EI1024/MT1024 "Programación Concurrente y Paralela" 2025–26	Entregable para Laboratorio
Nombre y apellidos (1): Maria Pallotti Romero	
Nombre y apellidos (2): Joel Gil Juan	
Tiempo empleado para tareas en casa en formato *h:mm* (obligatorio): 2:30	la05_g

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

Tema 07. Concurrencia en Colecciones de Java

1 Se dispone del siguiente código que define una interfaz gráfica de tiro al blanco.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.lang.reflect.InvocationTargetException;
import java.util.*;
import java.util.concurrent.*;

// ================================================================
public class GUITiroAlBlanco {
  // ================================================================

  // Declaracion de constantes (para tamanyos de ventana).
  static final int  maxVentanaX = 800 ;
  static final int  maxVentanaY = 600 ;

  // Declaracion de variables.
  JFrame             jframe;
  JPanel             jpanel;
  CanvasCampoTiro    cnvCampoTiro;
  JTextField         txfInformacion;
  JTextField         txfVelocidadInicial;
  JTextField         txfAnguloInicial;
  JButton            btnDispara;
  Point              objetivo;
  static int         numIters;
//  MiHebraCalculadora                hebra ; // Ejercicio 4
//  LinkedBlockingQueue<NuevoDisparo> listaD; // Ejercicio 4

  // ----------------------------------------------------------------
  public static void main( String args[] ) {
/*
    try {
      numIters = Integer.parseInt( args[ 0 ] );
      if( numIters <= 0 ) {
        System.err.print( "Uso: [ java programa <numIters> ] donde ( numIters > 0 )" );
        System.exit( -1 );
      }
    } catch( NumberFormatException ex ) {
      numIters = -1;
      System.out.println( "ERROR: Numeros de entrada incorrectos." );
      System.exit( -1 );
    }
```

```
*/
    GUITiroAlBlanco gui = new GUITiroAlBlanco();
    gui.go();
  }

  // ------------------------------------------------------------------------
  public void go() {
    SwingUtilities.invokeLater( new Runnable() {
      public void run() {
        generaGUI();
      }
    } );
  }

  // ------------------------------------------------------------------------
  public void generaGUI() {

    // Declaracion de variables locales.
    JPanel  tablero, informacion, controles, incdec;
    JButton btnVelInc100, btnVelDec100, btnVelInc5, btnVelDec5;
    double  velIni, angIni;
    Font    miFuenteP, miFuenteM, miFuenteG;

    // Crea el JFrame principal.
    jframe = new JFrame( "GUI Tiro Al Blanco " );
    jpanel = ( JPanel ) jframe.getContentPane();
    jpanel.setPreferredSize( new Dimension( maxVentanaX, maxVentanaY ) );
    jpanel.setLayout( new BorderLayout() );

    //
    // Creacion del canvas para el campo de tiro.
    //
    cnvCampoTiro = new CanvasCampoTiro();

    //
    // Creacion del panel de informacion (aciertos, fallos, etc.).
    //
    informacion = new JPanel();
    informacion.setLayout( new FlowLayout() );

    // Crea y anyade el campo de mensajes.
    JLabel labInformacion = new JLabel( "Informacion:" );
    miFuenteM = labInformacion.getFont().deriveFont( Font.PLAIN, 15.0F );
    labInformacion.setFont( miFuenteM );
    informacion.add( labInformacion );

    txfInformacion = new JTextField( 45 );
    txfInformacion.setFont( miFuenteM );
    txfInformacion.setEditable( false );
    txfInformacion.setHorizontalAlignment( JTextField.CENTER );
    informacion.add( txfInformacion );

    //
    // Creacion del panel de controles de disparo.
    //
    controles = new JPanel();
    controles.setLayout( new FlowLayout() );

    // Crea y anyade el control de velocidad inicial.
    JLabel labVelocidadInicial = new JLabel( "Velocidad: " );
    miFuenteG = labVelocidadInicial.getFont().deriveFont( Font.PLAIN, 18.0F );
```

```
labVelocidadInicial.setFont( miFuenteG );
controles.add( labVelocidadInicial );

velIni = 100.0 * Math.round( 50.0 + Math.random() * 10.0 );
txfVelocidadInicial  = new JTextField( String.valueOf( velIni ), 7 );
txfVelocidadInicial.setFont( miFuenteG );
controles.add( txfVelocidadInicial );

// Creacion del minipanel de incrementos/decrementos.
incdec = new JPanel();
incdec.setLayout( new GridLayout( 2, 2 ) );

// Crea y anyade el boton para incrementar la velocidad en 100.
btnVelInc100 = new JButton( "+100" );
miFuenteP = btnVelInc100.getFont().deriveFont( Font.PLAIN, 10.0F );
btnVelInc100.setFont( miFuenteP );
incdec.add( btnVelInc100 );

// Anyade el codigo para procesar la pulsacion del boton "btnVelInc100".
btnVelInc100.addActionListener( new ActionListener() {
  public void actionPerformed( ActionEvent e ) {
    // En las llamadas a getText/setText de objetos graficos aqui no hace
    // falta el invokeLater dado que este codigo lo ejecuta la
    // hebra event-dispatching.
    double vel;
    try {
      vel = Double.parseDouble( txfVelocidadInicial.getText().trim() );
      vel += 100.0;
      txfVelocidadInicial.setText( String.valueOf( vel ) );
    } catch( NumberFormatException ex ) {
      txfInformacion.setText( "ERROR: Numeros incorrectos." );
    }
  }
} );

// Crea y anyade el boton para incrementar la velocidad en 5.
btnVelInc5 = new JButton( "+5" );
btnVelInc5.setFont( miFuenteP );
incdec.add( btnVelInc5 );

// Anyade el codigo para procesar la pulsacion del boton "btnVelInc5".
btnVelInc5.addActionListener( new ActionListener() {
  public void actionPerformed( ActionEvent e ) {
    // En las llamadas a getText/setText de objetos graficos aqui no hace
    // falta el invokeLater dado que este codigo lo ejecuta la
    // hebra event-dispatching.
    double vel;
    try {
      vel = Double.parseDouble( txfVelocidadInicial.getText().trim() );
      vel += 5.0;
      txfVelocidadInicial.setText( String.valueOf( vel ) );
    } catch( NumberFormatException ex ) {
      txfInformacion.setText( "ERROR: Numeros incorrectos." );
    }
  }
} );

// Crea y anyade el boton para decrementar la velocidad en 100.
btnVelDec100 = new JButton( "-100" );
btnVelDec100.setFont( miFuenteP );
incdec.add( btnVelDec100 );
```

```
// Anyade el codigo para procesar la pulsacion del boton "btnVelDec100".
btnVelDec100.addActionListener( new ActionListener() {
  public void actionPerformed( ActionEvent e ) {
    // En las llamadas a getText/setText de objetos graficos aqui no hace
    // falta el invokeLater dado que este codigo lo ejecuta la
    // hebra event-dispatching.
    double vel;
    try {
      vel = Double.parseDouble( txfVelocidadInicial.getText().trim() );
      vel -= 100.0;
      txfVelocidadInicial.setText( String.valueOf( vel ) );
    } catch( NumberFormatException ex ) {
      txfInformacion.setText( "ERROR: Numeros incorrectos." );
    }
  }
} );

// Crea y anyade el boton para decrementar la velocidad en 5.
btnVelDec5 = new JButton( "-5" );
btnVelDec5.setFont( miFuenteP );
incdec.add( btnVelDec5 );

// Anyade el codigo para procesar la pulsacion del boton "btnVelDec5".
btnVelDec5.addActionListener( new ActionListener() {
  public void actionPerformed( ActionEvent e ) {
    // En las llamadas a getText/setText de objetos graficos aqui no hace
    // falta el invokeLater dado que este codigo lo ejecuta la
    // hebra event-dispatching.
    double vel;
    try {
      vel = Double.parseDouble( txfVelocidadInicial.getText().trim() );
      vel -= 5.0;
      txfVelocidadInicial.setText( String.valueOf( vel ) );
    } catch( NumberFormatException ex ) {
      txfInformacion.setText( "ERROR: Numeros incorrectos." );
    }
  }
} );

// Anyade el nuevo minipanel de incrementos/decrementos al panel de control.
controles.add( incdec );

// Crea y anyade un cierto espacio de separacion.
JLabel labSeparacion1 = new JLabel( "     " );
labSeparacion1.setFont( miFuenteG );
controles.add( labSeparacion1 );

// Crea y anyade el control del angulo inicial.
JLabel labAnguloInicial = new JLabel( "angulo: " );
labAnguloInicial.setFont( miFuenteG );
controles.add( labAnguloInicial );

angIni = Math.round( 45.0 + Math.random() * 15.0 );
txfAnguloInicial = new JTextField( String.valueOf( angIni ), 5 );
txfAnguloInicial.setFont( miFuenteG );
controles.add( txfAnguloInicial );

// Crea y anyade un cierto espacio de separacion.
JLabel labSeparacion2 = new JLabel( "     " );
labSeparacion2.setFont( miFuenteG );
```

```
    controles.add( labSeparacion2 );

    // Crea y anyade el boton de disparo.
    btnDispara = new JButton( "Dispara" );
    btnDispara.setFont( miFuenteG );
    controles.add( btnDispara );

    //
    // Creacion del panel de tablero que contiene los minipaneles de
    // informacion y controles.
    //
    tablero = new JPanel();
    tablero.setLayout( new BorderLayout() );
    tablero.add( "Center", informacion );
    tablero.add( "South", controles );

    //
    // Anyade el canvas y el tablero al panel principal.
    //
    jpanel.add( "Center", cnvCampoTiro );
    jpanel.add( "South", tablero );

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

    // Inicializa la posicion del objetivo.
    this.objetivo = generaCoordenadasDeObjetivo();
    System.out.println( "generaGUI. Coordenadas del objetivo: " +
              this.objetivo.x + "," + this.objetivo.y);
    cnvCampoTiro.guardaCoordenadasObjetivo( this.objetivo );

/* ========= INICIO CODIGO A MODIFICAR EN EJERCICIO 2 ========= */
    // Anyade el codigo para procesar la pulsacion del boton "Dispara".
    btnDispara.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent e ) {
        // En las llamadas a getText/setText de objetos graficos aqui no hace
        // falta el invokeLater dado que este codigo lo ejecuta la
        // hebra event-dispatching.
        double vel, ang;
        try {
          vel = Double.parseDouble( txfVelocidadInicial.getText().trim() ) / 100.0;
          ang = Double.parseDouble( txfAnguloInicial.getText().trim() );
          if( ( 0.0 <= ang )&&( ang < 90 )&&( vel > 0 ) ) {
            txfInformacion.setText( "Calculando y dibujando trayectoria..." );
            creaYMueveProyectil( new NuevoDisparo( vel, ang ) );
          } else {
            txfInformacion.setText( "ERROR: Datos incorrectos." );
          }
        } catch( NumberFormatException ex ) {
          txfInformacion.setText( "ERROR: Numeros incorrectos." );
        }
      }
    } );
/* =========   FIN  CODIGO A ANALIZAR EN EJERCICIO 2 ========== */
  }

  // ------------------------------------------------------------------
  Point generaCoordenadasDeObjetivo() {
```

```
    int      maxDimX, maxDimY, distanciaAlBorde, objetivoX, objetivoY;
    double   mitadX, posicionX;

    // Obten las dimensiones del canvas.
    maxDimX = cnvCampoTiro.getWidth();
    maxDimY = cnvCampoTiro.getHeight();

    // Genera una posicion aleatoria en la segunda mitad.
    mitadX  = ( ( double ) ( maxDimX - 1 ) ) / 2.0 ;
    posicionX = Math.round( mitadX + Math.random() * mitadX );

    // Controla que el objetivo no esta muy cerca de los bordes.
    distanciaAlBorde = 50;
    objetivoX = Math.max( distanciaAlBorde,
                  Math.min( maxDimX - distanciaAlBorde, ( int ) posicionX ) );
    objetivoY = 0;

    return new Point( objetivoX, objetivoY );
  }

  //-----------------------------------------------------------------------
  public void creaYMueveProyectil( NuevoDisparo d ) {
    Proyectil  p;
    boolean    impactado;

    p = new Proyectil( d.velocidadInicial, d.anguloInicial, cnvCampoTiro );
    impactado = false;
    while( ! impactado ) {
      // Muestra en pantalla los datos del proyectil p.
      p.imprimeEstadoProyectilEnConsola();

      // Mueve un incremental de tiempo el proyectil p.
      p.mueveUnIncremental();

      // Actualiza en pantalla la posicion del proyectil p.
      p.actualizaDibujoDeProyectil();

      // Comprueba si el proyectil p ha impactado en el suelo.
      impactado = determinaEstadoProyectil( p );

      duermeUnPoco( 2L );
    }
  }

  //-----------------------------------------------------------------------
  boolean determinaEstadoProyectil( Proyectil p ) {
    // Devuelve cierto si el proyectil ha impactado contra el suelo o contra
    // el objetivo.
    boolean  impactado;
    String   mensaje;

    if ( ( p.intPosX == objetivo.x )&&( p.intPosY == objetivo.y ) ) {
      // El proyectil ha acertado el objetivo.
      impactado = true;
      mensaje = " Destruido!!!";
      muestraMensajeEnCampoInformacion( mensaje );

    } else if( ( p.intPosY <= 0 )&&( p.velY < 0.0 ) ) {
      // El proyectil ha impactado contra el suelo, pero no ha acertado.
      impactado = true;
      mensaje = "Has fallado. Esta en " + objetivo.x + ". " +
```

```
        "Has disparado a " + p.intPosX + ".";
        muestraMensajeEnCampoInformacion( mensaje );
      } else {
        // El proyectil continua en vuelo.
        impactado = false;
      }
      return impactado;
    }

    //-----------------------------------------------------------------------
    void muestraMensajeEnCampoInformacion( String mensaje ) {
      // Muestra mensaje en el cuadro de texto de informacion.

/* ========= INICIO CODIGO A ANALIZAR EN EJERCICIO 2.e) ========= */
      String miMensaje = mensaje;
      txfInformacion.setText( miMensaje );
/* =========    FIN  CODIGO A ANALIZAR EN EJERCICIO 2.e) ========== */
    }

    //-----------------------------------------------------------------------
    void duermeUnPoco( long millis ) {
      try {
        Thread.sleep( millis );
      } catch( InterruptedException ex ) {
        ex.printStackTrace();
      }
    }
}

//=============================================================================
class CanvasCampoTiro extends Canvas {
  //===========================================================================

  // Declaracion de constantes.
  static final int  tamProyectil  = 5;
  static final int  tamObjetivoX  = 20;
  static final int  tamObjetivoY  = 30;
  static final int  tamCanyonX  = 40;
  static final int  tamCanyonY  = 40;

  // Declaracion de variables.
  int objetivoX, objetivoY;

  //-----------------------------------------------------------------------
  public void paint( Graphics g ) {

    // Fija el color de fondo.
    this.setBackground( Color.gray );

    // Dibuja el borde.
    g.setColor( Color.black );
    g.drawRect( 0, 0, this.getWidth() - 1, this.getHeight() - 1 );

    // Dibuja el canyon y el objetivo.
    dibujaCanyon( 0, 0 );
    dibujaObjetivo( objetivoX, objetivoY );
  }

  //-----------------------------------------------------------------------
  public void dibujaProyectil( int x, int y, int xOld, int yOld ) {
    Graphics g = this.getGraphics();
```

```
      // Borra posicion anterior.
      g.setColor( Color.white );
      g.fillOval( coorX( xOld ), coorY( yOld ), tamProyectil, tamProyectil );
      // Dibuja posicion nueva.
      g.setColor( Color.red );
      g.fillOval( coorX( x ), coorY( y ), tamProyectil, tamProyectil );
    }

    // ---------------------------------------------------------------------
    public void dibujaCanyon( int x, int y ) {
      Graphics g = this.getGraphics();
      g.setColor( Color.green );
      g.fillOval( coorX( x ) - tamCanyonX / 2, coorY( y ) - tamCanyonY / 2,
            tamCanyonX, tamCanyonY );
    }

    // ---------------------------------------------------------------------
    public void dibujaObjetivo( int x, int y ) {
      Graphics g = this.getGraphics();
      g.setColor( Color.yellow );
      g.fillRect( coorX( x ) - tamObjetivoX / 2, coorY( y ) - tamObjetivoY / 2,
            tamObjetivoX, tamObjetivoY );
    }

    // ---------------------------------------------------------------------
    int coorX( int x ) {
      return x;
    }

    // ---------------------------------------------------------------------
    int coorY( int y ) {
      return ( this.getHeight() - 1 - y );
    }

    // ---------------------------------------------------------------------
    void guardaCoordenadasObjetivo( Point objetivo ) {
      this.objetivoX = objetivo.x;
      this.objetivoY = objetivo.y;
    }
  }

  // =======================================================================
  class NuevoDisparo {
    // =====================================================================

    final double velocidadInicial, anguloInicial;

    // ---------------------------------------------------------------------
    public NuevoDisparo( double velocidadInicial, double anguloInicial ) {
      this.velocidadInicial = velocidadInicial;
      this.anguloInicial  = anguloInicial;
    }
  }

  // =======================================================================
  class Proyectil {
    // =====================================================================

    // Declaracion de constantes.
    static final double  GRAVITY = 9.8;
    static final double  TO_RAD  = ( 2.0 * Math.PI ) / 360.0;
```

```
  static final double  DELTA_T = 5.0E-3;

  // Declaracion de variables.
  CanvasCampoTiro cnvCampoTiro;
/* ========= INICIO CODIGO A ANALIZAR EN EJERCICIO 2.c) y 5.a) ========= */
  // Posiciones, angulo y velocidades con precision doble.
  double       posX, posY;
  double       anguloRad, velX, velY;
  // Posiciones exactas enteras.
  int          intPosX, intPosY, intPosXOld, intPosYOld;
/* =========   FIN  CODIGO A ANALIZAR EN EJERCICIO 2.c) y 5.a) ========= */

  //-------------------------------------------------------------------------
  Proyectil( double velocidadInicial, double anguloInicial,
        CanvasCampoTiro cnvCampoTiro ) {
    this.posX      = 0.0;
    this.posY      = 0.0;
    this.anguloRad  = anguloInicial * TO_RAD;
    this.velX      = Math.cos( anguloRad ) * velocidadInicial;
    this.velY      = Math.sin( anguloRad ) * velocidadInicial;
    this.cnvCampoTiro = cnvCampoTiro;
  }

  //-------------------------------------------------------------------------
  void mueveUnIncremental() {
    // Actualiza la posicion y la velocidad.
    this.posX += this.velX * DELTA_T;
    this.posY += this.velY * DELTA_T;
    //// this.velX = this.velX;  Esta velocidad no cambia.
    this.velY -= GRAVITY * DELTA_T;

    // Guarda la anterior posicion entera.
    this.intPosXOld = intPosX;
    this.intPosYOld = intPosY;

    // Calcula la nueva posicion entera.
    this.intPosX = ( int ) posX;
    this.intPosY = ( int ) posY;
  }

  //-------------------------------------------------------------------------
  void imprimeEstadoProyectilEnConsola() {
    System.out.format( "  Pos:( %6.2f %6.2f )" +
              " Vel:( %6.2f %6.2f )" + " IntPos:( %4d %4d )%n",
              this.posX, this.posY,
              this.velX, this.velY, this.intPosX, this.intPosY );
  }

  //-------------------------------------------------------------------------
  public void actualizaDibujoDeProyectil() {
    // Dibuja la nueva posicion del proyectil solo si la nueva posicion es
    // distinta de la anterior.
    if( ( this.intPosX != this.intPosXOld )||
       ( this.intPosY != this.intPosYOld ) ) {
/* ========= INICIO CODIGO A ANALIZAR EN EJERCICIO 2.d) ========= */
      cnvCampoTiro.dibujaProyectil( intPosX, intPosY,
                        intPosXOld, intPosYOld );
/* =========   FIN  CODIGO A ANALIZAR EN EJERCICIO 2.d) ========= */
    }
  }
}
```

No revises el código, únicamente compílalo y prueba su funcionamiento.

Cuando un proyectil se encuentra en movimiento, ¿responde la aplicación a alguna acción, como por ejemplo intentar cambiar la velocidad o el ángulo, o realizar otro disparo? ¿Por qué ocurre?

No, la aplicación no es responsiva. No hace nada hasta que el disparo se para de calcular, luego lo ejecuta todo.

2 El objetivo de esta práctica es modificar la aplicación para que la interfaz gráfica sea más interactiva. En la primera implementación, cada disparo será movido por una hebra nueva.

Para ello, debes definir una nueva clase de hebras (subclase de la clase `Thread`) que se podría denominar, por ejemplo, `MiHebraCalculadoraUnDisparo`. El constructor de esta clase debe recibir cuatro argumentos: **el canvas** (`cnvCampoTiro`), **el cuadro de texto de mensajes** (`txfInformacion`), **el objetivo** (`objetivo`) y **el nuevo disparo**. Por su parte, el código del método `run` de la hebra debe **crear un proyectil** a partir del disparo recibido y moverlo hasta que alcance el suelo, exactamente lo que hace el método `creaYMueveProyectil`.

De este modo, cada vez que el usuario pulse el botón de disparo (`btnDispara`), la hebra *event-dispatching* debe comprobar, en el `ActionListener` asociado, si los parámetros que ha definido el usuario son correctos, creando un nuevo disparo (`d`) si lo fueran. (Busca EJERCICIO 2 en el código) Seguidamente **debe crear un objeto (`t`)** de la clase `MiHebraCalculadoraUnDisparo` que se encargue de mover el proyectil asociado al disparo.

Ten en cuenta que para completar estos cambios, es muy recomendale "mover" métodos desde `GUITiroAlBlanco` a `MiHebraCalculadoraUnDisparo`, como `determinaEstadoProyectil`, `muestraMensajeEnCampoInformacion` o `duermeUnPoco`.

En la descripción anterior, aparecen situaciones que pueden generar problemas de visibilidad y/o de atomicidad, cuya resolución puede modificar tu implementación. Con este fin, se va a analizar el código, indicar qué líneas pueden ser problemáticas, y actuar en consecuencia. El código incluye tres áreas que deberían ser analizadas.

Seguidamente se plantean cuestiones que ayudan a detectar y resolver estos problemas:

a) Cuando se crea y arranca la hebra auxilar. ¿Se pueden producir problemas de visibilidad y/o de atomicidad? Razona tu respuesta.

No, no se producen problemas de visibilidad ya que al arrancar una hebra hay sincronización explícita. Tampoco hay problemas de atomicidad, ya que los valores los gestiona una hebra.

b) Cuando se crea el proyectil. ¿Se pueden producir problemas de visibilidad y/o de atomicidad? Razona tu respuesta, pensando dónde se crea y quién lo maneja.

Siguen sin haber ni problemas de visibilidad ni de atomicidad, ya que cada hebra es la que gestiona su propio disparo.

c) Al mover un proyectil. ¿Se pueden producir problemas de visibilidad y/o de atomicidad? Razona tu respuesta, pensando dónde se crea y quién lo maneja. (Buscar 2.c en el código)

Al mover un proyectil tampoco, ya que se crea uno nuevo para cada hebra.

d) Cuando se actualiza la trayectoria del proyectil (método `actualizaDibujoDeProyectil`). ¿Cuál es el objeto gráfico? ¿Quién debe actualizarlo y cómo puede conseguirlo? ¿Se pueden producir problemas de visibilidad y/o de atomicidad? ¿Cómo se podrían resolver? ¿Sería útil el uso del modificador `final`? Razona tu respuesta. (Buscar 2.d en el código)

El objeto es cnvCampoTiro. El que lo debe analizar es la hebra event-dispatching. Puede que al pasarle los datos a la hebra se generen problemas de atomicidad. Se puede resolver añadiendo el final a las variables para que no cambien.

e) Cuando se imprime el estado de un proyectil en el cuadro de texto de mensajes (método `muestraMensajeEnCampoInformacion`). ¿Cuál es el objeto gráfico? ¿Quién debe actualizarlo y cómo puede conseguirlo? ¿Se producen problemas de visibilidad y/o de atomicidad? ¿Cómo se podrían resolver? ¿Sería útil el uso del modificador `final`? Razona tu respuesta. (Buscar 2.e en el código)

El objeto es txfInformacion , el objeto gráfico es el cuadro de texto debajo del gráfico. Lo debe actualizar la hebra event-dispatching, usando un invokeLater. Aquí también se pueden producir problemas de atomicidad al pasarle los datos, se puede resolver con el operador final.

Escribe a continuación la parte de tu código que realiza la tarea descrita: la definición de la clase `MiHebraCalculadoraUnDisparo` y los cambios a introducir en el código del método `generaGUI`.

3 ¿Piensas que es realista la implementación? ¿Qué pasaría si varias hebras estuviesen moviendo diferentes proyectiles y una de ellas perdiera la CPU?

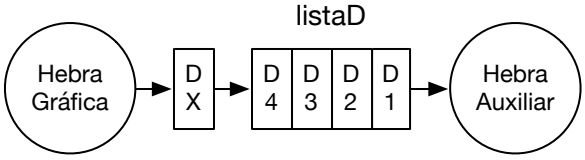
Si una pierde la CPU mientras mueve los puntos, esto causaría que al volver, los valores estuviesen cambiados y se produjese un salto en la línea.

4 El objetivo de esta segunda implementación es modificar la aplicación para que la interfaz gráfica sea más interactiva y también más realista. No olvides crear una **copia del código original** (GUITiroAlBlancoUnaHebra.java).

Para lograrlo, todos los proyectiles deben ser movidos por **una única hebra auxiliar** (clase `MiHebraCalculadora`) que será creada junto con el interfaz gráfico. Esta hebra se debe bloquear mientras no haya ningún proyectil en el aire, es decir, no se puede emplear espera activa.

Con este objetivo, se pretende que la hebra auxiliar trabaje con dos colecciones.

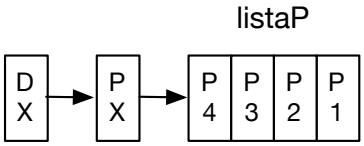
Una **primera colección**, denominada lista de disparos (`listaD`), es utilizada por la hebra gráfica para comunicar a la hebra auxiliar los datos de los nuevos disparos (objetos de la clase `NuevoDisparo`) que el usuario ha producido. Esta colección debe ser *thread-safe* porque tanto la hebra gráfica como la hebra auxiliar accederán a la información que contiene.



Como la hebra auxiliar puede bloquearse a la espera de nuevos disparos (si no hay ningún proyectil en el aire), se puede emplear un objeto de la clase `LinkedBlockingQueue`. ¿Qué métodos de la clase `LinkedBlockingQueue` permiten realizar una inserción bloqueante y una extracción bloqueante en un objeto de esta clase?

.put() y .take() respectivamente

La **segunda colección**, denominada lista de proyectiles (`listaP`), debe ser local a la hebra auxiliar, por lo que no necesita ser *thread-safe*. En esta lista, la hebra auxiliar guarda todos los proyectiles que están en el aire, y debe eliminar un proyectil cuando llega al suelo. Periódicamente, la hebra auxiliar debe consultar la lista de disparos para comprobar si no está vacía, y en tal caso, coger los disparos, crear los proyectiles asociados e insertarlos en su lista local. Esta segunda colección debe ser de una clase que permita eliminar cualquier componente de la colección, puesto que no se conoce de antemano la posición de los proyectiles que llegan al suelo.



Una opción sería emplear, la clase `ArrayList`. ¿Qué métodos de la clase `ArrayList` permiten realizar una inserción y un borrado en un objeto de esta clase?

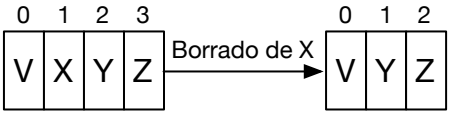
.add() y .remove() respectivamente

A continuación se describe con más detalle el proceso iterativo que debe realizar la hebra auxiliar:

1. La hebra auxiliar, antes de proceder a mover todos los proyectiles que se encuentran en vuelo, debe comprobar si la lista de disparos no está vacía.

   Si la hebra gráfica ha dejado uno o varios nuevos disparos en la `listaD`, la hebra auxiliar debe extraerlos, crear los proyectiles e insertarlos en su lista local de proyectiles (`listaP`).

   Si la hebra gráfica no ha dejado trabajo y `listaP` está vacía, la hebra auxiliar no puede hacer nada. En tal caso, la hebra debe bloquearse a la espera de recibir nuevos disparos en `listaD`, evitando realizar una espera activa, para lo cual debe utilizar el método adecuado.
2. Tras vaciar la lista de disparos, y en el caso que existan proyectiles en el aire, la hebra auxiliar debe mover **todos los proyectiles** que están en `listaP` como si hubiese transcurrido **un incremental de tiempo**.
3. Si algún proyectil de los que están en vuelo alcanza el suelo (y estalla), la hebra auxiliar debe eliminarlo de `listaP` utilizando el procedimiento adecuado.



4. Repetir los pasos anteriores hasta que la interfaz gráfica termine. Para ello, las acciones anteriores deben estar en un bucle infinito.

A continuación se muestra el algoritmo del cuerpo de la hebra en pseudo-código:

```
// Bucle infinito en el cuerpo de la hebra.
while( true )
  // Bucle para coger todos los nuevos disparos dejados por la hebra grafica.
  while( ( listaD no este vacia ) || ( listaP este vacia ) ) {
    Tomar un nuevo disparo de listaD (d), bloqueandose si no hubiera.
    Crear un nuevo proyectil (p) a partir del nuevo disparo (d).
    Anyadir el nuevo proyectil (p) a listaP.
  }
  // Procesado de la lista local de proyectiles.
  for( todos los proyectiles de listaP) {
    Sea p el proyectil actual de listaP.
    Muestra en pantalla los datos del proyectil p.
    Mueve un incremental de tiempo  el proyectil p.
    Actualiza en pantalla la posicion del proyectil p.
    Comprueba si el proyectil p ha impactado en el suelo.
    if( El proyectil p ha impactado sobre el suelo ) {
      Eliminar el proyectil p de la listaP.
    }
  }
}
```

Fíjate que la mayoría de las acciones en el cuerpo del bucle `for` coinciden con las que aparecían en el método `creaYMueveProyectil`.

IMPORTANTE: No olvides crear la `hebra` y la `listaD`, que ya están definidas en el código. Para la `hebra`, ten en cuenta que los objetos que aparecen en su constructor ya deben estar creados.

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

Una vez terminado el código, realiza las siguientes comprobaciones:

1. Comprueba que el nuevo código mueve simultáneamente varios disparos en el aire.
2. Comprueba que no hay espera activa: Ejecuta el comando `top` en Linux o el comando similar en *Windows*. Comprueba la carga mientras no hay proyectiles en el aire. Comprueba la carga mientras hay varios proyectiles en el aire.
3. Comprueba que la hebra auxiliar no acceda a ningún método de los dos objetos gráficos (excepto si estos aparecen dentro de: `invokeAndWait`, `invokeLater`, etc.).
4. Comprueba que la hebra auxiliar no acceda a ningún dato modificado por la hebra gráfica que no esté protegido con `synchronized` o con `volatile`, o que no sea `final`.
5. Comprueba que la hebra gráfica no acceda a ningún dato modificado por la hebra auxiliar que no esté protegido con `synchronized` o con `volatile`, o que no sea `final`.

Escribe a continuación la parte de tu código que realiza tal tarea: la definición de la clase `MiHebraCalculadora` y los cambios a introducir en el código del método `generaGUI`.

5 El objetivo de la última implementación es acelerar el funcionamiento de la aplicación. Crea una **copia del código del ejercicio anterior** (GUITiroAlBlancoUnaHebraYVirtuales.java). A continuación descomenta las líneas que aparecen en las primeras líneas del método `main`, que obliga al programa a recibir un parámetro.

Habréis podido comprobar, cómo se ralentiza la aplicación cuando aumenta el número de proyectiles en vuelo, ya que una única hebra auxiliar es la encargada de realizar todos los cálculos asociados al procesado de la lista de proyectiles. Existen diferentes alternativas para resolver este problema, pero de las diferentes posibles soluciones se propone el uso de hebras virtuales.

La idea es que **las encargadas de mover los proyectiles sean hebras virtuales**. Para ello, se crea una hebra virtual en cada una de las iteraciones del bucle `for` que mueve los proyectiles en la hebra auxiliar, que se encargará del correspondiente proyectil de `listaP`. Así, una hebra virtual debería mostrar los datos del proyectil, moverlo y dibujarlo, mientras que la hebra auxiliar únicamente se encargaría de comprobar si el proyectil ha impactado o no, **tras comprobar que la hebra virtual ha finalizado**, por si hubiese que eliminarlo de `listaP`.

A continuación aparece el pseudo-código de la gestión las hebras virtuales en la hebra auxiliar.

```
// Procesado de la lista local de proyectiles.
Crea vector de hebras virtuales de dimension igual que tamanyo de listaP.
for( todos los proyectiles de listaP) {
  Sea p el proyectil actual de listaP.
  Crea y arranca una hebra virtual que gestione el proyectil p.
}
Espera la finalizacion de las hebras virtuales antes de comprobar
  que proyectiles de lista P han impactado en el suelo para eliminarlos.
```

Para limitar el número de hebras virtuales creadas, se puede permitir que una **hebra virtual realice un número de incrementales sobre un proyectil**, añadiendo un parámetro al constructor que fuese el número máximo de iteraciones (`numIters`). Este parámetro es un número máximo, porque la hebra virtual también debería comprobar si el proyectil ha impactado, en cuyo caso finalizaría prematuramente su ejecución. La implementación más sencilla sería añadir un bucle `for` al cuerpo de la hebra virtual (de 0 a *numIters*), en el que se ejecutase un `break` cuando el proyectil hubiese impactado.

A continuación aparece el pseudo-código correspondiente al cuerpo de las hebras virtuales.

```
// CUERPO DE LA HEBRA VIRTUAL
  Sea p el proyectil gestionado por la hebra virtual.
  for(int i=0; i<numIters; i++) {
    Muestra en pantalla los datos del proyectil p.
    Mueve un incremental de tiempo  el proyectil p.
    Actualiza en pantalla la posicion del proyectil p.
    Comprueba si el proyectil p ha impactado en el suelo.
    if( El proyectil p ha impactado sobre el suelo )
      break;
  }
```

Seguidamente se plantean cuestiones que ayudan a detectar y resolver estos problemas:

a) Cuando una hebra virtual mueve un proyectil. ¿Se pueden producir problemas de visibilidad y/o de atomicidad? (Buscar 5.a en el código) Razona tu respuesta, pensando dónde se crea el proyectil y quién lo maneja.

Siguen sin producirse problemas, porque cada proyectil es independiente encargándose cada hebra virtual de su propio proyectil.

b) Ejecuta la aplicación, con difentes valores de `numIters` ($= 1, 10, 50, 100$) ¿Han mejorado las prestaciones de la aplicación? Razona tu respuesta.

Cuando hay más numIters la aplicación funciona bastante mejor, aunque llega un punto que ya no se nota la diferencia (habría que crear muchas hebras para notarlo).

Escribe a continuación la parte de tu código que realiza esta última implementación: la definición de la clase `MiHebraCalculadora` y la definición de la clase `MiHebraVirtual`.