```
class AppleThread extends Thread {
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.println("Apple" + i);
    }
}


-----------------------------------------------

class OrangeThread extends Thread {
    public void run() {
        for (int i = 0; i < 6; i++)
            System.out.println("Orange" + i);
    }

}


-----------------------------------------------

public class Fruit {
    public static void main(String[] args) {
        AppleThread apple = new AppleThread();
        OrangeThread orange = new OrangeThread();
        apple.start();
//        try {
//            apple.join();
//        } catch (InterruptedException e) {
//            e.printStackTrace();
//        }
        orange.start();
        System.out.println("Finishing the main thread ...");
//        Thread.sleep(10);
    }
}


-----------------------------------------------

public class MyThread extends Thread {

    public MyThread(String name) {
        super(name);
    }

    @Override
    public void run() {
        System.out.println("MyThread - START " + Thread.currentThread().getName());
        try {
            Thread.sleep(1000);
            //Get database connection, delete unused data from DB
            doDBProcessing();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("MyThread - END " + Thread.currentThread().getName());
    }

    private void doDBProcessing() throws InterruptedException {
        Thread.sleep(5000);
    }
}

-----------------------------------------------

public class HeavyWorkRunnable implements Runnable {

    @Override
```

```
70      public void run() {
71          System.out.println("Doing heavy processing - START " + Thread.currentThread().
            getName());
72          try {
73              Thread.sleep(1000);
74              //Get database connection, delete unused data from DB
75              doDBProcessing();
76          } catch (InterruptedException e) {
77              e.printStackTrace();
78          }
79          System.out.println("Doing heavy processing - END " + Thread.currentThread().
            getName());
80      }
81
82      private void doDBProcessing() throws InterruptedException {
83          Thread.sleep(5000);
84      }
85
86  }
87
88  ------------------------------------------------
89
90  public class ThreadRunExample {
91
92      public static void main(String[] args){
93          Thread t1 = new Thread(new HeavyWorkRunnable(), "t1");
94          Thread t2 = new Thread(new HeavyWorkRunnable(), "t2");
95          System.out.println("Starting Runnable threads");
96          t1.start();
97          t2.start();
98          System.out.println("Runnable Threads has been started");
99          Thread t3 = new MyThread("t3");
100         Thread t4 = new MyThread("t4");
101         System.out.println("Starting MyThreads");
102         t3.start();
103         t4.start();
104         System.out.println("MyThreads has been started");
105
106     }
107 }
108
109 ------------------------------------------------
110
111 // Fig. 23.3: PrintTask.java
112 // PrintTask class sleeps for a random time from 0 to 5 seconds
113 import java.security.SecureRandom;
114
115 public class PrintTask implements Runnable
116 {
117    private final static SecureRandom generator = new SecureRandom();
118    private final int sleepTime; // random sleep time for thread
119    private final String taskName; // name of task
120
121    // constructor
122    public PrintTask(String taskName)
123    {
124       this.taskName = taskName;
125
126       // pick random sleep time between 0 and 5 seconds
127       sleepTime = generator.nextInt(5000); // milliseconds
128    }
129
130    // method run contains the code that a thread will execute
131    public void run()
132    {
133       try // put thread to sleep for sleepTime amount of time
134       {
135          System.out.printf("%s going to sleep for %d milliseconds.%n",
136             taskName, sleepTime);
```

```
            Thread.sleep(sleepTime); // put thread to sleep
         }
         catch (InterruptedException exception)
         {
            exception.printStackTrace();
            Thread.currentThread().interrupt(); // re-interrupt the thread
         }

         // print task name
         System.out.printf("%s done sleeping%n", taskName);
      }
   } // end class PrintTask


   ------------------------------------------------------

   // Fig. 23.4: TaskExecutor.java
   // Using an ExecutorService to execute Runnables.
   import java.util.concurrent.Executors;
   import java.util.concurrent.ExecutorService;

   public class TaskExecutor
   {
      public static void main(String[] args)
      {
         // create and name each runnable
         PrintTask task1 = new PrintTask("task1");
         PrintTask task2 = new PrintTask("task2");
         PrintTask task3 = new PrintTask("task3");

         System.out.println("Starting Executor");

         // create ExecutorService to manage threads
         ExecutorService executorService = Executors.newCachedThreadPool();

         // start the three PrintTasks
         executorService.execute(task1); // start task1
         executorService.execute(task2); // start task2
         executorService.execute(task3); // start task3

         // shut down ExecutorService--it decides when to shut down threads
         executorService.shutdown();

         System.out.printf("Tasks started, main ends.%n%n");
      }
   } // end class TaskExecutor

   ================================================================================
   UnsynchronizedExample:

   import java.security.SecureRandom;
   import java.util.Arrays;

   public class SimpleArray // CAUTION: NOT THREAD SAFE!
   {
      private static final SecureRandom generator = new SecureRandom();
      private final int[] array; // the shared integer array
      private int writeIndex = 0; // shared index of next element to write

      // construct a SimpleArray of a given size
      public SimpleArray(int size)
      {
         array = new int[size];
      }

      // add a value to the shared array
      public void add(int value)
      {
         int position = writeIndex; // store the write index
```

```
206
207         try
208         {
209            // put thread to sleep for 0-499 milliseconds
210            Thread.sleep(generator.nextInt(500));
211         }
212         catch (InterruptedException ex)
213         {
214            Thread.currentThread().interrupt(); // re-interrupt the thread
215         }
216
217         // put value in the appropriate element
218         array[position] = value;
219         System.out.printf("%s wrote %2d to element %d.%n",
220            Thread.currentThread().getName(), value, position);
221
222         ++writeIndex; // increment index of element to be written next
223         System.out.printf("Next write index: %d%n", writeIndex);
224      }
225
226      // used for outputting the contents of the shared integer array
227      public String toString()
228      {
229         return Arrays.toString(array);
230      }
231   } // end class SimpleArray
232   ------------------------------------------------------------------------
233   import java.lang.Runnable;
234
235   public class ArrayWriter implements Runnable
236   {
237      private final SimpleArray sharedSimpleArray;
238      private final int startValue;
239
240      public ArrayWriter(int value, SimpleArray array)
241      {
242         startValue = value;
243         sharedSimpleArray= array;
244      }
245
246      public void run()
247      {
248         for (int i = startValue; i < startValue + 3; i++)
249         {
250            sharedSimpleArray.add(i); // add an element to the shared array
251         }
252      }
253   } // end class ArrayWriter
254   ------------------------------------------------------------------------
255   import java.util.concurrent.Executors;
256   import java.util.concurrent.ExecutorService;
257   import java.util.concurrent.TimeUnit;
258
259   public class SharedArrayTest
260   {
261      public static void main(String[] arg)
262      {
263         // construct the shared object
264         SimpleArray sharedSimpleArray = new SimpleArray(6);
265
266         // create two tasks to write to the shared SimpleArray
267         ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
268         ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);
269
270         // execute the tasks with an ExecutorService
271         ExecutorService executorService = Executors.newCachedThreadPool();
272         executorService.execute(writer1);
273         executorService.execute(writer2);
274
```

```
275         executorService.shutdown();
276
277         try
278         {
279            // wait 1 minute for both writers to finish executing
280            boolean tasksEnded =
281               executorService.awaitTermination(1, TimeUnit.MINUTES);
282
283            if (tasksEnded)
284            {
285               System.out.printf("%nContents of SimpleArray:%n");
286               System.out.println(sharedSimpleArray); // print contents
287            }
288            else
289               System.out.println(
290                  "Timed out while waiting for tasks to finish.");
291         }
292         catch (InterruptedException ex)
293         {
294            ex.printStackTrace();
295         }
296      } // end main
297   } // end class SharedArrayTest
298   ================================================================================
299   SynchronizedExample:
300
301   import java.security.SecureRandom;
302   import java.util.Arrays;
303
304   public class SimpleArray
305   {
306      private static final SecureRandom generator = new SecureRandom();
307      private final int[] array; // the shared integer array
308      private int writeIndex = 0; // index of next element to be written
309
310      // construct a SimpleArray of a given size
311      public SimpleArray(int size)
312      {
313         array = new int[size];
314      }
315
316      // add a value to the shared array
317      public synchronized void add(int value)
318      {
319         int position = writeIndex; // store the write index
320
321         try
322         {
323            // in real applications, you shouldn't sleep while holding a lock
324            Thread.sleep(generator.nextInt(500)); // for demo only
325         }
326         catch (InterruptedException ex)
327         {
328            Thread.currentThread().interrupt();
329         }
330
331         // put value in the appropriate element
332         synchronized (this){
333            array[position] = value;
334         }
335         System.out.printf("%s wrote %2d to element %d.%n",
336            Thread.currentThread().getName(), value, position);
337
338         ++writeIndex; // increment index of element to be written next
339         System.out.printf("Next write index: %d%n", writeIndex);
340      }
341
342      // used for outputting the contents of the shared integer array
343      public synchronized String toString()
```

```
344      {
345         return Arrays.toString(array);
346      }
347   } // end class SimpleArray
348   ----------------------------------------------------------------
349   import java.lang.Runnable;
350
351   public class ArrayWriter implements Runnable
352   {
353      private final SimpleArray sharedSimpleArray;
354      private final int startValue;
355
356      public ArrayWriter(int value, SimpleArray array)
357      {
358         startValue = value;
359         sharedSimpleArray= array;
360      }
361
362      public void run()
363      {
364         for (int i = startValue; i < startValue + 3; i++)
365         {
366            sharedSimpleArray.add(i); // add an element to the shared array
367         }
368      }
369   } // end class ArrayWriter
370   ----------------------------------------------------------------
371   import java.util.concurrent.Executors;
372   import java.util.concurrent.ExecutorService;
373   import java.util.concurrent.TimeUnit;
374
375   public class SharedArrayTest
376   {
377      public static void main(String[] arg)
378      {
379         // construct the shared object
380         SimpleArray sharedSimpleArray = new SimpleArray(6);
381
382         // create two tasks to write to the shared SimpleArray
383         ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
384         ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);
385
386         // execute the tasks with an ExecutorService
387         ExecutorService executorService = Executors.newCachedThreadPool();
388         executorService.execute(writer1);
389         executorService.execute(writer2);
390
391         executorService.shutdown();
392
393         try
394         {
395            // wait 1 minute for both writers to finish executing
396            boolean tasksEnded =
397               executorService.awaitTermination(1, TimeUnit.MINUTES);
398
399            if (tasksEnded)
400            {
401               System.out.printf("%nContents of SimpleArray:%n");
402               System.out.println(sharedSimpleArray); // print contents
403            }
404            else
405               System.out.println(
406                  "Timed out while waiting for tasks to finish.");
407         }
408         catch (InterruptedException ex)
409         {
410            System.out.println(
411               "Interrupted while waiting for tasks to finish.");
412         }
```

```
   } // end main
} // end class SharedArrayTest
/**************************************************************************
ProdConsumExample:

// Fig. 23.9: Buffer.java
// Buffer interface specifies methods called by Producer and Consumer.
public interface Buffer
{
   // place int value into Buffer
   public void blockingPut(int value) throws InterruptedException;

   // obtain int value from Buffer
   public int blockingGet() throws InterruptedException;
} // end interface Buffer
//-------------------------------------------------------------
import java.util.concurrent.ArrayBlockingQueue;

public class BlockingBuffer implements Buffer
{
   private final ArrayBlockingQueue<Integer> buffer; // shared buffer

   public BlockingBuffer()
   {
      buffer = new ArrayBlockingQueue<Integer>(1);
   }

   // place value into buffer
   public void blockingPut(int value) throws InterruptedException
   {
      buffer.put(value); // place value in buffer
      System.out.printf("%s%2d\t%s%d%n", "Producer writes ", value,
         "Buffer cells occupied: ", buffer.size());
   }

   // return value from buffer
   public int blockingGet() throws InterruptedException
   {
      int readValue = buffer.take(); // remove value from buffer
      System.out.printf("%s %2d\t%s%d%n", "Consumer reads ",
         readValue, "Buffer cells occupied: ", buffer.size());

      return readValue;
   }
} // end class BlockingBuffer
//-------------------------------------------------------------
import java.security.SecureRandom;

public class Producer implements Runnable
{
   private static final SecureRandom generator = new SecureRandom();
   private final Buffer sharedLocation; // reference to shared object

   // constructor
   public Producer(Buffer sharedLocation)
   {
      this.sharedLocation = sharedLocation;
   }

   // store values from 1 to 10 in sharedLocation
   public void run()
   {
      int sum = 0;

      for (int count = 1; count <= 10; count++)
      {
         try // sleep 0 to 3 seconds, then place value in Buffer
         {
            Thread.sleep(generator.nextInt(3000)); // random sleep
```

```
482               sharedLocation.blockingPut(count); // set value in buffer
483               sum += count; // increment sum of values
484            }
485            catch (InterruptedException exception)
486            {
487               Thread.currentThread().interrupt();
488            }
489         }
490
491         System.out.printf(
492            "Producer done producing%nTerminating Producer%n");
493      }
494   } // end class Producer
495   ----------------------------------------------------------
496   import java.security.SecureRandom;
497
498   public class Consumer implements Runnable
499   {
500      private static final SecureRandom generator = new SecureRandom();
501      private final Buffer sharedLocation; // reference to shared object
502
503      // constructor
504      public Consumer(Buffer sharedLocation)
505      {
506         this.sharedLocation = sharedLocation;
507      }
508
509      // read sharedLocation's value 10 times and sum the values
510      public void run()
511      {
512         int sum = 0;
513
514         for (int count = 1; count <= 10; count++)
515         {
516            // sleep 0 to 3 seconds, read value from buffer and add to sum
517            try
518            {
519               Thread.sleep(generator.nextInt(3000));
520               sum += sharedLocation.blockingGet();
521            }
522            catch (InterruptedException exception)
523            {
524               Thread.currentThread().interrupt();
525            }
526         }
527
528         System.out.printf("%n%s %d%n%s%n",
529            "Consumer read values totaling", sum, "Terminating Consumer");
530      }
531   } // end class Consumer
532   ----------------------------------------------------------
533   import java.util.concurrent.ExecutorService;
534   import java.util.concurrent.Executors;
535   import java.util.concurrent.TimeUnit;
536
537   public class BlockingBufferTest
538   {
539      public static void main(String[] args) throws InterruptedException
540      {
541         // create new thread pool with two threads
542         ExecutorService executorService = Executors.newCachedThreadPool();
543
544         // create BlockingBuffer to store ints
545         Buffer sharedLocation = new BlockingBuffer();
546
547         executorService.execute(new Producer(sharedLocation));
548         executorService.execute(new Consumer(sharedLocation));
549
550         executorService.shutdown();
```

```
551         executorService.awaitTermination(1, TimeUnit.MINUTES);
552      }
553   } // end class BlockingBufferTest
554   ================================================================================
555   WaitNotifyExample:
556
557   // Fig. 23.16: SynchronizedBuffer.java
558   // Synchronizing access to shared mutable data using Object
559   // methods wait and notifyAll.
560   public class SynchronizedBuffer implements Buffer
561   {
562      private int buffer = -1; // shared by producer and consumer threads
563      private boolean occupied = false;
564   //   private Object o;
565
566      // place value into buffer
567      public synchronized void blockingPut(int value)
568         throws InterruptedException
569      {
570         // while there are no empty locations, place thread in waiting state
571
572         while (occupied)
573         {
574            // output thread information and buffer information, then wait
575            System.out.println("Producer tries to write."); // for demo only
576            displayState("Buffer full. Producer waits." + Thread.currentThread().getName
               ()); // for demo only
577            wait();
578         }
579
580         buffer = value; // set new buffer value
581
582         // indicate producer cannot store another value
583         // until consumer retrieves current buffer value
584         occupied = true;
585
586         displayState("Producer writes " + buffer); // for demo only
587
588         notifyAll(); // tell waiting thread(s) to enter runnable state
589      } // end method blockingPut; releases lock on SynchronizedBuffer
590
591      // return value from buffer
592      public synchronized int blockingGet() throws InterruptedException
593      {
594         // while no data to read, place thread in waiting state
595         while (!occupied)
596         {
597            // output thread information and buffer information, then wait
598            System.out.println("Consumer tries to read."); // for demo only
599            displayState("Buffer empty. Consumer waits."); // for demo only
600            wait();
601         }
602
603         // indicate that producer can store another value
604         // because consumer just retrieved buffer value
605         occupied = false;
606
607         displayState("Consumer reads " + buffer); // for demo only
608
609         notifyAll(); // tell waiting thread(s) to enter runnable state
610
611         return buffer;
612      } // end method blockingGet; releases lock on SynchronizedBuffer
613
614      // display current operation and buffer state; for demo only
615      private synchronized void displayState(String operation)
616      {
617         System.out.printf("%-40s%d\t\t%b%n%n", operation, buffer,
618            occupied);
```

```
    }
} // end class SynchronizedBuffer
-------------------------------------------------------
import java.security.SecureRandom;

public class Producer implements Runnable {
    private static final SecureRandom generator = new SecureRandom();
    private final Buffer sharedLocation; // reference to shared object

    // constructor
    public Producer(Buffer sharedLocation) {
        this.sharedLocation = sharedLocation;
    }

    // store values from 1 to 10 in sharedLocation
    public void run() {
        int sum = 0;

        for (int count = 1; count <= 10; count++) {
            try // sleep 0 to 3 seconds, then place value in Buffer
            {
                Thread.sleep(generator.nextInt(3000)); // random sleep
                sharedLocation.blockingPut(count); // set value in buffer
                sum += count; // increment sum of values
            } catch (InterruptedException exception) {
                Thread.currentThread().interrupt();
            }
        }

        System.out.printf(
                "Producer done producing%nTerminating Producer%n");
    }
} // end class Producer
-----------------------------------------------------------------
import java.security.SecureRandom;

public class Consumer implements Runnable
{
   private static final SecureRandom generator = new SecureRandom();
   private final Buffer sharedLocation; // reference to shared object

   // constructor
   public Consumer(Buffer sharedLocation)
   {
      this.sharedLocation = sharedLocation;
   }

   // read sharedLocation's value 10 times and sum the values
   public void run()
   {
      int sum = 0;

      for (int count = 1; count <= 10; count++)
      {
         // sleep 0 to 3 seconds, read value from buffer and add to sum
         try
         {
            Thread.sleep(generator.nextInt(3000));
            sum += sharedLocation.blockingGet();
         }
         catch (InterruptedException exception)
         {
            Thread.currentThread().interrupt();
         }
      }

      System.out.printf("%n%s %d%n%s%n",
         "Consumer read values totaling", sum, "Terminating Consumer");
   }
```

```
} // end class Consumer
-------------------------------------------------------------------------
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class SharedBufferTest2
{
   public static void main(String[] args) throws InterruptedException
   {
      // create a CachedThreadPool
      ExecutorService executorService = Executors.newCachedThreadPool();

      // create SynchronizedBuffer to store ints
      Buffer sharedLocation = new SynchronizedBuffer();

      System.out.printf("%-40s%s\t\t%s%n%-40s%s%n%n", "Operation",
         "Buffer", "Occupied", "---------", "------\t\t--------");

      // execute the Producer and Consumer tasks
      executorService.execute(new Producer(sharedLocation));
      executorService.execute(new Consumer(sharedLocation));

      executorService.shutdown();
      executorService.awaitTermination(1, TimeUnit.MINUTES);
   }
} // end class SharedBufferTest2

=============================================================================
SwingWorkerExample:

// Calculates the first n primes, displaying them as they are found.

import javax.swing.JTextArea;
import javax.swing.JLabel;
import javax.swing.JButton;
import javax.swing.SwingWorker;
import java.security.SecureRandom;
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CancellationException;
import java.util.concurrent.ExecutionException;

public class PrimeCalculator extends SwingWorker<Integer, Integer> {
    private static final SecureRandom generator = new SecureRandom();
    private final JTextArea intermediateJTextArea; // displays found primes
    private final JButton getPrimesJButton;
    private final JButton cancelJButton;
    private final JLabel statusJLabel; // displays status of calculation
    private final boolean[] primes; // boolean array for finding primes

    // constructor
    public PrimeCalculator(int max, JTextArea intermediateJTextArea, JLabel statusJLabel
    , JButton getPrimesJButton,
            JButton cancelJButton) {
        this.intermediateJTextArea = intermediateJTextArea;
        this.statusJLabel = statusJLabel;
        this.getPrimesJButton = getPrimesJButton;
        this.cancelJButton = cancelJButton;
        primes = new boolean[max];

        Arrays.fill(primes, true); // initialize all primes elements to true
    }

    // finds all primes up to max using the Sieve of Eratosthenes
    public Integer doInBackground() {
        int count = 0; // the number of primes found

        // starting at the third value, cycle through the array and put
```

```
            // false as the value of any greater number that is a multiple
            for (int i = 2; i < primes.length; i++) {
                if (isCancelled()) // if calculation has been canceled
                    return count;
                else {
                    setProgress(100 * (i + 1) / primes.length);

                    try {
                        Thread.sleep(generator.nextInt(5));
                    } catch (InterruptedException ex) {
                        statusJLabel.setText("Worker thread interrupted");
                        return count;
                    }

                    if (primes[i]) // i is prime
                    {
                        publish(i); // make i available for display in prime list
                        ++count;

                        for (int j = i + i; j < primes.length; j += i)
                            primes[j] = false; // i is not prime
                    }
                }
            }

            return count;
        }

        // displays published values in primes list
        protected void process(List<Integer> publishedVals) {
            for (int i = 0; i < publishedVals.size(); i++)
                intermediateJTextArea.append(publishedVals.get(i) + "\n");
        }

        // code to execute when doInBackground completes
        protected void done() {
            getPrimesJButton.setEnabled(true); // enable Get Primes button
            cancelJButton.setEnabled(false); // disable Cancel button

            try {
                // retrieve and display doInBackground return value
                statusJLabel.setText("Found " + get() + " primes.");
            } catch (InterruptedException | ExecutionException | CancellationException ex) {
                statusJLabel.setText(ex.getMessage());
            }
        }
    } // end class PrimeCalculator
    ------------------------------------------------------------------------
    // Fig 23.27: FindPrimes.java

    // Using a SwingWorker to display prime numbers and update a JProgressBar
    // while the prime numbers are being calculated.

    import javax.swing.JFrame;
    import javax.swing.JTextField;
    import javax.swing.JTextArea;
    import javax.swing.JButton;
    import javax.swing.JProgressBar;
    import javax.swing.JLabel;
    import javax.swing.JPanel;
    import javax.swing.JScrollPane;
    import javax.swing.ScrollPaneConstants;
    import java.awt.BorderLayout;
    import java.awt.GridLayout;
    import java.awt.event.ActionListener;
    import java.awt.event.ActionEvent;
    import java.beans.PropertyChangeListener;
    import java.beans.PropertyChangeEvent;
```

```
825 public class FindPrimes extends JFrame {
826     private final JTextField highestPrimeJTextField = new JTextField();
827     private final JButton getPrimesJButton = new JButton("Get Primes");
828     private final JTextArea displayPrimesJTextArea = new JTextArea();
829     private final JButton cancelJButton = new JButton("Cancel");
830     private final JProgressBar progressJProgressBar = new JProgressBar();
831     private final JLabel statusJLabel = new JLabel();
832     private PrimeCalculator calculator;
833
834     // constructor
835     public FindPrimes() {
836         super("Finding Primes with SwingWorker");
837         setLayout(new BorderLayout());
838
839         // initialize panel to get a number from the user
840         JPanel northJPanel = new JPanel();
841         northJPanel.add(new JLabel("Find primes less than: "));
842         highestPrimeJTextField.setColumns(5);
843         northJPanel.add(highestPrimeJTextField);
844         getPrimesJButton.addActionListener(new ActionListener() {
845             public void actionPerformed(ActionEvent e) {
846                 progressJProgressBar.setValue(0); // reset JProgressBar
847                 displayPrimesJTextArea.setText(""); // clear JTextArea
848                 statusJLabel.setText(""); // clear JLabel
849
850                 int number; // search for primes up through this value
851
852                 try {
853                     // get user input
854                     number = Integer.parseInt(highestPrimeJTextField.getText());
855                 } catch (NumberFormatException ex) {
856                     statusJLabel.setText("Enter an integer.");
857                     return;
858                 }
859
860                 // construct a new PrimeCalculator object
861                 calculator = new PrimeCalculator(number, displayPrimesJTextArea,
                    statusJLabel, getPrimesJButton,
862                         cancelJButton);
863
864                 // listen for progress bar property changes
865                 calculator.addPropertyChangeListener(new PropertyChangeListener() {
866                     public void propertyChange(PropertyChangeEvent e) {
867                         // if the changed property is progress,
868                         // update the progress bar
869                         if (e.getPropertyName().equals("progress")) {
870                             int newValue = (Integer) e.getNewValue();
871                             progressJProgressBar.setValue(newValue);
872                         }
873                     }
874                 } // end anonymous inner class
875                 ); // end call to addPropertyChangeListener
876
877                 // disable Get Primes button and enable Cancel button
878                 getPrimesJButton.setEnabled(false);
879                 cancelJButton.setEnabled(true);
880
881                 calculator.execute(); // execute the PrimeCalculator object
882             }
883         } // end anonymous inner class
884         ); // end call to addActionListener
885         northJPanel.add(getPrimesJButton);
886
887         // add a scrollable JList to display results of calculation
888         displayPrimesJTextArea.setEditable(false);
889         add(new JScrollPane(displayPrimesJTextArea, ScrollPaneConstants.
            VERTICAL_SCROLLBAR_ALWAYS,
890                 ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER));
891
```

```
892          // initialize a panel to display cancelJButton,
893          // progressJProgressBar, and statusJLabel
894          JPanel southJPanel = new JPanel(new GridLayout(1, 3, 10, 10));
895          cancelJButton.setEnabled(false);
896          cancelJButton.addActionListener(new ActionListener() {
897              public void actionPerformed(ActionEvent e) {
898                  calculator.cancel(true); // cancel the calculation
899              }
900          } // end anonymous inner class
901          ); // end call to addActionListener
902          southJPanel.add(cancelJButton);
903          progressJProgressBar.setStringPainted(true);
904          southJPanel.add(progressJProgressBar);
905          southJPanel.add(statusJLabel);
906
907          add(northJPanel, BorderLayout.NORTH);
908          add(southJPanel, BorderLayout.SOUTH);
909          setSize(350, 300);
910          setVisible(true);
911      } // end constructor
912
913      // main method begins program execution
914      public static void main(String[] args) {
915          FindPrimes application = new FindPrimes();
916          application.setDefaultCloseOperation(EXIT_ON_CLOSE);
917      } // end main
918  } // end class FindPrimes
```