

DESCRIPCION: En este documento se explica el concepto de hilos de ejecución en el lenguaje de programación Java, considerando los fundamentos y característica teóricas de este concepto, como también los mecanismos soportados por Java para el tratamiento de hilos; especialmente en lo relativo a las clases e interfaces que este lenguaje dispone para permitirnos implementar procesos en segundo plano en nuestras aplicaciones. En este orden de ideas, se estudian los mecanismos de hilos de Java, considerando los métodos principales de las clases e interfaces más comúnmente usadas para aplicar esta técnica de programación. Como ejemplo práctico, se desarrolla una aplicación de interfaz gráfica de usuario (GUI) hecha en el IDE *NetBeans*, implementando igualmente un diseño UML de clases acorde a un problema planteado, para cuya solución es necesario el tratamiento de hilos.

OBJETIVOS:

- Comprender las características e importancia de los hilos de ejecución como técnica para programar procesos paralelos.
- Identificar mecanismos, clases e interfaces disponibles en Java para el tratamiento de hilos, con sus respectivos métodos y relaciones.
- Diseñar e implementar en Java las clases necesarias para mover mediante hilos sobre la ventana un conjunto de etiquetas, cuya apariencia debe ser circular y aplicando funciones matemáticas para definir la trayectoria de movimiento de las figuras.

PALABRAS CLAVES: Hilos de ejecución, procesos en segundo plano, clases e interface Java para hilos, clase *Thread*, interface *Runnable*, colecciones de hilos en *ArrayList*, aplicaciones de ventana en Java, movimiento de figuras en la ventana, herencia desde componentes swing, sobre escritura del método *paintComponent*.

1. Hilos de ejecución.

Los hilos de ejecución nos permiten incluir en nuestros programas la capacidad de ejecutar dos o más tareas (llamadas procesos) de manera simultánea; es decir, en paralelo, a lo cual también se le conoce como ejecución asincrónica. El uso de esta técnica de ejecución en nuestras aplicaciones, nos ofrece como ventaja que el usuario puede seguir interactuando con el programa después de haber iniciado ciertos procesos, funcionalidades o tareas, aun cuando éstas no hayan terminado completamente; permitiendo que la aplicación siga respondiendo a nuevas solicitudes de los usuarios, lo cual es importante especialmente en aquellos casos que el programa debe realizar procesamientos intensos y/o que consumen un buen tiempo de ejecución; de modo que no se genere en el usuario la impresión de que el programa está bloqueado o no responde, cuando lo que sucede realmente es que no ha terminado de realizar las tareas solicitadas. En este sentido, considere como ejemplo tareas en las que el programa debe ejecutar ciclos que eventualmente pueden tardar, ya sea porque tienen muchas iteraciones o por que tienen varios niveles de anidamiento.

El aplicar hilos en este tipo de procesamiento, permite que el programa no se “pase” ante el usuario mientras tales procesos no hayan finalizado; facilitando así por ejemplo, que el usuario vea el progreso de la tarea o que utilice otras prestaciones de nuestra aplicación; incluyendo también la posibilidad de cancelar o abortar la tarea que ejecutamos en paralelo el hilo. No obstante, en este punto es importante resaltar que el concepto de ejecución paralela real (ejecutar dos o más tareas o procesos de forma simultánea) es posible siempre que tengamos más de un procesador en nuestra máquina; pues en caso contrario el sistema operativo conmuta (comparte) procesador con cada proceso de cada programa en ejecución, haciéndolo tan rápido que da la impresión que los procesos corren simultáneamente. Algo así por ejemplo vemos cuando descargamos varios archivos de forma simultánea.

Por otra parte para implementar hilos de ejecución en Java tenemos varias alternativas, una de ellas es heredando de la clase **Thread**, de manera que la clase hija sobre escriba el método *run*, implementándolo para que ejecute las tareas que se deben realizar en segundo plano; para tal efecto, tome en cuenta que este método no retorna un resultado (es **void**) y tampoco recibe o requiere parámetros. Así mismo, la clase **Thread** hereda el

método *run* de la interface **Runnable**, toda vez que implementa (realiza) a dicha interface; y ya que la clase **Thread** no es abstracta, concluimos que incluye o tienen una implementación propia para el método *run*; sin embargo, es completamente válido sobre escribir este método en cualquier descendiente de **Thread**. No obstante, es posible usar la clase **Thread** sin crear un descendiente de ella; es decir, empleando directamente una instancia de esta clase; en cuyo caso, podemos hacer uso de uno de los constructores de la clase **Thread**, al cual se le pasa por parámetro una instancia de interface **Runnable**, que como es lógico, se trata de una instancia de una clase cualquiera que implemente (realice) a dicha interface. Una ventaja derivada de esta opción, es que en las últimas versiones de Java la interface **Runnable** es funcional, por lo que en consecuencia nos ofrece además la posibilidad de implementarla usando expresiones lambdas; dándonos así más alternativas y mayor flexibilidad a la hora de codificar o indicar cuál es la tarea o métodos que deseamos ejecutar en segundo plano; es decir, concretizar que código queremos que se ejecuten de forma asincrónica. En cuanto a los constructores, la clase **Thread** tiene entre otros los siguientes:

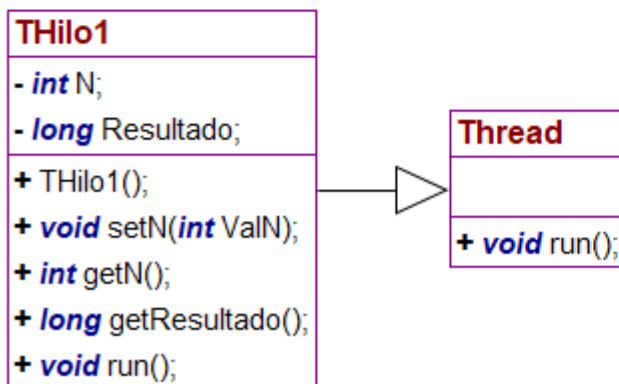
- **Thread()**: Que es el constructor por defecto y sin parámetros.
- **Thread(String name)**: Crea una nueva instancia considerando por parámetro el nombre para el hilo.
- **Thread(Runnable target)**: Crea una nueva instancia de la clase en base a una instancia de clase (target) que implemente a la interface **Runnable**.
- **Thread(Runnable target, String name)**: Este constructor es una combinación de los dos anteriores.

Por otra parte, dentro de los métodos públicos de la clase **Thread** tenemos entre otros los siguientes:

- **long getId()**: Devuelve el identificador un del hilo.
- **String getName()**: Retorna el nombre del hilo.
- **int getPriority()**: Devuelve la prioridad del hilo.
- **Thread.State getState()**: Retorna el estado del hilo como un miembro de la enumeración **Thread.State** (definida internamente en la clase **Thread**).

- **boolean** *isAlive*(): Comprueba si este el hilo está vivo (activo o en ejecución).
- **void** *join*(): Detiene la ejecución del hilo llamante y lo hace esperar a que este hilo termine.
- **void** *join* (**long** millis): Similar al caso anterior, pero solo detiene la ejecución del hilo llamante un máximo de tiempo en milisegundos o antes si este hilo termina primero.
- **void** *setName* (**String** name): Asigna o cambia el nombre del hilo.
- **void** *setPriority* (**int** newPriority): cambia la prioridad del hilo.
- **static void** *sleep*(**long** millis): Duerme o detiene temporalmente al ejecución de hilo, por un tiempo dado en milisegundos.
- **void** *start*(): Inicia la ejecución de las tareas delegadas al hilo.

En este punto, es importante tomar en cuenta que el prototipo del método *run* no retorna un resultado ni tampoco requiere parámetros, lo que no es compatible en casos cuando, por ejemplo, requerimos que se retorne un resultado después que el hilo termine, como también cuando necesitamos pasarle parámetros al código ejecutado por el hilo. No obstante, ambos casos lo podemos resolver sin mayor esfuerzo; así por ejemplo, el resultado de la ejecución el hilo puede guardarse en un atributo de la instancia hija de *Thread*; así mismo, los parámetros requeridos bien pueden ser sustituidos por atributos en la clase hija. Al respecto, consideremos que necesitamos hallar la suma de los N primeros números naturales, haciendo uso de un hilo, para lo cual diseñamos la clase **THilo1** que hereda de la clase **Thread**.



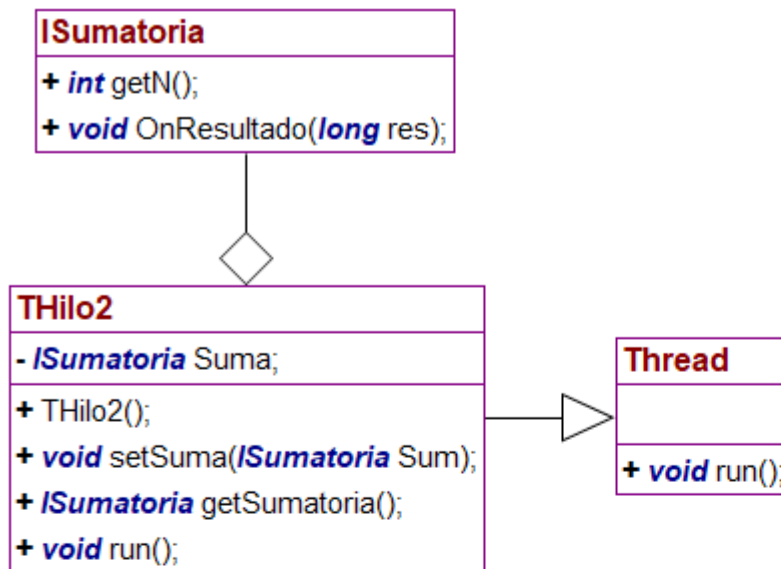
Según las especificaciones anteriores, una implementación para esta clase puede ser la siguiente:


```

1 public class THilo1 extends Thread {
2
3     private int N;
4     private long Resultado;
5
6     public THilo1(){
7         super();
8         N=0;
9         Resultado=0;
10    }
11
12    public void setN(int ValN){
13        N=ValN;
14    }
15
16    public int getN(){
17        return N;
18    }
19
20    public long getResultado(){
21        return Resultado;
22    }
23
24    @Override
25    public void run(){
26        int i;
27        Resultado=0;
28        for(i=1;i<=N;i++){
29            Resultado=Resultado + i;
30        }
31    }
32
33 }

```

Sin embargo, para estos mismos efectos, igualmente podemos tener como atributo una referencia a una interface, a cuyos métodos se les pase como parámetro los resultados obtenidos en la terminación del hilo; es más, esta misma interface podría incluir métodos que retorne los parámetros que requiere el hilo para su ejecución. De esta manera, en la interface **ISumatoria** que se muestra más abajo, tenemos que el método *getN()* retorna un valor que sustituye a un hipotético parámetro N, mientras que el método *OnResultado* –definido al estilo de un evento– se utiliza para notificar a la instancia de la interface cual fue el resultado obtenido por el hilo una vez finalice su cálculo y, consecuentemente su ejecución. El diseño UML para este último caso es el siguiente:



El código para la interface **ISumatoria** es el siguiente:

```

1 interface ISumatoria(){
2
3     public int getN();
4     public void OnResultado(long res);
5
6 }
  
```

Por su parte el código para la clase **THilo2** es como sigue:

```

1 public class THilo2 extends Thread {
2
3     private ISumatoria Suma;
4
5     public THilo2(){
6         super();
7         Suma=null;
8     }
9
10    public void setSuma(ISumatoria Sum){
11        Suma=Sum;
12    }
13
14    public ISumatoria getSumatoria(){
15        return Suma;
16    }
  
```

```

17
18 @Override
19 public void run(){
20     int i, N;
21     long res;
22     res=0;
23     N=Suma.getN();//Solicitamos el valor del parametro N
24     for(i=1;i<=N;i++){
25         res=res + i;
26     }
27     Suma.OnResultado(res);//al terminar notificamos el resultado
28 }
29
30 }

```

Otra opción que podemos usar para trabajar con hilos en java es la clase **Timer** del paquete **javax.swing**, la cual se especializa en ejecutar un código asignado cada intervalo de tiempo de forma repetitiva o cíclica. Así los objetos de esta clase se construyen especificando tanto un parámetro de retraso o intervalo de tiempo, como también una instancia de la interface **ActionListener**. El parámetro de demora se interpreta en milisegundos y se usa para establecer la demora inicial y demora entre la ejecución del método (evento) **actionPerformed** de la interface. Una vez que se ha iniciado el temporizador, se espera el retraso inicial antes de disparar por vez primera el evento de los oyentes o escuchas de la interface. Después de este primer evento, se continúa disparando el evento cada vez que transcurre o se consume el intervalo de tiempo de espera o retraso entre eventos, hasta que el **Timer** sea detenido.

En este sentido, tome en cuenta que todos las instancia de **Timer** realizan o consumen su tiempo de espera utilizando un solo hilo compartido, el cual a su vez es creado por el primer objeto **Timer** que se ejecuta; pero los manejadores de eventos de las acciones a ejecutar por los temporizadores se ejecutan en otro hilo, específicamente en el hilo de envío o de notificación de eventos. Esto significa que los manejadores de eventos para los **Timer** pueden realizar operaciones de manera segura en los componentes *Swing*. Sin embargo, también significa que los controladores deben ejecutarse rápidamente para que la GUI responda y no quede pasmada. En cuanto a los métodos definidos en la clase **Timer**, a continuación se describen algunos de estos métodos.

- **Timer(int delay, ActionListener listener):** Este es el constructor, al cual se le pasa por parámetro el tiempo en milisegundos para el espera inicial y el intervalo o periodo de ejecución del evento. El segundo parámetro es una instancia de la interface **ActionListener**, de modo que el **Timer** ejecuta de esta el método **actionPerformed** cada periodo de tiempo indicado por el primer parámetro.
- **void addActionListener(ActionListener listener):** Agrega un oyente a la acción o evento del temporizador.
- **int getDelay():** Retorna el retraso, en milisegundos, entre las ejecuciones de los eventos.
- **int getInitialDelay():** Devuelve el retraso inicial del temporizador.
- **boolean isRepeats():** Devuelve true si el temporizador debe ejecutar el evento cíclicamente.
- **boolean isRunning():** Retorna true si el temporizador se está ejecutando.
- **void restart():** Reinicia el temporizador, cancela los eventos pendientes y hace que se reactive con su retraso inicial.
- **void setDelay(int delay):** Establece el tiempo de espera en milisegundos entre cada ejecución del evento del temporizador.
- **void setRepeats(boolean flag):** Si *flag* es **false**, indica al temporizador que ejecuta una sola vez el evento que tiene asociado.
- **void start():** Inicia el temporizador a disparar o ejecutar los eventos de sus oyentes.
- **void stop():** Detiene la ejecución del temporizador.

Existe otra clase que también se llama **Timer**, pero que se encuentra en el paquete **java.util** que es una clase de utilidad pensada para programar tareas que se ejecutarán como sub procesos en segundo plano. Estas tareas pueden planificarse para una ejecución única o para una ejecución repetida a intervalos regulares; es decir, en forma cíclica. Ahora bien, para cada instancia de la clase **Timer** hay un único subproceso en segundo plano que se utiliza para ejecutar todas las tareas del **Timer** secuencialmente; por lo tanto, es importante que las tareas del **Timer** deben completarse rápidamente, pues si una tarea de un **Timer** tarda demasiado en completarse, "acapara" el hilo de

ejecución de la tarea del **Timer**. Esto puede, a su vez, retrasar la ejecución de tareas posteriores, que pueden "agruparse" y ejecutarse sucesivamente de manera rápida solo cuando la tarea infractora finalice. Esta clase representa a un hilo seguro, ya que múltiples subprocesos (hilos) pueden compartir una sola instancia de la clase **Timer** sin la necesidad de sincronización externa. Entre algunos de los métodos de esta clase tenemos los siguientes:

- **Timer()**: Constructor por defecto (sin parámetros).
- **Timer(String name)**: Crea un nuevo temporizador cuyo hilo asociado tiene el nombre especificado.
- **int purge()**: Elimina todas las tareas canceladas de la cola de tareas del **Timer**.
- **void schedule(TimerTask task, Date time)**: Programa la tarea especificada para su ejecución a la fecha y hora especificada.
- **void schedule(TimerTask task, Date firstTime, long period)**: Planifica la tarea pasada por parámetro, para que sea ejecutada repetidamente cada periodo de tiempo indicado, pero comenzando a la fecha y hora especificada.
- **void schedule(TimerTask task, long delay)**: Programa la tarea pasada por parámetro para su ejecución después que transcurra el tipo del retraso especificado.
- **void schedule(TimerTask task, long delay, long period)**: Planifica la tarea especificada para su ejecución repetida cada periodo de tiempo y después del retraso especificado.
- **void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)**: Programa la tarea especificada para su ejecución repetida en un periodo de tiempo fijo, comenzando a la hora especificada.
- **void scheduleAtFixedRate(TimerTask task, long delay, long period)**: Planifica la tarea especificada para su ejecución repetida a una velocidad fija, comenzando después del retraso especificado.
- **void cancel()**: Termina la ejecución del temporizador, descartando cualquier tarea programada o planificada actualmente.

En la mayoría de los métodos anteriores, vemos que se pasa como parámetro una instancia de la clase **TimerTask**, pues bien, esta es una clase que representa una tarea que puede planificarse para ser ejecutada por un temporizador (*Timer*), tanto para una ejecución única como para una ejecución repetida o cíclica. Esta clase además hereda el método `run` de la interface **Runnable**, pero no lo implementa; por lo tanto, se trata de una clase abstracta, cuyo constructor además es protegido (**protected**). Los métodos de esta clase comprenden los siguientes:

- **boolean cancel()**: Cancela la tarea que está ejecutando el temporizador.
- **abstract void run()**: Implemente este método en un descendiente para definir el código o la tarea que el temporizador (*Timer*) debe realizar en segundo plano.
- **long scheduledExecutionTime()**: Retorna el tiempo de ejecución programado de la ejecución real más reciente de esta tarea.

2. Presentación ejemplo hilos.

Para este ejemplo de hilos, vamos a considerar mover o desplazar por la ventana figuras circulares, que serán construidas mediante etiquetas (**JLabel**), de manera que el movimiento de cada figura estará a cargo de un hilo; uno diferente para cada figura. Así mismo, las figuras se moverán bajo una trayectoria definida por algunas funciones matemáticas; para lo cual, se considerará un intervalo del dominio de la función que será ingresado por el usuario, considerando igualmente que ingresa para la función el valor mínimo y máximo de la misma; que como es lógico, se corresponde con intervalo del rango de la función en el intervalo del dominio indicado. En este sentido, es importante escalar o ajustar los valores matemáticos de **X** e **Y** para la función en cuestión, de modo que se muestren en el área visible de la ventana; por lo tanto, que se adapten en relación al ancho y alto de la ventana, siendo importante el tratamiento de valores negativos; pues como es bien sabido las coordenadas negativas están por fuera del área de visibilidad de la ventana.

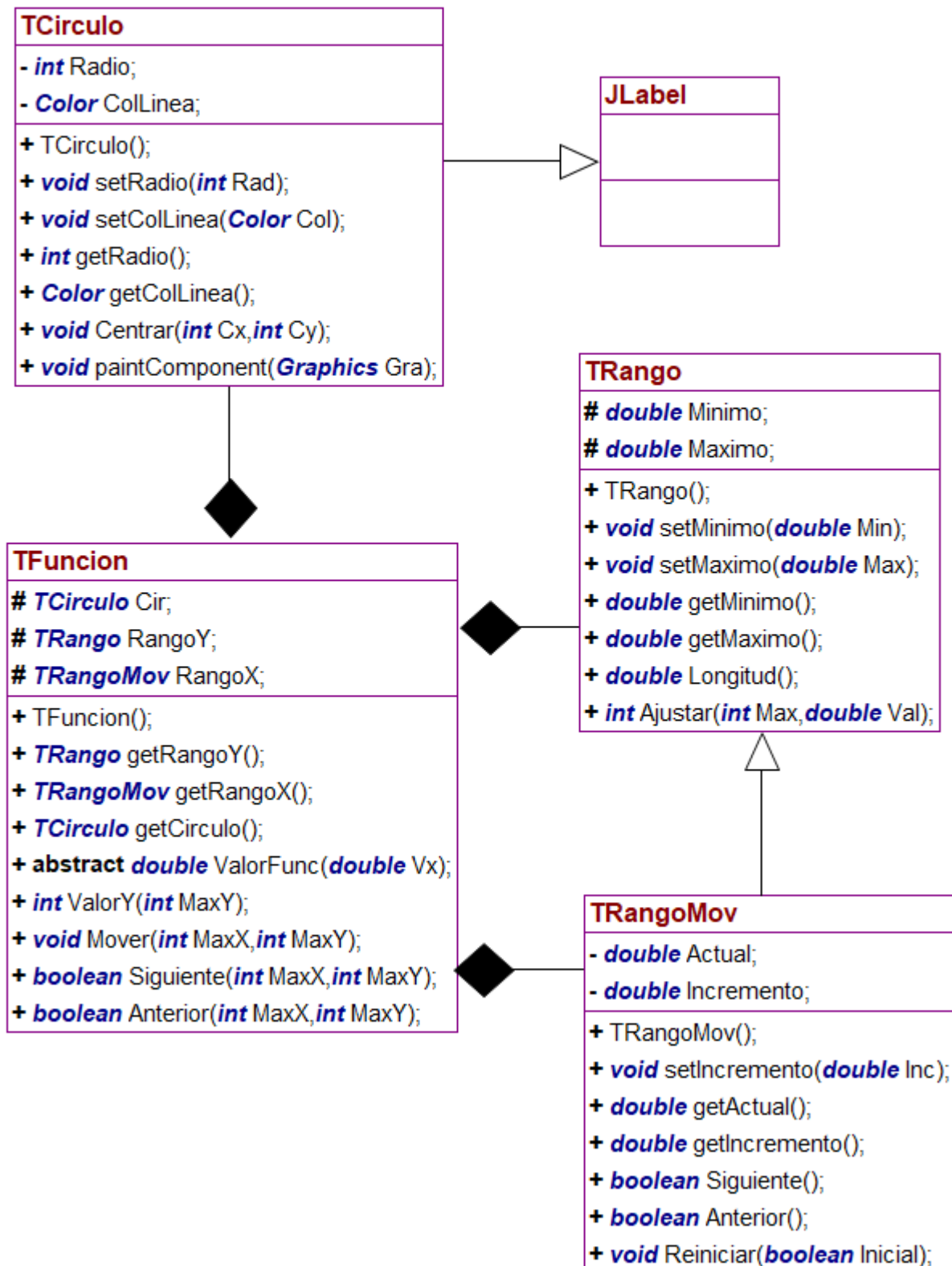
A efectos del desplazamiento de la figura, el hilo asigna valores a la coordenada **X** de la figura tomándolos del intervalo del dominio de la función, usando para ello un incremento o decremento dado por el usuario; además esta tarea la realiza haciendo pausas dadas en milisegundo, de modo que el usuario pueda controlar la velocidad de movimiento de

la figura usando estos dos parámetros; siendo importante además, que los valores de **X** e **Y** así generados por el hilo, se correspondan con el centro de la figura (el **JLabel** circular). El intervalo del dominio de la función, así mismo podrá ser recorrido desde el valor menor al mayor; es decir, en el sentido de izquierda a derecha. Pero una vez finalizado, el hilo deberá reiniciar desde el principio el movimiento de la figura; no obstante, en la implementación de este proceso se prevé que la figura también pueda ser desplazada en el sentido inverso; o sea de derecha a izquierda, caso para el cual consideramos que el intervalo del dominio de la función se recorre desde su valor mayor a su valor menor; esto es, disminuyendo el valor de la coordenada **X** actual de la figura que se mueve. Para este efecto, se usa una clase que representen el intervalo de valores en el eje **X**; es decir, del dominio de la función; esta clase además tiene un atributo que representa el valor actual para la posición **X** de la figura y tiene también otro atributo que indica el valor con el cual se incrementa **X**, que es usado para obtener la siguiente posición en **X**. Sin embargo, el incremento (que bien puede ser negativo), se aplica a través de dos métodos públicos en esta clase: uno llamado *Siguiente* que hace un incremento y otro llamado *Anterior* que por el contrario resta al **X** actual el valor del incremento.

Ahora bien, para el trazado de la figura circular que se mueve en la ventana, desarrolla una clase hija que herede de un **JLabel**, de modo que sobre escribiendo su método *paintComponent*, el componente adapte la apariencia de un círculo; para lo cual el usuario podrá escoger el color de línea (contorno), el color de fondo y el valor del radio. Al respecto, el valor del radio como es de esperarse, determina la dimensión del **JLabel** en tiempo de ejecución, por lo que el ancho y alto de este componente será el doble del valor de su radio. De igual manera, un hilo mueve y manda a centrar a este **JLabel** usando este radio y los valores **X** e **Y** generados por la función y ajustados a las dimensiones (ancho y alto) de la ventana. Finalmente para el tratamiento de las funciones matemáticas, consideraremos una jerarquía de clase con una clase base abstracta, de la cual se derivan clases para las funciones trigonométricas de seno, coseno y tangente por ejemplo; la aplicación debe permitirle al usuario crear todos los hilos que desee, modificar los parámetros de cada uno de ellos, eliminarlos o detenerlos cuando quiera, así como también configurar los círculos y las funciones matemáticas disponibles a su gusto.

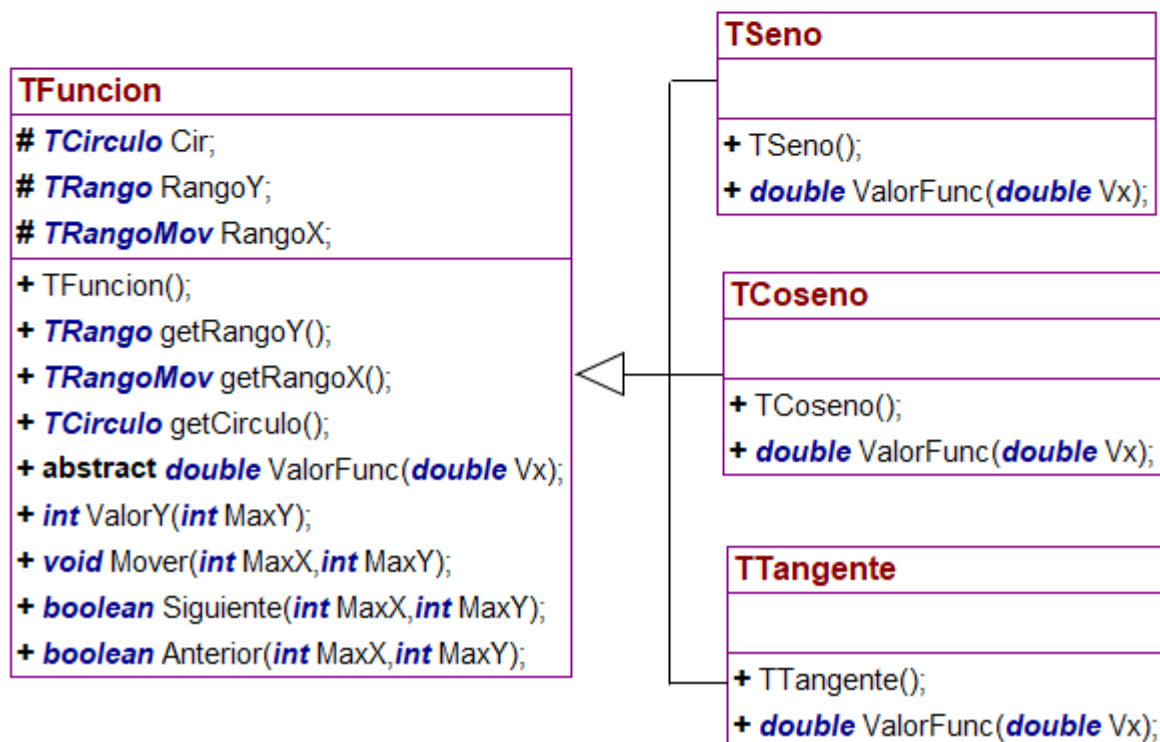
3. Diagrama UML de clases.

a) Composición clase abstracta para la función matemática.



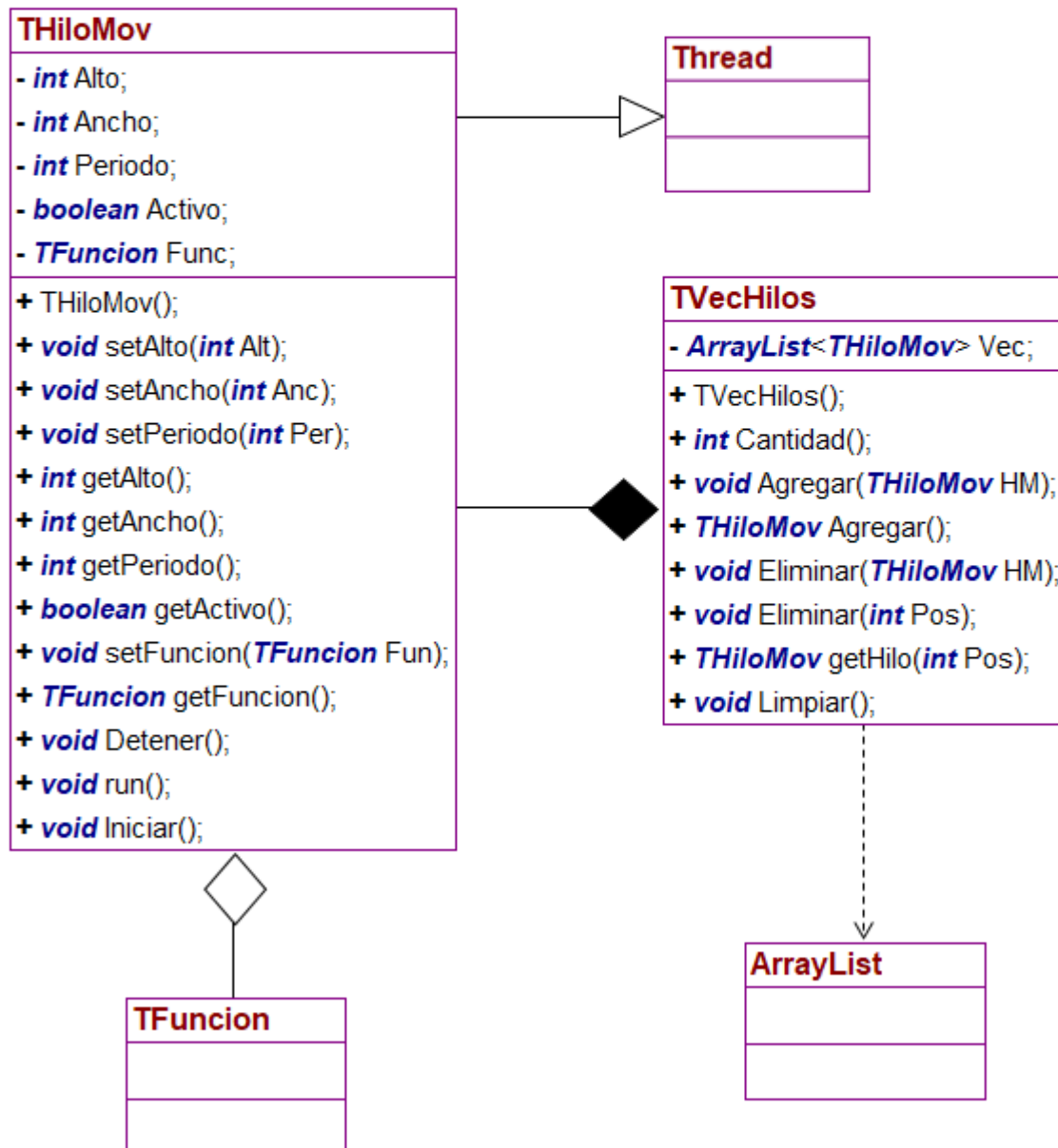
En el diagrama anterior, vemos que la clase **TCirculo** heredera de un **JLabel** y agrega dos atributo, uno para radio y otro para el color de la línea. Igualmente tiene el método *Centrar* para centrarlo en función del radio con respecto a las coordenadas Cx y Cy pasadas por parámetro. El método *paintComponent* es el encargado entonces de dibujar la silueta circular para el **JLabel**. La clase **TRango** por su parte, representa un intervalo de valores entre un valor máximo y uno mínimo; tiene el método para *Ajustar* que escala un valor dado respecto un valor máximo, aplicando una proporción en relación a la longitud del intervalo representado por los atributos de esta clase. De esta clase hereda la clase **TRangoMov**, que añade el atributo *Actual* como valor comprendido entre el mínimo y el máximo, el cual varía entre estos dos valores según el valor del atributo *Incremento*; de modo que éste se suma con el método *Siguiente* o se resta con el método *Anterior*. En cuanto a la case **TFuncion**, vemos que se compone de una instancia de la clase **TCirculo** que es la figura a mover; también contiene una instancia de la clase **TRango** que representa el intervalo de valores máximo y mínimo de la posición de la figura en el eje **Y**, e incluye una instancia de la clase **TRangoMov**, empleada para representar el intervalo de valores máximo y mínimo de la posición en el eje **X**.

b) Clases hijas para las funciones matemáticas.



En la clase **TFuncion** del diagrama anterior, observamos que los métodos *Mover*, *Siguiente* y *Anterior* reciben dos parámetros, que representan las dimensiones (ancho y alto) de la ventana sobre la se mueve la figura. Esta clase tiene como abstracto al método *ValorY*, que calcula el valor de la función para un valor de **X** dado; así las clases hijas concretas **TSeno**, **TCoseno** y **TTangente**, implementan este método respecto a la función trigonométrica que le corresponde.

c) Clases para los hilos.

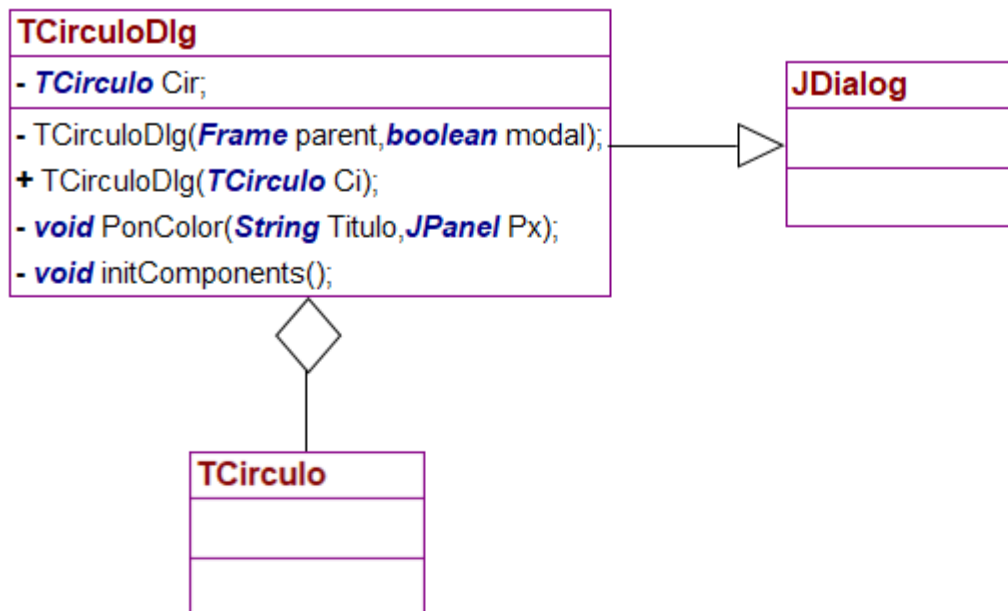


En el diagrama anterior, tenemos la clase **THiloMov** que hereda de la clase **Thread**, por lo cual es la encargada de realizar la tarea que se desarrollará en segundo plano; esto es, la de mover la figura dentro de la ventana siguiendo una función matemática particular. Para tal efecto y como es de esperarse, la clase **THiloMov** sobrescribe el método *run* heredado de **Thread**, pero considerando que el proceso del hilo es arrancado y finalizado mediante los métodos *Iniciar* y *Detener*, que así mismo cambian el estado del atributo *Activo*, cuyo propósito es de controlar el inicio y finalización de un ciclo **while** (mientras). Este ciclo es el responsable de cambiar el valor actual (coordenada **X**) en el intervalo del dominio de la función, para luego usarlo en el cálculo del valor **Y** que le corresponda. Enseguida escala estos dos valores acorde a las dimensiones de la ventana, de modo una vez ajustado, los usa para mover y central la figura en la nueva posición así obtenida. El cambio del valor **X** actual y el movimiento de la figura en la ventana, se efectúa periódicamente cada vez que se cumple o transcurre un intervalo de tiempo; para ello la clase **THiloMov** define el atributo *Periodo*, que es un número entero (*int*) que indica un tiempo medido en milisegundos; por lo tanto, es una pausa o espera que se hace entre cada paso que la figura hace en un su movimiento sobre la ventana.

En este punto la clase **THiloMov** conoce las dimensiones de la ventana mediante los atributos *Alto* y *Ancho*, que son de tipo entero (*int*); mientras que la función matemática a aplicar esta referenciada en el atributo *Fun*, que es instancia de la clase **TFuncion**; en específico instancia de algunos de sus descendientes. Entre tanto, la clase **TVecHilos** representa un conjunto o grupo de hilos, que en específico son instancias de la clase **THiloMov**. Para este propósito, nótese que la clase **TVecHilos** define como atributo una colección (un **ArrayList**) de instancias de la clase **THiloMov**, incluyendo además métodos para agregar, eliminar, consultar o limpiar todos los objetos (hilos en este caso) que son del tipo **THiloMov**.

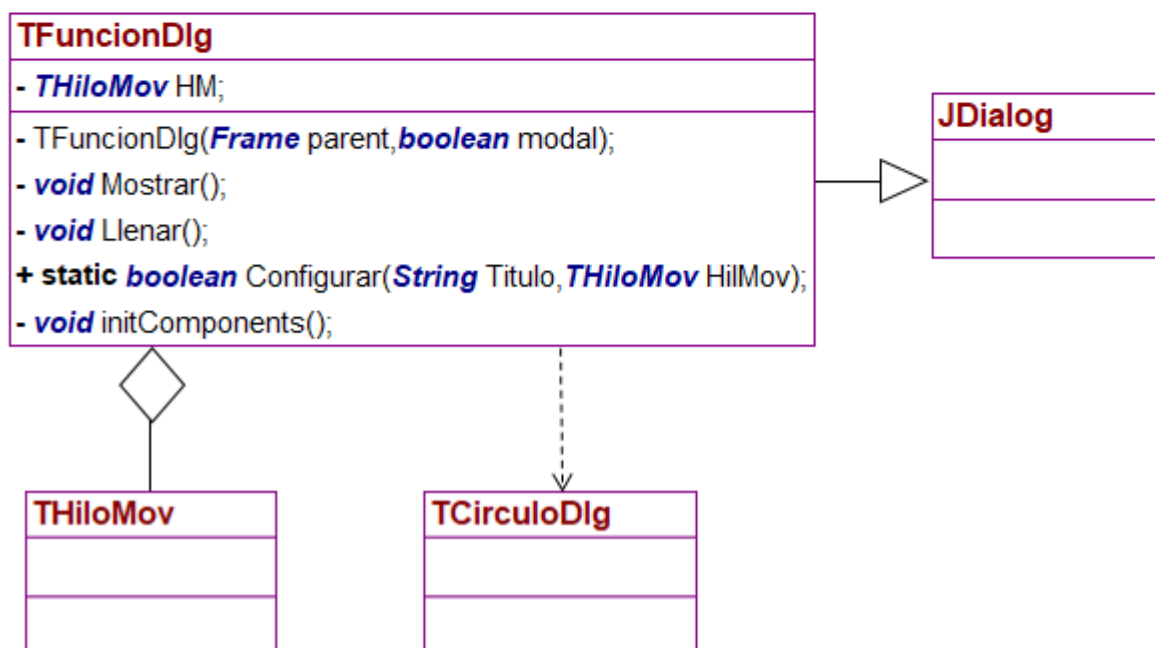
De esta manera vemos que esta aplicación manejará un grupo de hilos de ejecución de manera simultánea e independientes entre sí, mientras que el proceso de sincronización no será relativo a dos o más hilos entre sí; por el contrario, consistirá en hacer que la figura (circulo) se mueva periódicamente y a una velocidad controlada, siempre siguiendo de forma estricta la trayectoria definida por una función matemática.

d) Clases cuadro de dialogo para el círculo.



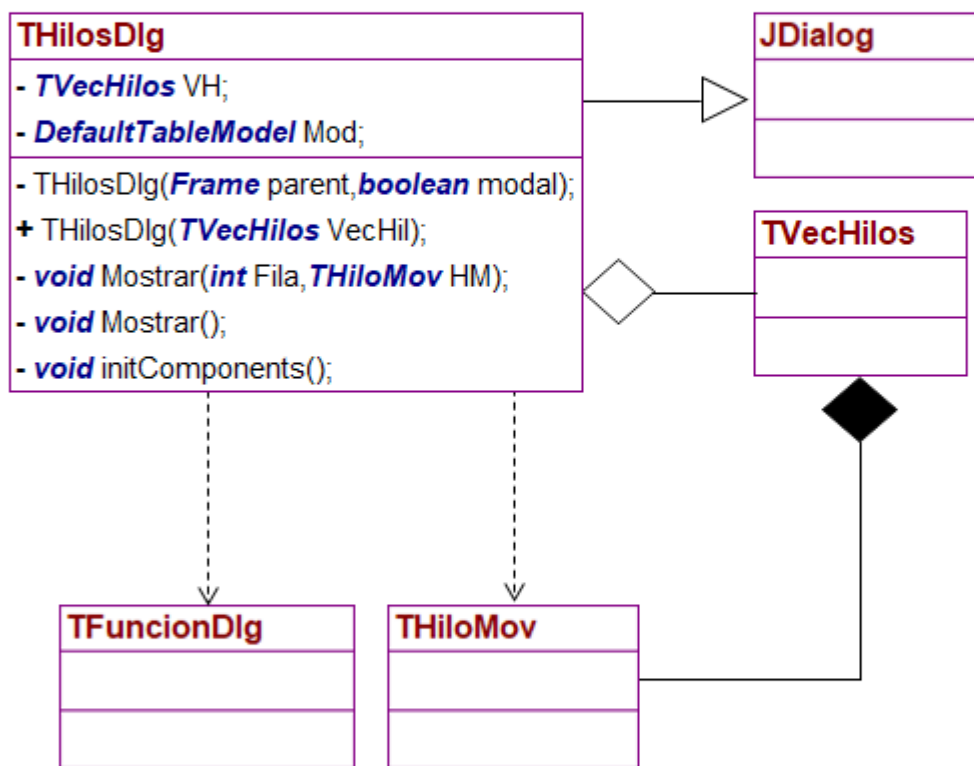
En este diagrama tenemos la clase **TCirculoDlg**, que representa a la ventana en la que se configura la apariencia de un círculo (el **JLabel**), que es la figura a mover con el hilo; de manera que esta ventana es un descendiente de un **JDIALOG** y tiene una relación de agregación por referencia con la clase **TCirculo**.

e) Clases cuadro de dialogo para la función



En este diagrama apreciamos la clase **TFuncionDlg**, que es una ventana descendiente de un **JDialog**; en ella se configuran los valores de la función para un hilo, considerando los valores máximos y mínimos para los intervalos en los ejes **X**, **Y**; así también el valor de incremento en **X** y el periodo de tiempo que el hilo espera entre un desplazamiento y otro de la figura. Por esta razón vemos la relación de agregación por referencia que esta clase hace respecto a la instancia del hilo (clase **THiloMov**); así mismo, como desde este dialogo se llama al dialogo del círculo, tenemos la relación de dependencia entre la clase de dialogo **TFuncionDlg** hacia la del dialogo **TCirculoDlg**.

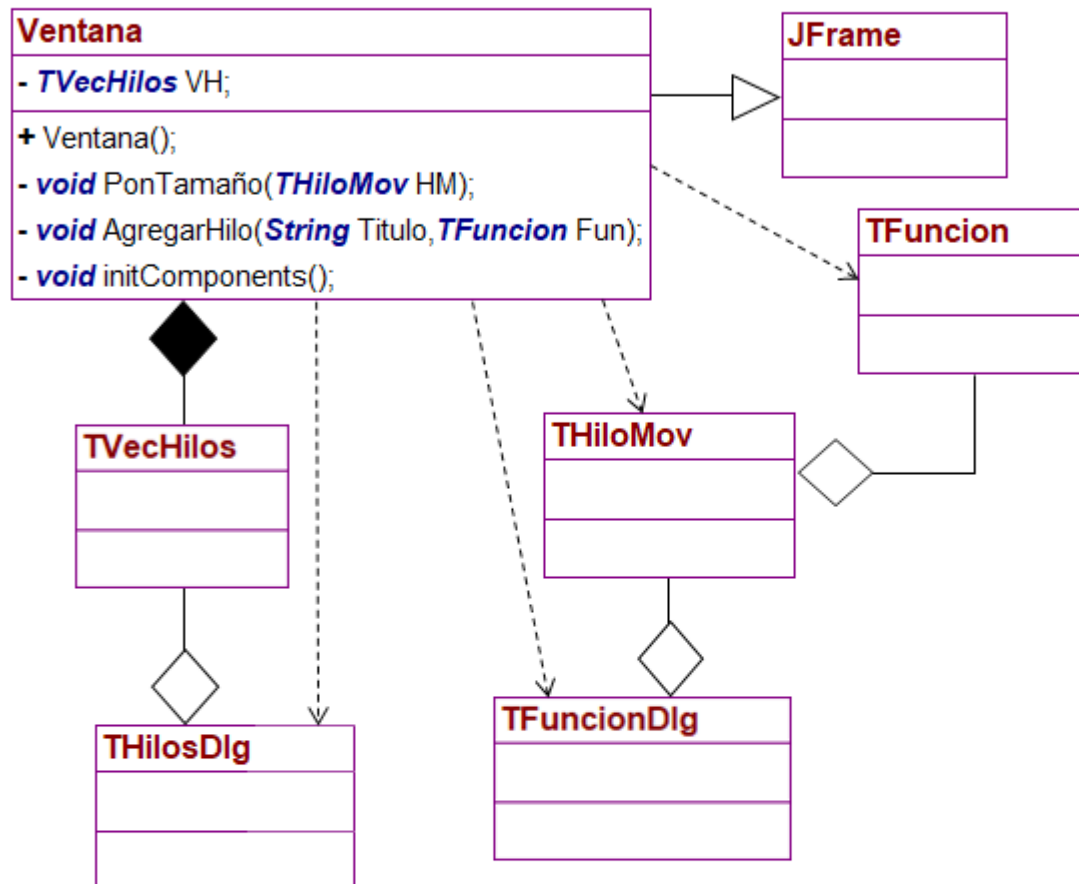
f) Clases cuadro de dialogo para la colección de hilos.



En este diagrama apreciamos que la clase **THilosDlg** también representa a una ventana descendiente de un **JDialog**; en particular, en esta ventana de podrán modificar o actualizar los datos de cualquiera de los hilos creados anteriormente. Para este efecto, en la clase **THilosDlg** se define el atributo **VH** como instancia de la clase **TVecHilos**; de allí la relación de agregación por referencia entre estas dos clases. Ahora bien, para mostrar la información de cada hilo (instancias de la clase **THiloMov**), el dialogo dispone de un **JTable**, cuyo modelo de datos es también atributo de la ventana y mostrará algunos de los atributos del hilo; por esta razón, tenemos una dependencia del dialogo **THilosDlg**

hacia la clase **THiloMov**. Finalmente como el hilo seleccionado en la tabla se configura mediante el dialogo **TFuncionDlg**, vemos que existe una dependencia del dialogo **THilosDlg** respecto al dialogo de la función.

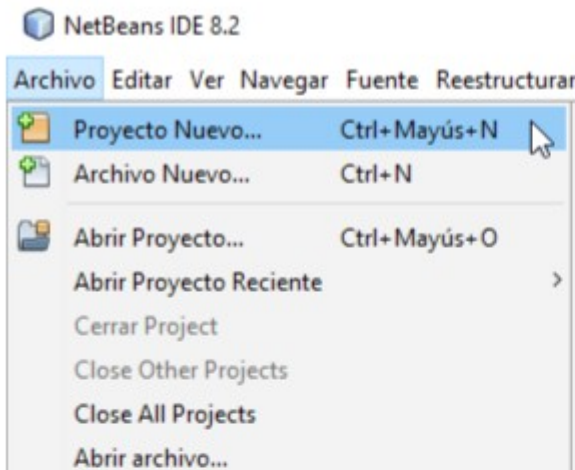
g) Clases para la ventana principal.



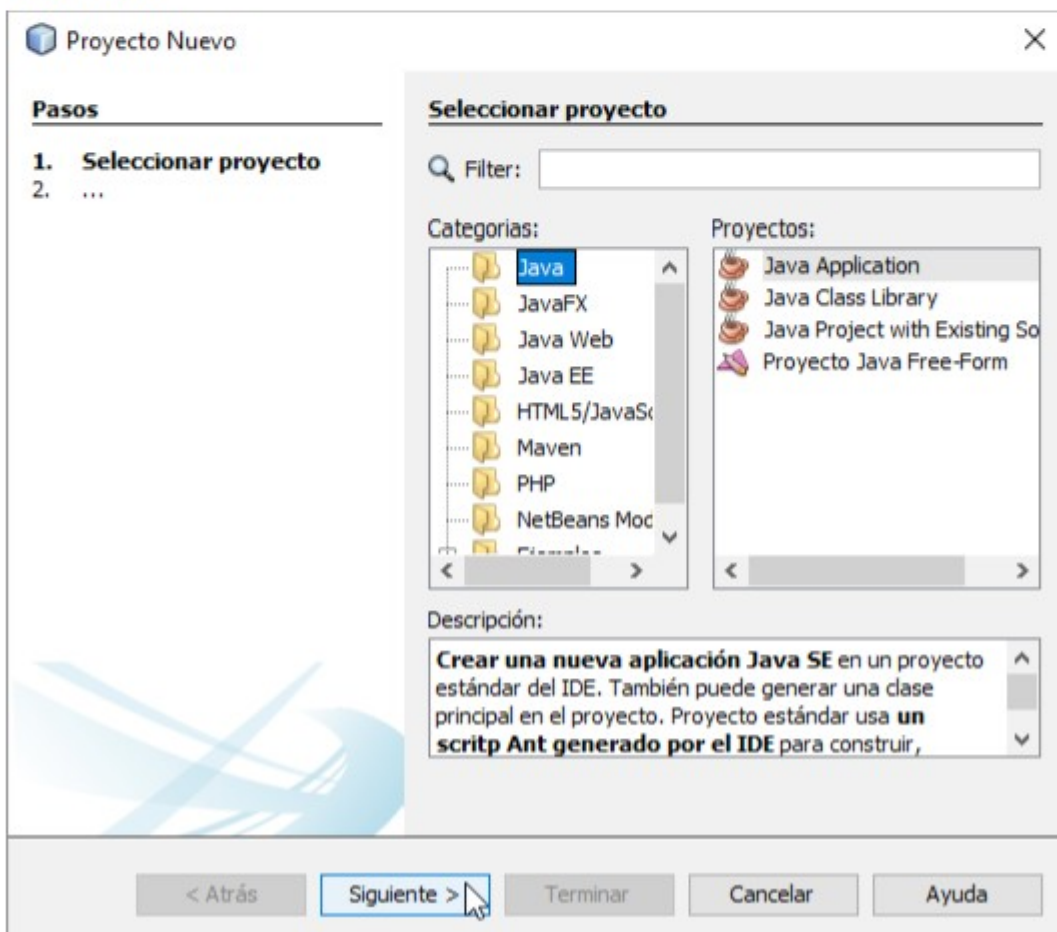
En este diagrama vemos que la ventana herede de un **JFrame** y tiene una relación de composición con una instancia (**VH**) de la clase **TVecHilos**; que en últimas es la colección de hilos que se ejecutaran en segundo plano en el programa. Así mismo, como las funciones son creadas desde esta ventana a petición del usuario, vemos que se establece una relación de dependencia de la clase **Ventana** hacia la clase **TFuncion**; en particular hacia sus descendientes. Por la misma razón, como los hilos son creados desde la ventana tenemos la relación de dependencia de esta hacia la clase **THiloMov**. Ahora bien, toda vez que desde la ventana se llama al diálogo **TFuncionDlg** cuando se crea una función, la ventana depende de este último dialogo; por idéntica razón, la ventana también depende del diálogo **THilosDlg**, pues lo necesita para mostrar al usuario todos los hilos creados para que este los configure.

4. Creación del proyecto en *NetBeans*

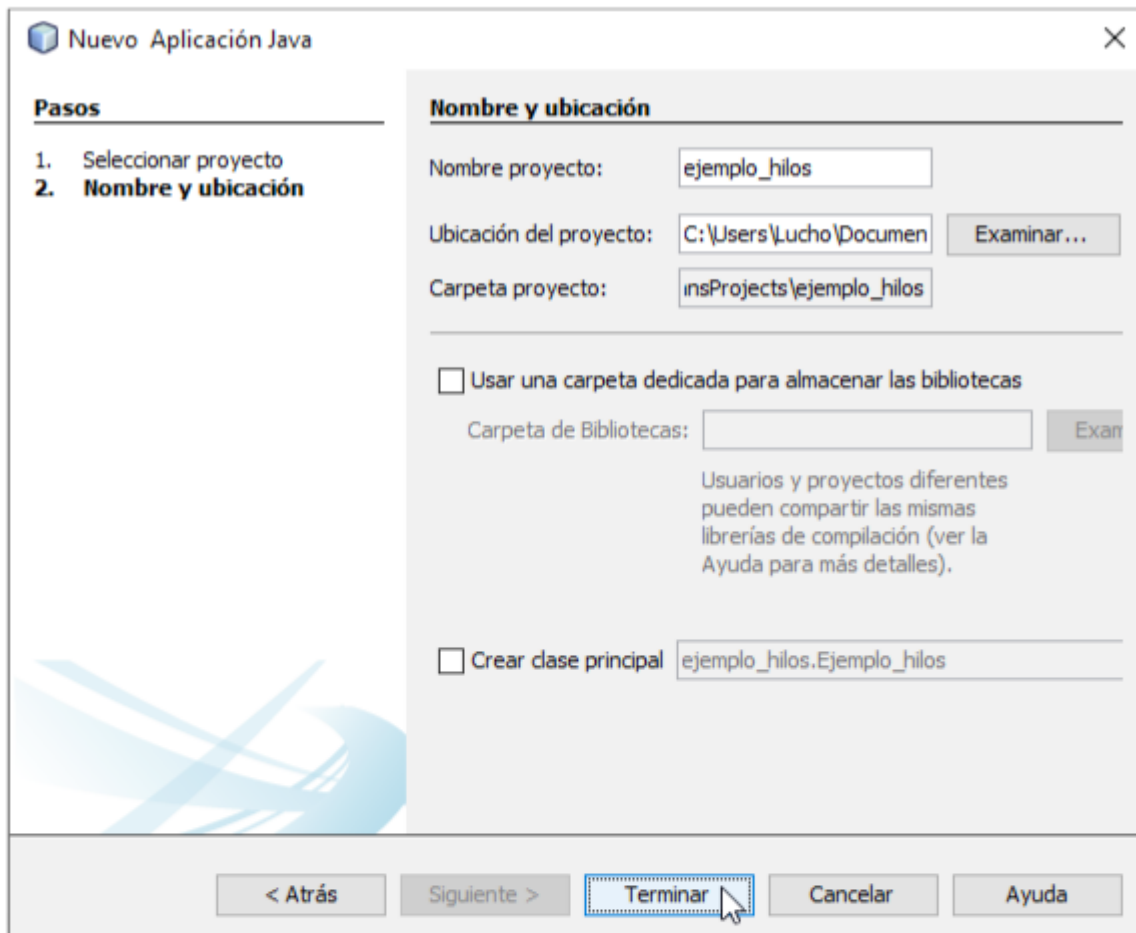
- ➡ Haga click en la opción de menú **Archivo** y escoja la opción **Proyecto Nuevo**:



- ➡ En la ventana desplegada en el panel "**Categorías**" escoja el nodo "**Java**"; en el panel "**Proyectos**" seleccione el ítem "**Java Application**" y haga click en el botón "**Siguiente**".



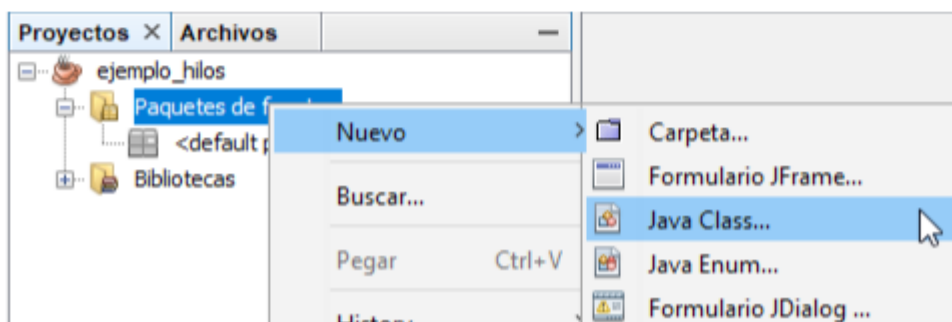
- En la entrada “**Nombre proyecto**” de esta ventana, escriba **ejemplo_hilos** y desmarque la casilla “**Crear clase principal**”, finalmente haga click en el botón **Terminar**.



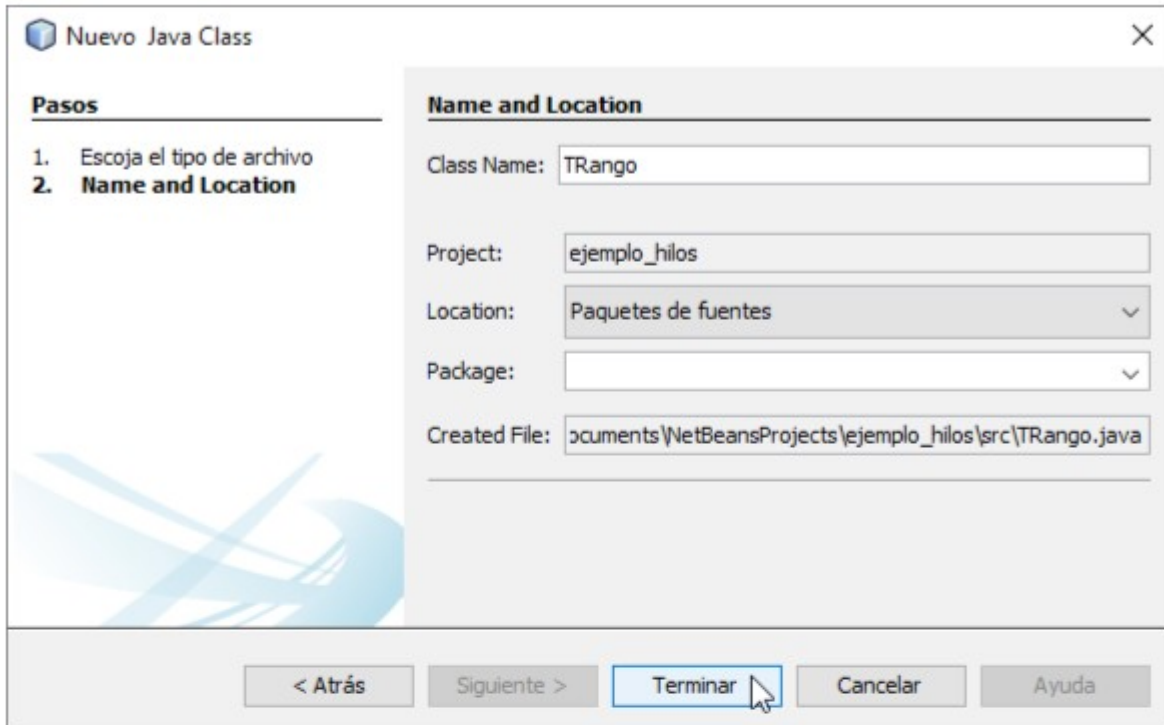
5. Implementación clases lógica de aplicación

5.1 Creación clase *TRango*

- En el nodo del proyecto (**ejemplo_hilos**) del panel **Proyectos**, haga click derecho sobre el nodo **Paquetes de fuentes**; después sobre las opciones **Nuevo** y **Java Class**, tal como se muestra abajo:



- ⇒ En esta ventana en la entrada **Class Name** ingrese **TRango** como nombre de la clase y haga click en el botón **Terminar**.



Nuevo Java Class

Pasos

1. Escoja el tipo de archivo
2. **Name and Location**

Name and Location

Class Name: TRango

Project: ejemplo_hilos

Location: Paquetes de fuentes

Package:

Created File: ocuments\NetBeansProjects\ejemplo_hilos\src\TRango.java

< Atrás Siguiente > **Terminar** Cancelar Ayuda

- ⇒ Implemente la clase **TRango** (archivo **TRango.java**) como sigue:

```
1 public class TRango {
2
3     protected double Minimo;
4     protected double Maximo;
5
6     public TRango(){
7         Minimo=0;
8         Maximo=0;
9     }
10
11     public void setMinimo(double Min){
12         Minimo=Min;
13     }
14
15     public void setMaximo(double Max){
16         Maximo=Max;
17     }
18
19     public double getMinimo(){
20         return Minimo;
21     }
22 }
```

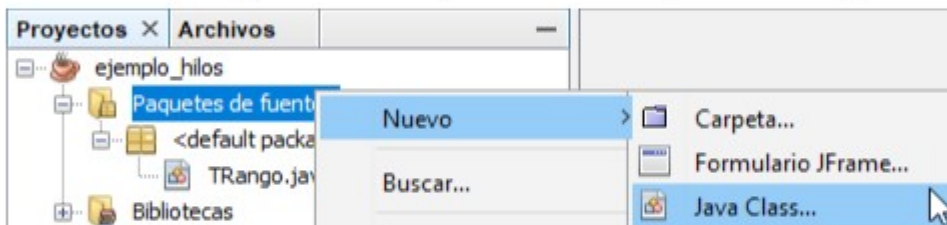
```

23 public double getMaximo(){
24     return Maximo;
25 }
26
27 public double Longitud(){
28     return Math.abs(Maximo-Minimo);
29 }
30
31 /*Escala el valor Val como una proporción de Max respecto
32 a la longitud del intervalo o rango */
33 public int Ajustar(int Max,double Val){
34     return (int)(Val*Max/Longitud());
35 }
36
37 }

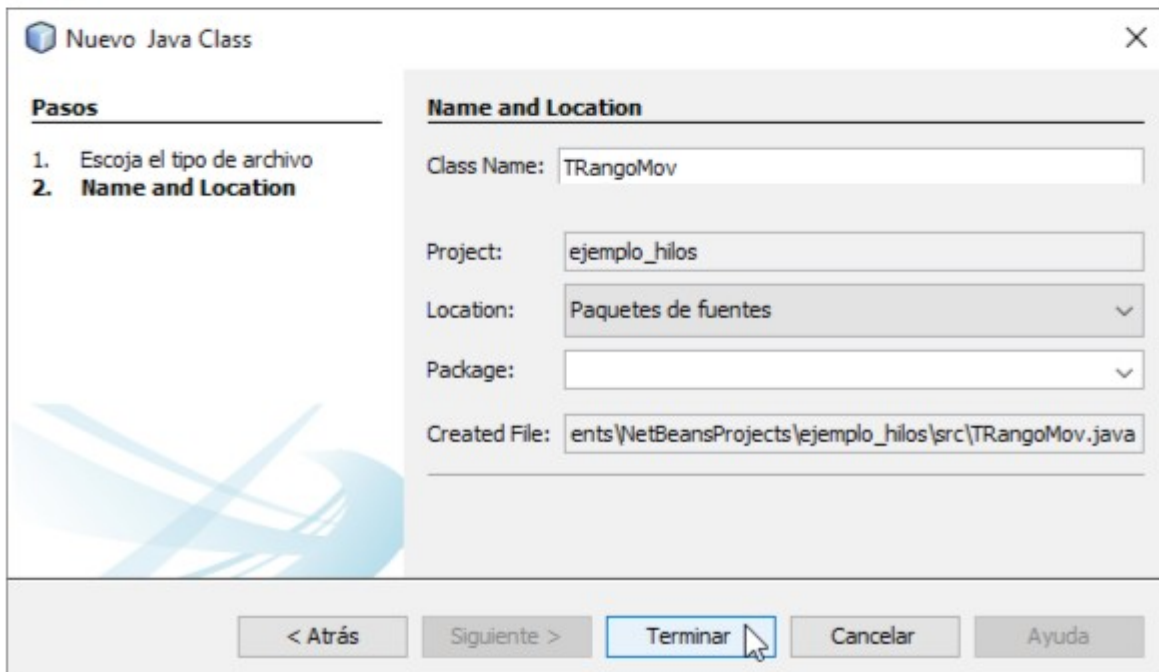
```

5.2 Creación clase *TRangoMov*

- En el nodo del proyecto (*ejemplo_hilos*) haga click derecho sobre el nodo **Paquetes de fuentes** y luego sobre las opciones **Nuevo** y **Java Class**, como se muestra abajo:



- En esta ventana en la entrada **Class Name** ingrese **TRango** como nombre de la clase y haga click en el botón **Terminar**.



➡ Implemente la clase **TRangoMov** (archivo *TRangoMov.java*) como sigue:

```
1 public class TRangoMov extends TRango {
2
3     private double Actual; //Valor activo entre Minimo y Maximo
4     private double Incremento; //Variacion para el valor actual
5
6     public TRangoMov(){
7         super();
8         Actual=0;
9         Incremento=0;
10    }
11
12    public void setIncremento(double Inc){
13        Incremento=Math.abs(Inc); //Incremento siempre será positivo
14    }
15
16    public double getActual(){
17        return Actual;
18    }
19
20    public double getIncremento(){
21        return Incremento;
22    }
23
24    //Obtiene un nuevo valor actual sumando el incremento
25    public boolean Siguiente(){
26        Actual=Actual + Incremento;
27        if(Actual>Maximo){ //Si es mayor al maximo permitido en el rango
28            Actual=Maximo; //Se ajusta al maximo posible
29            return false; //No se obtuvo un siguiente dentro del rango
30        }
31        else{
32            return true; //El nuevo actual esta en el rango
33        }
34    }
35
36    public boolean Anterior(){ //Obtiene nuevo actual restando el incremento
37        Actual=Actual - Incremento;
38        if(Actual<Minimo){ //Si es menor al minimo permitido en el rango
39            Actual=Minimo; //Se ajusta al minimo posible
40            return false; //No se obtuvo un anterior dentro del rango
41        }
42        else{
43            return true; //El nuevo actual esta en el rango
44        }
45    }
```

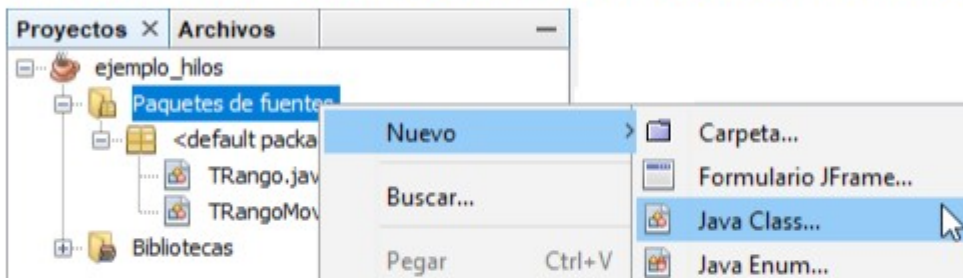
```

46
47 //Establece el valor actual al principio o al final del rango
48 public void Reiniciar(boolean Inicial){
49     if(Inicial){
50         Actual=Minimo; //Restaura con el inicio del rango
51     }
52     else{
53         Actual=Maximo; //Restaura con el final del rango
54     }
55 }
56
57 }

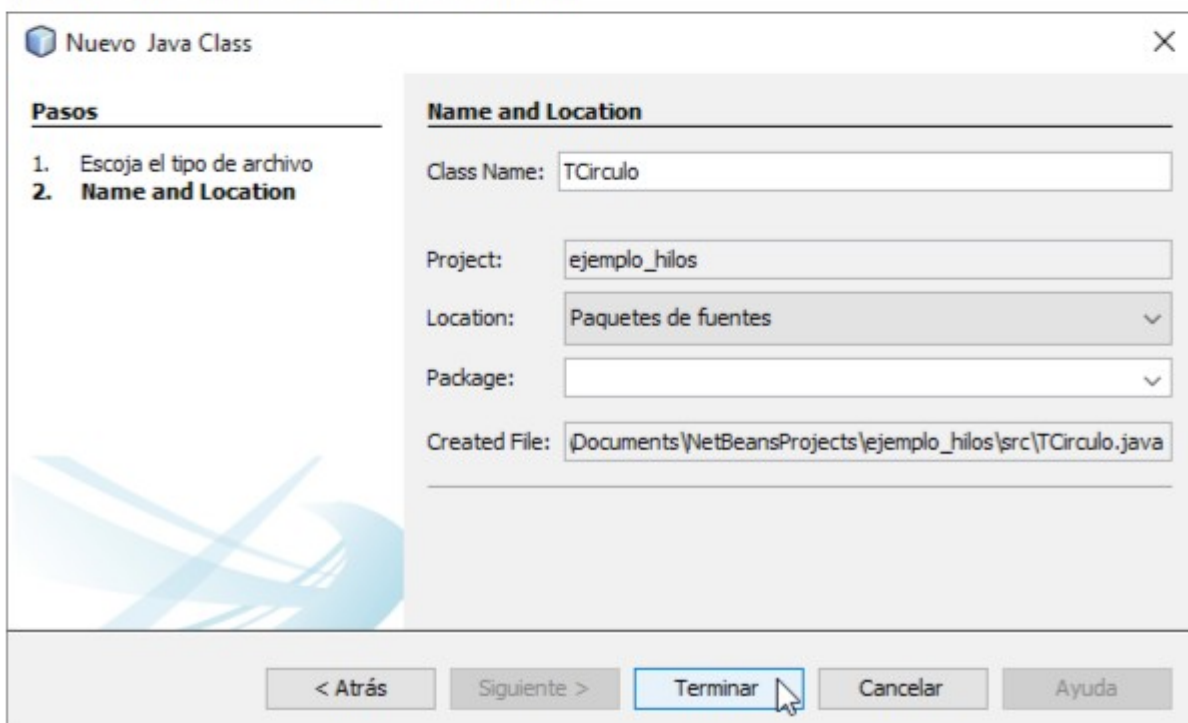
```

5.3 Creación clase *TCirculo*

- ➡ En el nodo del proyecto (*ejemplo_hilos*) haga click derecho sobre el nodo **Paquetes de fuentes** y luego sobre las opciones **Nuevo** y **Java Class**, como se muestra abajo:



- ➡ En esta ventana en la entrada **Class Name** ingrese **TCirculo** como nombre de la clase y haga click en el botón **Terminar**.



➡ Implemente la clase **TCirculo** (archivo *TCirculo.java*) como sigue:

```
1 import java.awt.Color;
2 import java.awt.Graphics;
3 import java.awt.Graphics2D;
4 import javax.swing.JLabel;
5 import java.awt.geom.Ellipse2D;
6
7 public class TCirculo extends JLabel {
8
9     private int Radio;
10    private Color ColLinea;
11
12    public TCirculo(){
13        super();
14        Radio=0;
15        setText("");
16        ColLinea=Color.BLACK;
17        setBackground(Color.WHITE);
18    }
19
20    public void setRadio(int Rad){
21        if(Rad>0){
22            Radio=Rad;
23            //Ajusta el tamaño (ancho y alto) al diametro del circulo
24            setSize(2*Radio,2*Radio);
25        }
26    }
27
28    public void setColLinea(Color Col){
29        ColLinea=Col;
30        repaint();//repintar el JLabel
31    }
32
33    public int getRadio(){
34        return Radio;
35    }
36
37    public Color getColLinea(){
38        return ColLinea;
39    }
40
41    public void Centrar(int Cx,int Cy){
42        //Cx-Radio y Cy-Radio es la posición superior izquierda del componente
43        setLocation(Cx-Radio,Cy-Radio);
44    }
```

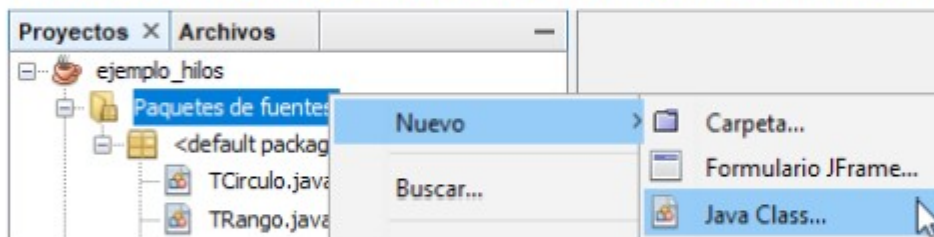
```

45
46 @Override
47 public void paintComponent(Graphics Gra){
48     Ellipse2D Cir;
49     Graphics2D G2D;
50     super.paintComponent(Gra); //ejecutamos la implementacion del padre
51     G2D=(Graphics2D)Gra;
52     //Creamos la geometria con dimensiones para un circulo
53     Cir=new Ellipse2D.Float(0,0,getWidth()-1,getHeight()-1);
54     G2D.setColor(getBackground()); //Asignamos color de fondo
55     G2D.fill(Cir); //Dibujamos de forma rellena
56     G2D.setColor(ColLinea); //Asignamos color de la silueta
57     G2D.draw(Cir); //Dibujamos la silueta
58 }
59
60 }

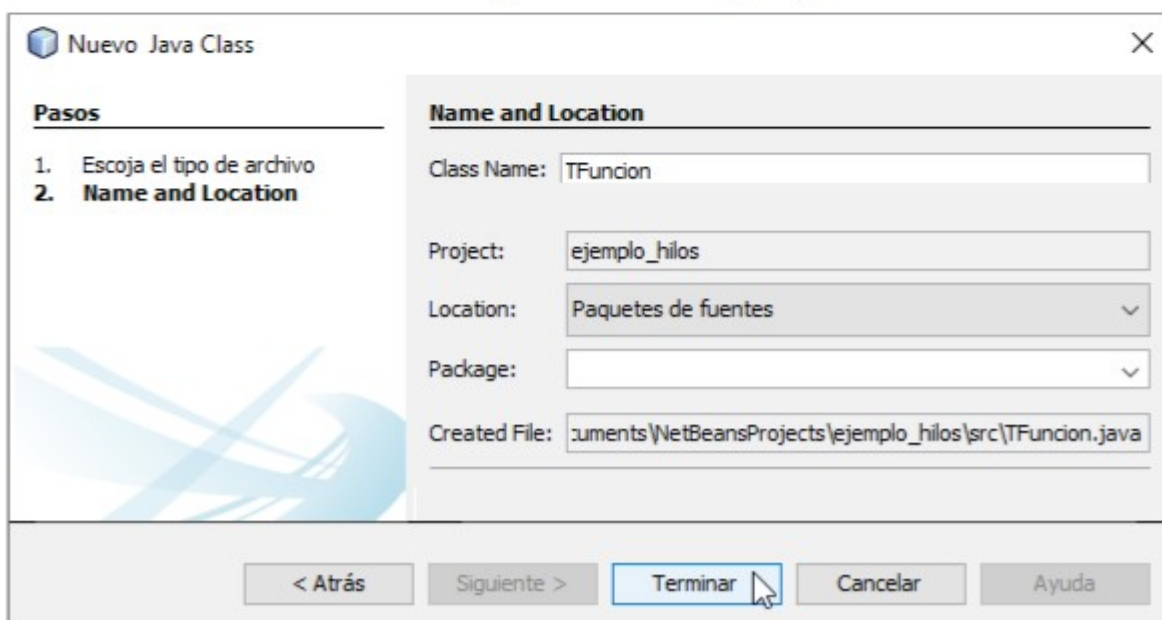
```

5.4 Creación clase TFuncion

- ➡ En el nodo del proyecto (**ejemplo_hilos**) haga click derecho sobre el nodo **Paquetes de fuentes** y luego sobre las opciones **Nuevo** y **Java Class**, como se muestra abajo:



- ➡ Ahora en la entrada **Class Name** ingrese **TFuncion** y haga click en el botón **Terminar**.



➡ Implemente la clase **TFuncion** (archivo *TFuncion.java*) como sigue:

```
1 public abstract class TFuncion {
2
3     //Atributos protegidos para acceder a ellos desde las clases hijas
4     protected TCirculo Cir; //Instancia del circulo (figura) a mover
5     protected TRango RangoY; //Rango de la funcion (para el eje Y)
6     protected TRangoMov RangoX; //Dominio de la funcion (para el eje X)
7
8     public TFuncion(){
9         Cir=new TCirculo();
10        RangoY=new TRango();
11        RangoX=new TRangoMov();
12    }
13
14    /* Solo metodos selectores para estos atributos,
15       pues es una relacion de composicion */
16    public TRango getRangoY(){
17        return RangoY;
18    }
19
20    public TRangoMov getRangoX(){
21        return RangoX;
22    }
23
24    public TCirculo getCirculo(){
25        return Cir;
26    }
27
28    //Calcula la funcion en el punto Vx (valor x)
29    public abstract double ValorFunc(double Vx);
30
31    //MaxY es el alto maximo del container (ventana o un panel)
32    public int ValorY(int MaxY){
33        return RangoY.Ajustar(MaxY,ValorFunc(RangoX.getActual()));
34    }
35
36    //MaxX es el ancho maximo del contenedor (ventana o un panel)
37    public void Mover(int MaxX,int MaxY){ //MaxY alto maximo del contenedor
38        int cx,cy;
39        //Se dividen por dos para ubicar origen en el centro de la ventana
40        cy=MaxY/2 - ValorY(MaxY);
41        cx=MaxX/2 + RangoX.Ajustar(MaxX,RangoX.getActual());
42        //Se centra el circulo en su nueva posicion
43        Cir.Centrar(cx,cy);
44    }
45
```



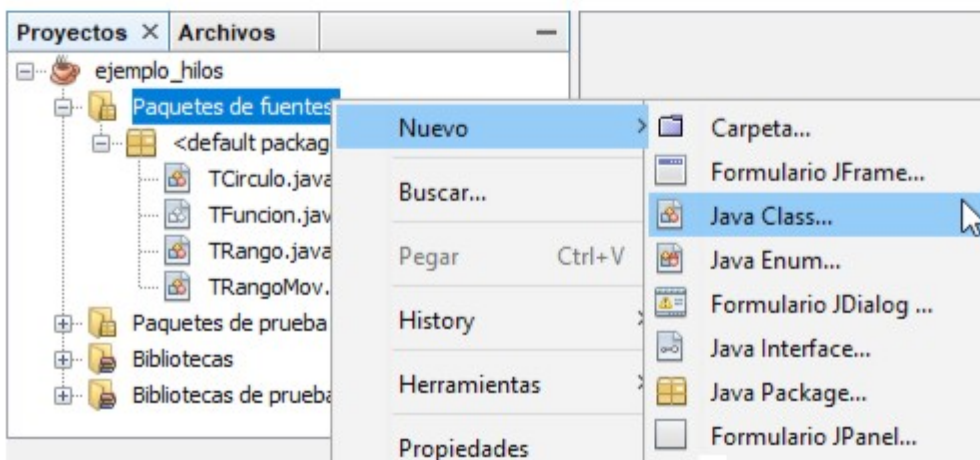
```

46 //Obtiene el la posicion siguiente aumentando el valor de X
47 public boolean Siguiente(int MaxX,int MaxY){
48     if(RangoX.Siguiente()){
49         Mover(MaxX,MaxY);
50         return true; //Se pudo desplazar
51     }
52     else{
53         return false; //Se llegó al extremo maximo (derecho)
54     }
55 }
56
57 //Obtiene el la posicion anterior disminuyendo el valor de X
58 public boolean Anterior(int MaxX,int MaxY){
59     if(RangoX.Anterior()){
60         Mover(MaxX,MaxY);
61         return true; //Se pudo desplazar
62     }
63     else{
64         return false; //Se llegó al extremo minimo (izquierdo)
65     }
66 }
67
68 }

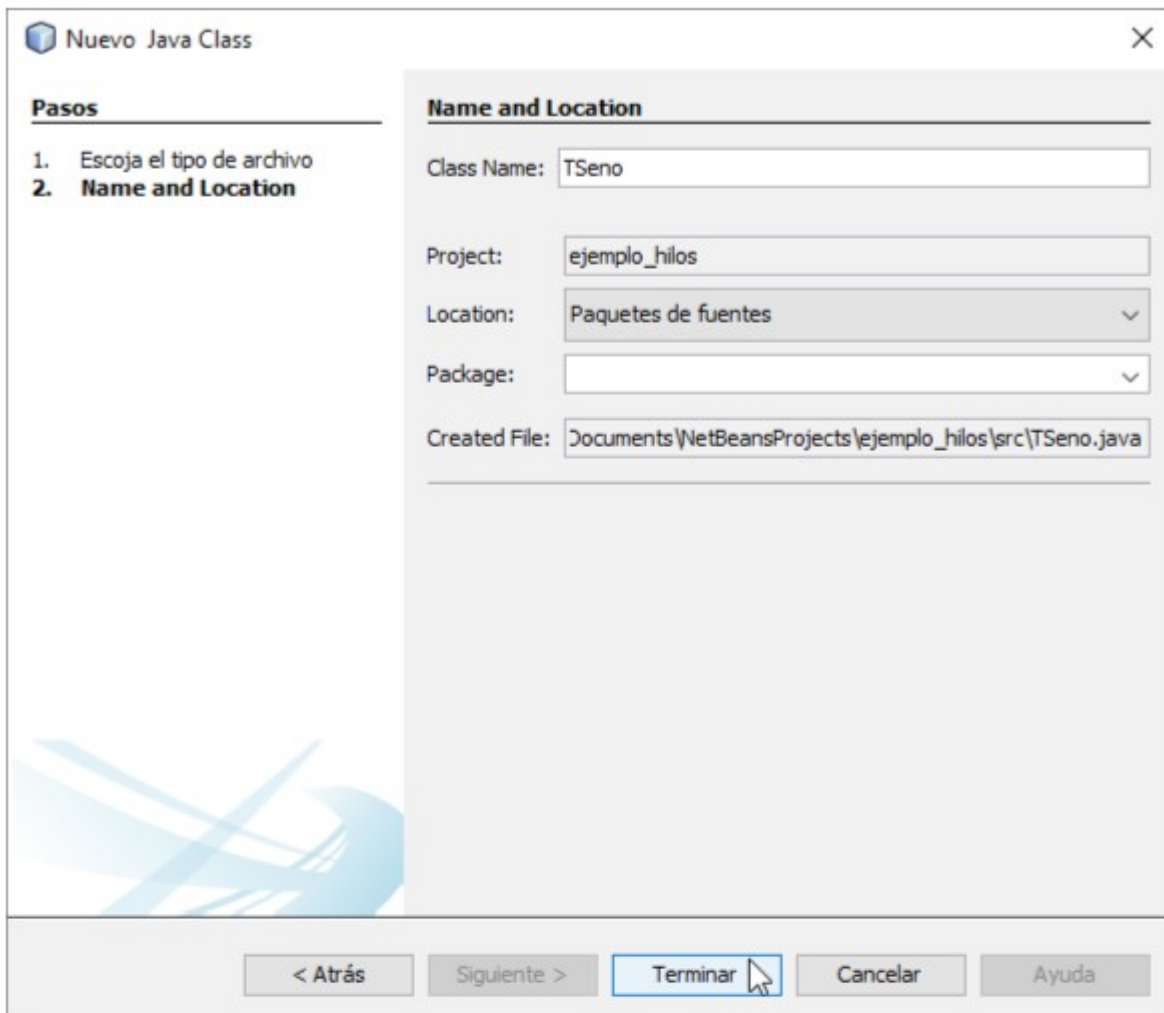
```

5.5 Creación clase TSeno

- ➡ A la izquierda en el panel de proyectos, en el nodo del proyecto (**ejemplo_hilos**) haga click derecho sobre el nodo **Paquetes de fuentes** y luego sobre las opciones **Nuevo** y **Java Class**, tal como se ilustra en la imagen de abajo:



- ➡ En esta ventana en la entrada **Class Name** ingrese **TSeno** como nombre de la clase y haga click en el botón **Terminar**.



Nuevo Java Class

Pasos

1. Escoja el tipo de archivo
2. **Name and Location**

Name and Location

Class Name:

Project:

Location:

Package:

Created File:

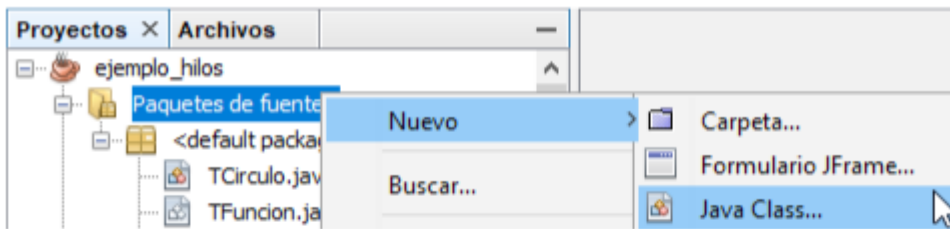
< Atrás Siguiente > **Terminar** Cancelar Ayuda

- ➡ Implemente la clase **TSeno** (archivo *TSeno.java*) como sigue:

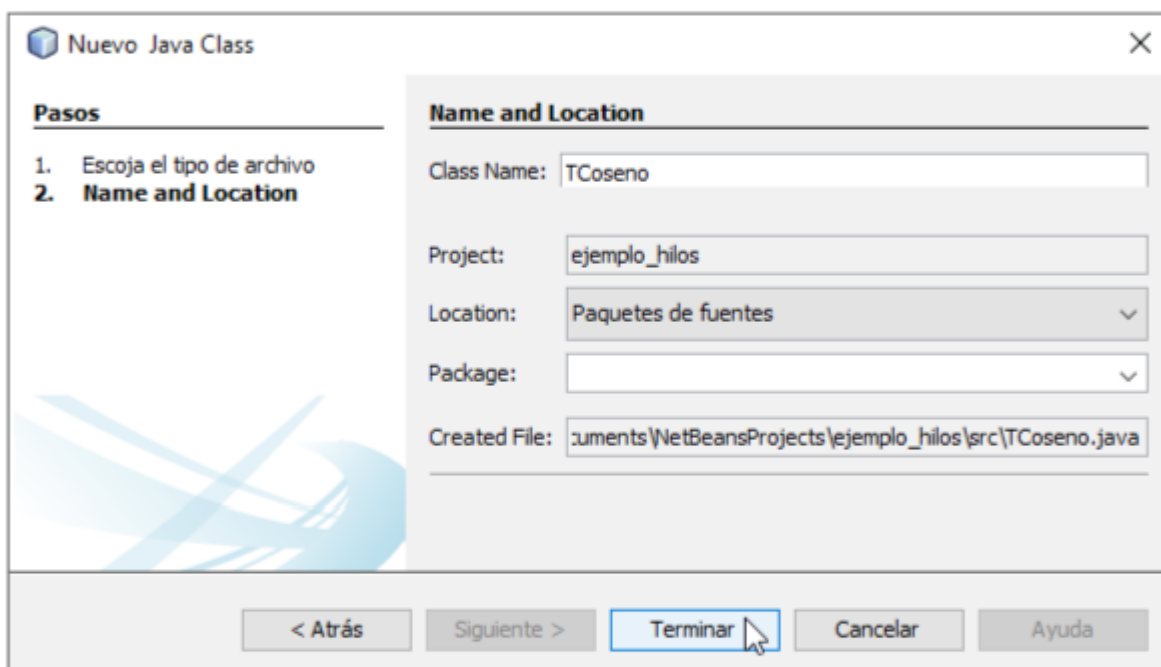
```
1 public class TSeno extends TFuncion {  
2  
3     public TSeno() {  
4         super();  
5     }  
6  
7     //Implementamos el metodo abstracto heredado  
8     @Override  
9     public double ValorFunc(double Vx) {  
10        //Esta multiplicacion convierte el angulo Vx a radian  
11        return Math.sin(Vx*0.0174532925);  
12    }  
13  
14 }
```

5.6 Creación clase *TCoseno*

- ➡ En el nodo del proyecto (*ejemplo_hilos*) haga click derecho sobre el nodo **Paquetes de fuentes** y luego sobre las opciones **Nuevo** y **Java Class**, como se muestra abajo:



- ➡ En esta ventana en la entrada **Class Name** ingrese **TCoseno** como nombre de la clase y haga click en el botón **Terminar**.

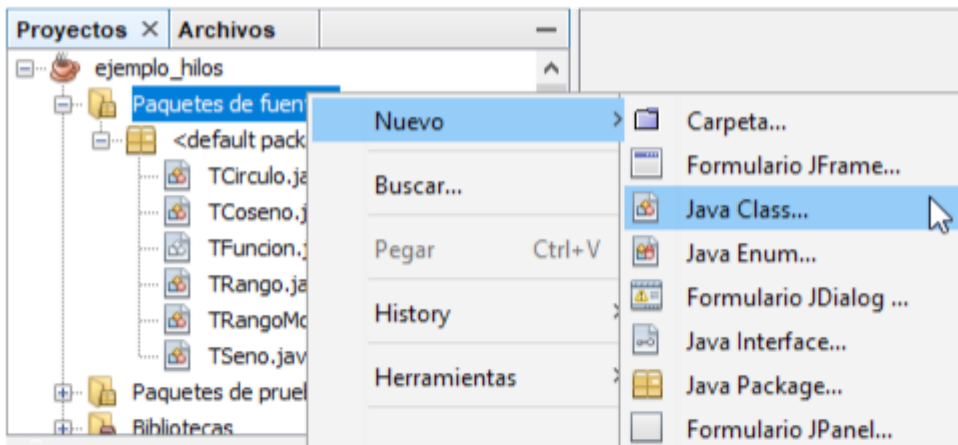


- ➡ Implemente la clase **TCoseno** (archivo *TCoseno.java*) como sigue:

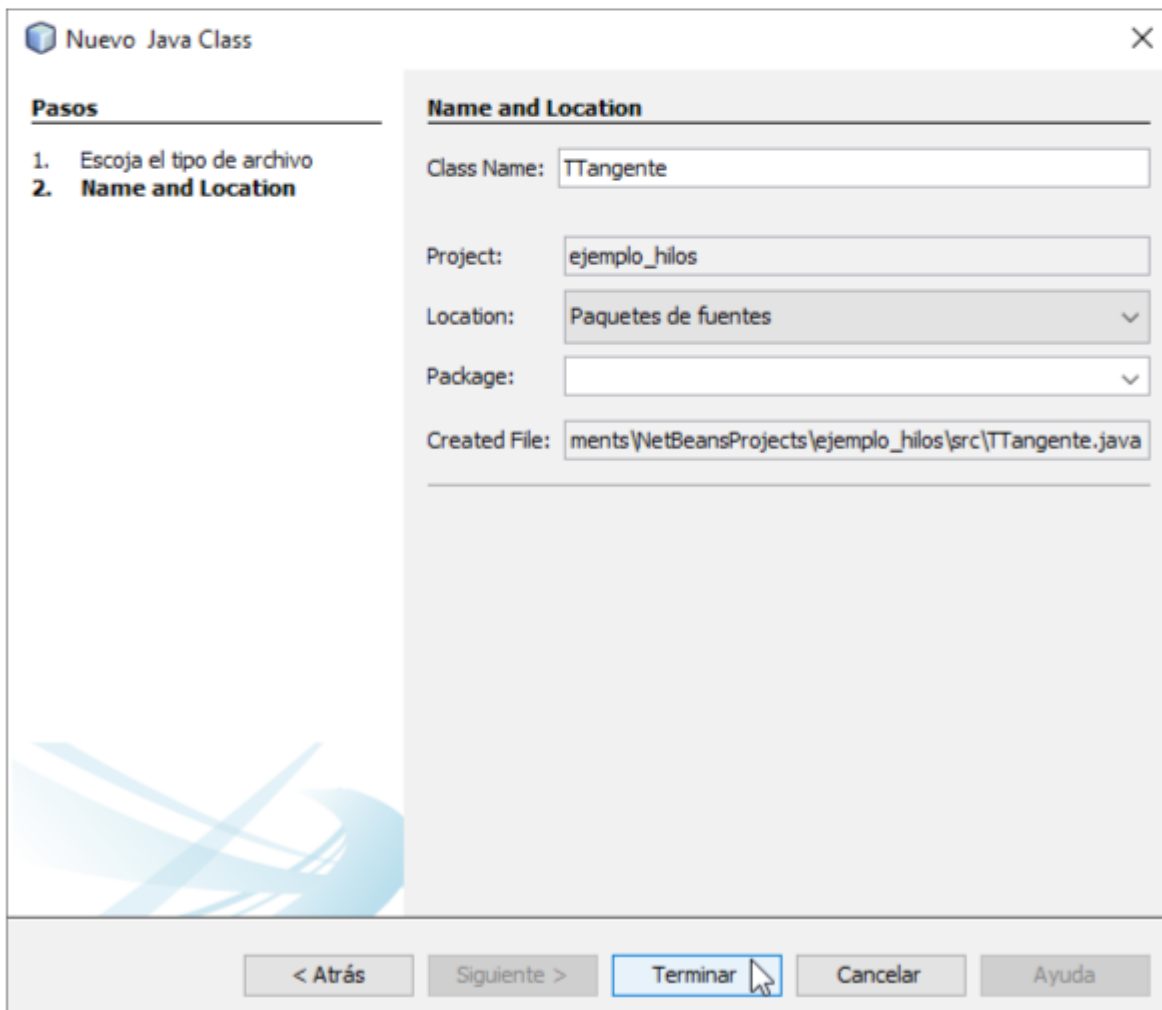
```
1 public class TCoseno extends TFuncion {  
2  
3     public TCoseno() {  
4         super();  
5     }  
6  
7     //Implementamos el metodo abstracto heredado  
8     @Override  
9     public double ValorFunc(double Vx) {  
10        return Math.cos(Vx*0.0174532925);  
11    }  
12  
13 }
```

5.7 Creación clase *TTangente*

- ➡ En el nodo del proyecto (*ejemplo_hilos*) haga click derecho sobre el nodo **Paquetes de fuentes** y luego sobre las opciones **Nuevo** y **Java Class**, como se muestra abajo:



- ➡ En esta ventana en la entrada **Class Name** ingrese **TTangente** como nombre de la clase y haga click en el botón **Terminar**.

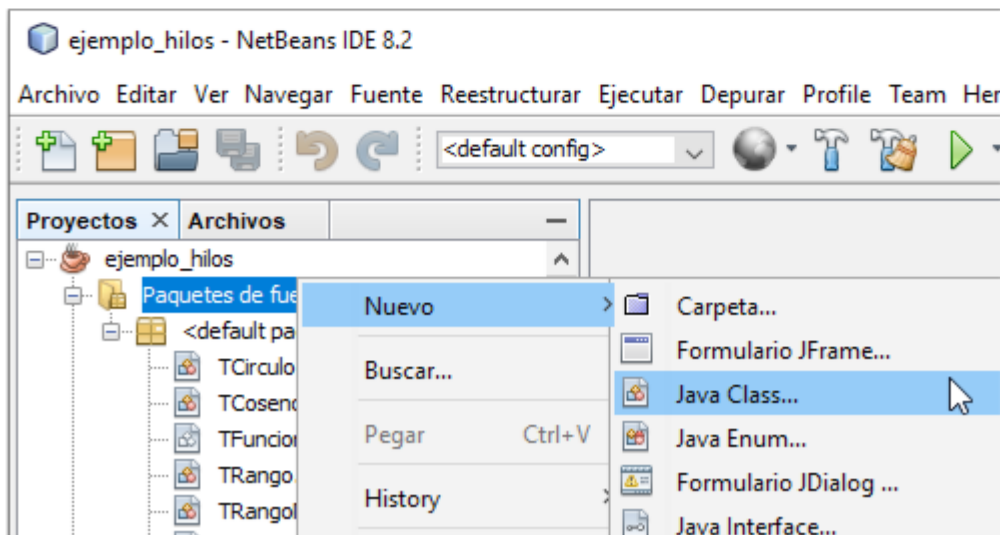


➡ Implemente la clase **TTangente** (archivo *TTangente.java*) como sigue:

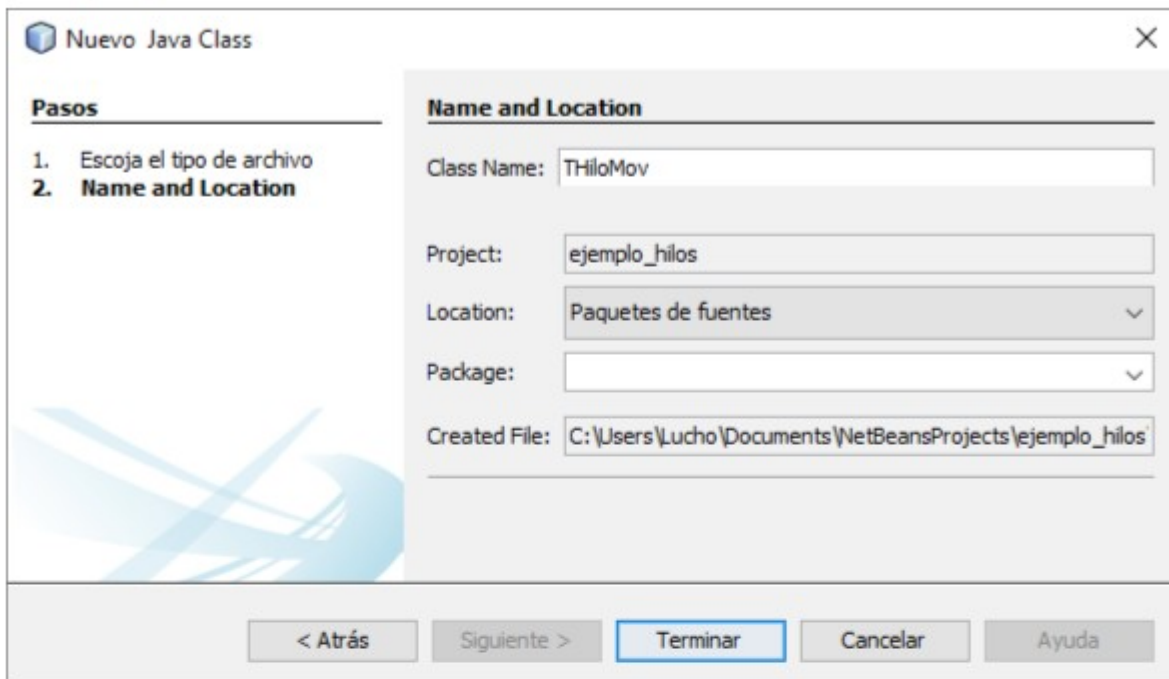
```
1 public class TTangente extends TFuncion {
2
3     public TTangente(){
4         super();
5     }
6
7     //Implementamos el metodo abstracto heredado
8     @Override
9     public double ValorFunc(double Vx){
10         double cx,ang;
11         ang=Vx*0.0174532925; //Convertimos a radian
12         cx=Math.cos(ang);
13         if(cx!=0){
14             return Math.sin(ang)/cx;
15         }
16         else{
17             return -1;
18         }
19     }
20 }
21 }
```

5.8 Creación clase *THiloMov*

➡ A la izquierda en el panel de proyectos, en el nodo del proyecto (*ejemplo_hilos*) haga click derecho sobre el nodo **Paquetes de fuentes** y luego sobre las opciones **Nuevo** y **Java Class**, tal como se ilustra en la imagen de abajo:



- ➡ En esta ventana en la entrada **Class Name** ingrese **THiloMov** como nombre de la clase y haga click en el botón **Terminar**.



Nuevo Java Class

Pasos

1. Escoja el tipo de archivo
2. **Name and Location**

Name and Location

Class Name: THiloMov

Project: ejemplo_hilos

Location: Paquetes de fuentes

Package:

Created File: C:\Users\Lucho\Documents\NetBeansProjects\ejemplo_hilos

< Atrás Siguiete > **Terminar** Cancelar Ayuda

- ➡ Implemente la clase **THiloMov** (archivo **THiloMov.java**) como sigue:

```
1 public class THiloMov extends Thread {
2
3     private int Alto; //alto del contenedor
4     private int Ancho; //ancho del contenedor
5     private int Periodo; //tiempo(ms) o pausa entre cada paso o movimiento
6     private boolean Activo;
7     private TFuncion Func; //instancia de la funcion
8
9     public THiloMov(){
10         Alto=0;
11         Ancho=0;
12         Periodo=0;
13         Activo=false;
14         Func=null;
15     }
16
17     public void setAncho(int Anc){
18         Ancho=Anc;
19     }
20
21     public void setAlto(int Alt){
22         Alto=Alt;
23     }
24 }
```

```

24
25 public void setPeriodo(int Per){
26     if(Per>0){
27         Periodo=Per;
28     }
29 }
30
31 public void setFuncion(TFuncion Fun){
32     Func=Fun;
33 }
34
35 public int getPeriodo(){
36     return Periodo;
37 }
38
39 public boolean getActivo(){
40     return Activo;
41 }
42
43 public int getAlto(){
44     return Alto;
45 }
46
47 public int getAncho(){
48     return Ancho;
49 }
50
51 public TFuncion getFuncion(){
52     return Func;
53 }
54
55 //Detiene la ejecucion del hilo
56 public synchronized void Detener(){
57     if(Activo && isAlive()){ //Si el hilo no esta activo
58         Activo=false;
59     }
60 }
61
62 //Aqui se codifica la tarea que se hará en segundo plano
63 @Override
64 public void run(){
65     while(Activo){ //repetir siempre que activo sea true
66         if(!Func.Siguiente(Ancho,Alto)){
67             Func.getRangoX().Reiniciar(true);
68         }

```

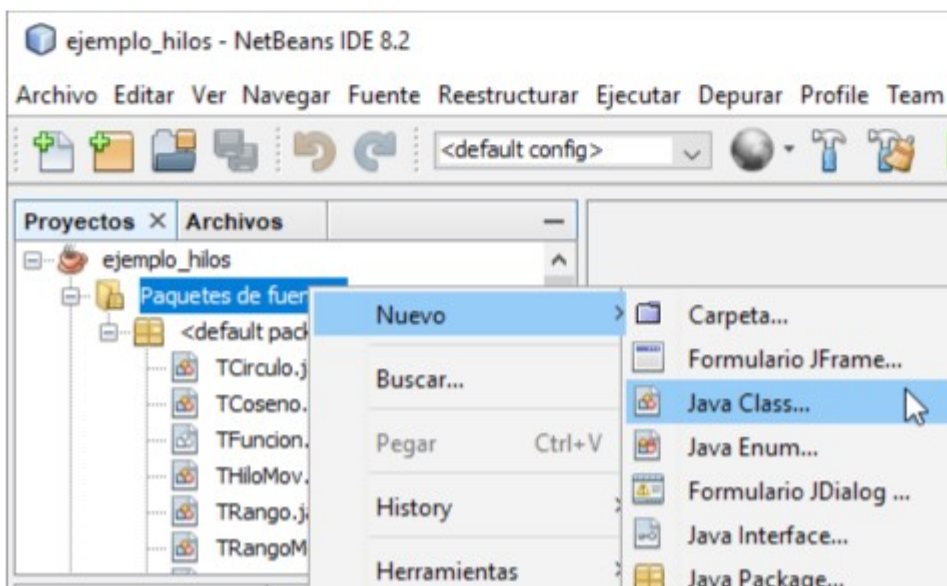
```

69     try{
70         Thread.sleep(Periodo); //Pausamos la ejecucion del ciclo
71     }
72     catch (InterruptedException Error){
73         Error.printStackTrace();
74         Detener();
75     }
76 }
77 }
78
79 public void Iniciar(){
80     try{
81         Detener();
82     }
83     finally{
84         Activo=true;
85         Func.getRangoX().Reiniciar(true);
86         start();//arranca formalmente el hilo
87     }
88 }
89
90 }

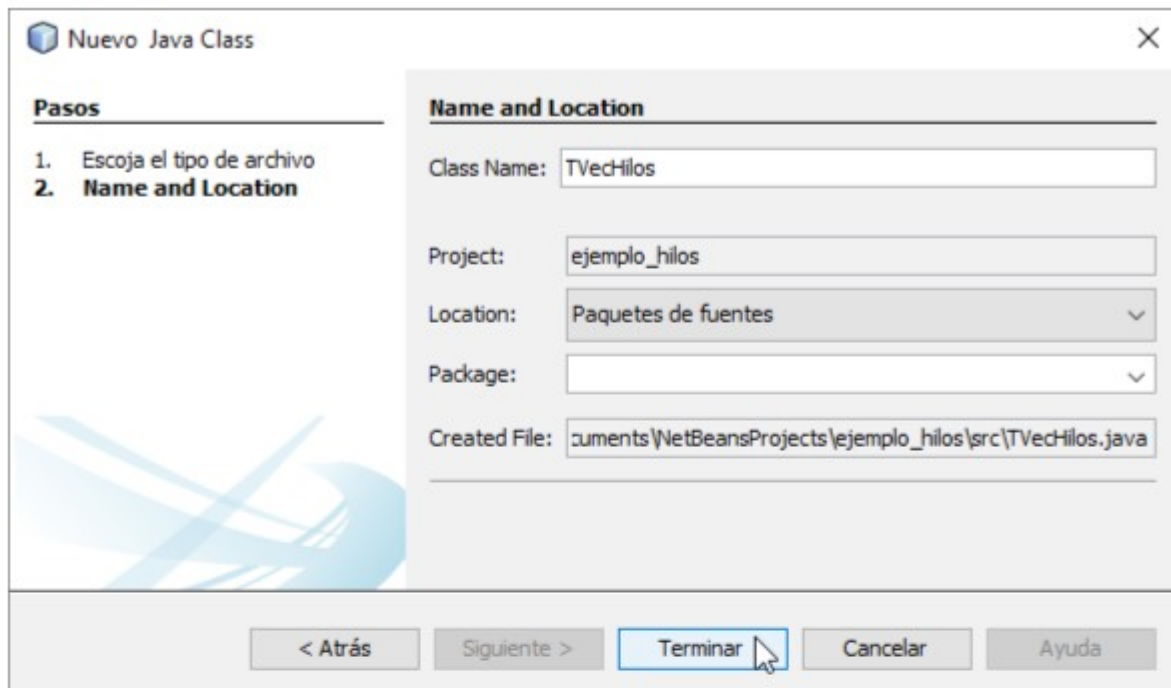
```

5.9 Creación clase *TVechilos*

- ➡ A la izquierda en el panel de proyectos, en el nodo del proyecto (**ejemplo_hilos**) haga click derecho sobre el nodo **Paquetes de fuentes** y luego sobre las opciones **Nuevo** y **Java Class**, tal como se ilustra en la imagen de abajo:



- ➡ En esta ventana en la entrada **Class Name** ingrese **TVecHilos** como nombre de la clase y haga click en el botón **Terminar**.



Nuevo Java Class

Pasos

1. Escoja el tipo de archivo
2. **Name and Location**

Name and Location

Class Name:

Project:

Location:

Package:

Created File:

< Atrás Siguiete > **Terminar** Cancelar Ayuda

- ➡ Implemente la clase **TVecHilos** (archivo *TVecHilos.java*) como sigue:

```
1 import java.util.ArrayList;
2
3 public class TVecHilos {
4
5     //Coleccion a arreglo de instancias de hilos
6     private ArrayList<THiloMov> Vec;
7
8     public TVecHilos(){
9         //Creamos el vector de hilos
10        Vec=new ArrayList<>();
11    }
12
13    public int Cantidad(){
14        return Vec.size();
15    }
16
17    public void Agregar(THiloMov HM){
18        //No se aceptan instancias nulas ni las ya agregadas
19        if(!Vec.contains(HM) && HM!=null){
20            Vec.add(HM);
21        }
22    }
23 }
```



```

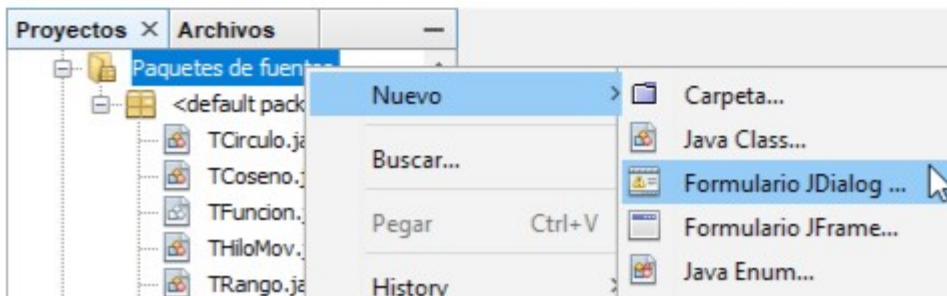
24 public THiloMov Agregar(){
25     THiloMov HM;
26     HM=new THiloMov();
27     Agregar(HM);
28     return HM;
29 }
30
31 public void Eliminar(THiloMov HM){
32     //Al eliminar un hilo detenemos su tarea
33     HM.Detener();
34     Vec.remove(HM);
35 }
36
37 public void Eliminar(int Pos){
38     Eliminar(Vec.get(Pos));
39 }
40
41 public THiloMov getHilo(int Pos){
42     return Vec.get(Pos);
43 }
44
45 public void Limpiar(){
46     int i,n;
47     n=Cantidad();
48     for(i=0;i<n;i++){
49         Eliminar(0);
50     }
51 }
52
53 }

```

6. Desarrollo dialogo *TCirculoDlg*

a) Creación del dialogo

- ➡ Haga click derecho en nodo **Paquetes de fuentes** del proyecto, escoja la opción **Nuevo** y luego **Formulario JDialog**, tal como se muestra en la siguiente imagen:



- ➡ En la ventana desplegada, en la entrada **Class Name**, ingrese **TCirculoDlg** y haga click en el botón **Terminar**.

Nuevo Formulario JDialog

Pasos

1. Escoja el tipo de archivo
2. **Name and Location**

Name and Location

Class Name:

Project:

Location:

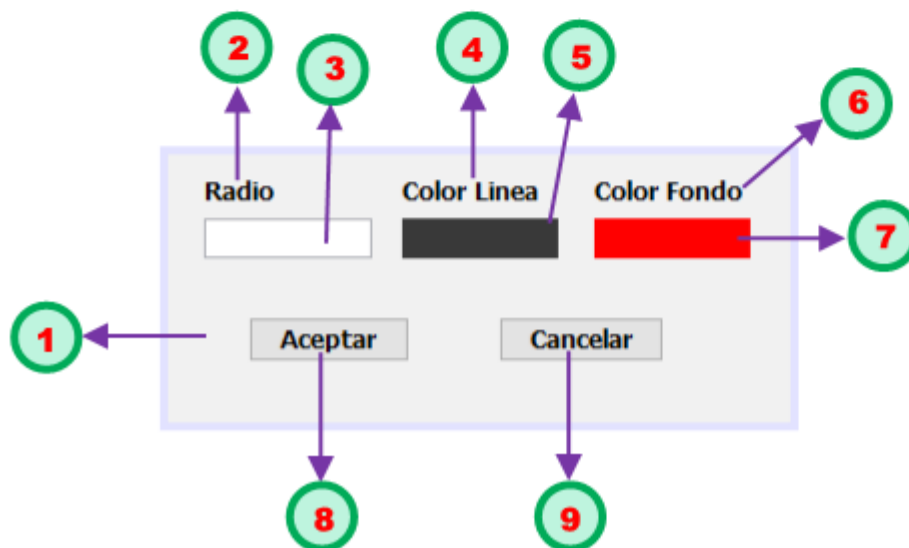
Package:

Created File:

< Atrás Siguiete > **Terminar** Cancelar Ayuda

b) Diseño gráfico de la ventana

Asegúrese de diseñar la ventana de la siguiente manera, tomando en cuenta las indicaciones descritas en la tabla más abajo y siguiendo el orden numérico.



c) Descripción de componentes del dialogo

Nº	Componente	Propiedad	Valor
1	JDialog	title	Propiedades del circulo
		Generar centro	<input checked="" type="checkbox"/> (Marcar la casilla)
2	JLabel	text	Radio
3	JTextField	<u>Nombre</u>	tf1
		text	Borre el texto y déjelo en blanco
4	JLabel	text	Color línea
5	JPanel	<u>Nombre</u>	pcl
		background	[0,0,0] Negro
6	JLabel	text	Color línea
7	JPanel	<u>Nombre</u>	pcf
		background	[255,0,0] Rojo
8	JButton	<u>Nombre</u>	B1
		text	Aceptar
9	JButton	<u>Nombre</u>	B2
		text	Cancelar

d) Definición de atributos e implementación de métodos del dialogo:

- ➡ Implemente la clase **TCirculoDlg** como sigue, pero considerando que no se muestran los métodos **initComponents** y **main**; particularmente este último puede borrarlo. Así mismo no se muestran otras partes generadas automáticamente por *NetBeans*, como por ejemplo los comentarios; por lo tanto, el código que se presenta enseguida no lo genera *NetBeans*, pues es el código que nosotros escribimos explícitamente, De esta manera es posible que la numeración de líneas no coincida con la de su archivo de código fuente; por ejemplo si usted no ha borrado los comentario.

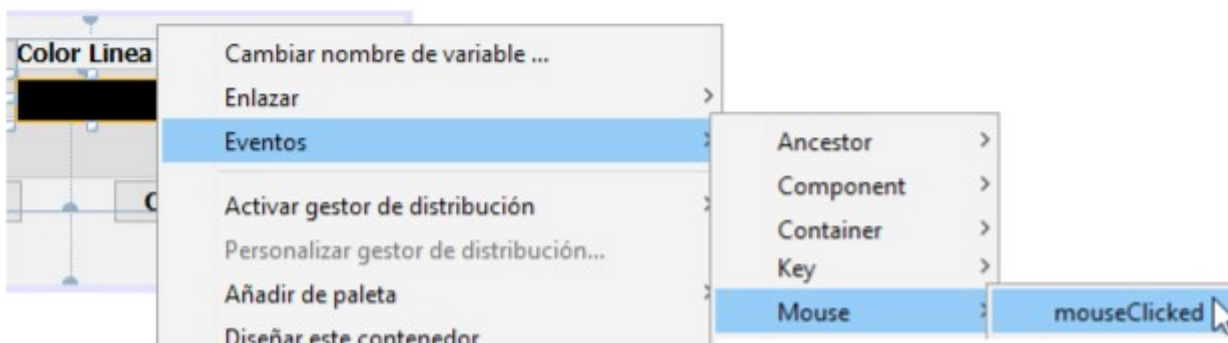
```

1 import java.awt.Color;
2 import javax.swing.JPanel;
3 import javax.swing.JColorChooser;
4
5 public class TCirculoDlg extends javax.swing.JDialog {
6
7     private TCirculo Cir;
8
9     private TCirculoDlg(java.awt.Frame parent, boolean modal) {
10         super(parent, modal);
11         initComponents();
12         Cir=null;
13     }
14
15     public TCirculoDlg(TCirculo Ci){
16         this(null,true);
17         Cir=Ci;
18         tf1.setText(Cir.getRadio()+"");
19         pcl.setBackground(Cir.getColLinea());
20         pcf.setBackground(Cir.getBackground());
21         setVisible(true);
22     }
23
24     private void PonColor(String Titulo,JPanel Px){
25         Color Col;
26         Col=JColorChooser.showDialog(null,Titulo,Px.getBackground());
27         if(Col!=null){
28             Px.setBackground(Col);
29         }
30     }

```

e) Implementación eventos del dialogo

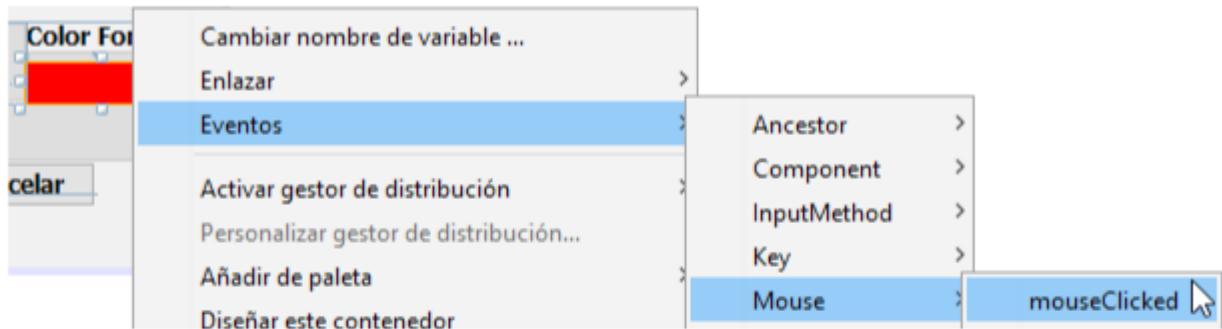
- ➡ Vaya al diseño de la ventana, haga click derecho sobre el panel **pcl** (el negro) y escoja las opciones **Eventos + Mouse + mouseClicked**:



- ➡ Implemente el evento para el panel de la siguiente manera:

```
private void pnlMouseClicked(java.awt.event.MouseEvent evt) {  
    PonColor("Color linea",pcl);  
}
```

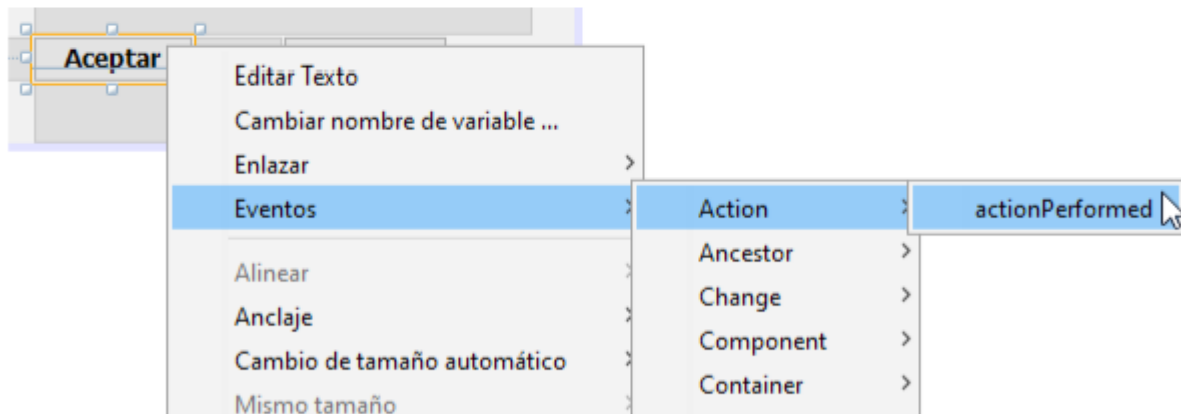
- ➡ Nuevamente pase al diseño de la ventana, haga click derecho sobre el panel **pcf** (el rojo) y escoja las opciones **Eventos + Mouse + mouseClicked**:



- ➡ Implemente el evento para este panel de la siguiente manera:

```
private void pcfMouseClicked(java.awt.event.MouseEvent evt) {  
    PonColor("Color fondo",pcf);  
}
```

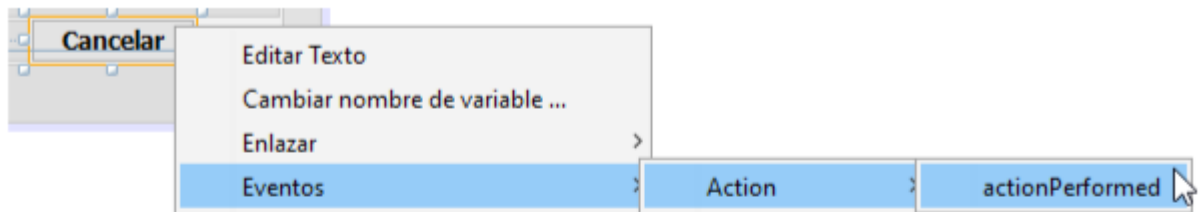
- ➡ Nuevamente vaya al diseño de la ventana, haga click sobre el botón **Aceptar** y escoja las opciones **Eventos + Action + actionPerformed**:



- ➡ El código para el evento de este botón es el siguiente:

```
private void blActionPerformed(java.awt.event.ActionEvent evt) {  
    Cir.setRadio(Integer.parseInt(tf1.getText()));  
    Cir.setColLinea(pcl.getBackground());  
    Cir.setBackground(pcf.getBackground());  
    dispose();  
}
```

- Otra vez vaya al diseño de la ventana, haga click sobre el botón **Cancelar** y escoja las opciones **Eventos + Action + actionPerformed** :



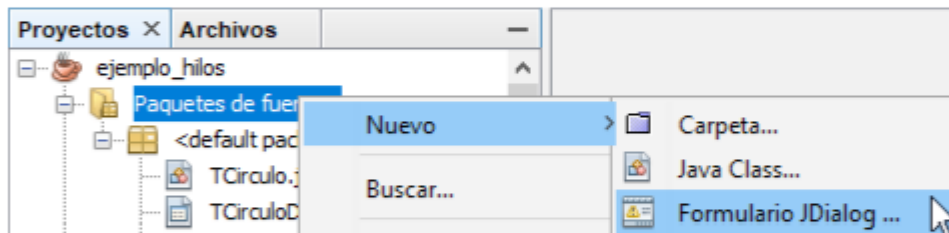
- El código para el evento de este botón es el siguiente:

```
private void B2ActionPerformed(java.awt.event.ActionEvent evt) {  
    dispose();  
}
```

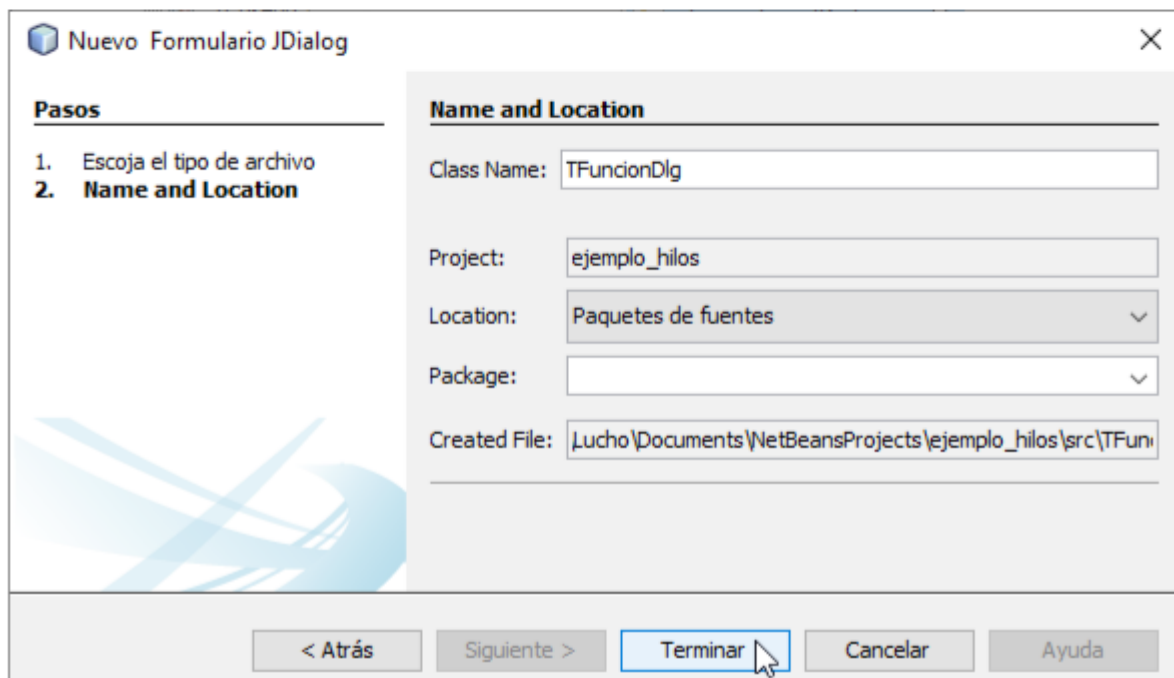
7. Desarrollo dialogo *TFuncionDlg*

a) Creación del dialogo

- Haga click derecho en nodo **Paquetes de fuentes** del proyecto, escoja la opción **Nuevo** y luego **Formulario JDialog**, tal como se muestra en la siguiente imagen:

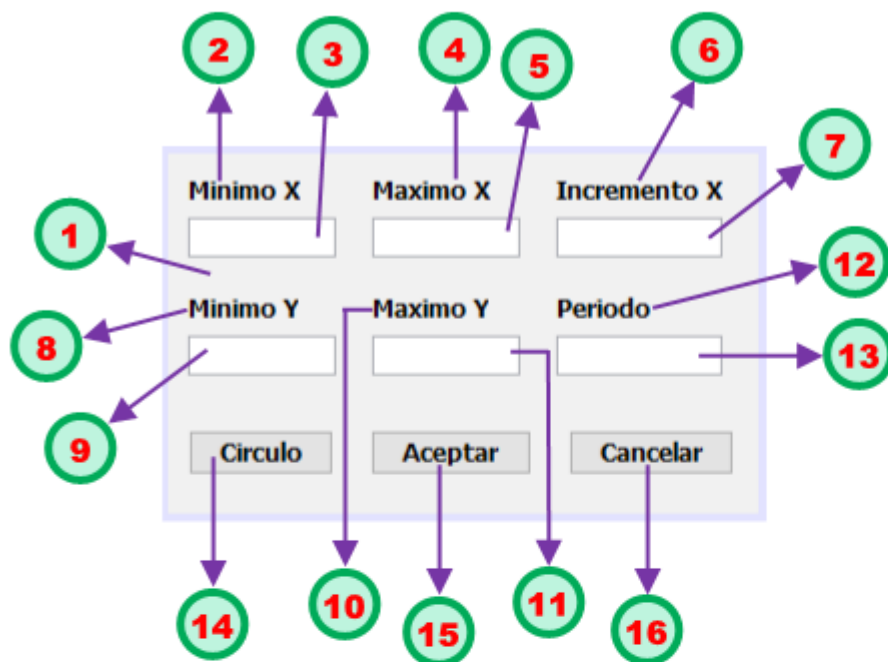


- En la entrada **Class Name**, ingrese **TFuncionDlg** y haga click en el botón **Terminar**.



b) Diseño gráfico de la ventana

Asegúrese de diseñar la ventana de la siguiente manera:



c) Descripción de componentes del dialogo

Nº	Com ponente	Propiedad	Valor
1	<i>JDialog</i>	Generar centro	<input checked="" type="checkbox"/> (Marcar la casilla)
2	<i>JLabel</i>	text	Mínimo X
3	<i>JTextField</i>	<u>Nombre</u>	tf1
		text	Borre el texto y déjelo en blanco
4	<i>JLabel</i>	text	Máximo X
5	<i>JTextField</i>	<u>Nombre</u>	tf2
		text	Borre el texto y déjelo en blanco
6	<i>JLabel</i>	text	Incremento X
7	<i>JTextField</i>	<u>Nombre</u>	tf3
		text	Borre el texto y déjelo en blanco

Nº	Componente	Propiedad	Valor
8	<i>JLabel</i>	text	Mínimo Y
9	<i>TextField</i>	<u>Nombre</u>	<i>tf4</i>
		text	<i>Borre el texto y déjelo en blanco</i>
10	<i>JLabel</i>	text	Máximo Y
11	<i>TextField</i>	<u>Nombre</u>	<i>tf5</i>
		text	<i>Borre el texto y déjelo en blanco</i>
12	<i>JLabel</i>	text	Periodo
13	<i>TextField</i>	<u>Nombre</u>	<i>tf6</i>
		text	<i>Borre el texto y déjelo en blanco</i>
14	<i>Button</i>	<u>Nombre</u>	<i>B1</i>
		text	Circulo
15	<i>Button</i>	<u>Nombre</u>	<i>B2</i>
		text	Aceptar
16	<i>Button</i>	<u>Nombre</u>	<i>B3</i>
		text	Cancelar

d) Definición de atributos e implementación de métodos del dialogo:

- ➡ Implemente la clase ***TFuncionDlg*** para este dialogo tal como se indica más abajo, de modo que no se muestra la codificación que *NetBeans* genera automáticamente; en especial a lo que respecta a los métodos ***initComponents*** y ***main***; igualmente como en el caso del dialogo anterior, bien puede borrar el método *main*, pues esta no va a ser ni la ventana ni la clase principal. En este punto, es importante recordar que no es necesario que el orden de las líneas de código, mostradas más adelante, coincida con el código escrito por usted; este orden puede variar por ejemplo por conservar, agregar, eliminar o modificar los comentarios del código.

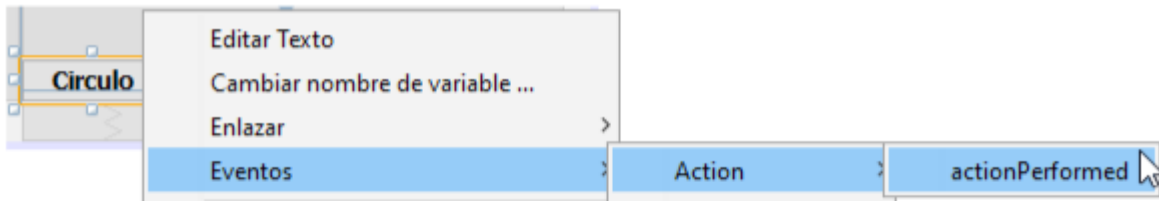

```

1 public class TFuncionDlg extends javax.swing.JDialog {
2
3     private THiloMov HM;
4
5     private TFuncionDlg(java.awt.Frame parent, boolean modal) {
6         super(parent, modal);
7         initComponents();
8     }
9
10    private void Mostrar(){
11        tf1.setText(HM.getFuncion().getRangoX().getMinimo()+"");
12        tf2.setText(HM.getFuncion().getRangoX().getMaximo()+"");
13        tf3.setText(HM.getFuncion().getRangoX().getIncremento()+"");
14        tf4.setText(HM.getFuncion().getRangoY().getMinimo()+"");
15        tf5.setText(HM.getFuncion().getRangoY().getMaximo()+"");
16        tf6.setText(HM.getPeriodo()+"");
17    }
18
19    private void Llenar(){
20        TRango Ry;
21        TRangoMov Rx;
22        Ry=HM.getFuncion().getRangoY();
23        Rx=HM.getFuncion().getRangoX();
24        Rx.setMinimo(Double.parseDouble(tf1.getText()));
25        Rx.setMaximo(Double.parseDouble(tf2.getText()));
26        Rx.setIncremento(Double.parseDouble(tf3.getText()));
27        Ry.setMinimo(Double.parseDouble(tf4.getText()));
28        Ry.setMaximo(Double.parseDouble(tf5.getText()));
29        HM.setPeriodo(Integer.parseInt(tf6.getText()));
30    }
31
32    public static boolean Configurar(String Titulo, THiloMov HilMov){
33        TFuncionDlg Dlg;
34        Dlg=new TFuncionDlg(null,true);
35        Dlg.HM=HilMov;
36        Dlg.Mostrar();
37        Dlg.setTitle("Funcion " + Titulo);
38        Dlg.setVisible(true);
39        return Dlg.HM==HilMov;
40    }

```

e) Implementación eventos del dialogo

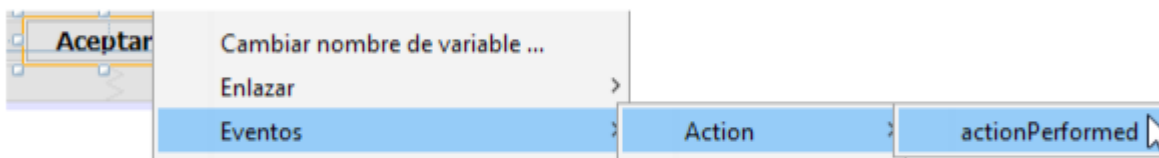
- ➡ Vaya al diseño de la ventana, haga click derecho sobre el botón **Circulo** y escoja las opciones **Eventos + Action + actionPerformed**:



- ➡ Implemente el evento para este botón de la siguiente manera:

```
private void B1ActionPerformed(java.awt.event.ActionEvent evt) {  
    new TCirculoDlg(HM.getFuncion().getCirculo());  
}
```

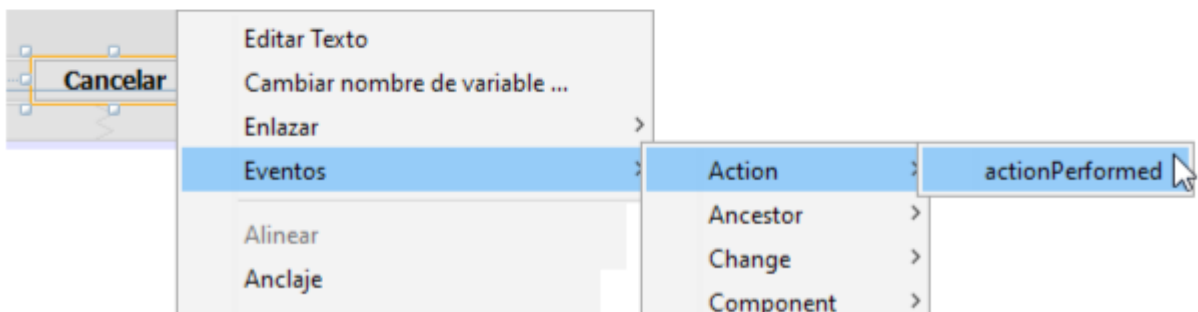
- ➡ Nuevamente pase al diseño de la ventana, haga click derecho sobre botón **Aceptar** y seleccione las opciones **Eventos + Action + actionPerformed**:



- ➡ Implemente el evento para el botón de la siguiente manera:

```
private void B2ActionPerformed(java.awt.event.ActionEvent evt) {  
    Llenar();  
    dispose();  
}
```

- ➡ Regrese al diseño de la ventana, haga click sobre el botón **Cancelar** y escoja las opciones **Eventos + Action + actionPerformed**:



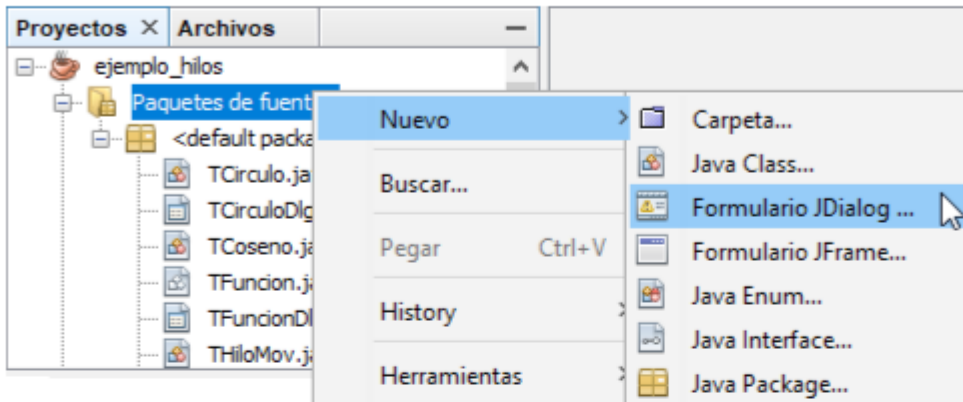
- ➡ El código para el evento de este botón es el siguiente:

```
private void B3ActionPerformed(java.awt.event.ActionEvent evt) {  
    HM=null;  
    dispose();  
}
```

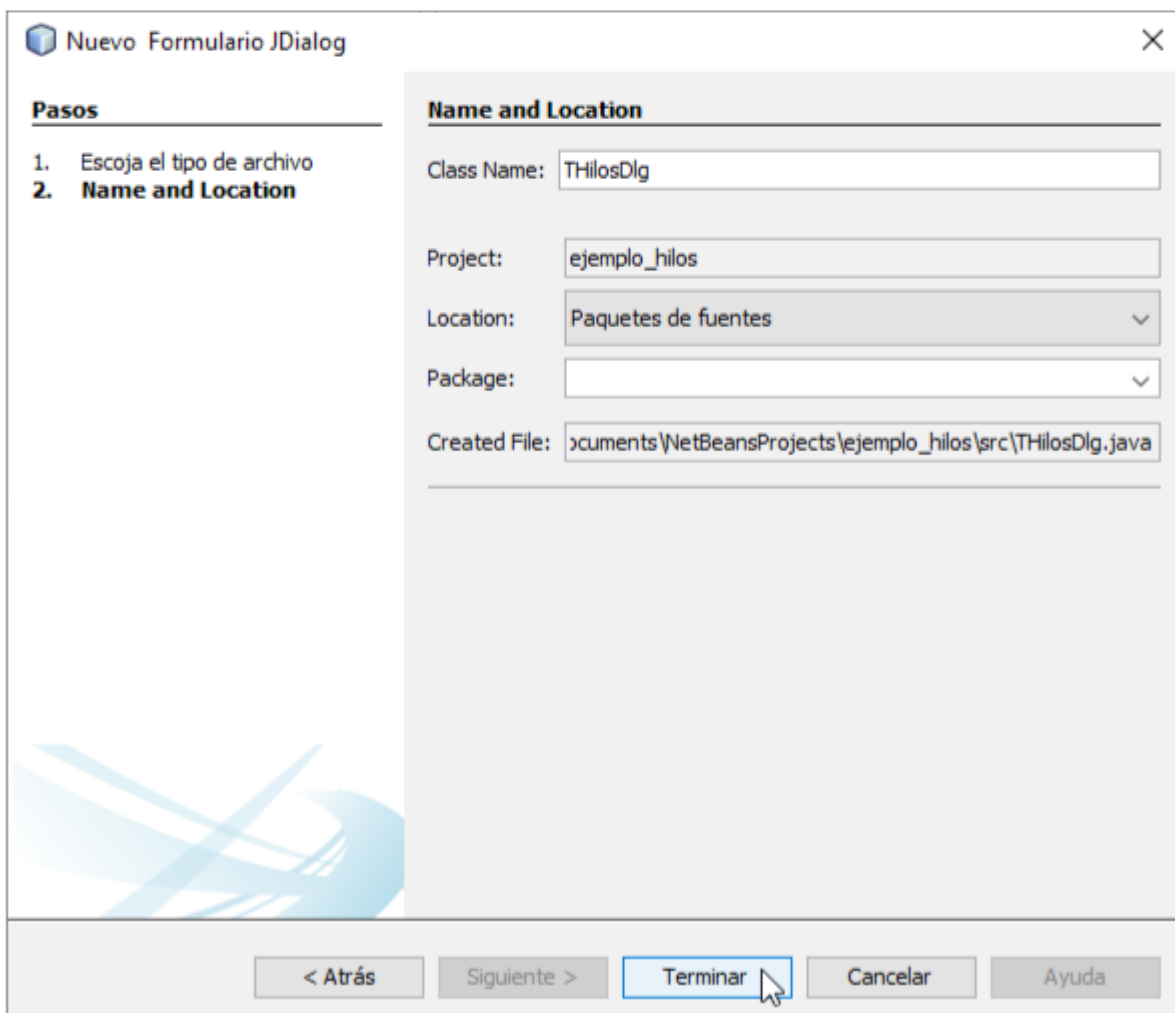
8. Desarrollo dialogo *THilosDlg*

a) Creación del dialogo

- ➡ Haga click derecho en nodo **Paquetes de fuentes** del proyecto, escoja la opción **Nuevo** y luego **Formulario JDialog**, tal como se muestra en la siguiente imagen:

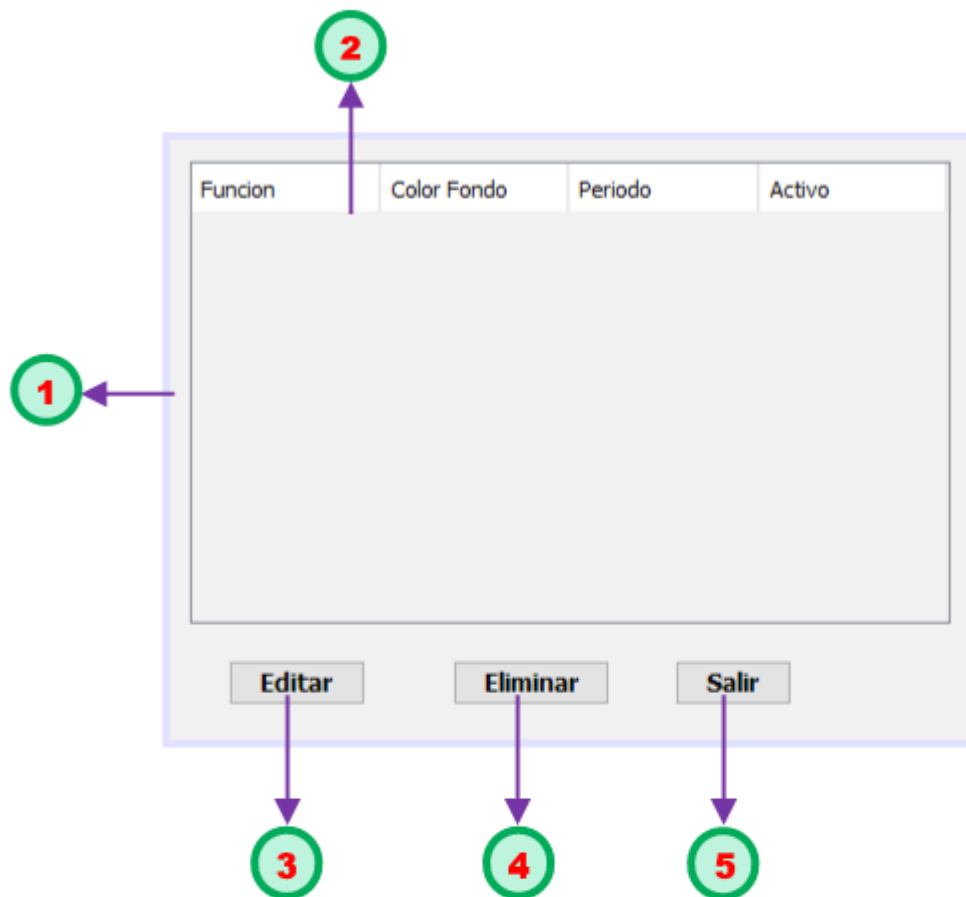


- ➡ En la entrada **Class Name**, ingrese **THilosDlg** y haga click en el botón **Terminar**.



b) Diseño gráfico de la ventana

Asegúrese de diseñar la ventana de la siguiente manera:



c) Descripción de componentes del dialogo

Nº	Componente	Propiedad	Valor
1	JDialog	title	<i>Hilos y funciones</i>
		Generar centro	<input checked="" type="checkbox"/> (Marcar la casilla)
2	JTable	<u>Nombre</u>	<i>tab</i>
		model	<i>Verlo en la imagen más abajo</i>
3	JButton	<u>Nombre</u>	<i>B1</i>
		text	Editar

Nº	Componente	Propiedad	Valor
4	JButton	<u>Nombre</u>	B2
		text	Eliminar
5	JButton	<u>Nombre</u>	B3
		text	Salir

Asegúrese que el modelo de datos para la tabla quede definido de modo que no tenga filas iniciales, pero que contenga cuatro columnas, acorde a la configuración mostrada en la siguiente imagen:

tab [JTable] - model

Establecer propiedad **tab's model** utilizando: Personalizador del modelo de tabla

Modelo de tabla

Ajustes de la tabla | Valores predeterminados

Indicar los tipos del título y de la columna aquí:

Columna	Título	Tipo	Editable
1	Funcion	String	<input type="checkbox"/>
2	Color Fondo	String	<input type="checkbox"/>
3	Periodo	Integer	<input type="checkbox"/>
4	Activo	Boolean	<input type="checkbox"/>

Filas: 0 + - Columnas: 4 + -

Insertar
Eliminar
Subir
Bajar

Aceptar Restablecer Valores por Defecto Cancelar

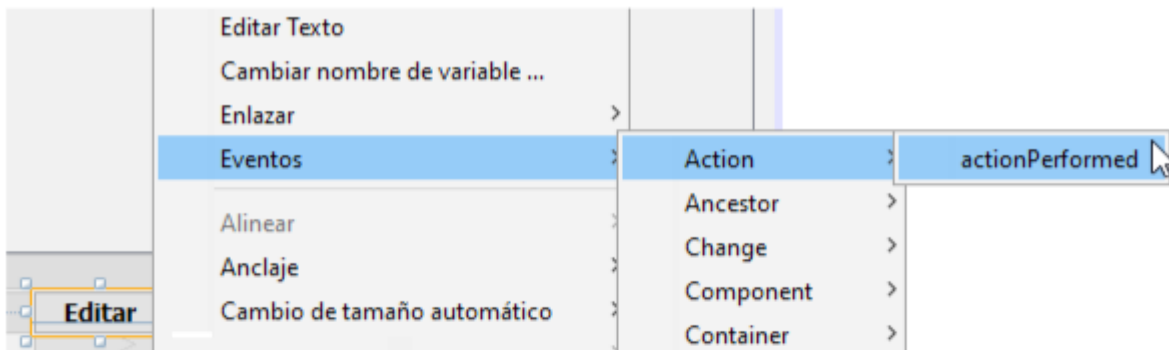
d) Definición de atributos e implementación de métodos del dialogo:

Implemente la clase **THilosDlg** para este dialogo, el cual no muestra los métodos **initComponents** y **main**; tal como se muestra enseguida:

```
1 import javax.swing.table.DefaultTableModel;
2
3 public class THilosDlg extends javax.swing.JDialog {
4
5     private TVecHilos VH;
6     private DefaultTableModel Mod;
7
8     private THilosDlg(java.awt.Frame parent, boolean modal){
9         super(parent,modal);
10        initComponents();
11        Mod=(DefaultTableModel)tab.getModel();
12    }
13
14    private void Mostrar(int Fila,THiloMov HM){
15        String Txt;
16        Txt=HM.getFuncion().getClass().getName();
17        Mod.setValueAt(Txt,Fila,0);
18        Txt=HM.getFuncion().getCirculo().getBackground()+"";
19        Mod.setValueAt(Txt.substring(Txt.indexOf("T")),Fila,1);
20        Mod.setValueAt(HM.getPeriodo(),Fila,2);
21        Mod.setValueAt(HM.getActivo(),Fila,3);
22    }
23
24    private void Mostrar(){
25        int i,N;
26        N=VH.Cantidad();
27        Mod.setRowCount(N);
28        for(i=0;i<N;i++){
29            Mostrar(i,VH.getHilo(i));
30        }
31    }
32
33    public THilosDlg(TVecHilos VecHil){
34        this(null,true);
35        VH=VecHil;
36        Mostrar();
37        setVisible(true);
38    }
```

e) Implementación eventos del dialogo

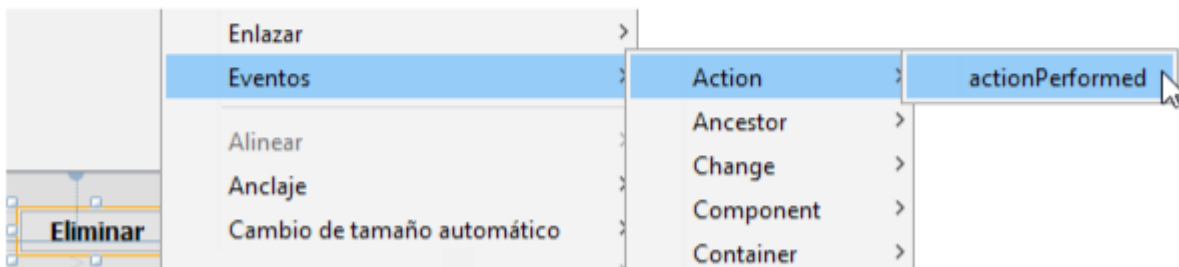
- ➡ Vaya al diseño de la ventana, haga click derecho sobre el botón **Editar** y escoja las opciones **Eventos + Action + actionPerformed**:



- ➡ Implemente el evento para este botón de la siguiente manera:

```
private void B1ActionPerformed(java.awt.event.ActionEvent evt) {  
    int Fila;  
    THiloMov HM;  
    Fila=tab.getSelectedRow();  
    if(Fila>=0){  
        HM=VH.getHilo(Fila);  
        if(TFuncionDlg.Configurar(tab.getValueAt(Fila,0)+"" ,HM)){  
            Mostrar(Fila,HM);  
        }  
    }  
}
```

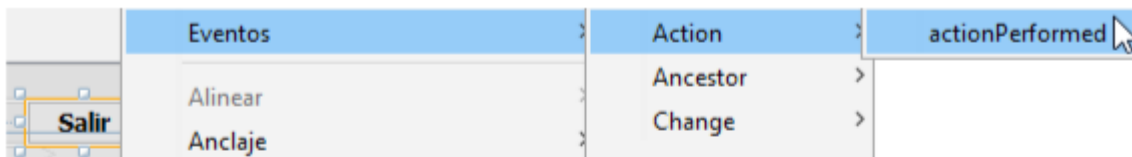
- ➡ Nuevamente pase al diseño de la ventana, haga click derecho sobre botón **Eliminar** y seleccione las opciones **Eventos + Action + actionPerformed**:



- ➡ Implemente el evento para el botón de la siguiente manera:

```
private void B2ActionPerformed(java.awt.event.ActionEvent evt) {  
    int Fila=tab.getSelectedRow();  
    if(Fila>=0){  
        Mod.removeRow(Fila);  
    }  
}
```

- Regrese al diseño de la ventana, haga click sobre el botón **Salir** y escoja las opciones **Eventos + Action + actionPerformed** :



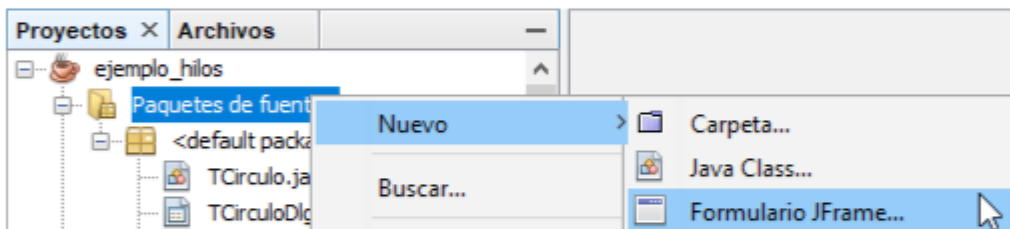
- El código para el evento de este botón es el siguiente:

```
private void B3ActionPerformed(java.awt.event.ActionEvent evt) {  
    dispose();  
}
```

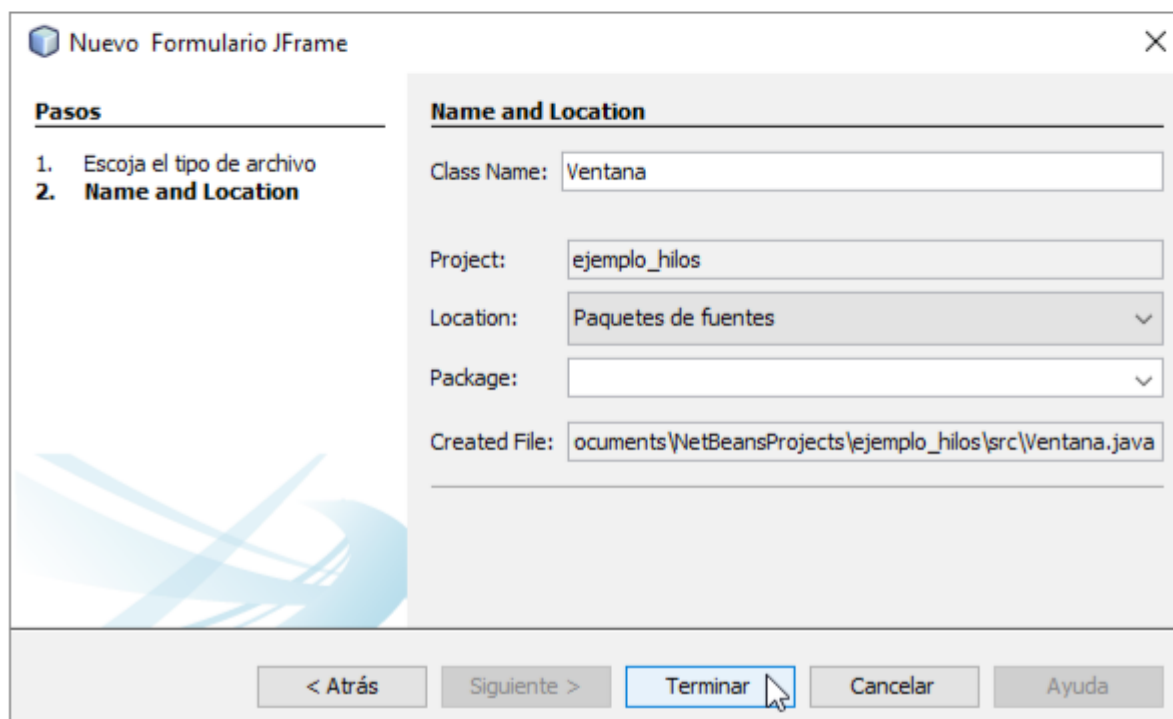
9. Desarrollo Ventana principal

a) Creación del formulario

- Haga click derecho en nodo **Paquetes de fuentes** del proyecto, escoja la opción **Nuevo** y luego **Formulario JFrame**, tal como se muestra en la siguiente imagen:

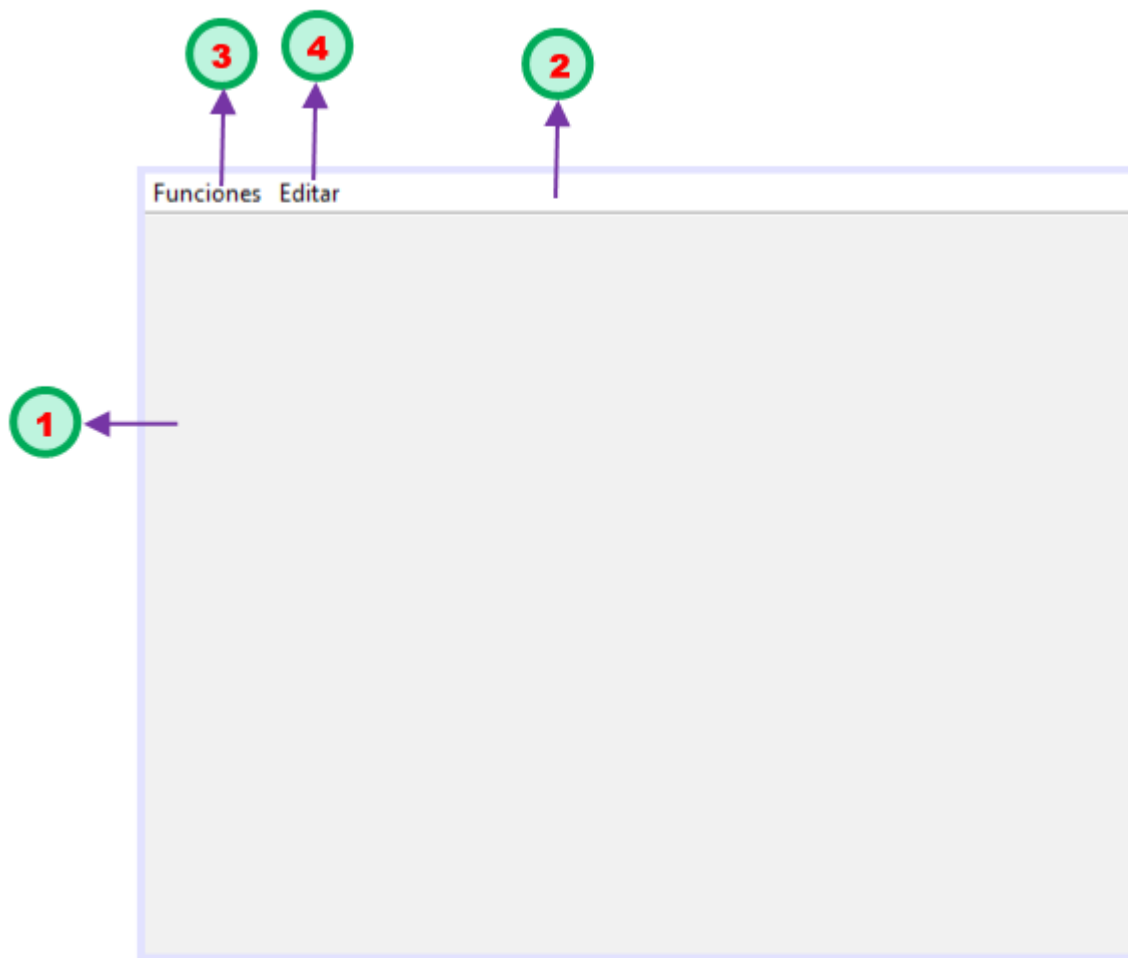


- En la entrada **Class Name**, ingrese **Ventana** y haga click en el botón **Terminar**.



b) Diseño gráfico de la ventana

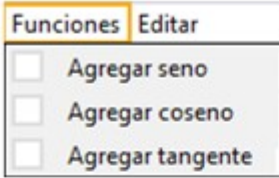
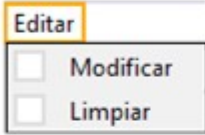
Asegúrese de diseñar la ventana de la siguiente manera:



c) Descripción de componentes del dialogo

Nº	Componente	Propiedad	Valor
1	JFrame	title	<i>Ejemplo hilos de ejecución</i>
		Generar centro	<input checked="" type="checkbox"/> (Marcar la casilla)
2	JMenuBar	<u>Nombre</u>	Bar
3	JMenu	text	Funciones
4	JMenu	text	Pila

➡ Las opciones de los menús **Matriz** y **Fila** se describen seguidamente:

Nº	Menú	Opciones	Propiedad	Valor
3		JMenuItem	text	Agregar seno
		JMenuItem	text	Agregar coseno
		JMenuItem	text	Agregar tangente
4		JMenuItem	text	Modificar
		JMenuItem	text	Limpiar

d) Definición de atributos e implementación de métodos del formulario:

Implemente la clase Ventana de la siguiente manera

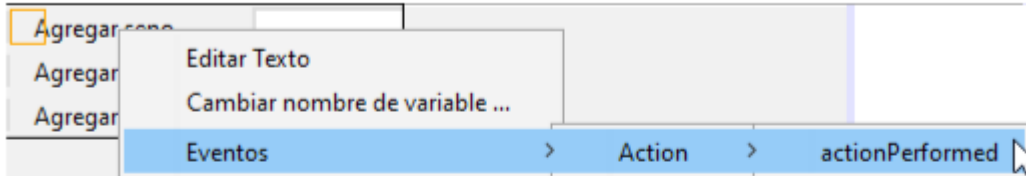
```

1 public class Ventana extends javax.swing.JFrame {
2
3     private TVecHilos VH;
4
5     public Ventana(){
6         initComponents();
7         VH=new TVecHilos();
8     }
9
10    private void PonTamaño(THiloMov HM){
11        HM.setAncho(getWidth());
12        HM.setAlto(getHeight()-Bar.getHeight()-26);
13        getContentPane().add(HM.getFuncion().getCirculo());
14    }
15
16    private void AgregarHilo(String Titulo,TFuncion Fun){
17        THiloMov HM;
18        HM=new THiloMov();
19        HM.setFuncion(Fun);
20        if(TFuncionDlg.Configurar(Titulo,HM)){
21            PonTamaño(HM);
22            VH.Agregar(HM);
23            HM.Iniciar();
24        }
25    }

```

e) Implementación eventos del formulario

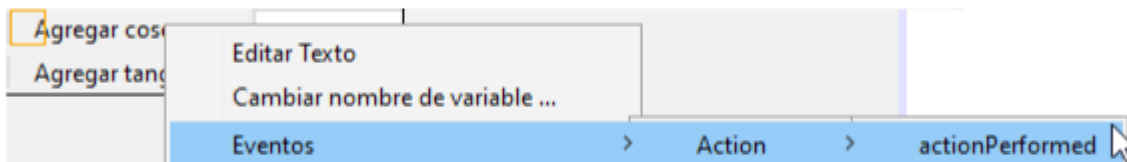
- ➡ Vaya al diseño de la ventana, haga click en la opción de menú **Funciones**, seleccione la opción **Agregar seno**, luego haga click derecho ella y escoja las opciones **Eventos** + **Action** + **actionPerformed** :



- ➡ El código para el evento de esta opción de menú es el siguiente:

```
private void jMenuItem1ActionPerformed(java.awt.event.ActionEvent evt) {  
    AgregarHilo("Seno",new TSeno());  
}
```

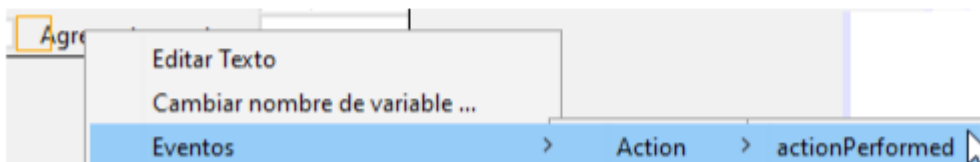
- ➡ Ahora haga click en la opción de menú **Funciones** seleccione la opción **Agregar coseno**, luego haga click derecho ella y escoja las opciones **Eventos** + **Action** + **actionPerformed** :



- ➡ El código para el evento de esta opción de menú es el siguiente:

```
private void jMenuItem2ActionPerformed(java.awt.event.ActionEvent evt) {  
    AgregarHilo("Coseno",new TCoseno());  
}
```

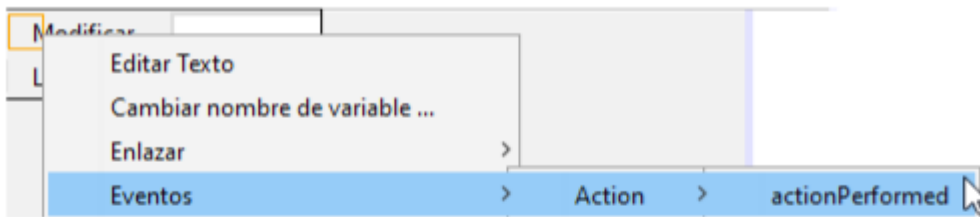
- ➡ Haga click en la opción de menú **Funciones**, seleccione la opción **Agregar tangente**, luego haga click derecho ella y escoja las opciones **Eventos** + **Action** + **actionPerformed** :



- ➡ El código para el evento de esta opción de menú es el siguiente:

```
private void jMenuItem3ActionPerformed(java.awt.event.ActionEvent evt) {  
    AgregarHilo("Tangente",new TTangente());  
}
```

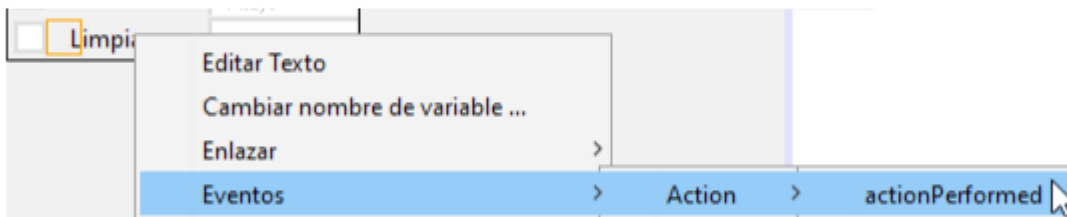
- ➡ Haga click en la opción de menú **Editar**, seleccione la opción **Modificar**, luego haga click derecho ella y escoja las opciones **Eventos + Action + actionPerformed** :



- ➡ El código para el evento de esta opción de menú es el siguiente:

```
private void jMenuItem4ActionPerformed(java.awt.event.ActionEvent evt) {  
    new THilosDlg(VH);  
}
```

- ➡ Haga click en la opción de menú **Editar**, seleccione la opción **Limpiar**, luego haga click derecho ella y escoja las opciones **Eventos + Action + actionPerformed** :

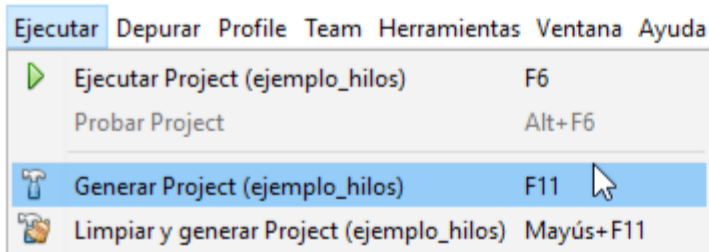


- ➡ El código para el evento de esta opción de menú es el siguiente:

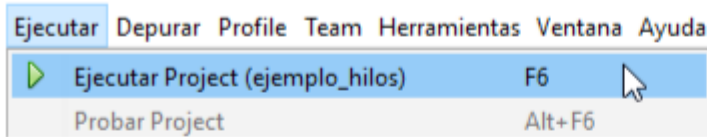
```
private void jMenuItem5ActionPerformed(java.awt.event.ActionEvent evt) {  
    VH.Limpiar();  
}
```

10. Compilación y ejecución del programa

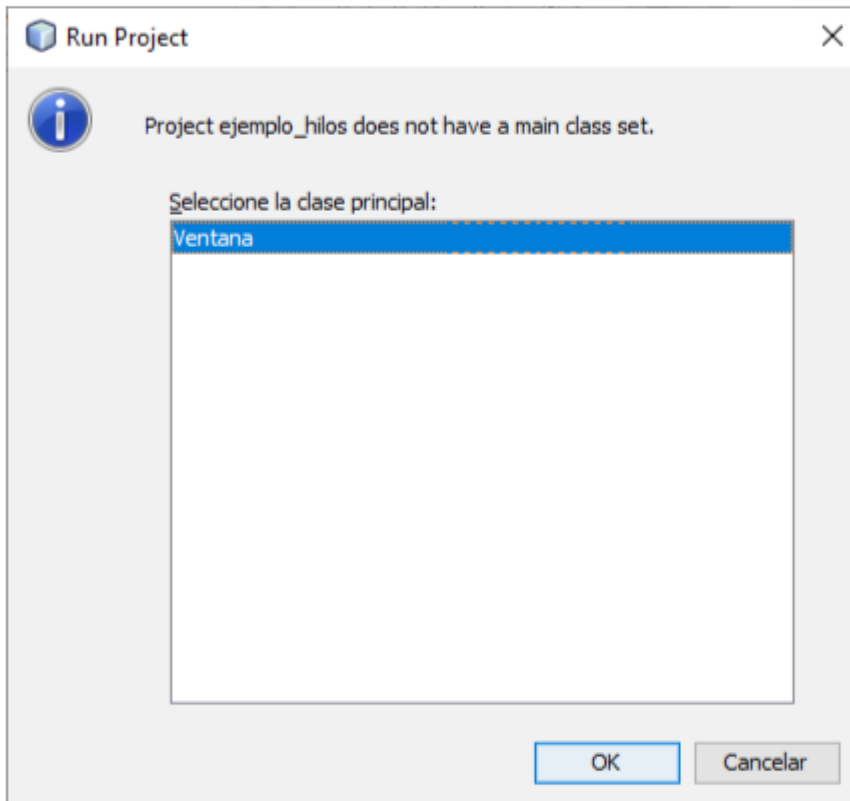
- ➡ Para compilar el proyecto seleccione la opción de menú **Ejecutar** y escoja la opción **Generar Project (ejemplo_hilos)** o pulse la tecla F11, tal como se indica abajo:



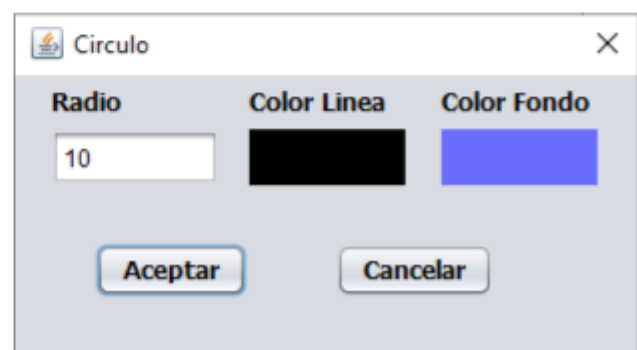
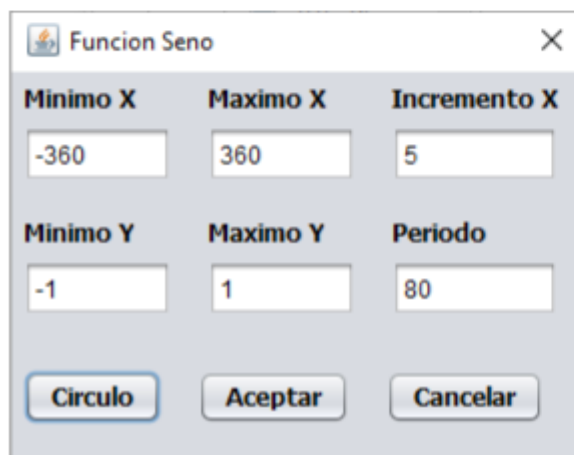
- Una vez corregido cualquier error ejecute el programa con la opción de menú **Ejecutar** y **Ejecutar Project (ejemplo_hilos)** o pulse la tecla F6.



- Cuando ejecuta el programa por primera vez, verá este cuadro de dialogo, de modo que escoja a **Ventana** como clase principal y haga click en el botón **OK**:



- Las siguientes imágenes son algunas capturas de la ejecución del programa.

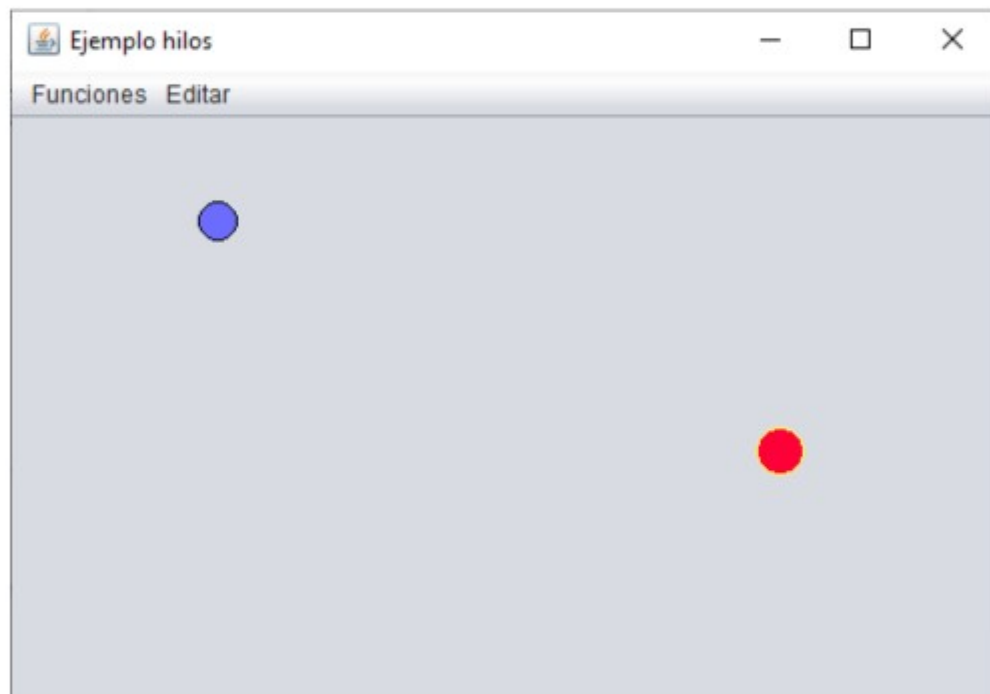


Funcion Coseno

Minimo X	Maximo X	Incremento X
-180	180	4
Minimo Y	Maximo Y	Periodo
-1	1	70

Circulo

Radio	Color Linea	Color Fondo
12		

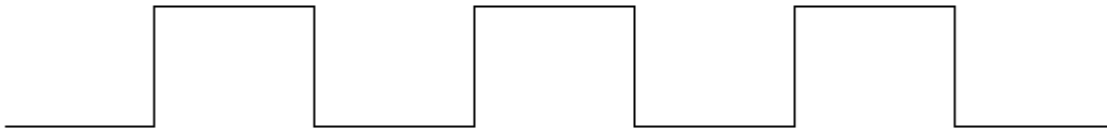


Hilos y funciones

Funcion	Color Fondo	Periodo	Activo
TSeno	[r=102,g=102,b=255]	80	<input checked="" type="checkbox"/>
TCoseno	[r=255,g=0,b=51]	70	<input checked="" type="checkbox"/>

Actividades propuestas

1. Modifique la implementación del método *run* de la clase **THiloMov**, de tal suerte que la figura se mueva recorriendo el dominio (atributo *RangoX*) de la función en ambos sentidos; es decir, inicialmente moviendo de izquierda a derecha (recorriendo desde el mínimo **X** hasta el máximo **X**); y una vez que llegue al extremo derecho se devuelve de derecha a izquierda (desde el máximo **X** hasta el mínimo **X**). Ahora bien, cuando llegue al extremo izquierdo, nuevamente se moverá al derecho y así sucesivamente.
2. Modifique la implementación del método *Centrar* de la clase **TCirculo**, de manera que reajuste las posición de la figura, tal que siempre este visible en forma completa (que no se vea oculta ninguna parte del circulo); es decir, que todo el circulo siempre se encuentre dentro del área visible de la ventana.
3. Diseñe en UML e implemente una clase hija para una función que mueva en forma lineal (ecuación $y=mx + b$) la figura, incluyendo su correspondiente dialogo de captura de datos.
4. Diseñe en UML e implemente una clase hija para una función que mueva el círculo siguiendo la trayectoria de una onda cuadrada.



5. Diseñe en UML e implemente una clase hija de **THiloMov** que agregue una segunda función, de manera que el hilo mueva dos círculos al tiempo, pero considerando que estos dos círculos se mueven describiendo la figura de la letra **X**; es decir, un circulo para cada línea de la letra.
6. Diseñe en UML e implemente una clase para un hilo que sincronice 8 círculos, que se mueven en pares entrando (acercándose) y saliendo (alejándose) de un punto común (el de color amarillo); cada circulo de un par sigue una dirección opuesta respecto al otro, pero siempre sobre la misma línea; según las trayectorias mostradas abajo:

