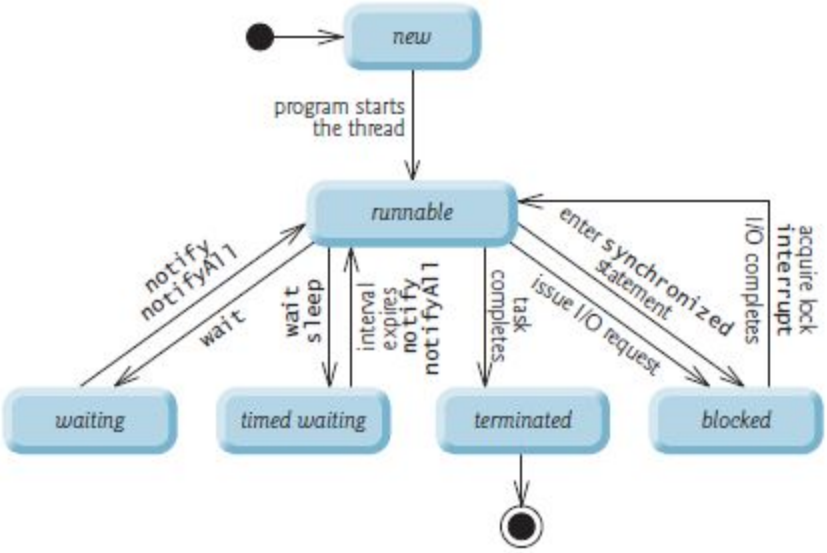
Thread States and Lifecycle



Example: 1 - PrintTask

Using the ExecutorService to Manage Threads that Execute PrintTasks

The TaskExecutor(Java 5) class uses an ExecutorService object to manage threads that execute PrintTasks (as defined in PrintTask class). Lines 11–13 create and name three PrintTasks to execute. Line 18 uses Executors(interface Java 5) method newCachedThreadPool to obtain an ExecutorService that's capable of creating new threads as they're needed by the application. These threads are used by ExecutorService to execute the Runnables.

```
// 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
```

```
// 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
```

Main Thread

The code in main executes in the main thread, which is created by the JVM. The code in the run method of PrintTask (lines 21–37 of PrintTask) executes whenever the Executor starts each PrintTask—again, this is sometime after they're passed to the ExecutorService's execute method (TaskExecutor, lines 21–23). When main terminates, the program itself continues running because there are still tasks that must finish executing. The program will not terminate until these tasks complete.

Example: 2 Without Synchronization

```
// Class that manages an integer array to be shared by multiple threads.
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




// Adds integers to an array shared with other Runnables
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
```

```
// Executing two Runnables to add elements to a shared SimpleArray.
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
```

Example: 3 With Synchronization

```
// Class that manages an integer array to be shared by multiple
// threads with synchronization.
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
      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()
   {
      return Arrays.toString(array);
   }
} // end class SimpleArray
```

```
// Adds integers to an array shared with other Runnables
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
```

// Executing two Runnables to add elements to a shared SimpleArray.

```
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
```

Example: 3 Multithreading GUI Example

SwingWorker class for creating worker threads

Class SwingWorker (in package javax.swing) enables you to perform an asynchronous task in a worker thread (such as a long-running computation) then update Swing components from the event dispatch thread based on the task's results. SwingWorker implements the Runnable interface, meaning that a SwingWorker object can be scheduled to execute in a separate thread. The SwingWorker class provides several methods to simplify performing a task in a worker thread and making its results available for display in a GUI. Some common SwingWorker methods are described in below table.

Method	Description
`doInBackground`	Defines a long computation and is called in a worker thread.
`done`	Executes on the event dispatch thread when `doInBackground` returns.
`execute`	Schedules the `SwingWorker` object to be executed in a worker thread.
`get`	Waits for the computation to complete, then returns the result of the computation (i.e., the return value of `doInBackground`).
`publish`	Sends intermediate results from the `doInBackground` method to the `process` method for processing on the event dispatch thread.
`process`	Receives intermediate results from the `publish` method and processes these results on the event dispatch thread.
`setProgress`	Sets the progress property to notify any property change listeners on the event dispatch thread of progress bar updates.

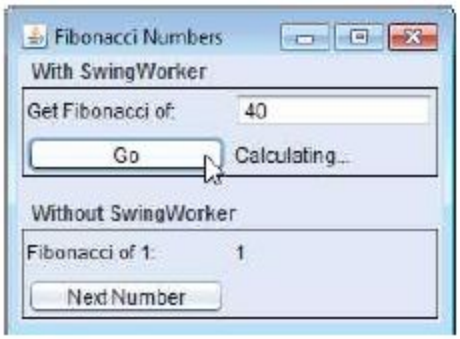
```
// SwingWorker subclass for calculating Fibonacci numbers
// in a background thread.
import javax.swing.SwingWorker;
import javax.swing.JLabel;
import java.util.concurrent.ExecutionException;

public class BackgroundCalculator extends SwingWorker< Long, Object >
{
   private final int n; // Fibonacci number to calculate
   private final JLabel resultJLabel; // JLabel to display the result

   // constructor
   public BackgroundCalculator(int n, JLabel resultJLabel)
   {
      this.n = n;
      this.resultJLabel = resultJLabel;
   }

   // long-running code to be run in a worker thread
   public Long doInBackground()
   {
      return fibonacci(n);
   }

   // code to run on the event dispatch thread when doInBackground returns
   protected void done()
   {
      try
      {
         // get the result of doInBackground and display it
         resultJLabel.setText(get().toString());
      }
      catch (InterruptedException ex)
      {
         resultJLabel.setText("Interrupted while waiting for results.");
      }
      catch (ExecutionException ex)
      {
         resultJLabel.setText(
            "Error encountered while performing calculation.");
      }
   }
```

```
   // recursive method fibonacci; calculates nth Fibonacci number
   public long fibonacci(long number)
   {
      if (number == 0 || number == 1)
         return number;
      else
         return fibonacci(number - 1) + fibonacci(number - 2);
   }
} // end class BackgroundCalculator



// Using SwingWorker to perform a long calculation with
// results displayed in a GUI.
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JLabel;
import javax.swing.JTextField;
import javax.swing.border.TitledBorder;
import javax.swing.border.LineBorder;
import java.awt.Color;
import java.util.concurrent.ExecutionException;

public class FibonacciNumbers extends JFrame
{
   // components for calculating the Fibonacci of a user-entered number
   private final JPanel workerJPanel =
      new JPanel(new GridLayout(2, 2, 5, 5));
   private final JTextField numberJTextField = new JTextField();
   private final JButton goJButton = new JButton("Go");
   private final JLabel fibonacciJLabel = new JLabel();

   // components and variables for getting the next Fibonacci number
   private final JPanel eventThreadJPanel =
      new JPanel(new GridLayout(2, 2, 5, 5));
   private long n1 = 0; // initialize with first Fibonacci number
   private long n2 = 1; // initialize with second Fibonacci number
   private int count = 1; // current Fibonacci number to display
   private final JLabel nJLabel = new JLabel("Fibonacci of 1: ");
   private final JLabel nFibonacciJLabel =
      new JLabel(String.valueOf(n2));
```

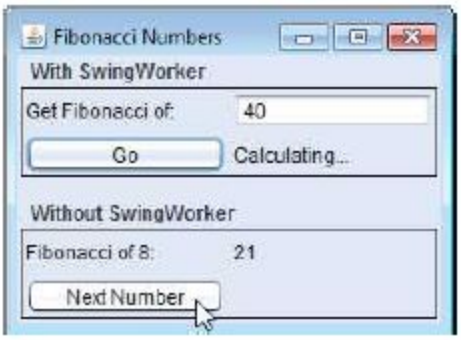
```
private final JButton nextNumberJButton = new JButton("Next Number");

// constructor
public FibonacciNumbers()
{
   super("Fibonacci Numbers");
   setLayout(new GridLayout(2, 1, 10, 10));

   // add GUI components to the SwingWorker panel
   workerJPanel.setBorder(new TitledBorder(
      new LineBorder(Color.BLACK), "With SwingWorker"));
   workerJPanel.add(new JLabel("Get Fibonacci of:"));
   workerJPanel.add(numberJTextField);
   goJButton.addActionListener(
      new ActionListener()
      {
         public void actionPerformed(ActionEvent event)
         {
            int n;

            try
            {
               // retrieve user's input as an integer
               n = Integer.parseInt(numberJTextField.getText());
            }
            catch(NumberFormatException ex)
            {
               // display an error message if the user did not
               // enter an integer
               fibonacciJLabel.setText("Enter an integer.");
               return;
            }

            // indicate that the calculation has begun
            fibonacciJLabel.setText("Calculating...");

            // create a task to perform calculation in background
            BackgroundCalculator task =
               new BackgroundCalculator(n, fibonacciJLabel);
            task.execute(); // execute the task
         }
      } // end anonymous inner class
   ); // end call to addActionListener
   workerJPanel.add(goJButton);
   workerJPanel.add(fibonacciJLabel);
```

```
      // add GUI components to the event-dispatching thread panel
      eventThreadJPanel.setBorder(new TitledBorder(
         new LineBorder(Color.BLACK), "Without SwingWorker"));
      eventThreadJPanel.add(nJLabel);
      eventThreadJPanel.add(nFibonacciJLabel);
      nextNumberJButton.addActionListener(
         new ActionListener()
         {
            public void actionPerformed(ActionEvent event)
            {
               // calculate the Fibonacci number after n2
               long temp = n1 + n2;
               n1 = n2;
               n2 = temp;
               ++count;

               // display the next Fibonacci number
               nJLabel.setText("Fibonacci of " + count + ": ");
               nFibonacciJLabel.setText(String.valueOf(n2));
            }
         } // end anonymous inner class
      ); // end call to addActionListener
      eventThreadJPanel.add(nextNumberJButton);

      add(workerJPanel);
      add(eventThreadJPanel);
      setSize(275, 200);
      setVisible(true);
   } // end constructor

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

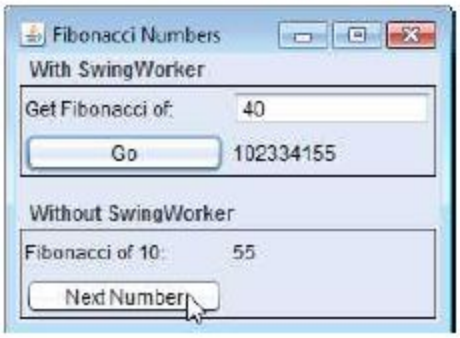
a) Begin calculating Fibonacci of 40 in the background



b) Calculating other Fibonacci values while Fibonacci of 40 continues calculating



c) Fibonacci of 40 calculation finishes



Example: 4 Multithreading GUI Prime Numbers

```
// 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
```

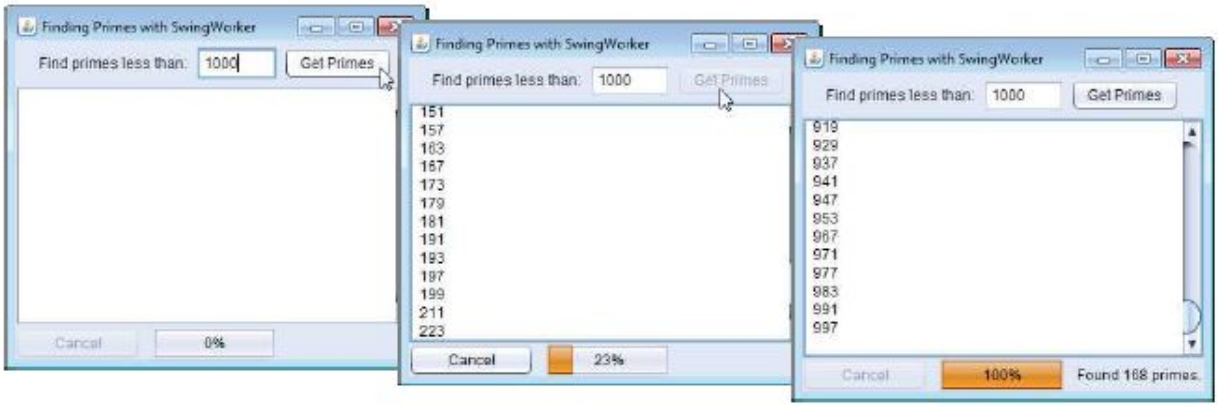
```
private final JButton getPrimesJButton;
private final JButton cancelJButton;
private final JLabel statusJLabel; // displays status of calculation
private final boolean[] primes; // boolean array for finding primes

// 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
```

```
// 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.util.concurrent.ExecutionException;
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeEvent;

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
```


Finding Primes with SwingWorker
Find primes less than:
1000
Get Primes
Cancel
0%
Finding Primes with SwingWorker
Find primes less than:
1000
Get Primes
151
157
163
167
173
179
181
191
193
197
199
211
223
Cancel
23%
Finding Primes with SwingWorker
Find primes less than:
1000
Get Primes
919
929
937
941
947
953
967
971
977
983
991
997
Cancel
100%
Found 168 primes.