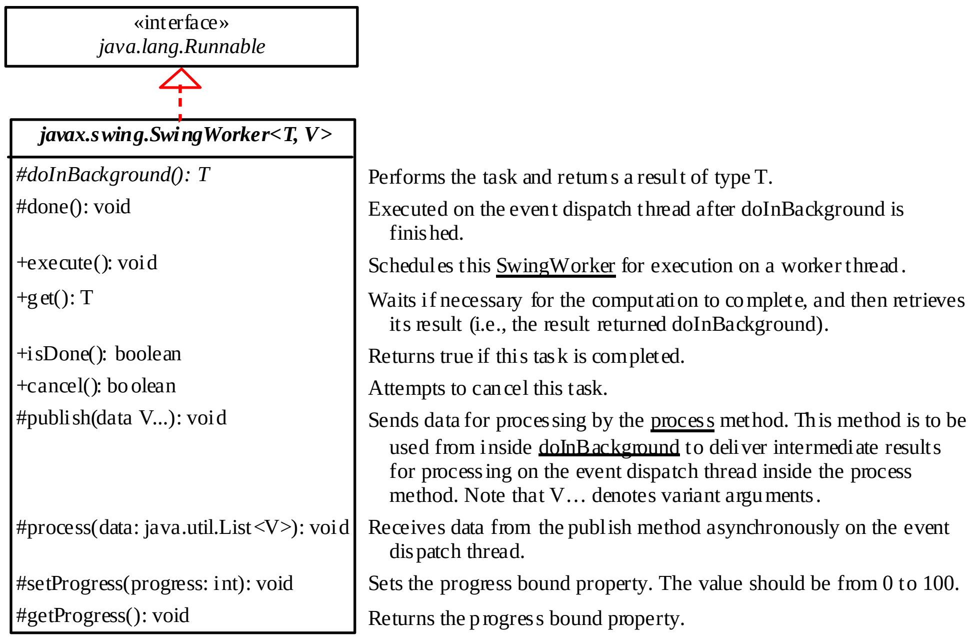
# Supplement: SwingWorker and JProgressBar
# For Introduction to Java Programming
**By Y. Daniel Liang**

This supplement introduces two advanced Swing components SwingWorker and JProgressBar. You can read this supplement after completing the chapter on threads.

## 1 SwingWorker

***<Side Remark: event dispatch thread>***
All Swing GUI events are processed in a single *event dispatch thread*. If an event requires a long time to process, the thread cannot attend to other tasks in the queue. To solve this problem, you should run the time-consuming task for processing the event in a separate thread. Java 6 introduced SwingWorker. SwingWorker is an abstract class that implements Runnable. You can define a task class that extends SwingWorker, run the time-consuming task, and update the GUI using the results produced from the task. Figure 1 defines SwingWorker.



| | |
|---|---|
| *#doInBackground(): T* | Performs the task and returns a result of type T. |
| #done(): void | Executed on the event dispatch thread after doInBackground is finished. |
| +execute(): void | Schedules this SwingWorker for execution on a worker thread. |
| +get(): T | Waits if necessary for the computation to complete, and then retrieves its result (i.e., the result returned doInBackground). |
| +isDone(): boolean | Returns true if this task is completed. |
| +cancel(): boolean | Attempts to cancel this task. |
| #publish(data V...): void | Sends data for processing by the process method. This method is to be used from inside doInBackground to deliver intermediate results for processing on the event dispatch thread inside the process method. Note that V… denotes variant arguments. |
| #process(data: java.util.List<V>): void | Receives data from the publish method asynchronously on the event dispatch thread. |
| #setProgress(progress: int): void | Sets the progress bound property. The value should be from 0 to 100. |
| #getProgress(): void | Returns the progress bound property. |

**Figure 1**
*The SwingWorker class can be used to process time-consuming tasks.*

This section demonstrates basic use of SwingWorker. The next section gives an example involving advanced features of SwingWorker.
***<Side Remark: doInBackground()>***
***<Side Remark: done()>***
To use SwingWorker, your task class should override doInBackground() to perform a time-consuming task and

override the done() method to update GUI components if necessary. Listing 1 gives an example that lets the user specify a number and displays the number of prime numbers less than or equal to the specified number, as shown in Figure 2.

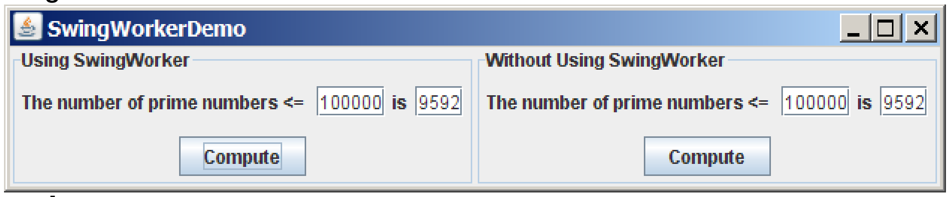

SwingWorkerDemo
Using SwingWorker
The number of prime numbers <= 100000 is 9592
Compute
Without Using SwingWorker
The number of prime numbers <= 100000 is 9592
Compute

**Figure 2**

*You can compare the effect of using versus not using SwingWorker.*

**Listing 1 SwingWorkerDemo.java**

******PD: Please add line numbers in the following code******
***<Side Remark line 6: GUI components>***
***<Side Remark line 13: create UI>***
***<Side Remark line 14: left panel>***
***<Side Remark line 27: right panel>***
***<Side Remark line 44: add listener>***
***<Side Remark line 46: create task>***
***<Side Remark line 47: create task>***
***<Side Remark line 51: add listener>***
***<Side Remark line 66: constructor>***
***<Side Remark line 72: override doInBackground()>***
***<Side Remark line 77: override done()>***
***<Side Remark line 87: getNumberOfPrimes>***
***<Side Remark line 116: main method omitted>***

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class SwingWorkerDemo extends JApplet {
  private JButton jbtComputeWithSwingWorker = new JButton("Compute");
  private JTextField jtfLimit1 = new JTextField(8);
  private JTextField jtfResult1 = new JTextField(6);
  private JButton jbtCompute = new JButton("Compute");
  private JTextField jtfLimit2 = new JTextField(8);
  private JTextField jtfResult2 = new JTextField(6);

  public SwingWorkerDemo() {
    JPanel panel1 = new JPanel(new GridLayout(2, 1));
    panel1.setBorder(BorderFactory.createTitledBorder(
      "Using SwingWorker"));
    JPanel panel11 = new JPanel();
    panel11.add(new JLabel("The number of prime numbers <= "));
    panel11.add(jtfLimit1);
    panel11.add(new JLabel("is"));
    panel11.add(jtfResult1);
    JPanel panel12 = new JPanel();
    panel12.add(jbtComputeWithSwingWorker);
    panel1.add(panel11);
    panel1.add(panel12);

    JPanel panel2 = new JPanel(new GridLayout(2, 1));
    panel2.setBorder(BorderFactory.createTitledBorder(
      "Without Using SwingWorker"));
    JPanel panel21 = new JPanel();
    panel21.add(new JLabel("The number of prime numbers <= "));
    panel21.add(jtfLimit2);
    panel21.add(new JLabel("is"));
    panel21.add(jtfResult2);
    JPanel panel22 = new JPanel();
    panel22.add(jbtCompute);
    panel2.add(panel21);
    panel2.add(panel22);

    setLayout(new GridLayout(1, 2));
    add(panel1);
    add(panel2);

    jbtComputeWithSwingWorker.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        new ComputePrime(Integer.parseInt(jtfLimit1.getText()),
          jtfResult1).execute(); // Execute SwingWorker
        }
      });

    jbtCompute.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        int count = ComputePrime.getNumberOfPrimes(
          Integer.parseInt(jtfLimit2.getText()));
        jtfResult2.setText(count + "");
      }
    });
  }

  /** Task class for SwingWorker */
  static class ComputePrime extends SwingWorker<Integer, Object> {
    private int limit;
    private JTextField result; // Text field in the UI

    /** Construct a runnable Task */
    public ComputePrime(int limit, JTextField result) {
      this.limit = limit;
      this.result = result;
    }

    /** Code run on a background thread */
    protected Integer doInBackground() {
      return getNumberOfPrimes(limit);
    }

    /** Code executed after the background thread finishes */
    protected void done() {
      try {
        result.setText(get().toString()); // Display in text field
      }
      catch (Exception ex) {
        result.setText(ex.getMessage());
      }
    }

    /** Return the number of primes <= limit */
    public static int getNumberOfPrimes(int limit) {
      int count = 0; // Count the number of prime numbers
      int number = 2; // A number to be tested for primeness

      // Repeatedly find prime numbers
      while (number <= limit) {
        // Print the prime number and increase the count
        if (isPrime(number)) {
          count++; // Increase the count
        }

        // Check if the next number is prime
        number++;
      }

      return count;
    }

    /** Check whether number is prime */
    private static boolean isPrime(int number) {
      for (int divisor = 2; divisor <= number / 2; divisor++) {
        if (number % divisor == 0) { // If true, number is not prime
          return false; // number is not a prime
        }
      }

      return true; // number is prime
    }
  }
}
```

***<Side Remark: using or not using SwingWorker>***
The UI consists of two panels. The left panel demonstrates how to compute the number of prime numbers using a SwingWorker. The right panel demonstrates how to compute the number of prime numbers without using a SwingWorker. You enter a number (e.g., 100000) in the first text field in the panel and click the *Compute* button to display the number of primes in the second text field. When you click the *Compute* button in the left panel, a SwingWorker task is created and executed (lines 46–47). Since the task is run on a separate thread, you can continue to use the GUI. However, when you click the *Compute* button in the right panel, the GUI is frozen, because the getNumberOfPrimes method is executed on the event dispatch thread (lines 53–54).
***<Side Remark: override doInBackground>***
***<Side Remark: override done>***
The inner class ComputePrime is a SwingWorker (line 61). It overrides the doInBackground method to run getNumberOfPrimes in a background thread (lines 72–74). It also overrides the done method to display the result in a text field, once the background thread finishes (lines 77–84).
***<Side Remark: static getNumberOfPrimes>***
The ComputePrime class defines the static getNumberOfPrimes method for computing the number of primes (lines 87–103). When you click the *Compute* button in the left panel, this method is executed on a background thread (lines 46–47). When you click the *Compute* button in the right panel, this method is executed in the event dispatch thread (lines 53–54).

TIP:
***<Side Remark: GUI and SwingWorker>***
Two things to remember when writing Swing GUI programs:

- Time-consuming tasks should be run in SwingWorker.
- Swing components should be accessed from the event dispatch thread only.

## 2 Displaying Progress Using JProgressBar

In the preceding example, it may take a long time to finish the computation in the background thread. It is better to inform the user the progress of the computation. You can use the JProgressBar to display the progress.

JProgressBar is a component that displays a value graphically within a bounded interval. A progress bar is typically used to show the percentage of completion of a lengthy operation; it comprises a rectangular bar that is "filled in" from left to right horizontally or from bottom to top vertically as the operation is performed. It provides the user with feedback on the progress of the operation. For example, when a file is being read, it alerts the user to the progress of the operation, thereby keeping the user attentive.

JProgressBar is often implemented using a thread to monitor the completion status of other threads. The progress bar can be displayed horizontally or vertically, as determined by its orientation property. The minimum, value, and maximum properties determine the minimum, current, and maximum lengths on the progress bar, as shown in Figure 3. Figure 4 lists frequently used features of JProgressBar.

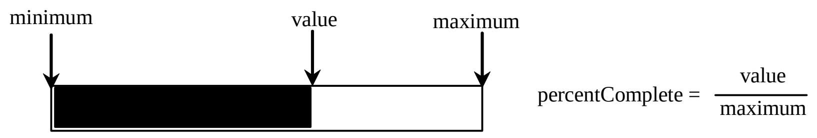
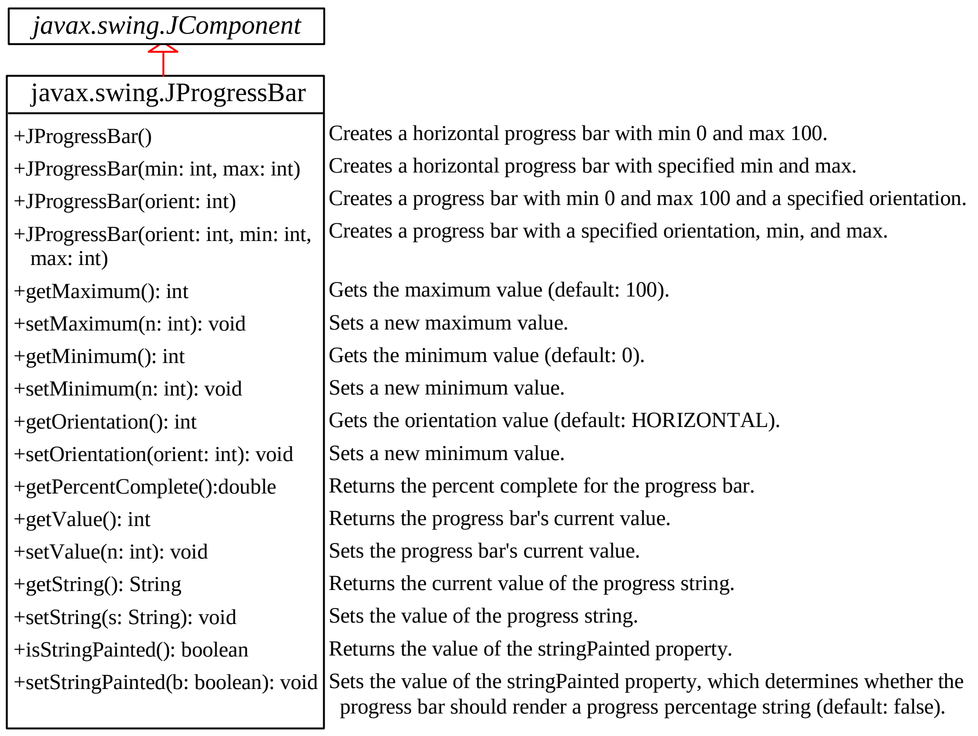


**Figure 3**
*JProgressBar displays the progress of a task.*

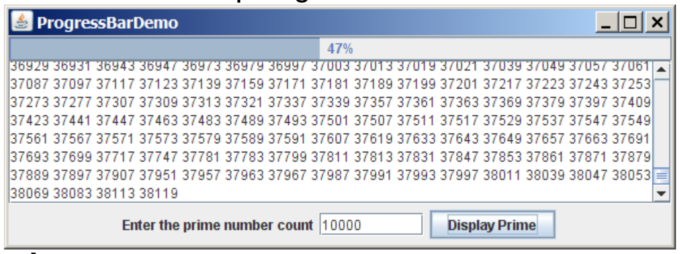
***<PD: UML Class Diagram>***



| | |
|---|---|
| +JProgressBar() | Creates a horizontal progress bar with min 0 and max 100. |
| +JProgressBar(min: int, max: int) | Creates a horizontal progress bar with specified min and max. |
| +JProgressBar(orient: int) | Creates a progress bar with min 0 and max 100 and a specified orientation. |
| +JProgressBar(orient: int, min: int, max: int) | Creates a progress bar with a specified orientation, min, and max. |
| +getMaximum(): int | Gets the maximum value (default: 100). |
| +setMaximum(n: int): void | Sets a new maximum value. |
| +getMinimum(): int | Gets the minimum value (default: 0). |
| +setMinimum(n: int): void | Sets a new minimum value. |
| +getOrientation(): int | Gets the orientation value (default: HORIZONTAL). |
| +setOrientation(orient: int): void | Sets a new minimum value. |
| +getPercentComplete():double | Returns the percent complete for the progress bar. |
| +getValue(): int | Returns the progress bar's current value. |
| +setValue(n: int): void | Sets the progress bar's current value. |
| +getString(): String | Returns the current value of the progress string. |
| +setString(s: String): void | Sets the value of the progress string. |
| +isStringPainted(): boolean | Returns the value of the stringPainted property. |
| +setStringPainted(b: boolean): void | Sets the value of the stringPainted property, which determines whether the progress bar should render a progress percentage string (default: false). |

**Figure 4**
*JProgressBar is a Swing component with many properties that enable you to customize a progress bar.*

Listing 2 gives an example that lets the user specify the number of primes, say *n*, and displays the first *n* primes starting from 2, as shown in Figure 5. The program displays the primes in the text area and updates the completion status in a progress bar.



```
36929 36931 36943 36947 36973 36979 36997 37003 37013 37019 37021 37039 37049 37057 37061
37087 37097 37117 37123 37139 37159 37171 37181 37189 37199 37201 37217 37223 37243 37253
37273 37277 37307 37309 37313 37321 37337 37339 37357 37361 37363 37369 37379 37397 37409
37423 37441 37447 37463 37483 37489 37493 37501 37507 37511 37517 37529 37537 37547 37549
37561 37567 37571 37573 37579 37589 37591 37607 37619 37633 37643 37649 37657 37663 37691
37693 37699 37717 37747 37781 37783 37799 37811 37813 37831 37847 37853 37861 37871 37879
37889 37897 37907 37951 37957 37963 37967 37987 37991 37993 37997 38011 38039 38047 38053
38069 38083 38113 38119
```

**Figure 5**
*The user enters the number of prime numbers and clicks the Display Prime button to display the primes starting from 2 to the text area.*

**Listing 2 ProgressBarDemo.java**

******PD: Please add line numbers in the following code******
***<Side Remark line 7: progress bar>***
***<Side Remark line 13: JProgressBar properties UI>***
***<Side Remark line 17: wrap word>***
***<Side Remark line 18: wrap line>***
***<Side Remark line 31: create task>***
***<Side Remark line 29: add button listener>***
***<Side Remark line 34: add property listener>***
***<Side Remark line 37: get property value>***
***<Side Remark line 42: execute task>***
***<Side Remark line 48: task class>***
***<Side Remark line 59: override doInBackground>***
***<Side Remark line 66: override process>***
***<Side Remark line 72: compute primes>***
***<Side Remark line 81: set progress property>***
***<Side Remark line 82: publish a prime>***

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;

public class ProgressBarDemo extends JApplet {
  private JProgressBar jpb = new JProgressBar();
  private JTextArea jtaResult = new JTextArea();
  private JTextField jtfPrimeCount = new JTextField(8);
  private JButton jbtDisplayPrime = new JButton("Display Prime");

  public ProgressBarDemo() {
    jpb.setStringPainted(true); // Paint the percent in a string
    jpb.setValue(0);
    jpb.setMaximum(100);

    jtaResult.setWrapStyleWord(true);
    jtaResult.setLineWrap(true);

    JPanel panel = new JPanel();
    panel.add(new JLabel("Enter the prime number count"));
    panel.add(jtfPrimeCount);
    panel.add(jbtDisplayPrime);

    add(jpb, BorderLayout.NORTH);
    add(new JScrollPane(jtaResult), BorderLayout.CENTER);
    add(panel, BorderLayout.SOUTH);

    jbtDisplayPrime.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        ComputePrime task = new ComputePrime(
          Integer.parseInt(jtfPrimeCount.getText()), jtaResult);

        task.addPropertyChangeListener(new PropertyChangeListener() {
          public void propertyChange(PropertyChangeEvent e) {
            if ("progress".equals(e.getPropertyName())) {
              jpb.setValue((Integer)e.getNewValue());
            }
          }
        });

        task.execute(); // Execute SwingWorker
      }
    });
  }

  /** Task class for SwingWorker */
  static class ComputePrime extends SwingWorker<Integer, Integer> {
    private int count;
    private JTextArea result; // Textarea in the UI

    /** Construct a runnable Task */
    public ComputePrime(int count, JTextArea result) {
      this.count = count;
      this.result = result;
    }

    /** Code run on a background thread */
    protected Integer doInBackground() {
      publishPrimeNumbers(count);

      return 0; // doInBackground must return a value
    }

    /** Override process to display published prime values */
    protected void process(java.util.List<Integer> list) {
      for (int i = 0; i < list.size(); i++)
        result.append(list.get(i) + " ");
    }

    /** Publish the first n primes number */
    private void publishPrimeNumbers(int n) {
      int count = 0; // Count the number of prime numbers
      int number = 2; // A number to be tested for primeness

      // Repeatedly find prime numbers
      while (count <= n) {
        // Print the prime number and increase the count
        if (isPrime(number)) {
          count++; // Increase the count
          setProgress(100 * count / n); // Update progress
          publish(number); // Publish the prime number
        }

        // Check if the next number is prime
        number++;
      }
    }

    /** Check whether number is prime */
    private static boolean isPrime(int number) {
      for (int divisor = 2; divisor <= number / 2; divisor++) {
        if (number % divisor == 0) { // If true, number is not prime
          return false; // number is not a prime
        }
      }

      return true; // number is prime
    }
  }
}
```

The SwingWorker class generates a PropertyChangeEvent whenever the setProgress method is invoked. The setProgress method (line 81) sets a new progress value between 0 and 100. This value is wrapped in the PropertyChangeEvent. The listener of this event can obtain the progress value using the getNewValue() method (line 37). The progress bar is updated using this new progress value (line 37).

The program creates a JProgressBar (line 7) and sets its properties (lines 13–15).

***<Side Remark: override doInBackground>***

The inner class ComputePrime is a SwingWorker (line 48). It overrides the doInBackground method to run publishPrimeNumbers in a background thread (line 60). The publishPrimeNumbers method finds the specified number of primes starting from 2. When a prime is found, the setProgress method is invoked to set a new progress value (line 81). This causes a PropertyChangeEvent to be fired, which is notified to the listener.

***<Side Remark: override process>***

When a prime is found, the publish method is invoked to send the data to the process method (line 82). The process method is overridden (lines 66–69) to display the primes sent from the publish method. The primes are displayed in the text area (line 68).