EI1024/MT1024 "Programación Concurrente y Paralela" 2025–26	Entregable para Laboratorio
Nombre y apellidos (1): Vicente Ventura Ninot	
Nombre y apellidos (2): Víctor-Nesta Reig Buendía	
Tiempo empleado para tareas en casa en formato *h:mm* (obligatorio): 1:30	la04_g

Tema 05. El Problema de la Atomicidad en Java

Tema 06. *Thread Pools* e Interfaces Gráficas en Java

1 Se desea calcular el número π mediante integración numérica de la siguiente función:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx.$$

Este método no es el más rápido para calcular el número π, pero sí uno de las más simples. Consiste en calcular la anterior integral mediante una aproximación numérica basada en el cálculo y acumulación del área de numerosos rectángulos pequeños.

Uno de los parámetros más importantes es el número de rectángulos cuya área se va a sumar. En este caso, este parámetro será pasado en la línea de argumentos, después del número de hebras.

El siguiente programa realiza el cálculo de forma secuencial. Con vistas a facilitar el desarrollo posterior de la versión paralela, este código secuencial contiene un fragmento de código comentado, además de la declaración e inicialización de la variables `numHebras` y `numRectangulos`. Ambas partes no son útiles en la versión secuencial, pero, la inclusión de este fragmento de código simplifica el desarrollo de la versión paralela.

```
/*
// ================================================================================
class Acumula {
// ================================================================================
   double suma;

   // ---------------------------------------------------------------------------
   Acumula() {
      // ...
   }

   // ---------------------------------------------------------------------------
   void acumulaDato( double dato ) {
      // ...
   }

   // ---------------------------------------------------------------------------
   double dameDato() {
      // ...
   }
}

// ================================================================================
class MiHebraMultAcumulaciones extends Thread {
```

```
// ============================================================================
  int       miId, numHebras;
  long      numRectangulos;
  Acumula   a;

  // ----------------------------------------------------------------------------
  MiHebraMultAcumulaciones( int miId, int numHebras, long numRectangulos,
                            Acumula a ) {
    // ...
  }

  // ----------------------------------------------------------------------------
  public void run() {
    // ...
  }
}

// ============================================================================
class MiHebraUnaAcumulacion extends Thread {
// ============================================================================
// ...
}

// ============================================================================
class MiHebraMultAcumulacionAtomic extends Thread {
// ============================================================================
// ...
}

// ============================================================================
class MiHebraUnaAcumulacionAtomic extends Thread {
// ============================================================================
// ...
}

*/

// ============================================================================
class EjemploNumeroPI {
// ============================================================================

  // ----------------------------------------------------------------------------
  public static void main( String args[] ) {
    long                        numRectangulos;
    double                      baseRectangulo, x, suma, pi;
    int                         numHebras;
    long                        t1, t2;
    double                      tSec, tPar;
    // Acumula                     a;
    // MiHebraMultAcumulaciones  vt[];

    // Comprobacion de los argumentos de entrada.
    if( args.length != 2 ) {
      System.out.println( "ERROR: numero de argumentos incorrecto." );
      System.out.println( "Uso: java programa <numHebras> <numRectangulos>" );
      System.exit( -1 );
    }
    try {
      numHebras      = Integer.parseInt( args[ 0 ] );
      numRectangulos = Long.parseLong( args[ 1 ] );
      if( ( numHebras <= 0 ) || ( numRectangulos <= 0 ) ) {
```

```
        System.err.print( "Uso: [ java programa <numHebras> <n> ] " );
        System.err.println( "donde ( numHebras > 0 ) y ( numRectangulos > 0 )" );
        System.exit( -1 );
      }
    } catch( NumberFormatException ex ) {
      numHebras      = -1;
      numRectangulos = -1;
      System.out.println( "ERROR: Numeros de entrada incorrectos." );
      System.exit( -1 );
    }
    System.out.println();
    System.out.println( "Calculo del numero PI mediante integracion." );
    //
    // Calculo del numero PI de forma secuencial.
    //
    System.out.println();
    System.out.println( "Inicio del calculo secuencial." );
    t1 = System.nanoTime();
    baseRectangulo = 1.0 / ( ( double ) numRectangulos );
    suma           = 0.0;
    for( long i = 0; i < numRectangulos; i++ ) {
      x = baseRectangulo * ( ( ( double ) i ) + 0.5 );
      suma += f( x );
    }
    pi = baseRectangulo * suma;
    t2 = System.nanoTime();
    tSec = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
    System.out.println( "Version secuencial. Numero PI: " + pi );
    System.out.println( "Tiempo secuencial (s.):        " + tSec );
/*
    //
    // Calculo del numero PI de forma paralela:
    // Multiples acumulaciones por hebra.
    //
    System.out.println();
    System.out.print( "Inicio del calculo paralelo: " );
    System.out.println( "Multiples acumulaciones por hebra." );
    t1 = System.nanoTime();
    // ...
    t2 = System.nanoTime();
    tPar = ( ( double ) ( t2 - t1 ) ) / 1.0e9;
    System.out.println( "Calculo del numero PI:    " + pi );
    System.out.println( "Tiempo ejecucion (s.):    " + tPar );
    System.out.println( "Incremento velocidad :    " + ... );
    //
    // Calculo del numero PI de forma paralela:
    // Una acumulacion por hebra.
    // ...
    //
    // Calculo del numero PI de forma paralela:
    // Multiples acumulaciones por hebra (Atomica)
    // ...
    //
    // Calculo del numero PI de forma paralela:
    // Una acumulacion por hebra (Atomica).
    // ...
*/
    System.out.println();
    System.out.println( "Fin de programa." );
  }
```

```
  // ---------------------------------------------------------------------
  static double f( double x ) {
    return ( 4.0/( 1.0 + x*x ) );
  }
}
```

1.1) Estudia el código anterior y paralelízalo mediante el uso de hebras con una `distribución cíclica`. Utiliza un objeto de la clase `Acumula` para almacenar el resultado.

En esta versión paralela cada vez que las hebras calculan el área de un rectángulo, deben acumular el valor obtenido sobre el objeto compartido de la clase `Acumula`. Para un correcto manejo del programa, hay que asegurar que el acceso al objeto compartido sea *thread-safe*.

No crees un nuevo programa. Haz que esta implementación paralela se ejecute a continuación de la versión secuencial dentro del mismo programa. Ello permitirá obtener los tiempos y calcular los incrementos de velocidad de forma más rápida y automatizada.

Escribe a continuación la parte de tu código que realiza esta tarea: la definición de la clase `MiHebraMultAcumulaciones` y el código incluido en el programa principal que permite gestionar los objetos de esta clase.

1.2) Modifica el programa anterior, de modo que en la versión paralela las hebras acumulen el área que han calculado en una variable local (`sumaL`), antes de sumarla al objeto compartido.

No crees un nuevo programa. Haz que esta implementación paralela se ejecute a continuación de la versión paralela desarrollada en el apartado anterior. Ello permitirá obtener los tiempos y los incrementos de velocidad de forma más rápida y automatizada.

Escribe a continuación la parte de tu código que realiza esta tarea: la definición de la clase `MiHebraUnaAcumulacion` y el código incluido en el programa principal que permite gestionar los objetos de esta clase.

1.3) En las **DOS** versiones paralelas anteriores se ha utilizado un objeto de la clase `Acumula` que permite acumular números reales con precisión doble de forma atómica, pero también se podría realizar empleando clases y operadores atómicos avanzados, como `DoubleAdder`.

Define las clases `MiHebraMultAcumulacionesAtomic` y `MiHebraUnaAcumulacionAtomic`, como réplica de las anteriores. Estas clases deben manejar un objeto de la clase `DoubleAdder`, utilizando el método `add` para acumular los valores, mientras que el valor final se obtendrá con el método `sum`. Recuerda que se debe **eliminar completamente** la clase `Acumula` y en su lugar utilizar la clase atómica.

Además, modifica el programa principal para que incluya la gestión de estas nuevas clases. Escribe a continuación los cambios realizados en el código.

2 Se dispone de una interfaz gráfica con un cuadro de texto y dos botones denominados `Inicia secuencia` y `Cancela secuencia`. Por el momento, la interfaz no hace nada cuando el usuario realiza alguna acción sobre los botones o sobre el cuadro de texto.

La interfaz está definida por el siguiente código:

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

/*
// ============================================================================
class ZonaIntercambio {
// ============================================================================
  // ...

        public ZonaIntercambio ( ... ) {
    // ...
  }

  // ----------------------------------------------------------------------------
  void setTiempo( ... ) {
    // ...
  }

  // ----------------------------------------------------------------------------
  long getTiempo( ... ) {
    // ...
  }
}
*/

// ============================================================================
public class GUISecuenciaPrimos {
// ============================================================================
  JFrame      container;
  JPanel      jpanel;
  JTextField  txfMensajes;
  JButton     btnIniciaSecuencia, btnCancelaSecuencia;
  JSlider     sldEspera;
//  HebraTrabajadora  t; // Ejercicio 2.2
//  ZonaIntercambio   z; // Ejercicio 2.3

  // ----------------------------------------------------------------------------
  public static void main( String args[] ) {
    GUISecuenciaPrimos gui = new GUISecuenciaPrimos();
    SwingUtilities.invokeLater(new Runnable(){
      public void run(){
        gui.go();
      }
    });
  }

  // ----------------------------------------------------------------------------
  public void go() {
    // Constantes.
    final int valorMaximo = 1000;
    final int valorMedio  = 500;
```

```
// Variables.
JPanel  tempPanel;

// Crea el JFrame principal.
container = new JFrame( "GUI Secuencia de Primos " );

// Consigue el panel principal del Frame "container".
jpanel = ( JPanel ) container.getContentPane();
jpanel.setLayout( new GridLayout( 3, 1 ) );

// Crea e inserta la etiqueta y el campo de texto para los mensajes.
txfMensajes = new JTextField( 20 );
txfMensajes.setEditable( false );
tempPanel = new JPanel();
tempPanel.setLayout( new FlowLayout() );
tempPanel.add( new JLabel( "Secuencia: " ) );
tempPanel.add( txfMensajes );
jpanel.add( tempPanel );

// Crea e inserta los botones de Inicia secuencia y Cancela secuencia.
btnIniciaSecuencia = new JButton( "Inicia secuencia" );
btnCancelaSecuencia = new JButton( "Cancela secuencia" );
tempPanel = new JPanel();
tempPanel.setLayout( new FlowLayout() );
tempPanel.add( btnIniciaSecuencia );
tempPanel.add( btnCancelaSecuencia );
jpanel.add( tempPanel );

// Crea e inserta el slider para controlar el tiempo de espera.
sldEspera = new JSlider( JSlider.HORIZONTAL, 0, valorMaximo , valorMedio );
tempPanel = new JPanel();
tempPanel.setLayout( new BorderLayout() );
tempPanel.add( new JLabel( "Tiempo de espera: " ) );
tempPanel.add( sldEspera );
jpanel.add( tempPanel );

// Activa inicialmente los 2 botones.
btnIniciaSecuencia.setEnabled( true );
btnCancelaSecuencia.setEnabled( true );

// Anyade codigo para procesar el evento del boton de Inicia secuencia.
btnIniciaSecuencia.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
      // ...
    }
} );

// Anyade codigo para procesar el evento del boton de Cancela secuencia.
btnCancelaSecuencia.addActionListener( new ActionListener() {
    public void actionPerformed( ActionEvent e ) {
      // ...
    }
} );

// Anyade codigo para procesar el evento del slider " Espera " .
sldEspera.addChangeListener( new ChangeListener() {
  public void stateChanged( ChangeEvent e ) {
    JSlider sl = ( JSlider ) e.getSource();
    if ( ! sl.getValueIsAdjusting() ) {
      long tiempoEnMilisegundos = ( long ) sl.getValue();
```

```
            System.out.println( "JSlider value = " + tiempoEnMilisegundos );
            // ...
          }
        }
      } );

      // Fija caracteristicas del container.
      container.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      container.pack();
      container.setResizable( false );
      container.setVisible( true );

      System.out.println( "% End of routine: go.\n" );
    }

    // ---------------------------------------------------------------------
    static boolean esPrimo( long num ) {
      boolean primo;
      if( num < 2 ) {
        primo = false;
      } else {
        primo = true;
        long i = 2;
        while( ( i < num )&&( primo ) ) {
          primo = ( num % i != 0 );
          i++;
        }
      }
      return( primo );
    }
}
```

2.1) Modifica la interfaz gráfica para que los botones `Inicia secuencia` y `Cancela secuencia` se activen y desactiven (`setEnabled`) de acuerdo a la siguiente lógica de funcionamiento:

- Inicialmente el botón `Inicia secuencia` debe estar activado y el botón `Cancela secuencia` debe estar desactivado (modificar método `go`).
- Cuando se presione el botón `Inicia secuencia`, éste se desactiva y se activa el botón `Cancela secuencia` (modificar `ActionListener` del primero).
- Cuando se presione el botón `Cancela secuencia`, éste se desactiva y se activa el botón `Inicia secuencia` (modificar `ActionListener` del primero).

ATENCIÓN: Los ejercicios anteriores deben realizarse en casa. Los siguientes, en el aula.

2.2) Modifica la anterior interfaz para que el programa muestre la secuencia de números primos en el cuadro de texto (siempre comenzando por el 2, 3, 5, 7, 11, etc.), cuando el usuario pulse el botón `Inicia secuencia`. Con este objetivo, la hebra *event-dispatching* creará una hebra trabajadora (`t`) en la que delegará dicho trabajo. La variable asociada a la hebra debe definirse como variable de clase, para que sea accesible desde todos los `Listener`. **Su definición aparece comentada en la plantilla suministrada.**

En cuanto el usuario pulse el botón `Cancela secuencia`, la generación de la secuencia debe terminar. Para detener la hebra, se fijará un valor especial en un atributo de la hebra (`fin`), cuyo valor será revisado por ésta cada vez que se genere un nuevo número primo.

Seguidamente se muestra la estructura del cuerpo de la hebra.

```
// Estructura del cuerpo de la hebra
long i = 1L;
while ( ! fin ) {
  if ( esPrimo ( i ) ) {
    // imprime ( i );
    // espera (ejer 2.3)
  }
  i++;
}
```

Una hebra trabajadora no puede llamar a ningún método de un objeto gráfico, ya que éstos sólo pueden ser llamados por la *event-dispatching*. Por tanto, cuando la hebra trabajadora desea escribir sobre el cuadro de texto (`txfMensajes`), debe utilizar los métodos `invokeAndWait` o `invokeLater`, que indican a la *event-dispatching* lo que debe realizar.

Estos métodos ejecutan un objeto Runnable que reciben como parámetro de entrada. El primer método bloquea la hebra hasta que la *event-dispatching* finaliza, por lo que es necesario gestionar dos excepciones, mientras que el segundo no bloquea la hebra.

Escribe a continuación la parte de tu código que realiza tal tarea: la definición de la clase `HebraTrabajadora` y el código a incluir en el programa principal que permite gestionar los objetos de esta clase.

2.3) Crea un copia del fichero anterior (GUISecuenciaPrimosSlider.java), en la que se va a gestionar la barra de deslizamiento horizontal (`JSlider`) que aparece en el interfaz. Con ella se pretende que el usuario pueda determinar la velocidad de generación de números primos. Si la barra está en un extremo, la hebra deberá generar números primos intercalando una demora (método `sleep`) de un segundo **tras la impresión en el cuadro de texto**. Si la barra está en el otro extremo, la hebra deberá generar números primos sin ninguna demora. Se recomienda definir y emplear una nueva clase denominada `ZonaIntercambio`, a través de la cual se comuniquen la hebra gráfica y la hebra calculadora. La hebra gráfica escribirá valores en un objeto de dicha clase y la hebra calculadora tomará valores de dicho objeto. Para que el objeto sea accesible desde todos los `Listener`, se debe definirse como variable de clase **(ver plantilla suministrada)**. Por último comentar que el tiempo de espera se expresa en milisegundos, y que el valor inicial definido en el código es 500 (`valorMedio`).
¿Cómo controlarías que los objetos de esta nueva clase sean accedidos por varias hebras?
Escribe a continuación la parte de tu código que realiza tal tarea: la definición de la clase `ZonaIntercambio`, el código para la gestión de la barra de desplazamiento, y los cambios en la clase `HebraTrabajadora`.

2.4) Se desea sustituir la barra de deslizamiento por dos botones adicionales: Un botón añadirá 0,1 segundos al tiempo de espera, mientras que el otro botón le restará 0,1 segundos.
No hagas ninguna implementación, pero responde a la siguiente pregunta. ¿Se podría realizar dicha modificación sólo con el operador `volatile` o habría que recurrir al modificador `synchronized`? Justifica la respuesta.

3 Este ejercicio es una continuación del ejercicio 1.

3.1) Completa la siguiente tabla para 500 000 000 de rectángulos. Obtén los resultados para 4 hebras en el ordenador del aula, y los resultados para 16 hebras en patan. Redondea los tiempos dejando sólo tres decimales y redondea los incrementos dejando dos decimales.
Justifica los resultados obtenidos.

Ejecución con 500 000 000 rectángulos				
	4 hebras (aula)		16 hebras (patan)	
	Tiempo	Incremento	Tiempo	Incremento
Secuencial	1,259	—	1,995	—
Paralela: Múltiples acumul.	13,097	0,096	82,967	0,024
Paralela: Una única acumul.	0,308	4,081	0,198	10,1
Paralela: Múltiples acumul. (clase atom.)	1,377	0,91	0,397	5,033
Paralela: Una única acumul. (clase atom.)	0,36	3,5	0,19	10,52

Podemos ver que acceder muchas veces a un recurso compartido que bloquea con sychronized produce retrasos muy importantes en la ejecución.

Utilizar un DoubleAdder con muchas inserciones es mejor pero sigue creando un cuello de botella que ralentiza el programa.

Las dos opciones que utilizan la ejecución en dos fases son mucho mejores, ya que aprovechan al máximo cada hebra, evitando bloqueos a la espera de guardar valores en el recurso compartido. Parece que la versión con el DoubleAdder es algo mejor debido a que es una clase optimizada para esta tarea y para ser accedida por varias hebras a la vez.