Programación Multi-hebra en Java
Parte II - Swing y Hebras

Java Multithreading
Part II - Swing & Threads

Samir Genaim

¿Qué Escribe este Programa?

```
public class List {
    private Cell data;

    class Cell {
        Integer value;
        Cell next;
    }

    public void add(Integer x) {
        Cell tmp = new Cell();
        tmp.value = x ;
        tmp.next = data;
        data = tmp;
    }

    public Integer length() {
        int i=0;
        Cell aux = data;
        while ( aux != null ) {
            aux = aux.next;
            i++;
        }
        return i;
    }
    ...
}
```

```
public class Test extends Thread {
    static List f = new List();
    private int n;
    private String id;

    Test( int n, String id ) {
        this.n = n;
        this.id = id;
    }

    public void run() {
        for(int i=0; i<n; i++) f.add(1);
    }

    static public void main(String[] args) ... {
        Thread t1 = new Test(1000,"T1");
        Thread t2 = new Test(1000,"T2");
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Length: "+f.length());
    }
}
```

see: examples.threads.ex5

Acceso (Read/Write) Simultáneo

- Cuando dos hebras que están ejecutando a la vez intentan a modificar un recurso común, este recurso puede quedar en un estado indeterminado …
- Para evitar este tipo de problemas, se puede usar un **mecanismo de sincronización** para que sólo una hebra puede acceder al recurso a la vez o **asegurarse de que todas la modificación se hacen de una sola hebra**
- El mecanismo de sincronización nos permite declarar que sólo una hebra puede ejecutar **un bloque de código** a la vez -- si una intenta a ejecutarlo mientras otra lo está ejecutando, pues se bloquea hasta que acabe la primera
- Hay varias formas de sincronización en Java, vamos a ver una en detalles más adelante …

En Java es más Complicado ...

- Aparte del problema de dejar recursos comunes en estado indeterminado, Java utiliza el "Weak memory model" ...
- Brevemente, un "Memory model" define lo que cada hebra ve cuando se lee una variable compartida ...

¿Este programa termina?

```
public class MemoryModelEx1 {
    static boolean f = false;

    public static void main(String[] args) throws ... {

        new Thread() {
            public void run() {
                while (!f) { do something ...}
            };
        }.start();

        Thread.sleep(2000);
        f = true;
    }
}
```

see: examples.threads.ex7.MemoryModelEx1

En Java es más Complicado ...

- Aparte del problema de dejar recursos comunes en estado indeterminado, Java utiliza el "Weak memory model" ...
- Brevemente, un "Memory model" define lo que cada hebra ve cuando se lee una variable compartida ...

¿Este programa termina? **¡Puede que no!**

```
public class MemoryModelEx1 {
    static boolean f = false;

    public static void main(String[] args) throws ... {

        new Thread() {
            public void run() {
                while (!f) { do something ...}
            };
        }.start();

        Thread.sleep(2000);
        f = true;
    }
}
```

Hebras pueden utilizar un caché local en lugar de la memoria principal, por lo que las modificaciones a los recursos compartidos pueden no ser visibles a todos. Para que sean visibles, el programa debe estar bien sincronizado ...

see: examples.threads.ex7.MemoryModelEx1

En Java es más Complicado ...

- Aparte del problema de dejar recursos comunes en estado indeterminado, Java utiliza el "Weak memory model" ...
- Brevemente, un "Memory model" define lo que cada hebra ve cuando se lee una variable compartida ...

¿Este programa termina? **¡Puede que no!**

```
public class MemoryModelEx1 {
    static boolean f = false;

    public static void main(String[] args) throws ... {

        new Thread() {
            public void run() {
                while (!f) { do something ...}
            };
        }.start();

        Thread.sleep(2000);
        f = true;
    }
}
```

volatile

Hebras pueden utilizar un caché local en lugar de la memoria principal, por lo que las modificaciones a los recursos compartidos pueden no ser visibles a todos. Para que sean visibles, el programa debe estar bien sincronizado ...

Por ahora, para asegurarse de que las modificaciones se sincronizan con la memoria principal usamos volatile, más adelante veremos más sobre lo que significa estar "bien sincronizado" ...

see: examples.threads.ex7.MemoryModelEx1

Thread Safety (Seguro ante Hebras)

- Decimos que un código, por ejemplo, una librería de Java, es **Thread Safe** si se puede usarlo en programas multi-hebras sin riesgo de tener algún problema al usarlo en varias hebras a la vez ...

- Muchas de las librerías de Java no son **Thread Safe!!** Porque sincronizar cuesta mucho (en tiempo de ejecución) -- el usuario tiene que decidir se quiere sincronizar el uso de este código ...

- No todas las **Collections** (ArrayList, HashMap, etc.) son **Thread Safe**, existen otras versiones que son **Thread Safe** en **java.util.concurrent**

- **Swing** tampoco es **Thread Safe!!**

Swing y Hebras

- Swing y AWT crear una nueva hebra que se utiliza para procesar, entre otras cosas, los eventos de entrada (teclado, ratón, etc.) — **Event Dispatch Thread**
- Para garantizar una interacción fluida, la hebra-de-eventos tiene más prioridad (6).
- La hebra-de-eventos se encarga también de llamar a los "observadores" (**actionPerformed**, etc.) cuando sea necesario. Esto implica que el código incluido en los observadores se ejecuta en la hebra de swing.
- Si el procesado del evento tarda mucho tiempo, los componentes de swing dejarán de responder, porque la hebra-de-eventos está ocupada manejando el último evento (la llamada a **actionPerformed**, etc.).

Swing y Hebras

- Para evitar este tipo de problemas, si el procesamiento de un evento podría tardar mucho, entonces el código correspondiente debe ser ejecutado en otra hebra.

- Swing no es **Thread Safe**, entonces se estamos realizando dos tareas a la vez que pueden modificar los componentes de swing podemos tener resultados inesperados.

- Para evitar esto, **sin sincronización**, cuando queremos modificar un componente, podemos pedir a la hebra-de-eventos que ejecuta el código que corresponde

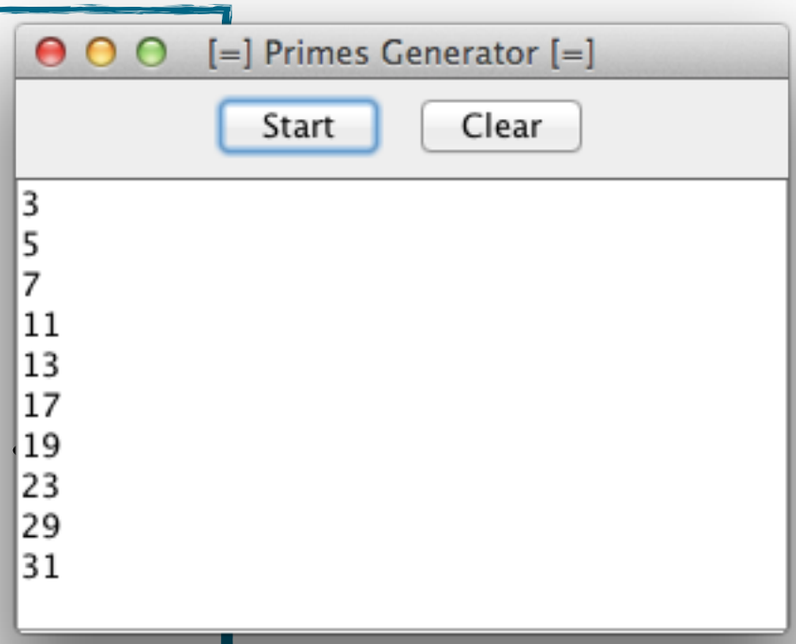
  1. SwingUtilities.invokeAndWait(Runnable)
  2. SwingUtilities.invokeLater(Runnable)

- También podríamos utilizar estos métodos para no bloquear la hebra-de-eventos ...

Swing y Hebras: Ejemplo I

```
public class SwingPrimes_I extends JFrame {
    private JTextArea primes;
    private BigInteger n = new BigInteger("1");
    ...
    private void initGUI() {
        ...
        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showNumbers();
            }
        });

        ...
        clearButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                primes.setText("");
                n = new BigInteger("1");
            }
        });
        ...
        primes = new JTextArea();
        ...
    }
    ...
}
```



¿Que pasa cuando se produce un click sobre **Start?**

```
private void showNumbers() {
    int i = 10;
    while (i > 0) {
        n = Primes.nextPrime(n);
        primes.append(n + "\n");
        System.out.println(n);
        sleepabit();
        i--;
    }
}
```

see: examples.threads.primes.SwingPrimes_1

Swing y Hebras: Ejemplo II

```
public class SwingPrimes_II extends JFrame {
    private JTextArea primes;
    private volatile BigInteger n = new BigInteger("1");
    private volatile Thread t;
    ...
    private void initGUI() {
        ...
        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showNumbers();
            }
        });
        ...
        clearButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                primes.setText("");
                n = new BigInteger("1");
            }
        });
        ...
        primes = new JTextArea();
        ...
    }
    ...
}
```

```
private void showNumbers() {
    int i = 10;
    while (i > 0) {
        n = Primes.nextPrime(n);
        primes.append(n + "\n");
        System.out.println(n);
        sleepabit();
        i--;
    }
}
```

see: examples.threads.primes.SwingPrimes_2

Swing y Hebras: Ejemplo II

```
public class SwingPrimes_II extends JFrame {
    private JTextArea primes;
    private volatile BigInteger n = new BigInteger("1");
    private volatile Thread t;
    ...
    private void initGUI() {
        ...
        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showNumbers();
            }
        });
        ...
        clearButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                primes.setText("");
                n = new BigInteger("1");
            }
        });
        ...
        primes = new JTextArea();
        ...
    }
    ...
}
```

```
if (t == null) {
    t = new Thread() {
        public void run() {
            showNumbers();
            t = null;
        }
    };
    t.start();
}
```

```
private void showNumbers() {
    int i = 10;
    while (i > 0) {
        n = Primes.nextPrime(n);
        primes.append(n + "\n");
        System.out.println(n);
        sleepabit();
        i--;
    }
}
```

see: examples.threads.primes.SwingPrimes_2

Swing y Hebras: Ejemplo II

```
public class SwingPrimes_II extends JFrame {
    private JTextArea primes;
    private volatile BigInteger n = new BigInteger("1");
    private volatile Thread t;
    ...
    private void initGUI() {
        ...
        startButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                showNumbers();
            }
        });
        ...
        clearButton.addActionListener(new ActionListen
            public void actionPerformed(ActionEvent e)
                primes.setText("");
                n = new BigInteger("1");
            }
        });
        ...
        primes = new JTextArea();
        ...
    }
    ...
}
```

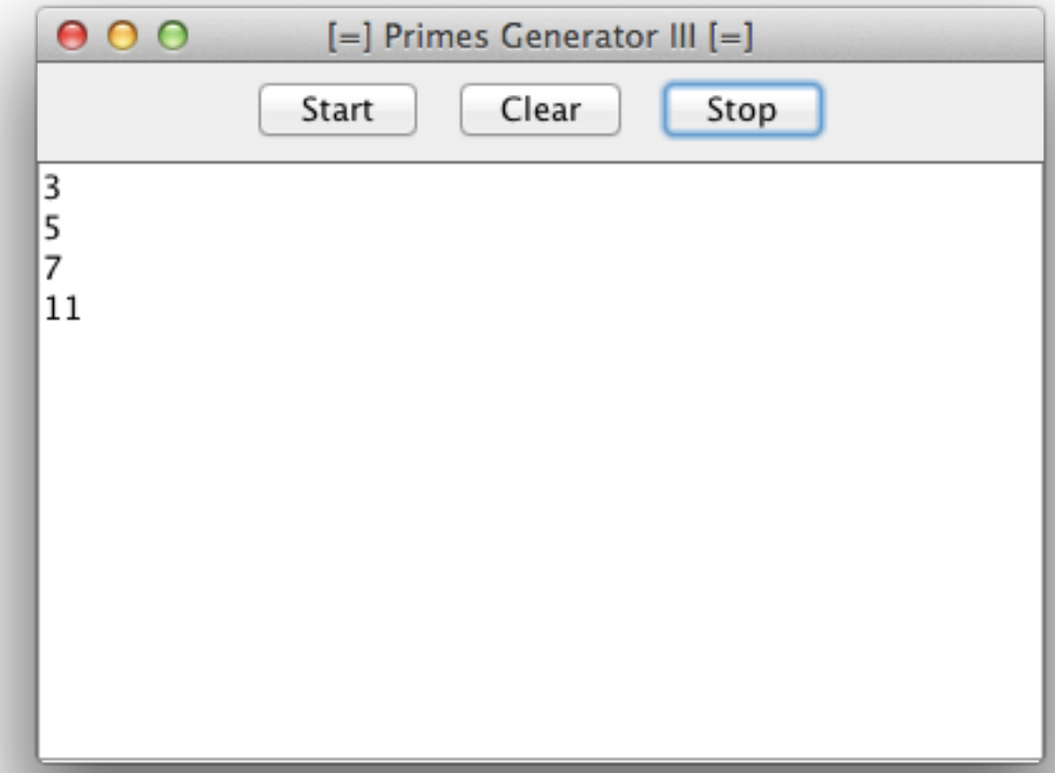
```
if (t == null) {
    t = new Thread() {
        public void run() {
            showNumbers();
            t = null;
        }
    };
    t.start();
}
```

```
final BigInteger x = n;
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        primes.append("" + x + "\n");
    }
});
```

```
private void showNum
    int i = 10;
    while (i > 0) {
        n = Primes.nextPrime(n);
        primes.append(n + "\n");
        System.out.println(n);
        sleepabit();
        i--;
    }
}
```

see: examples.threads.primes.SwingPrimes_2

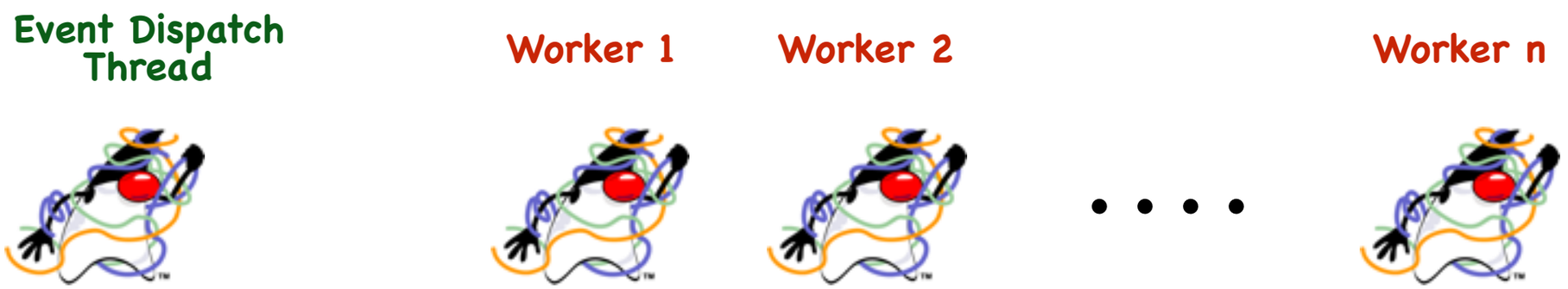
Ejercicio: Añadir un Botón Stop



1. Versión 1: usando interrupts
2. Versión 2: sin usar interrupts

see: examples.threads.primes.SwingPrimes_3 examples.threads.primes.SwingPrimes_4

Swing Workers



- Swing proporciona un marco (SwingWorker) para ejecutar código en otra hebra fácilmente, sin la necesidad de crear una nueva hebra cada vez, etc.
- Imagínese que Swing tiene varios "Workers" al que podemos enviar tareas para ejecutar, etc.
- Este marco está diseñado para ayudarnos a utilizar swing de una manera "thread safe" ...

Enviar Tareas a "Swing Workers"

```
public class SomeTask extends SwingWorker<T1,T2> {

    protected T1 doInBackground() throws Exception {
        ...
    }

    protected void done() {
        ...
    }

    protected void process(List<T2> chunks) {
        ...
    }
}
```

Invocando w.execute() se programa una tarea en un "worker" para ejecutar el método doInBackground() de w. La llamada a execute no espera hasta que acaba la tarea:

```
w = new SomeTask();
w.execute();
```

Cuando doInBackground() termina, se invoca al método done desde la hebra del "Event-Dispatch"

Podemos consultar la salida de doInBackground() usando:

```
w.get();
w.get(1000);
```

La primera bloquea hasta que ese valor está disponible, la segunda bloquea con timeout

Cancelar una Tarea

```
public class SomeTask extends SwingWorker<Integer, Integer> {
    protected Integer doInBackground() throws Exception {
        ...
        while ( !isCanceled() ) {
            ...
        }
        ...
    }
    ...
}
```

Para cancelar una tarea usamos el método cancel:

w.cancel(true) or w.cancel(false)

Esto sólo la marca cómo cancelada. En el método doInBackground() podemos consultar si la tarea ha sido cancelada usando isCanceled(). El parámetro de cancel indica si, además de cancelar la tarea, queremos interrumpir la hebra en el que está ejecutando.

Publicar Resultados Intermedios

Normalmente en doInBackground() hacemos sólo computaciones. Publicar los resultados, por ejemplo mostrándolos usando swing, se hace en el método process mediante llamadas a publish ...

```
public class SomeTask extends SwingWorker<Integer, Integer> {
    protected Integer doInBackground() throws Exception {
        ...
        publish(10);
        ...
        publish(5);
    }
    protected void process(List<Integer> chunks) {
        ...
    }
}
```

Los parámetros de las distintas llamadas a 'publish' se agrupan en chunks (trozos) y se pasan al método 'process' — no podemos suponer nada sobre el tamaño de estos "chunks". La llamada al método process se hace desde la hebra del Event-Dispatch!.

Comunicar Información

Las tarea pueden comunicar valores de of "propiedad" a observadores. Una "propiedad" consiste en un nombre (String) y un valor (Object)

```
public class SomeTask extends SwingWorker<Integer, Integer> {

    protected Integer doInBackground() throws Exception {
      …
      getPropertyChangeSupport().firePropertyChange("name", oldValue, newValue);
      …
    }
}
```

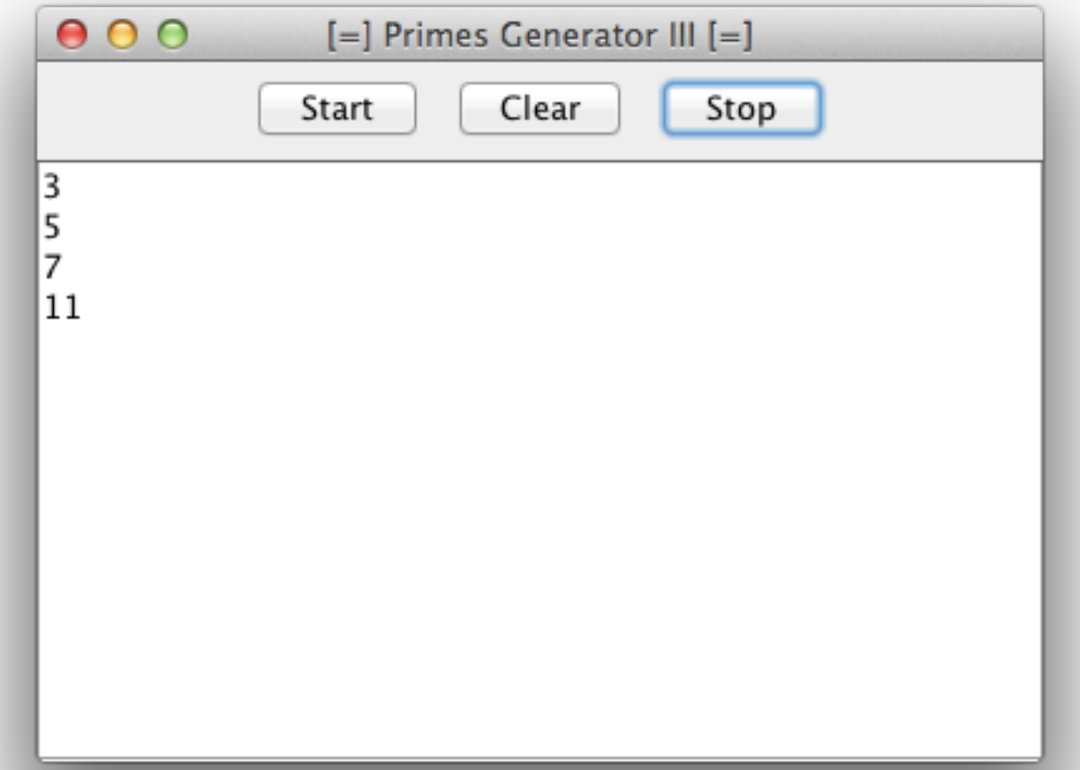
Las llamadas a los observadores se hacen desde la hebra del Event-Dispatch.

```
w.addPropertyChangeListener(new PropertyChangeListener() {
    public void propertyChange(PropertyChangeEvent evt) {
          System.out.println(evt.getPropertyName());
          System.out.println(evt.getOldValue());
          System.out.println(evt.getNewValue());
    }
});
```

Hay algunas propiedades predefinidas:

1. "progress": setProgress(n) cambia el valor de "progress" a n.
2. "state": cambia de valor cuando la tarea empieza, acaba, etc.

Ejercicio



Implementa el ejemplo de los números primos usando Swing Workers

see: examples.threads.primes.SwingPrimes_6 examples.threads.primes.SwingPrimes_7