Multitarea En Swing

1.- Introducción.

En las aplicaciones Java que usan Swing es particularmente importante manejar con cuidado la concurrencia. Una aplicación Java que usa Swing y que está bien desarrollada usa la concurrencia para crear una interfaz de usuario que nunca se bloquea, de modo que la aplicación siempre es capaz de responder a la interacción del usuario en un momento dado con independencia de las tareas que esté llevando a cabo la aplicación en ese momento. Para desarrollar una aplicación que tiene esta capacidad de respuesta, el desarrollador debe aprender cómo Swing emplea los hilos.

Las aplicaciones Java que usan Swing tienen tres tipos de hilos:

- Un hilo inicial o principal.
- Un hilo de despacho o expedición de eventos.
- Varios hilos trabajadores, también conocidos como hilos en segundo plano.

2.- Crear e inicializar la IGU de la manera correcta.

En toda aplicación Java, la máquina virtual arranca la aplicación creando el hilo principal de dicha aplicación y, a continuación, le cede el control para que se empiece a ejecutar. Este hilo invoca al método `main()` de la aplicación, el cual constituye el punto de entrada o de inicio de la aplicación.

En las aplicaciones Java que usan Swing, la principal tarea del hilo principal es proveer la creación e inicialización de la interfaz gráfica de usuario (IGU) de la aplicación. Una vez que se finaliza esta tarea, generalmente termina la ejecución del método `main()` de la aplicación y, por tanto, también termina de ejecutarse el hilo principal.

En dichas aplicaciones Java que usan Swing, existe un hilo de despacho de eventos dedicado a la IGU. Este hilo es el que dibuja y actualiza los componentes de la IGU, y también es el que responde a las interacciones del usuario con la IGU invocando los correspondientes manejadores de eventos de la aplicación. De esta explicación podemos sacar dos conclusiones:

- Todos los manejadores de eventos son ejecutados por el hilo de despacho de eventos.
- Toda interacción con los componentes de la IGU y con sus modelos de datos asociados debe realizarse únicamente por el hilo de despacho de eventos.

Por tanto, acceder a los componentes de la IGU o a sus manejadores de eventos desde otros hilos distintos al hilo de despacho de eventos puede causar errores de dibujo y actualización de la IGU y, en el peor de los casos, interbloqueo.

Por todo ello, llegamos a la primera regla para trabajar con Swing:

Regla 1
No se debe interactuar con componentes Swing excepto desde el hilo de despacho de eventos.

Veamos una aplicación práctica de esta regla. Es frecuente encontrar que el hilo principal de una aplicación Java que usa Swing tiene una forma parecida a la siguiente:

```
public class VentanaPrincipal extends javax.swing.JFrame
{
  ...

  public static void main(String[] args)
  {
    new VentanaPrincipal().setVisible(true);
  }
}
```

Aunque pueda parecer un código inocuo, viola la primera regla para trabajar con Swing, ya que se está interaccionando con componentes Swing desde un hilo distinto del hilo de despacho de eventos. Este error es fácil de cometer y, además, los problemas de sincronización que se pueden presentar no son inmediatamente obvios. Para evitar este error, el hilo principal debe tener la forma siguiente:

```
public class VentanaPrincipal extends javax.swing.JFrame
{
  ...

  public static void main(String[] args)
  {
    SwingUtilities.invokeLater(new Runnable()
    {
      public void run()
      {
        new VentanaPrincipal().setVisible(true);
      }
    });
  }
}
```

El hilo principal provee la creación e inicialización de la IGU creando un objeto `Runnable` que crea e inicializa la IGU pero cuyo método `run()` no es ejecutado por el hilo principal sino por el hilo de despacho de eventos. Para que esto ocurra, se pasa dicho objeto como argumento al método estático `invokeLater()` o al método estático `invokeAndWait()`, donde ambos pertenecen a la clase `javax.swing.SwingUtilities`.

De este modo, se actúa directamente sobra la cola de despacho de eventos que es procesada por el hilo de despacho de eventos, ya que estos métodos estáticos colocan el objeto `Runnable` que reciben como argumento al final de la cola de despacho de eventos existentes en ese momento. De este modo, el código del método `run()` de dicho objeto será ejecutado por el hilo de despacho de eventos después de que haya ejecutado los demás eventos pendientes que están en la cola de despacho de eventos.

Por tanto, los métodos estáticos `invokeLater()` e `invokeAndWait()` reciben como argumento un objeto que implementa la interfaz `Runnable`. Aunque es frecuente usar un objeto `Runnable` para crear un nuevo hilo, en este caso el código del método `run()` es ejecutado por el hilo de despacho de eventos y no por uno nuevo.

Ambos métodos estáticos son similares, pero presentan tres importantes diferencias semánticas.

En primer lugar, el método `invokeLater()` es asíncrono de modo que, una vez que ha insertado el objeto que recibe como argumento en la cola de despacho de eventos, retorna inmediatamente sin esperar a que el hilo de despacho de eventos ejecute el código, de tal forma que el hilo que invocó dicho método puede seguir ejecutándose. Sin embargo, el método `invokeAndWait()` es síncrono de modo que, una vez que ha insertado el objeto que recibe como argumento en la cola de despacho de eventos, no retorna hasta que el hilo de despacho de eventos haya ejecutado el método `run()` de ese objeto, permaneciendo a la espera mientras tanto el hilo que invocó dicho método.

Como normal general, se debería usar el método `invokeAndWait()` para leer el valor de componentes Swing, para actualizar la IGU o para asegurar que algo se muestra por pantalla antes de continuar la ejecución de la aplicación. En otro caso, se puede usar el método `invokeLater()`. Si es el hilo principal el que invoca uno de estos métodos como en el ejemplo anterior, suele dar igual cuál de los dos métodos se usa dado que proveer la creación e inicialización de la IGU es normalmente lo último que hace el hilo principal.

En segundo lugar, el método `invokeAndWait()` no puede ser invocado por el propio hilo de despacho de eventos. Esto es debido a que el hilo que ejecuta el método `invokeAndWait()` debe esperar para poder continuar su ejecución a que el hilo de despacho de eventos ejecute el código del método `run()` del objeto que dicho método ha recibido como argumento. Pero ningún hilo, incluyendo el hilo de despacho de eventos, puede quedarse esperando a que él mismo realice una tarea. En consecuencia, si el hilo de despacho de eventos ejecuta el método `invokeAndWait()`, la máquina virtual lanza la excepción `java.lang.Error`. En caso de que no sepa qué hilo está ejecutando determinado código, puede determinar si dicho código está siendo ejecutado por el hilo de despacho de eventos mediante el método `isEventDispatchThread()` de la clase `javax.swing.SwingUtilities`.

En tercer lugar, el método `invokeAndWait()` puede lanzar una excepción `InterruptedException` si el hilo que ejecuta este método es interrumpido mientras espera a que el hilo de despacho de eventos termine de ejecutar el método `run()`, o bien una excepción `InvocationTargetExcepcion` si el objeto `Runnable` lanza una excepción `java.lang.Error` o `java.lang.RuntimeException` mientras el hilo de despacho de eventos está ejecutando su método `run()`. El método `invokeLater()` no lanza ninguna excepción.

3.- Reducir la carga del hilo de despacho de eventos.

Una vez que la IGU ha sido creada e inicializada del modo explicado anteriormente, una aplicación Java que usa Swing está fundamentalmente guiada por eventos producidos por las interacciones del usuario con la IGU, cada uno de los cuales provoca que el hilo de despacho de eventos ejecute el correspondiente manejador de eventos asociado.

Es importante recordar cuando se escribe código para manejar eventos que todo ese código se ejecuta en el mismo hilo: el hilo de despacho de eventos. Esto implica que mientras el código de

manejo de un cierto evento se está ejecutando, ningún otro evento puede ser procesado. El código para manejar el siguiente evento empezará a ejecutarse sólo cuando termine de ejecutarse el manejador de eventos que se está ejecutando actualmente. Por tanto, la capacidad de respuesta de la aplicación a las interacciones del usuario con la IGU es dependiente de cuánto tiempo cuesta ejecutar los manejadores de eventos. Por tanto, los manejadores de eventos deben ejecutarse en tan poco tiempo como sea posible.

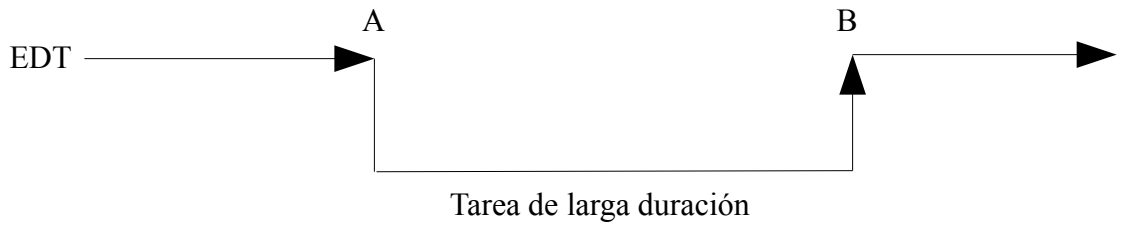
Por ello, los manejadores de eventos ejecutados por el hilo de despacho de eventos deben finalizar rápidamente de modo que la IGU pueda tener una buen tiempo de respuesta ante las interacciones del usuario. Si el hilo de despacho de eventos realiza tareas de larga duración, los eventos se acumulan en el hilo de despacho de eventos y la aplicación puede parecer bloqueada ya que no puede responder a evento alguno.

Aunque es menos frecuente, una vez que la IGU ha sido creada e inicializada puede ocurrir que un hilo de la aplicación envíe tareas al hilo de despacho de eventos para que sean ejecutadas por éste a través del método `run()` de un objeto `Runnable` pasado como argumento al método estático `invokeLater()` o al método estático `invokeAndWait()`. Por las mismas razones explicadas en los párrafos anteriores, es importante también que estas tareas sean cortas.

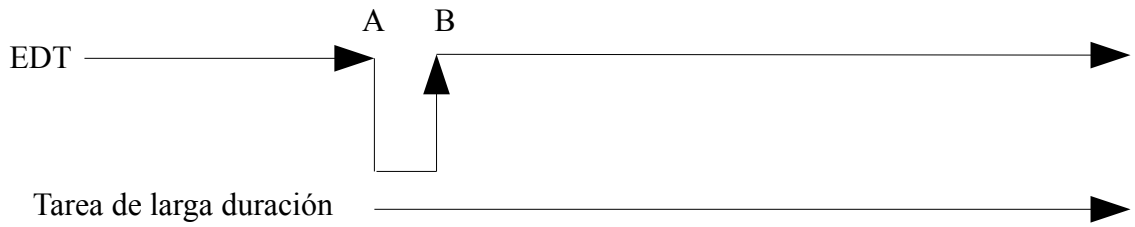
Por todo ello, llegamos a la segunda regla para trabajar con Swing:

Regla 2
Si un manejador de eventos debe realizar una tarea que requiere mucho tiempo o que se puede bloquear, dicha tarea debe ser ejecutada por un nuevo hilo, que recibe el nombre de hilo trabajador o hilo en segundo plano.

La siguiente figura muestra a un hilo de despacho de eventos -que se representará mediante el acrónimo HDE- que no puede procesar evento alguno en el espacio de tiempo que existe entre los puntos A y B dado que en dicho espacio de tiempo se está ejecutando una tarea de larga duración o que se puede bloquear:



La siguiente figura muestra lo que ocurre al utilizar un hilo trabajador. El hilo de despacho de eventos lanza un hilo trabajador para que realice su tarea asíncronamente y rápidamente retorna para seguir procesando eventos. En este caso, el espacio de tiempo entre los puntos Ay B es corto, de modo que el hilo de despacho de eventos puede seguir procesando eventos de la IGU sin demorarse en exceso:



Se puede hacer cumplir esta regla extendiendo la clase abstracta `SwingWorker<T,V>` del paquete `javax.swing`.

El ciclo de vida de un objeto que pertenece a una subclase de `SwingWorker<T,V>` involucra tres hilos:

- El hilo actual es el hilo que ejecuta el método `execute()` de `SwingWorker<T,V>`. Este método asíncrono asocia el `SwingWorker<T,V>` con el hilo trabajador que ha de ejecutarlo y retorna inmediatamente al hilo actual. Hay un cierto número de hilos trabajadores disponible. En el caso de que todos los hilos trabajadores estén ocupados ejecutando otros `SwingWorker<T,V>`, este `SwingWorker<T,V>` es insertado en una cola de espera. Es frecuente que el hilo actual en el que se instancia la subclase de `SwingWorker<T,V>` sea el hilo de despacho de eventos.

- El hilo trabajador es el hilo que ejecuta en segundo plano el método `doInBackground()` de `SwingWorker<T,V>`. Este método es el que lleva a cabo la tarea de larga duración o que se puede bloquear.

- El hilo de despacho de eventos es el hilo que ejecuta los métodos `process()` y `done()` de `SwingWorker<T,V>`. Estos métodos son los que interaccionan con los componentes de la IGU.

Para utilizar correctamente la clase abstracta `SwingWorker<T,V>` es necesario extender esta clase e implementar el método `doInBackground()`, ya que este método es el que realiza la tarea de larga duración o que se puede bloquear.

La clase `SwingWorker<T,V>` está diseñada para que el método `doInBackground()` se ejecute una única vez, de modo que ejecutar más de una vez el método `execute()` no hará que se ejecute más de una vez el método `doInBackground()`. Por ello, si es necesario ejecutar una tarea en segundo plano más de una vez, será necesario instanciar más de una vez la subclase que extiende a `SwingWorker<T,V>`.

Suponiendo que el hilo actual es el hilo de despacho de eventos, emplear un hilo trabajador suele implicar los siguientes pasos:

- El hilo de despacho de eventos instancia la subclase de `SwingWorker<T,V>`, de modo que `SwingWorker<T,V>` entra en el estado `SwingWorker.StateValue.PENDING` -o sea, PENDIENTE-, puesto que aún no se ha asociado con un hilo trabajador.

- El hilo de despacho de eventos invoca el método `execute()` de `SwingWorker<T,V>`, que asocia el `SwingWorker<T,V>` con el hilo trabajador que ha de ejecutarlo. Internamente el hilo de despacho de eventos notifica a los oyentes registrados para recibir la notificación de una

modificación en el estado de `SwingWorker<T,V>` que dicho estado ha cambiado a `SwingWorker.StateValue.STARTED` -o sea, iniciado-.

- El hilo trabajador ejecuta el método `doInBackground()` de `SwingWorker<T,V>`. Si es necesario realizar algún procesamiento sobre los resultados intermedios que el hijo trabajador produce antes de proporcionar un resultado final, es posible publicar dichos resultados intermedios, que son de tipo `V`, tendremos que invocar dentro de `doInBackground()` al método `publish()` de `SwingWorker<T,V>`.

  Este método `publish()` internamente envía resultados intermedios en forma de segmentos de datos al método `process()` de `SwingWorker<T,V>`, de modo que estos resultados intermedios serán procesados por el hilo de despacho de eventos cuando éste ejecute el método `process()`. Por tanto, `publish()` es ejecutado por el hilo trabajador y `process()` por el hilo de despacho de eventos. Lo que ocurre realmente es que `publish()` crea un objeto `Runnable` cuyo método `run()` invoca a `process()` y después mediante `invokeLater()` este objeto `Runnable` es enviado a la cola de despacho de eventos. Ya vimos que `invokeLater()` es asíncrono, por lo que el método `process()` es invocado asíncronamente por el hilo de despacho de eventos. Por tanto, pueden producirse varias invocaciones de `publish()` antes de que sea ejecutado el método `process()` por el hilo de despacho de eventos; con el propósito de lograr un buen rendimiento, todas esas invocaciones de `publish()` se fusionan internamente en una sola invocación con los respectivos argumentos concatenados. Lo habitual, por tanto, en caso de necesitar trabajar con resultados intermedios, es invocar `publish()` dentro de `doInBackground()` y sobreescribir `process()`.

- Una vez que el hilo trabajador termina de ejecutar `doInBackground()`, internamente el hilo de despacho de eventos notifica a los oyentes registrados para recibir la notificación de una modificación en el estado de `SwingWorker<T,V>` que dicho estado ha cambiado a `SwingWorker.StateValue.DONE` -o sea, terminado-. Además, automáticamente el hilo de despacho de eventos ejecuta `done()`; por ello, no se debe invocar `done()` directamente. La implementación por defecto de `done()` no hace nada así que, en caso de que sea necesario actualizar un componente de la IGU en función del resultado final generado por el hilo trabajador, será necesario sobreescribir `done()`.

  Se usa el método `get()` de `SwingWorker<T,V>` para recuperar el resultado final del método `doInBackground()`. De hecho, tanto `doInBackground()` como `get()` devuelven valores del tipo T. Es importante tener en cuenta que este método es síncrono, de modo que el hilo que lo invoca permanece bloqueado hasta que el hilo trabajador ha finalizado. Por ello, no debe ser invocado por el hilo de despacho de eventos hasta que el resultado final generado por el hilo trabajador esté disponible ya que, de lo contrario, la IGU se bloquearía y el usuario tendría que esperar a que el hilo trabajador finalizara. Por ello, es buena idea usar el método `get()` dentro de `done()`.

Finalmente, tiene una propiedad `progress` -o sea, progreso-. Mientras el hilo trabajador progresa, es posible actualizar esta propiedad con un valor entero entre 0 y 100. Igualmente, el hilo trabajador puede notificar a los respectivos oyentes registrados cuando esta propiedad cambia.

Si se desea permitir que un usuario pueda cancelar un hilo trabajador, el código de la subclase de `SwingWorker<T,V>` debería comprobar periódicamente si el usuario ha solicitado dicha cancelación. Para ello, se usa el método `isCancelled()` de `SwingWorker<T,V>`. Es una buena idea invocar este método dentro de bucles, por ejemplo, ya que la idea es identificar una petición de cancelación tan pronto como sea posible para no continuar con la operación actualmente en curso.

La siguiente aplicación de ejemplo es simple pero ejemplifica bien el uso de la mayoría de métodos de la clase `SwingWorker<T,V>`, que es utilizada para generar un lista de números primos.

El fichero `Main.java`:

```
package org.jomaveger.aplicaciomes.multitareaswing;

import javax.swing.SwingUtilities;

/**
 *
 * @author jomaveger
 */
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                new VentanaPrincipal();
            }
        });
    }
}
```

El fichero `VentanaPrincipal.java`:

```
package org.jomaveger.aplicaciomes.multitareaswing;

import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JProgressBar;
import javax.swing.JScrollPane;

/**
 *
 * @author jomaveger
 */
public class VentanaPrincipal extends JFrame implements ActionListener {

    private JButton botonInicio, botonParada;
    private JScrollPane panelDesplazamiento;
    private JList lista;
    private DefaultListModel modeloLista;
    private JProgressBar barraProgreso;
    private GeneraPrimos generaPrimos;
```

```
    public VentanaPrincipal() {
        super("Multitarea Swing");
        this.panelDesplazamiento = new JScrollPane();
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.getContentPane().setLayout(new FlowLayout());
        this.botonInicio = this.construyeBoton("Iniciar");
        this.botonParada = this.construyeBoton("Parar");
        this.botonParada.setEnabled(false);
        this.barraProgreso = this.construyeBarraProgreso(0, 99);
        this.modeloLista = new DefaultListModel();
        this.lista = new JList(this.modeloLista);
        this.panelDesplazamiento.setViewportView(this.lista);
        this.getContentPane().add(this.panelDesplazamiento);
        this.pack();
        this.setVisible(true);
    }

    private JButton construyeBoton(String titulo) {
        JButton b = new JButton(titulo);
        b.setActionCommand(titulo);
        b.addActionListener(this);
        this.getContentPane().add(b);
        return b;
    }

    private JProgressBar construyeBarraProgreso(int min, int max) {
        JProgressBar progressBar = new JProgressBar();
        progressBar.setMinimum(min);
        progressBar.setMaximum(max);
        progressBar.setStringPainted(true);
        progressBar.setBorderPainted(true);
        this.getContentPane().add(progressBar);
        return progressBar;
    }

    public void actionPerformed(ActionEvent e) {
        if("Iniciar".equals(e.getActionCommand())) {
            this.modeloLista.clear();
            this.botonInicio.setEnabled(false);
            this.botonParada.setEnabled(true);
            this.generaPrimos = new GeneraPrimos(this.modeloLista,
                this.barraProgreso, this.botonInicio, this.botonParada);
            this.generaPrimos.execute();
        }
        else if("Parar".equals(e.getActionCommand())) {
            this.botonInicio.setEnabled(true);
            this.botonParada.setEnabled(false);
            this.generaPrimos.cancel(true);
            this.generaPrimos = null;
        }
    }
}
```

El fichero `GeneraPrimos.java`:

```
package org.jomaveger.aplicaciomes.multitareaswing;

import java.util.ArrayList;
import java.util.concurrent.ExecutionException;
import javax.swing.DefaultListModel;
import javax.swing.JButton;
import javax.swing.JProgressBar;
import javax.swing.SwingWorker;

/**
 *
 * @author jomaveger
 */
public   class   GeneraPrimos   extends   SwingWorker<ArrayList<Integer>,
Integer> {

    private DefaultListModel modeloLista;
    private JProgressBar barraProgreso;
    private JButton botonInicio, botonParada;

   public   GeneraPrimos(DefaultListModel   modeloLista,   JProgressBar
barraProgreso, JButton botonInicio, JButton botonParada) {
        this.modeloLista = modeloLista;
        this.barraProgreso = barraProgreso;
        this.botonInicio = botonInicio;
        this.botonParada = botonParada;
    }

    protected ArrayList<Integer> doInBackground() {
        Integer valorTemp = new Integer(1);
        ArrayList<Integer> lista = new ArrayList<Integer>();
        for (int i = 0; i < 100; i++) {
            for (int j = 0; j < 100 && !isCancelled(); j++) {
               valorTemp = encuentraSiguientePrimo(valorTemp.intValue());
            }
            publish(new Integer(i));
            lista.add(valorTemp);
        }
        return lista;
    }
    @Override
    protected void process(java.util.List<Integer> lista) {
        if(!isCancelled()) {
            Integer parteCompletada = lista.get(lista.size() - 1);
            barraProgreso.setValue(parteCompletada.intValue());
        }
    }

    @Override
    protected void done() {
        if(!isCancelled()) {
            try {
                ArrayList<Integer> results = get();
                for (Integer i : results)
                    modeloLista.addElement(i.toString());
```

```
                this.botonInicio.setEnabled(true);
                this.botonParada.setEnabled(false);
            }
            catch (ExecutionException ex) {
                ex.printStackTrace();
            } catch (InterruptedException ex) {
                ex.printStackTrace();
            }
        }
    }

    private Integer encuentraSiguientePrimo(int num ) {
        do  {
            if(num % 2 == 0)
                num++;
            else
                num = num + 2;
        } while (!esPrimo(num));
        return new Integer(num);
    }

    private boolean esPrimo(int num) {
        int i;
        for (i = 2; i <= num / 2; i++ ) {
            if(num % i == 0)
                return false;
        }
        return true;
    }
}
```

Bibliografía:

- Improve Application Performance With SwingWorker in Java SE 6
- More Enhancements in Java SE 6
- *Java Threads*, Scott Oaks & Henry Wong, Ed. O'Reilly, 3ª Edición, 2004
- *Core Java 2 Vol. 2 – Características Avanzadas*, Cay Horstmann & Gary Cornell, Ed. Prentice Hall – Pearson Educación, 7ª Edición, 2005

José María Vegas Gertrudix