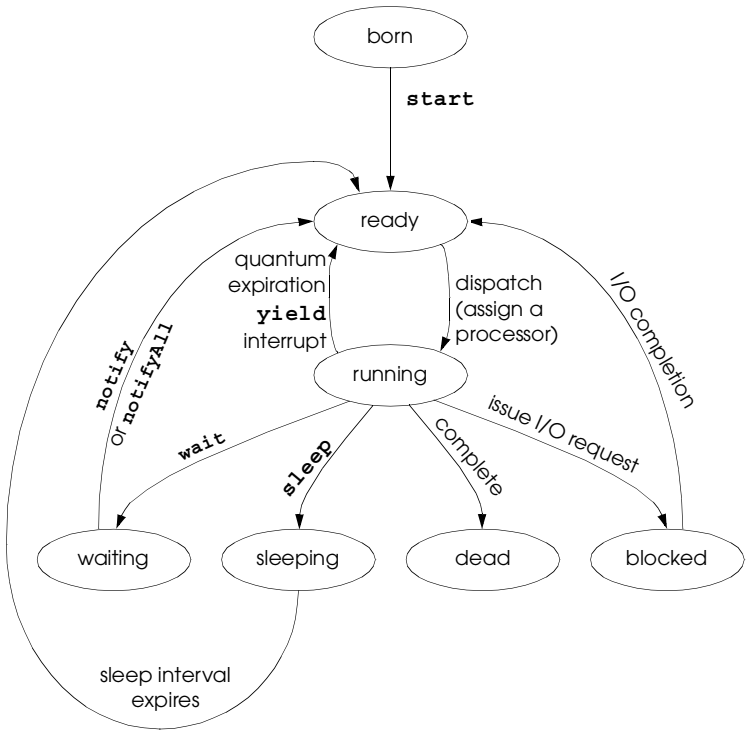

Fig. 15.1 Life cycle of a thread.

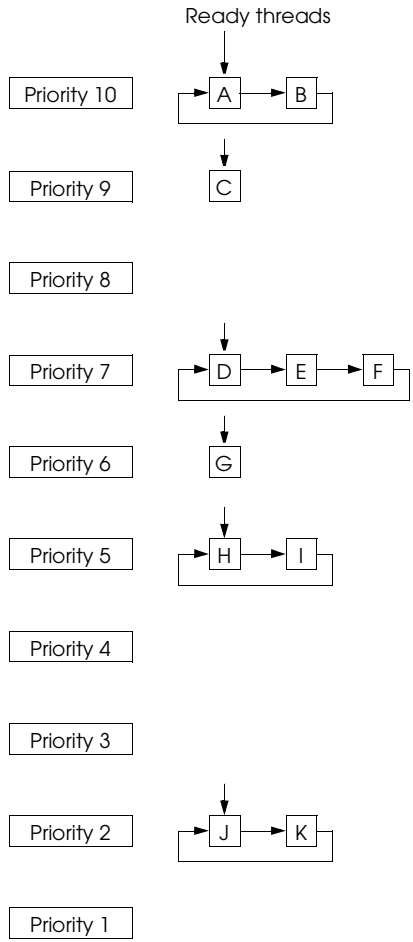

Fig. 15.2 Java thread priority scheduling.

```
1  // Fig. 15.3: ThreadTester.java
2  // Show multiple threads printing at different intervals.
3
4  public class ThreadTester {
5     public static void main( String args[] )
6     {
7        PrintThread thread1, thread2, thread3, thread4;
8
9        thread1 = new PrintThread( "thread1" );
10       thread2 = new PrintThread( "thread2" );
11       thread3 = new PrintThread( "thread3" );
12       thread4 = new PrintThread( "thread4" );
13
14       System.err.println( "\nStarting threads" );
15
16       thread1.start();
17       thread2.start();
18       thread3.start();
19       thread4.start();
20
21       System.err.println( "Threads started\n" );
22    }
23 }
24
25 class PrintThread extends Thread {
26    private int sleepTime;
27
28    // PrintThread constructor assigns name to thread
29    // by calling Thread constructor
30    public PrintThread( String name )
31    {
32       super( name );
33
34       // sleep between 0 and 5 seconds
35       sleepTime = (int) ( Math.random() * 5000 );
36
37       System.err.println( "Name: " + getName() +
38                           ";  sleep: " + sleepTime );
39    }
40
41    // execute the thread
42    public void run()
43    {
44       // put thread to sleep for a random interval
45       try {
46          System.err.println( getName() + " going to sleep" );
47          Thread.sleep( sleepTime );
48       }
49       catch ( InterruptedException exception ) {
50          System.err.println( exception.toString() );
51       }
52
```

Fig. 15.3 Multiple threads printing at random intervals (part 1 of 2).

```
53         // print thread name
54         System.err.println( getName() + " done sleeping" );
55      }
56   }
```

```
Name: thread1;  sleep: 1653
Name: thread2;  sleep: 2910
Name: thread3;  sleep: 4436
Name: thread4;  sleep: 201

Starting threads
Threads started

thread1 going to sleep
thread2 going to sleep
thread3 going to sleep
thread4 going to sleep
thread4 done sleeping
thread1 done sleeping
thread2 done sleeping
thread3 done sleeping
```

```
Name: thread1;  sleep: 3876
Name: thread2;  sleep: 64
Name: thread3;  sleep: 1752
Name: thread4;  sleep: 3120

Starting threads
Threads started

thread2 going to sleep
thread4 going to sleep
thread1 going to sleep
thread3 going to sleep
thread2 done sleeping
thread3 done sleeping
thread4 done sleeping
thread1 done sleeping
```

Fig. 15.3 Multiple threads printing at random intervals (part 2 of 2).

```
// Fig. 15.4: SharedCell.java
// Show multiple threads modifying shared object.
public class SharedCell {
   public static void main( String args[] )
   {
      HoldIntegerUnsynchronized h =
         new HoldIntegerUnsynchronized();
      ProduceInteger p = new ProduceInteger( h );
      ConsumeInteger c = new ConsumeInteger( h );

      p.start();
      c.start();
   }
}
```

Fig. 15.4 Threads modifying a shared object without synchronization (part 1 of 4).

```
// Fig. 15.4: ProduceInteger.java
// Definition of threaded class ProduceInteger
public class ProduceInteger extends Thread {
   private HoldIntegerUnsynchronized pHold;

   public ProduceInteger( HoldIntegerUnsynchronized h )
   {
      super( "ProduceInteger" );
      pHold = h;
   }

   public void run()
   {
      for ( int count = 1; count <= 10; count++ ) {
         // sleep for a random interval
         try {
            Thread.sleep( (int) ( Math.random() * 3000 ) );
         }
         catch( InterruptedException e ) {
            System.err.println( e.toString() );
         }

         pHold.setSharedInt( count );
      }

      System.err.println( getName() +
         " finished producing values" +
         "\nTerminating " + getName() );
   }
}
```

Fig. 15.4 Threads modifying a shared object without synchronization (part 2 of 4).

```
// Fig. 15.4: ConsumeInteger.java
// Definition of threaded class ConsumeInteger
public class ConsumeInteger extends Thread {
   private HoldIntegerUnsynchronized cHold;
```

```
49
50    public ConsumeInteger( HoldIntegerUnsynchronized h )
51    {
52       super( "ConsumeInteger" );
53       cHold = h;
54    }
55
56    public void run()
57    {
58       int val, sum = 0;
59
60       do {
61          // sleep for a random interval
62          try {
63             Thread.sleep( (int) ( Math.random() * 3000 ) );
64          }
65          catch( InterruptedException e ) {
66             System.err.println( e.toString() );
67          }
68
69          val = cHold.getSharedInt();
70          sum += val;
71       } while ( val != 10 );
72
73       System.err.println(
74          getName() + " retrieved values totaling: " + sum +
75          "\nTerminating " + getName() );
76    }
77 }
```

Fig. 15.4 Threads modifying a shared object without synchronization (part 3 of 4).

```
78 // Fig. 15.4: HoldIntegerUnsynchronized.java
79 // Definition of class HoldIntegerUnsynchronized
80 public class HoldIntegerUnsynchronized {
81    private int sharedInt = -1;
82
83    public void setSharedInt( int val )
84    {
85       System.err.println( Thread.currentThread().getName() +
86          " setting sharedInt to " + val );
87       sharedInt = val;
88    }
89
90    public int getSharedInt()
91    {
92       System.err.println( Thread.currentThread().getName() +
93          " retrieving sharedInt value " + sharedInt );
94       return sharedInt;
95    }
```

```
96    }
```

```
ConsumeInteger retrieving sharedInt value -1
ConsumeInteger retrieving sharedInt value -1
ProduceInteger setting sharedInt to 1
ProduceInteger setting sharedInt to 2
ConsumeInteger retrieving sharedInt value 2
ProduceInteger setting sharedInt to 3
ProduceInteger setting sharedInt to 4
ProduceInteger setting sharedInt to 5
ConsumeInteger retrieving sharedInt value 5
ProduceInteger setting sharedInt to 6
ProduceInteger setting sharedInt to 7
ProduceInteger setting sharedInt to 8
ConsumeInteger retrieving sharedInt value 8
ConsumeInteger retrieving sharedInt value 8
ProduceInteger setting sharedInt to 9
ConsumeInteger retrieving sharedInt value 9
ConsumeInteger retrieving sharedInt value 9
ProduceInteger setting sharedInt to 10
ProduceInteger finished producing values
Terminating ProduceInteger
ConsumeInteger retrieving sharedInt value 10
ConsumeInteger retrieved values totaling: 49
Terminating ConsumeInteger
```

Fig. 15.4 Threads modifying a shared object without synchronization (part 4 of 4).

```
1  // Fig. 15.5: SharedCell.java
2  // Show multiple threads modifying shared object.
3  public class SharedCell {
4     public static void main( String args[] )
5     {
6        HoldIntegerSynchronized h =
7           new HoldIntegerSynchronized();
8        ProduceInteger p = new ProduceInteger( h );
9        ConsumeInteger c = new ConsumeInteger( h );
10
11       p.start();
12       c.start();
13    }
14 }
```

Fig. 15.5 Threads modifying a shared object with synchronization (part 1 of 5).

```
15 // Fig. 15.5: ProduceInteger.java
16 // Definition of threaded class ProduceInteger
17 public class ProduceInteger extends Thread {
18    private HoldIntegerSynchronized pHold;
19
20    public ProduceInteger( HoldIntegerSynchronized h )
21    {
22       super( "ProduceInteger" );
23       pHold = h;
24    }
25
26    public void run()
27    {
28       for ( int count = 1; count <= 10; count++ ) {
29          // sleep for a random interval
30          try {
31             Thread.sleep( (int) ( Math.random() * 3000 ) );
32          }
33          catch( InterruptedException e ) {
34             System.err.println( e.toString() );
35          }
36
37          pHold.setSharedInt( count );
38       }
39
40       System.err.println( getName() +
41          " finished producing values" +
42          "\nTerminating " + getName() );
43    }
44 }
```

Fig. 15.5 Threads modifying a shared object with synchronization (part 2 of 5).

```
45 // Fig. 15.5: ConsumeInteger.java
46 // Definition of threaded class ConsumeInteger
```

```
47  public class ConsumeInteger extends Thread {
48     private HoldIntegerSynchronized cHold;
49
50     public ConsumeInteger( HoldIntegerSynchronized h )
51     {
52        super( "ConsumeInteger" );
53        cHold = h;
54     }
55
56     public void run()
57     {
58        int val, sum = 0;
59
60        do {
61           // sleep for a random interval
62           try {
63              Thread.sleep( (int) ( Math.random() * 3000 ) );
64           }
65           catch( InterruptedException e ) {
66              System.err.println( e.toString() );
67           }
68
69           val = cHold.getSharedInt();
70           sum += val;
71        } while ( val != 10 );
72
73        System.err.println(
74           getName() + " retrieved values totaling: " + sum +
75           "\nTerminating " + getName() );
76     }
77  }
```

Fig. 15.5 Threads modifying a shared object with synchronization (part 3 of 5).

```
78  // Fig. 15.5: HoldIntegerSynchronized.java
79  // Definition of class HoldIntegerSynchronized that
80  // uses thread synchronization to ensure that both
81  // threads access sharedInt at the proper times.
82  public class HoldIntegerSynchronized {
83     private int sharedInt = -1;
84     private boolean writeable = true;  // condition variable
85
86     public synchronized void setSharedInt( int val )
87     {
88        while ( !writeable ) {  // not the producer's turn
89           try {
90              wait();
91           }
92           catch ( InterruptedException e ) {
93              e.printStackTrace();
94           }
95        }
96
97        System.err.println( Thread.currentThread().getName() +
98           " setting sharedInt to " + val );
```

```
99          sharedInt = val;
100
101         writeable = false;
102         notify();  // tell a waiting thread to become ready
103      }
104
105      public synchronized int getSharedInt()
106      {
107         while ( writeable ) {   // not the consumer's turn
108            try {
109               wait();
110            }
111            catch ( InterruptedException e ) {
112               e.printStackTrace();
113            }
114         }
115
116         writeable = true;
117         notify();  // tell a waiting thread to become ready
118
119         System.err.println( Thread.currentThread().getName() +
120            " retrieving sharedInt value " + sharedInt );
121         return sharedInt;
122      }
123   }
```

Fig. 15.5 Threads modifying a shared object with synchronization (part 4 of 5).

```
ProduceInteger setting sharedInt to 1
ConsumeInteger retrieving sharedInt value 1
ProduceInteger setting sharedInt to 2
ConsumeInteger retrieving sharedInt value 2
ProduceInteger setting sharedInt to 3
ConsumeInteger retrieving sharedInt value 3
ProduceInteger setting sharedInt to 4
ConsumeInteger retrieving sharedInt value 4
ProduceInteger setting sharedInt to 5
ConsumeInteger retrieving sharedInt value 5
ProduceInteger setting sharedInt to 6
ConsumeInteger retrieving sharedInt value 6
ProduceInteger setting sharedInt to 7
ConsumeInteger retrieving sharedInt value 7
ProduceInteger setting sharedInt to 8
ConsumeInteger retrieving sharedInt value 8
ProduceInteger setting sharedInt to 9
ConsumeInteger retrieving sharedInt value 9
ProduceInteger setting sharedInt to 10
ProduceInteger finished producing values
Terminating ProduceInteger
ConsumeInteger retrieving sharedInt value 10
ConsumeInteger retrieved values totaling: 55
Terminating ConsumeInteger
```

Fig. 15.5 Threads modifying a shared object with synchronization.

```
1  // Fig. 15.6: SharedCell.java
2  // Show multiple threads modifying shared object.
3  import java.text.DecimalFormat;
4  import java.awt.*;
5  import java.awt.event.*;
6  import javax.swing.*;
7
8  public class SharedCell extends JFrame {
9     public SharedCell()
10    {
11       super( "Demonstrating Thread Synchronization" );
12       JTextArea output = new JTextArea( 20, 30 );
13
14       getContentPane().add( new JScrollPane( output ) );
15       setSize( 500, 500 );
16       show();
17
18       // set up threads and start threads
19       HoldIntegerSynchronized h =
20          new HoldIntegerSynchronized( output );
21       ProduceInteger p = new ProduceInteger( h, output );
22       ConsumeInteger c = new ConsumeInteger( h, output );
23
24       p.start();
25       c.start();
26    }
27
28    public static void main( String args[] )
29    {
30       SharedCell app = new SharedCell();
31       app.addWindowListener(
32          new WindowAdapter() {
33             public void windowClosing( WindowEvent e )
34             {
35                System.exit( 0 );
36             }
37          }
38       );
39    }
40 }
```

Fig. 15.6 Threads modifying a shared array of cells (part 1 of 8).

```
41 // Fig. 15.6: ProduceInteger.java
42 // Definition of threaded class ProduceInteger
43 import javax.swing.JTextArea;
44
45 public class ProduceInteger extends Thread {
46    private HoldIntegerSynchronized pHold;
47    private JTextArea output;
48
```

Fig. 15.6 Threads modifying a shared array of cells (part 2 of 8).

```
49     public ProduceInteger( HoldIntegerSynchronized h,
50                            JTextArea o )
51     {
52        super( "ProduceInteger" );
53        pHold = h;
54        output = o;
55     }
56
57     public void run()
58     {
59        for ( int count = 1; count <= 10; count++ ) {
60           // sleep for a random interval
61           // Note: Interval shortened purposely to fill buffer
62           try {
63              Thread.sleep( (int) ( Math.random() * 500 ) );
64           }
65           catch( InterruptedException e ) {
66              System.err.println( e.toString() );
67           }
68
69           pHold.setSharedInt( count );
70        }
71
72        output.append( "\n" + getName() +
73           " finished producing values" +
74           "\nTerminating " + getName() + "\n" );
75     }
76  }
```

Fig. 15.6 Threads modifying a shared array of cells (part 3 of 8).

```
77  // Fig. 15.6: ConsumeInteger.java
78  // Definition of threaded class ConsumeInteger
79  import javax.swing.JTextArea;
80
81  public class ConsumeInteger extends Thread {
82     private HoldIntegerSynchronized cHold;
83     private JTextArea output;
84
85     public ConsumeInteger( HoldIntegerSynchronized h,
86                            JTextArea o )
87     {
88        super( "ConsumeInteger" );
89        cHold = h;
90        output = o;
91     }
92
93     public void run()
94     {
95        int val, sum = 0;
96
```

Fig. 15.6 Threads modifying a shared array of cells (part 4 of 8).

```
97          do {
98             // sleep for a random interval
99             try {
100               Thread.sleep( (int) ( Math.random() * 3000 ) );
101            }
102            catch( InterruptedException e ) {
103               System.err.println( e.toString() );
104            }
105
106            val = cHold.getSharedInt();
107            sum += val;
108         } while ( val != 10 );
109
110         output.append( "\n" + getName() +
111            " retrieved values totaling: " + sum +
112            "\nTerminating " + getName() + "\n" );
113      }
114   }
```

Fig. 15.6 Threads modifying a shared array of cells (part 5 of 8).

```
115   // Fig. 15.6: HoldIntegerSynchronized.java
116   // Definition of class HoldIntegerSynchronized that
117   // uses thread synchronization to ensure that both
118   // threads access sharedInt at the proper times.
119   import javax.swing.JTextArea;
120   import java.text.DecimalFormat;
121
122   public class HoldIntegerSynchronized {
123      private int sharedInt[] = { -1, -1, -1, -1, -1 };
124      private boolean writeable = true;
125      private boolean readable = false;
126      private int readLoc = 0, writeLoc = 0;
127      private JTextArea output;
128
129      public HoldIntegerSynchronized( JTextArea o )
130      {
131         output = o;
132      }
133
134      public synchronized void setSharedInt( int val )
135      {
136         while ( !writeable ) {
137            try {
138               output.append( " WAITING TO PRODUCE " + val );
139               wait();
140            }
141            catch ( InterruptedException e ) {
142               System.err.println( e.toString() );
143            }
144         }
145
```

Fig. 15.6 Threads modifying a shared array of cells (part 6 of 8).

```
146         sharedInt[ writeLoc ] = val;
147         readable = true;
148
149         output.append( "\nProduced " + val +
150                        " into cell " + writeLoc );
151
152         writeLoc = ( writeLoc + 1 ) % 5;
153
154         output.append( "\twrite " + writeLoc +
155                        "\tread " + readLoc);
156         displayBuffer( output, sharedInt );
157
158         if ( writeLoc == readLoc ) {
159            writeable = false;
160            output.append( "\nBUFFER FULL" );
161         }
162
163         notify();
164      }
165
166      public synchronized int getSharedInt()
167      {
168         int val;
169
170         while ( !readable ) {
171            try {
172               output.append( " WAITING TO CONSUME" );
173               wait();
174            }
175            catch ( InterruptedException e ) {
176               System.err.println( e.toString() );
177            }
178         }
179
180         writeable = true;
181         val = sharedInt[ readLoc ];
182
183         output.append( "\nConsumed " + val +
184                        " from cell " + readLoc );
185
186         readLoc = ( readLoc + 1 ) % 5;
187
188         output.append( "\twrite " + writeLoc +
189                        "\tread " + readLoc );
190         displayBuffer( output, sharedInt );
191
192         if ( readLoc == writeLoc ) {
193            readable = false;
194            output.append( "\nBUFFER EMPTY" );
195         }
196
197         notify();
```

Fig. 15.6 Threads modifying a shared array of cells (part 7 of 8).

```
198         return val;
199      }
200
201      public void displayBuffer( JTextArea out, int buf[] )
202      {
203         DecimalFormat formatNumber = new DecimalFormat( " #;-#" );
204         output.append( "\tbuffer: " );
205
206         for ( int i = 0; i < buf.length; i++ )
207            out.append( " " + formatNumber.format( buf[ i ] ));
208      }
209   }
```

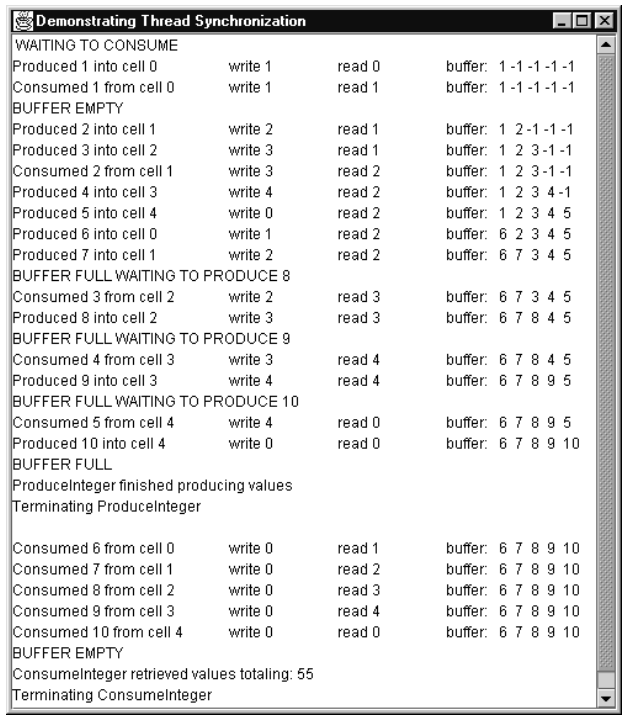

Fig. 15.6 Threads modifying a shared array of cells (part 8 of 8).

```
// Fig. 15.7: RandomCharacters.java
// Demonstrating the Runnableinterface
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RandomCharacters extends JApplet
                              implements Runnable,
                                         ActionListener {
   private String alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
   private JLabel outputs[];
   private JCheckBox checkboxes[];
   private final static int SIZE = 3;

   private Thread threads[];
   private boolean suspended[];

   public void init()
   {
      outputs = new JLabel[ SIZE ];
      checkboxes = new JCheckBox[ SIZE ];
```

Fig. 15.7 Demonstrating the **`Runnable`** interface (part 1 of 3).

```
      threads = new Thread[ SIZE ];
      suspended = new boolean[ SIZE ];

      Container c = getContentPane();
      c.setLayout( new GridLayout( SIZE, 2, 5, 5 ) );

      for ( int i = 0; i < SIZE; i++ ) {
         outputs[ i ] = new JLabel();
         outputs[ i ].setBackground( Color.green );
         outputs[ i ].setOpaque( true );
         c.add( outputs[ i ] );

         checkboxes[ i ] = new JCheckBox( "Suspended" );
         checkboxes[ i ].addActionListener( this );
         c.add( checkboxes[ i ] );
      }
   }

   public void start()
   {
      // create threads and start every time start is called
      for ( int i = 0; i < threads.length; i++ ) {
         threads[ i ] =
            new Thread( this, "Thread " + (i + 1) );
         threads[ i ].start();
      }
   }

   public void run()
```

```
51       {
52          Thread currentThread = Thread.currentThread();
53          int index = getIndex( currentThread );
54          char displayChar;
55
56          while ( threads[ index ] == currentThread ) {
57             // sleep from 0 to 1 second
58             try {
59                Thread.sleep( (int) ( Math.random() * 1000 ) );
60
61                synchronized( this ) {
62                   while ( suspended[ index ] &&
63                           threads[ index ] == currentThread )
64                      wait();
65                }
66             }
67             catch ( InterruptedException e ) {
68                System.err.println( "sleep interrupted" );
69             }
70
71             displayChar = alphabet.charAt(
72                               (int) ( Math.random() * 26 ) );
```

Fig. 15.7 Demonstrating the **`Runnable`** interface (part 2 of 3).

```
73             outputs[ index ].setText( currentThread.getName() +
74                ": " + displayChar );
75          }
76
77          System.err.println(
78             currentThread.getName() + " terminating" );
79       }
80
81       private int getIndex( Thread current )
82       {
83          for ( int i = 0; i < threads.length; i++ )
84             if ( current == threads[ i ] )
85                return i;
86
87          return -1;
88       }
89
90       public synchronized void stop()
91       {
92          // stop threads every time stop is called
93          // as the user browses another Web page
94          for ( int i = 0; i < threads.length; i++ )
95             threads[ i ] = null;
96
97          notifyAll();
98       }
99
100      public synchronized void actionPerformed( ActionEvent e )
101      {
102         for ( int i = 0; i < checkboxes.length; i++ ) {
```

```
103            if ( e.getSource() == checkboxes[ i ] ) {
104               suspended[ i ] = !suspended[ i ];
105
106               outputs[ i ].setBackground(
107                  !suspended[ i ] ? Color.green : Color.red );
108
109               if ( !suspended[ i ] )
110                  notify();
111
112               return;
113            }
114         }
115      }
116   }
```

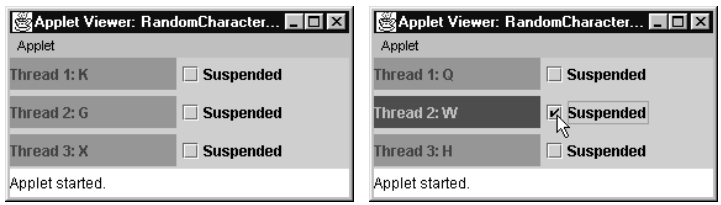

Fig. 15.7 Demonstrating the **`Runnable`** interface (part 3 of 3).