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

Gregorio Quintana ©

Departamento de Ingeniería y Ciencia de Computadores
Universidad Jaume I

Curso 2025–26

# Contenido

1. *Thread Pools*
2. Diseño de Interfaces Gráficas Multihebra

# Bibliografía

- Oaks. Capítulo 10 *"Thread Pools"*.
- Goetz. Capítulo 8 *"Applying Thread Pools"*.
- Oaks. Capítulo 7 *"Threads and Swing"*.
- Goetz. Capítulo 9 *"GUI Applications"*.

# Dos Problemas Típicos

- Problema 1: Contar cuántos números primos ha tecleado el usuario, aprovechando el sistema, pero sin saturarlo.

  En algún momento el usuario podría teclear 5000 números primos muy altos y en otros momentos podría irse a almorzar.

- Problema 2: Una aplicación gráfica con una barra de progreso de una tarea y un botón para cancelar la tarea.

# Objetivos

1. Ser capaz de emplear *Thread Pools* para resolver problemas paralelos con un número de tareas muy variable y con un instante de llegada imprevisible.
2. Ser capaz de diseñar e implementar interfaces gráficas que respondan rápidamente.

# *Thread Pools*

# El Problema de las Cargas de Tareas Masivas e Imprevistas

- En muchas aplicaciones hay que procesar en paralelo una cantidad de tareas imprevistas que en algunos momentos pueden llegar a ser enorme.
- Ejemplo: Servidores *web*.
- Ante este problema (y otros), cuando se trabaja con hebras, las dos alternativas son las siguientes:
  1. Creación de una hebra por cada tarea.
  2. Creación de un número fijo de hebras.

# El Problema de las Cargas de Tareas Masivas e Imprevistas (Cont.)

- **Primera alternativa: Creación de una hebra por cada tarea:**
  - Ventaja: La programación es muy sencilla.
  - Inconveniente 1: Puede conllevar la creación de un número excesivo de hebras (incluso miles de hebras).

    Puede provocar el bloqueo total o la saturación del sistema, con miles de hebras creadas, cada una de las cuales recibe muy poca CPU.
  - Inconveniente 2: El sobrecoste de crear y destruir una hebra para cada tarea puede ser prohibitivo si el coste de cada tarea es muy bajo.

# El Problema de las Cargas de Tareas Masivas e Imprevistas (Cont.)

- **Segunda alternativa: Creación de un número fijo de hebras:**
  - Ventaja: El sistema no se satura con facilidad (si el número de hebras es reducido).
  - Inconveniente 1: Si se crea un número fijo de hebras antes de comenzar, puede que sean pocas.
  - Inconveniente 2: Si alguna hebra termina abruptamente (por ejemplo debido a un desbordamiento), ninguna la reemplazará y las prestaciones del sistema se reducirán.
  - Inconveniente 3: La programación es más complicada.

# El Problema de las Cargas de Tareas Masivas e Imprevistas (Cont.)

- La primera alternativa (**una hebra por tarea**) no debería abordarse con hebras plataforma. ¿Por qué?

  Para emplear una hebra por tarea en tareas limitadas por la E/S sin los inconvenientes antes mencionados, Java 19 ha aportado el concepto de hebras virtuales.

  - Se pueden crear tantas como se quieran.
  - Netflix.

- Para emplear la segunda alternativa (**creación de un número fijo de tareas**) sin los inconvenientes antes mencionados, Java 8 ha aportado el concepto de *Thread Pool*.

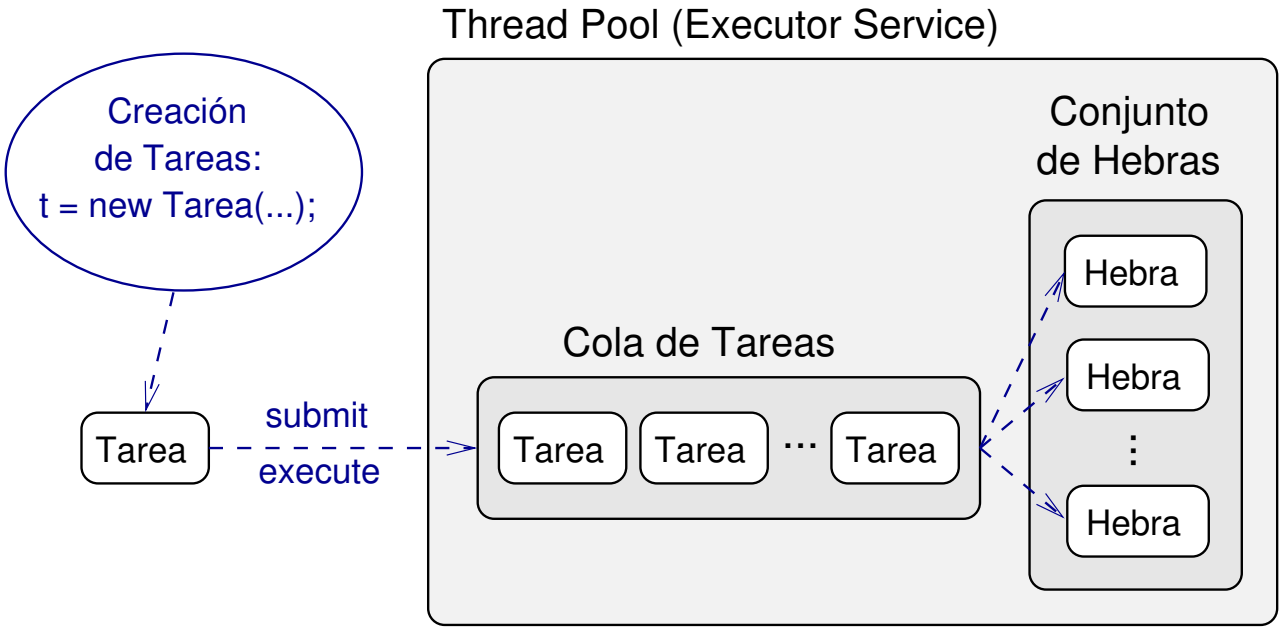
  - Gestión de tareas con reparto dinámico de forma muy sencilla.

# *Thread Pools* en Java

- El concepto de *Thread Pool* facilita la labor del programador cuando desea un número fijo de hebras.
- Concepto muy extendido. Por ejemplo, Twitter los emplea en muchos de sus servicios.
- El *Thread Pool* combina dos herramientas:
  1. Conjunto de hebras.
  2. Cola de tareas.
- Ventajas del *Thread Pool*:
  1. Las hebras se reutilizan.
  2. Programación mucho más sencilla.
- El *Thread Pool* ofrece mucha verstatilidad, tanto sobre las hebras (número habitual, número máximo, etc.) como sobre la cola (limitada, ilimitada, con prioridades, etc.).

# Esquema de un *Thread Pool*

- Las tareas primero se crean y después se encolan.
  De todo lo demás, se encarga el *Thread Pool*.




- Reparto dinámico: En cuanto una hebra del *Thread Pool* queda libre, automáticamente coge la siguiente tarea y comienza a ejecutarla.

# Coste Mínimo de la Tarea

- El *Thread Pool* de Java es una herramienta muy potente y versátil. Su implementación es muy eficiente.
- Sin embargo, el *Thread Pool* tiene varios sobrecostes:
  1. La gestión de la cola:
     1. El coste de insertar la tarea en la cola.
     2. El coste de extraer la tarea de la cola.
  2. La gestión de las hebras:
     1. El coste de despertar las hebras.
     2. El coste de dormir las hebras.
- Si la tarea tiene un coste muy bajo, los sobrecostes anularán el posible beneficio de la paralelización.
- Ejemplo de tarea: Determinar si un carácter es un blanco.

# Métodos Más Importantes para Encolar Tareas

- Método `execute( Runnable task )`: Encola una tarea en el *Thread Pool*. La tarea será ejecutada en cuanto una hebra esté disponible.
- Método `submit( Callable<V> task )`: Similar al anterior, pero este método devuelve un objeto `Future<V>` para poder seguirle la pista a la tarea.

# Métodos Más Importantes para Terminar

- Método `shutdown()`: El *Thread Pool* no acepta más tareas y terminará en cuanto todas las tareas pendientes (encoladas y en ejecución) sean procesadas.
- Método `shutdownNow()`: El *Thread Pool* no acepta más tareas y tampoco termina las encoladas. Muy drástico.
- Método `boolean isTerminated()`: Indica si el *Thread Pool* ha concluido: No hay tareas pendientes (ni encoladas ni en ejecución).
- Método `boolean awaitTermination( long timeout, TimeUnit unit )`: Se bloquea hasta que todas las hebras del *Thread Pool* hayan terminado o hasta transcurrido el tiempo indicado (lo que antes ocurra).
  Devuelve cierto si todas las hebras han terminado y falso si el tiempo se ha agotado.

# Terminación de un *Thread Pool*

- Método muy malo: ¿Por qué?

```
1  // Codigo ejecutado por la hebra principal.
2  exec.shutdown();
3  while( ! exec.isTerminated() ) {
4  }
```

- Método mejor:

```
1  // Codigo ejecutado por la hebra principal.
2  exec.shutdown();
3  try {
4    while( ! exec.awaitTermination( 2L,
5                    TimeUnit.MILLISECONDS ) ) {
6    }
7  } catch( InterruptedException ex ) {
8    ex.printStackTrace();
9  }
```

# Ejemplo Sencillo de Creación y Uso de un *Thread Pool*

```
1   // Crea el thread pool.
2   ExecutorService exec =
3       Executors.newFixedThreadPool( 4 );
4
5   // Encola las tareas.
6   for( int i = 0; i < numTareas; i++ ) {
7     exec.execute( tarea[ i ] );
8   }
9
10  // Espera a que el thread pool haya terminado.
11  exec.shutdown();
12  try {
13    while( ! exec.awaitTermination( 2L,
14                  TimeUnit.MILLISECONDS ) ) {
15    }
16  } catch( InterruptedException ex ) {
17    ex.printStackTrace();
18  }
```

# Tipos de Tareas Encoladas

- Los *Thread Pool* permiten encolar dos tipos de tareas diferentes.
- Para cada uno de los dos tipos de tareas, se debe emplear una interfaz diferente:
  1. Interfaz `Runnable`:
     - Tareas que (realizan un trabajo pero) no devuelven ningún valor.
     - Tareas que no pueden ser canceladas (una vez encoladas).
     - Encoladas con el método `execute`.
  2. Interfaz `Callable`:
     - Tareas que devuelven un valor.
     - Tareas que pueden ser canceladas (una vez encoladas).
     - Encoladas con el método `submit`.

# Tareas que Implementan la Interfaz `Runnable`

- Es el tipo de tareas más sencillo y habitual, pero tiene dos limitaciones:
  1. Las hebras no pueden devolver un resultado directamente. Por tanto, normalmente deberán "dejar" su resultado en un objeto compartido.
  2. No es posible realizar un seguimiento a una tarea y tampoco es posible posible cancelarla.
- Las tareas deben ser objetos de una clase que implemente la interfaz `Runnable`. Por tanto, deben tener un método con la cabecera: `public void run()`.
- El procesamiento de una tarea de este tipo consistirá en ejecutar el método `run` del objeto recibido como tarea.

# Tareas que Implementan la Interfaz `Callable`

- Este tipo de tareas es más complejo, pero más potente, pues ofrece las siguientes ventajas:
  1. Permite recoger un resultado una vez dicha tarea ha terminado.
  2. Permite realizar un seguimiento a la tarea y cancelarla si se desea.
- Si `V` es el tipo de datos del resultado que devuelve la tarea, la tarea debe ser encolada con el método `submit`, el cual:
  1. Debe recibir un objeto de una clase que implemente la interfaz `Callable<V>`.
     Es decir, las tareas deben ser objetos de una clase que implemente la interfaz `Callable<V>`.
  2. Devuelve un objeto de una clase que implemente la interfaz `Future<V>`.
     - La clase `V` debe coincidir en `Callable` y `Future`.

# Interfaz `Callable`

- La interfaz `Callable` es genérica: Puede devolver cualquier tipo de datos.

  Si `V` es el resultado de la tarea, `V` puede ser `Integer`, `Boolean`, etc.

- La interfaz `Callable<V>` debe tener un método `call` que devuelva un objeto de la clase `V`.

```
1  interfaz Callable<V> {
2    V call();
3  }
```

- El objetivo de esta interfaz es doble:
  1. Especificar el código a ejecutar.
     El procesamiento de una tarea de este tipo consistirá en ejecutar el método `call` del objeto recibido como tarea.
  2. Especificar el tipo de datos que devolverá dicho código.

# Interfaz `Future`

- El método `submit` encola la tarea y devuelve inmediatamente (sin esperar a que la tarea haya terminado) un objeto de la interfaz `Future<V>`.
- La interfaz `Future<V>` permite gestionar la tarea una vez ha sido encolada. Ofrece varios métodos:
  1. `boolean cancel( boolean mayInterruptIfRunning)`: Puede cancelar una tarea.
  2. `V get()`:
     Bloquea a la hebra invocadora hasta que la tarea ha sido completada, tras lo cual retorna el resultado devuelto por la hebra.
  3. `boolean isDone()`:
  4. etc.

# Ejemplo de Encolado de Tareas Runnable (parte 1)

```
1  public static void ejecutaThreadPoolConRunnable() {
2    final int numTareas = 10;
3
4    // Crea el thread pool.
5    ExecutorService exec =
6        Executors.newFixedThreadPool( 4 );
7
8    // Define la clase Tarea1b.
9    class Tarea1b implements Runnable {
10     public void run() {
11       System.out.println( "¡Hola!");
12     }
13   }
14
15   // Crea y encola las tareas.
16   for( int i = 0; i < numTareas; i++ ) {
17     exec.execute( new Tarea1b() );
18   }
19
20   // Continúa en la transparencia siguiente...
```

# Ejemplo de Encolado de Tareas Runnable (y parte 2)

```
1    // Viene de la transparencia anterior...
2
3    // Espera a que el thread pool haya terminado.
4    exec.shutdown();
5    try {
6      while( ! exec.awaitTermination( 2L,
7                    TimeUnit.MILLISECONDS ) ) {
8      }
9    } catch( InterruptedException ex ) {
10     ex.printStackTrace();
11   }
12 }
```

# Ejemplo de Encolado de Tareas Callable (parte 1)

```
1  public static void ejecutaThreadPoolConCallable() {
2    final int numTareas = 10;
3    Future<Double> f;
4
5    // Crea el thread pool.
6    ExecutorService exec =
7        Executors.newFixedThreadPool( 4 );
8
9    // Crea el objeto para recoger los resultados.
10   ArrayList<Future<Double>> alf =
11       new ArrayList<Future<Double>>( numTareas );
12
13   // Define la clase Tarea2b.
14   class Tarea2b implements Callable<Double> {
15     public Double call() {
16       return( Math.random() );
17     }
18   }
19
20   // Continúa en la transparencia siguiente...
```

# Ejemplo de Encolado de Tareas `Callable` (y parte 2)

```
1   // Viene de la transparencia anterior...
2   // Crea y encola las tareas.
3   for( int i = 0; i < numTareas; i++ ) {
4     f = exec.submit( new Tarea2b() );
5     alf.add( f );
6   }
7   // Indica al TP que termine y muestra los resultados.
8   exec.shutdown();
9   for( int i = 0; i < numTareas; i++ ) {
10    try {
11      f = alf.get( i );
12      // Espera a que el resultado esté disponible.
13      Double resultado = f.get();
14      System.out.println( resultado );
15    } catch ( ExecutionException ex ) {
16      ex.printStackTrace();
17    } catch ( InterruptedException ex ) {
18      ex.printStackTrace();
19    }
20  }
21 }
```

# Sincronización con *Thread Pools*

- Los *Thread Pools* emplean hebras en su interior, por lo que pueden aparecer Problemas de Visibilidad y Atomicidad.
- Los *Thread Pools* requieren sincronización de datos en los accesos a los datos compartidos:
  1. Sincronización Implícita:
     - Los métodos `execute` y `submit` realizan una sincronización implícita.
     - Los métodos de comprobación de terminación (como `get` de `Future`) realizan una sincronización implícita.
     - Además, es conveniente emplear el modificador `final`.
  2. Sincronización Explícita:
     - Modificador `volatile`.
     - Modificador/Bloque `synchronized`.
- Para detectar las transferencias de información, es mejor emplear clases con nombre en lugar de clases anónimas.

# Diseño de Interfaces Gráficas Multihebra

# Ejemplo de Implementación de Interfaces Gráficas

```
1  ...
2  // Creación del nuevo objeto.
3  btnPulsaAqui = new JButton( "Pulsa aqui" );
4
5  // Adición de comportamiento al nuevo objeto.
6  btnPulsaAqui.addActionListener(
7    new ActionListener() {
8      public void actionPerformed( ActionEvent e ) {
9        numVecesPulsado++;
10       txfMensajes.setText( "Has pulsado " +
11                            numVecesPulsado +
12                            " veces el boton" );
13     }
14   }
15 );
16
17 // Inserción del nuevo objeto en la interfaz.
18 tmpPanel.add( btnPulsaAqui );
19 ...
```

# AWT, Swing y la Hebra *event-dispatch* (EDT)

- AWT y Swing son paquetes de Java que permiten implementar aplicaciones gráficas.
- La interfaz gráfica almacena automáticamente en una cola de eventos todos los eventos gráficos generados: pulsaciones de botones, movimiento del cursor, etc.
- La hebra *event-dispatch* (*event-dispatch thread* o EDT) se encarga de procesar **todos los eventos** de la interfaz gráfica.

  La EDT irá extrayendo y ejecutando los eventos uno a uno.

  Por ejemplo, la EDT se encargará de ejecutar el método `actionPerformed` del botón `btnPulsaAqui` (ver transparencia anterior), cuando sea pulsado.

# La Hebra *event-dispatch* (EDT)

- La EDT es una hebra del sistema.

  Es decir, el sistema se encarga de crearla al comenzar la aplicación y destruirla al terminar.

  De hecho, cuando se cierra una interfaz gráfica, todas las hebras (tanto *daemon* como no) son eliminadas.

- Por otro lado, es labor del programador sincronizarla correctamente con el resto de hebras (si hay).

  Debe ser tratada como una hebra más en cuanto a sincronización y comunicación.

# Dos Limitaciones Serias

- **Limitación 1**: La EDT no debe realizar trabajo costoso (computación y E/S) pues entonces no puede atender a los eventos y la interfaz gráfica se queda congelada mientras se realiza la operación.

  En algún caso extremo, si la EDT ejecuta un trabajo muy largo, la interfaz se podría congelar durante mucho tiempo.

- **Limitación 2**: Ninguna hebra salvo la EDT puede llamar a métodos de componentes de la interfaz gráfica dado que AWT y Swing no son *thread-safe*.

  La aplicación podría no funcionar correctamente si cualquier hebra accede a los objetos gráficos (ejecuta métodos de los objetos gráficos).

# Solución a las Dos Limitaciones Serias: División de Trabajo

1. La EDT delega en hebras auxiliares todo el trabajo costoso (computación y E/S).
   - ¿Por qué?
     Para poder atender y procesar rápidamente otros eventos.
   - ¿Cómo?
     Creando hebras auxiliares o dándoles trabajo.
2. Las hebras auxiliares delegan en la EDT todo el tratamiento gráfico.
   - ¿Por qué?
     La interfaz gráfica no es *thread-safe* y sólo puede acceder a ella la EDT.
   - ¿Cómo?
     Con los métodos `invokeLater` y `invokeAndWait`.

# Métodos `invokeLater` e `invokeAndWait`

- La hebras creadas por el programador (auxiliares) no pueden tocar nada de la interfaz gráfica, pero le pueden mandar trabajo a la EDT mediante:
  1. Método `invokeLater`.
  2. Método `invokeAndWait`.
- Ambos métodos insertan la tarea en la cola de eventos, para que sea ejecutada por la EDT.
- Ambos métodos deben recibir como parámetro un objeto que implemente la interfaz `Runnable`.

  La EDT ejecutará el método `run` del objeto recibido.

# Métodos `invokeLater` e `invokeAndWait` (Cont.)

- Método `invokeLater`: La hebra llamadora le indica a la EDT que realice la tarea *cuando pueda*.
  - → Cuando la hebra llamadora ha retornado de ejecutar este método, la EDT podría no haber terminado (ni siquiera empezado) la tarea encomendada.
- Método `invokeAndWait`: La hebra llamadora le indica a la EDT que realice la tarea, y a continuación la hebra llamadora se bloquea hasta que la tarea haya sido completada por la EDT.
  - → Cuando la hebra llamadora termina de ejecutar este método, la EDT ha terminado completamente la tarea encomendada.

# Ejemplo de Uso del Método `invokeLater` con Clases Anónimas y con Funciones Lambda

```
1  import javax.swing.*;
2  class HebraAuxiliar extends Thread {
3    JTextField  txfMensaje; // Cuadro de texto.
4
5    public void run() {
6      // ...
7      // Uso incorrecto.
8      txfMensaje.setText( "¡Hola!" );
9
10     // ...
11     // Uso correcto.
12     SwingUtilities.invokeLater(
13         () -> txfMensaje.setText( "¡Hola!" )
14     );
15     // ...
16   }
17 }
```

# Excepciones en el Método `invokeAndWait`

- El método `invokeAndWait` puede activar dos excepciones, por lo que hay que procesarlas o pasarlas al nivel superior:
  1. `InterruptedException`: Se activa si la hebra llamadora es interrumpida mientras está a la espera de que la EDT termine la tarea encomendada.
  2. `java.lang.reflect.InvocationTargetException`: Se activa si la hebra llamadora es la propia EDT.

     Esto es un error dado que una hebra no puede bloquearse a la espera de que ella misma realice una tarea. Si se permitiera, sería un interbloqueo.