```java
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
    public void run() {
        System.out.println("Doing heavy processing - 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("Doing heavy processing - END " + Thread.currentThread().getName());
    }

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

}

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

public class ThreadRunExample {

    public static void main(String[] args){
        Thread t1 = new Thread(new HeavyWorkRunnable(), "t1");
        Thread t2 = new Thread(new HeavyWorkRunnable(), "t2");
        System.out.println("Starting Runnable threads");
        t1.start();
        t2.start();
        System.out.println("Runnable Threads has been started");
        Thread t3 = new MyThread("t3");
        Thread t4 = new MyThread("t4");
        System.out.println("Starting MyThreads");
        t3.start();
        t4.start();
        System.out.println("MyThreads has been started");

    }
}

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

// Fig. 23.3: PrintTask.java
// PrintTask class sleeps for a random time from 0 to 5 seconds
import java.security.SecureRandom;

public class PrintTask implements Runnable
{
   private final static SecureRandom generator = new SecureRandom();
   private final int sleepTime; // random sleep time for thread
   private final String taskName; // name of task

   // constructor
   public PrintTask(String taskName)
   {
      this.taskName = taskName;

      // pick random sleep time between 0 and 5 seconds
      sleepTime = generator.nextInt(5000); // milliseconds
   }

   // method run contains the code that a thread will execute
   public void run()
   {
      try // put thread to sleep for sleepTime amount of time
      {
         System.out.printf("%s going to sleep for %d milliseconds.%n",
            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

      try
      {
         // put thread to sleep for 0-499 milliseconds
         Thread.sleep(generator.nextInt(500));
      }
      catch (InterruptedException ex)
      {
         Thread.currentThread().interrupt(); // re-interrupt the thread
      }

      // put value in the appropriate element
      array[position] = value;
      System.out.printf("%s wrote %2d to element %d.%n",
         Thread.currentThread().getName(), value, position);

      ++writeIndex; // increment index of element to be written next
      System.out.printf("Next write index: %d%n", writeIndex);
   }

   // used for outputting the contents of the shared integer array
   public String toString()
   {
      return Arrays.toString(array);
   }
} // end class SimpleArray
---------------------------------------------------------------------
import java.lang.Runnable;

public class ArrayWriter implements Runnable
{
   private final SimpleArray sharedSimpleArray;
   private final int startValue;

   public ArrayWriter(int value, SimpleArray array)
   {
      startValue = value;
      sharedSimpleArray= array;
   }

   public void run()
   {
      for (int i = startValue; i < startValue + 3; i++)
      {
         sharedSimpleArray.add(i); // add an element to the shared array
      }
   }
} // end class ArrayWriter
----------------------------------------------------------------------
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class SharedArrayTest
{
   public static void main(String[] arg)
   {
      // construct the shared object
      SimpleArray sharedSimpleArray = new SimpleArray(6);

      // create two tasks to write to the shared SimpleArray
      ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
      ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);

      // execute the tasks with an ExecutorService
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(writer1);
      executorService.execute(writer2);

      executorService.shutdown();

      try
      {
         // wait 1 minute for both writers to finish executing
         boolean tasksEnded =
            executorService.awaitTermination(1, TimeUnit.MINUTES);

         if (tasksEnded)
         {
            System.out.printf("%nContents of SimpleArray:%n");
            System.out.println(sharedSimpleArray); // print contents
         }
         else
            System.out.println(
               "Timed out while waiting for tasks to finish.");
      }
      catch (InterruptedException ex)
      {
         ex.printStackTrace();
      }
   } // end main
} // end class SharedArrayTest
==============================================================================
SynchronizedExample:

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

public class SimpleArray
{
   private static final SecureRandom generator = new SecureRandom();
   private final int[] array; // the shared integer array
   private int writeIndex = 0; // index of next element to be written

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

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

      try
      {
         // in real applications, you shouldn't sleep while holding a lock
         Thread.sleep(generator.nextInt(500)); // for demo only
      }
      catch (InterruptedException ex)
      {
         Thread.currentThread().interrupt();
      }

      // put value in the appropriate element
      synchronized (this){
         array[position] = value;
      }
      System.out.printf("%s wrote %2d to element %d.%n",
         Thread.currentThread().getName(), value, position);

      ++writeIndex; // increment index of element to be written next
      System.out.printf("Next write index: %d%n", writeIndex);
   }

   // used for outputting the contents of the shared integer array
   public synchronized String toString()
```

```java
   {
      return Arrays.toString(array);
   }
} // end class SimpleArray
----------------------------------------------------------------
import java.lang.Runnable;

public class ArrayWriter implements Runnable
{
   private final SimpleArray sharedSimpleArray;
   private final int startValue;

   public ArrayWriter(int value, SimpleArray array)
   {
      startValue = value;
      sharedSimpleArray= array;
   }

   public void run()
   {
      for (int i = startValue; i < startValue + 3; i++)
      {
         sharedSimpleArray.add(i); // add an element to the shared array
      }
   }
} // end class ArrayWriter
-----------------------------------------------------------------
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;

public class SharedArrayTest
{
   public static void main(String[] arg)
   {
      // construct the shared object
      SimpleArray sharedSimpleArray = new SimpleArray(6);

      // create two tasks to write to the shared SimpleArray
      ArrayWriter writer1 = new ArrayWriter(1, sharedSimpleArray);
      ArrayWriter writer2 = new ArrayWriter(11, sharedSimpleArray);

      // execute the tasks with an ExecutorService
      ExecutorService executorService = Executors.newCachedThreadPool();
      executorService.execute(writer1);
      executorService.execute(writer2);

      executorService.shutdown();

      try
      {
         // wait 1 minute for both writers to finish executing
         boolean tasksEnded =
            executorService.awaitTermination(1, TimeUnit.MINUTES);

         if (tasksEnded)
         {
            System.out.printf("%nContents of SimpleArray:%n");
            System.out.println(sharedSimpleArray); // print contents
         }
         else
            System.out.println(
               "Timed out while waiting for tasks to finish.");
      }
      catch (InterruptedException ex)
      {
         System.out.println(
            "Interrupted while waiting for tasks to finish.");
      }
   } // 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
            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 BlockingBufferTest
{
   public static void main(String[] args) throws InterruptedException
   {
      // create new thread pool with two threads
      ExecutorService executorService = Executors.newCachedThreadPool();

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

      executorService.execute(new Producer(sharedLocation));
      executorService.execute(new Consumer(sharedLocation));

      executorService.shutdown();
      executorService.awaitTermination(1, TimeUnit.MINUTES);
   }
} // end class BlockingBufferTest
=============================================================================
WaitNotifyExample:

// Fig. 23.16: SynchronizedBuffer.java
// Synchronizing access to shared mutable data using Object
// methods wait and notifyAll.
public class SynchronizedBuffer implements Buffer
{
   private int buffer = -1; // shared by producer and consumer threads
   private boolean occupied = false;
//   private Object o;

   // place value into buffer
   public synchronized void blockingPut(int value)
      throws InterruptedException
   {
      // while there are no empty locations, place thread in waiting state

      while (occupied)
      {
         // output thread information and buffer information, then wait
         System.out.println("Producer tries to write."); // for demo only
         displayState("Buffer full. Producer waits." + Thread.currentThread().getName
         ()); // for demo only
         wait();
      }

      buffer = value; // set new buffer value

      // indicate producer cannot store another value
      // until consumer retrieves current buffer value
      occupied = true;

      displayState("Producer writes " + buffer); // for demo only

      notifyAll(); // tell waiting thread(s) to enter runnable state
   } // end method blockingPut; releases lock on SynchronizedBuffer

   // return value from buffer
   public synchronized int blockingGet() throws InterruptedException
   {
      // while no data to read, place thread in waiting state
      while (!occupied)
      {
         // output thread information and buffer information, then wait
         System.out.println("Consumer tries to read."); // for demo only
         displayState("Buffer empty. Consumer waits."); // for demo only
         wait();
      }

      // indicate that producer can store another value
      // because consumer just retrieved buffer value
      occupied = false;

      displayState("Consumer reads " + buffer); // for demo only

      notifyAll(); // tell waiting thread(s) to enter runnable state

      return buffer;
   } // end method blockingGet; releases lock on SynchronizedBuffer

   // display current operation and buffer state; for demo only
   private synchronized void displayState(String operation)
   {
      System.out.printf("%-40s%d\t\t%b%n%n", operation, buffer,
         occupied);
```

```java
   }
} // 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;

```

```java
public class FindPrimes extends JFrame {
    private final JTextField highestPrimeJTextField = new JTextField();
    private final JButton getPrimesJButton = new JButton("Get Primes");
    private final JTextArea displayPrimesJTextArea = new JTextArea();
    private final JButton cancelJButton = new JButton("Cancel");
    private final JProgressBar progressJProgressBar = new JProgressBar();
    private final JLabel statusJLabel = new JLabel();
    private PrimeCalculator calculator;

    // constructor
    public FindPrimes() {
        super("Finding Primes with SwingWorker");
        setLayout(new BorderLayout());

        // initialize panel to get a number from the user
        JPanel northJPanel = new JPanel();
        northJPanel.add(new JLabel("Find primes less than: "));
        highestPrimeJTextField.setColumns(5);
        northJPanel.add(highestPrimeJTextField);
        getPrimesJButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                progressJProgressBar.setValue(0); // reset JProgressBar
                displayPrimesJTextArea.setText(""); // clear JTextArea
                statusJLabel.setText(""); // clear JLabel

                int number; // search for primes up through this value

                try {
                    // get user input
                    number = Integer.parseInt(highestPrimeJTextField.getText());
                } catch (NumberFormatException ex) {
                    statusJLabel.setText("Enter an integer.");
                    return;
                }

                // construct a new PrimeCalculator object
                calculator = new PrimeCalculator(number, displayPrimesJTextArea,
                statusJLabel, getPrimesJButton,
                        cancelJButton);

                // listen for progress bar property changes
                calculator.addPropertyChangeListener(new PropertyChangeListener() {
                    public void propertyChange(PropertyChangeEvent e) {
                        // if the changed property is progress,
                        // update the progress bar
                        if (e.getPropertyName().equals("progress")) {
                            int newValue = (Integer) e.getNewValue();
                            progressJProgressBar.setValue(newValue);
                        }
                    }
                } // end anonymous inner class
                ); // end call to addPropertyChangeListener

                // disable Get Primes button and enable Cancel button
                getPrimesJButton.setEnabled(false);
                cancelJButton.setEnabled(true);

                calculator.execute(); // execute the PrimeCalculator object
            }
        } // end anonymous inner class
        ); // end call to addActionListener
        northJPanel.add(getPrimesJButton);

        // add a scrollable JList to display results of calculation
        displayPrimesJTextArea.setEditable(false);
        add(new JScrollPane(displayPrimesJTextArea, ScrollPaneConstants.
        VERTICAL_SCROLLBAR_ALWAYS,
                ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER));

        // initialize a panel to display cancelJButton,
        // progressJProgressBar, and statusJLabel
        JPanel southJPanel = new JPanel(new GridLayout(1, 3, 10, 10));
        cancelJButton.setEnabled(false);
        cancelJButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                calculator.cancel(true); // cancel the calculation
            }
        } // end anonymous inner class
        ); // end call to addActionListener
        southJPanel.add(cancelJButton);
        progressJProgressBar.setStringPainted(true);
        southJPanel.add(progressJProgressBar);
        southJPanel.add(statusJLabel);

        add(northJPanel, BorderLayout.NORTH);
        add(southJPanel, BorderLayout.SOUTH);
        setSize(350, 300);
        setVisible(true);
    } // end constructor

    // main method begins program execution
    public static void main(String[] args) {
        FindPrimes application = new FindPrimes();
        application.setDefaultCloseOperation(EXIT_ON_CLOSE);
    } // end main
} // end class FindPrimes
```