphics2D ;
import java.awt.RenderingHints ;
import java.util.Calendar ;
import java.util.TimeZone ;

/**
 * A program to display an analogue clock face with hour, minute and second hands that is updated every
 * second. The time displayed is for the selected time zone.
 *
 * @author Graham Roberts and Russel Winder
 * @version 2006-11-11
 */
public class WorldRoundClockFacePanel extends DrawPanel {
    /**
     * Instances of this inner class call the repaint method in the parent object (which is a
     * DrawPanel that is a RoundClockFacePanel application) every second.
     *
     * @authors Graham Roberts and Russel Winder
     * @version 2006-12-03
     */
    private class SecondTicker extends Thread {
        public void run ( ) {
            while ( true ) {
                try { sleep ( 1000 ) ; }
                catch ( Exception e ) { }
                repaint ( ) ;
            }
        }
    }
    private final SecondTicker ticker ;
    private final String location ;
    private final int centerX ;
```

```

private final int centerY ;
private final int diameter ;
private final int radius ;
private final int boundaryWidth ;
private final Color backgroundColour ;
private final Color faceColour ;
private final Line2D line ;
private final BasicStroke secondHandStroke ;
private final BasicStroke minuteHandStroke ;
private final BasicStroke hourHandStroke ;
private final BasicStroke tickStroke ;
private final BasicStroke thickTickStroke ;
private final BasicStroke clockBoundary ;
private final Ellipse2D clockFace ;
private final Rectangle2D clockBackground ;
public WorldRoundClockFacePanel ( final String clockLocation , final int width , final int height ,
                                     final int margin , final Color background , final Color face ) {

    super ( width , height ) ;
    location = clockLocation ;
    centerX = getWidth ( ) / 2 ;
    centerY = getHeight ( ) / 2 ;
    diameter = Math.min ( getWidth ( ) , getHeight ( ) ) - margin ;
    radius = diameter / 2 ;
    boundaryWidth = length ( 0.03 ) ;
    backgroundColour = background ;
    faceColour = face ;
    ticker = new SecondTicker ( ) ;
    ticker.start ( ) ;
    line = new Line2D.Double ( 0 , 0 , 0 , 0 ) ;
    secondHandStroke =
        new BasicStroke ( length ( 0.01 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    minuteHandStroke =
        new BasicStroke ( length ( 0.03 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    hourHandStroke =
        new BasicStroke ( length ( 0.05 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    tickStroke =
        new BasicStroke ( length ( 0.01 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    thickTickStroke =
        new BasicStroke ( length ( 0.02 ) , BasicStroke.CAP_ROUND , BasicStroke.JOIN_ROUND ) ;
    clockFace = new Ellipse2D.Double ( centerX - radius , centerY - radius , diameter , diameter ) ;
    clockBackground = new Rectangle2D.Double ( 0 , 0 , getWidth ( ) , getHeight ( ) ) ;
    clockBoundary = new BasicStroke ( boundaryWidth ) ;
}
public WorldRoundClockFacePanel ( final String clockLocation , final int width ,
                                     final int height , final int margin ) {
    this ( clockLocation , width , height , margin , Color.black , Color.white ) ;
}
private void drawAngledLine ( final Graphics2D g2d ,
                               final int x , final int y ,
                               final int startLength , final int endLength ,
                               final BasicStroke stroke , final int position , final int scale ) {
    g2d.setPaint ( Color.black ) ;
    g2d.setStroke ( stroke ) ;
    final double angle = ( Math.PI * ( ( position * 360.0 ) / scale ) - 90.0 ) / 180.0 ;
    line.setLine (
        x + ( startLength * Math.cos ( angle ) ) ,

```

```

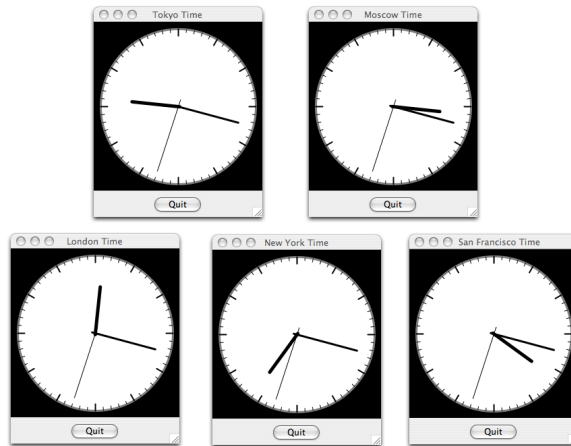
        y + ( startLength * Math.sin ( angle ) ),
        x + ( endLength * Math.cos ( angle ) ),
        y + ( endLength * Math.sin ( angle )
    );
    g2d.draw ( line );
}
private int length ( final double scale ) {
    return (int) ( radius * scale );
}
private void drawTicks ( final Graphics2D g2d, final int boundaryWidth ) {
    for ( int tick = 0; tick < 60; tick++ ) {
        final int end = radius - ( ( tick % 5 ) == 0 ) ? length ( 0.09 ) : length ( 0.05 );
        final BasicStroke stroke = ( tick % 5 == 0 ) ? thickTickStroke : tickStroke ;
        drawAngledLine ( g2d, centerX, centerY, radius - boundaryWidth + 3, end, stroke, tick, 60 );
    }
}
private void drawClockFace ( final Graphics2D g2d ) {
    g2d.setPaint ( backgroundColour );
    g2d.fill ( clockBackground );
    g2d.setPaint ( faceColour );
    g2d.fill ( clockFace );
    g2d.setPaint ( Color.gray );
    g2d.setStroke ( clockBoundary );
    g2d.draw ( clockFace );
    drawTicks ( g2d, boundaryWidth );
}
private void drawSecondHand ( final Graphics2D g2d, final Calendar time ) {
    int second = time.get ( Calendar.SECOND );
    drawAngledLine ( g2d, centerX, centerY, length ( 0.88 ), - length ( 0.1 ), secondHandStroke, second, 60 );
}
private void drawMinuteHand ( final Graphics2D g2d, final Calendar time ) {
    int minute = ( time.get ( Calendar.MINUTE ) * 10 ) + (int) ( 10 * ( time.get ( Calendar.SECOND ) / 60.0 ) );
    drawAngledLine ( g2d, centerX, centerY, length ( 0.8 ), - length ( 0.05 ), minuteHandStroke, minute, 600 );
}
private void drawHourHand ( final Graphics2D g2d, final Calendar time ) {
    int hour = ( time.get ( Calendar.HOUR ) * 10 ) + (int) ( 10 * ( time.get ( Calendar.MINUTE ) / 60.0 ) );
    drawAngledLine ( g2d, centerX, centerY, length ( 0.6 ), - length ( 0.02 ), hourHandStroke, hour, 120 );
}
private void drawHands ( final Graphics2D g2d ) {
    final Calendar time = Calendar.getInstance ( );
    time.setTimeZone ( TimeZone.getTimeZone ( location ) );
    drawSecondHand ( g2d, time );
    drawMinuteHand ( g2d, time );
    drawHourHand ( g2d, time );
}
public void paint ( final Graphics g ) {
    final Graphics2D g2d = (Graphics2D) g ;
    g2d.setRenderingHint ( RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON );
    g2d.setRenderingHint ( RenderingHints.KEY_RENDERING, RenderingHints.VALUE_RENDER_QUALITY );
    drawClockFace ( g2d );
    drawHands ( g2d );
}
public static void main ( final String[] args ) {
    DrawFrame.display ( "Tokyo Time", new WorldRoundClockFacePanel ( "Asia/Tokyo", 250, 250, 20 ) );
    DrawFrame.display ( "Moscow Time", new WorldRoundClockFacePanel ( "Europe/Moscow", 250, 250, 20 ) );
    DrawFrame.display ( "London Time", new WorldRoundClockFacePanel ( "Europe/London", 250, 250, 20 ) );
}

```

```
DrawFrame.display ( "New York Time" , new WorldRoundClockFacePanel ( "America/New_York" , 250 , 250 , 20 ) );  
DrawFrame.display ( "San Francisco Time" , new WorldRoundClockFacePanel ( "America/Los_Angeles" , 250 , 250 , 20 ) );  
}  
}
```

Note that the time zone name strings for Tokyo, Moscow, London and New York happen to use the city name (there are also alternative names for all of these zones). However, there is no "America/San_Francisco" name as the string "America/Los_Angeles" is used as the name for the time zone that San Francisco is in. A web site like <http://www.timeanddate.com/worldclock/> can be used to confirm that the correct times are displayed when the program is run.

When run the program displays the following (the windows need to be positioned on the screen manually):



9.4 *Extend Exercise 9.2 so that the ClockPanel displays four clock faces, one updated continuously, one once per second, one once per minute and one every five minutes.*