Asignatura 780014
Programación Avanzada

Tema 6 – CONTROL DE HILOS, TAREAS Y POOLS

Control de hilos, tareas y pools

- Objetivo del tema:
  - Analizar el comportamiento de hilos y las posibilidades de controlarlos desde un nivel de abstracción superior

Índice

1. Estados y control de hilos
   - Interrupción de hilos
2. Tareas y pools de hilos
   - Estrategias de ejecución de tareas
   - Tipos de pools
   - Interfaces Callable y Future

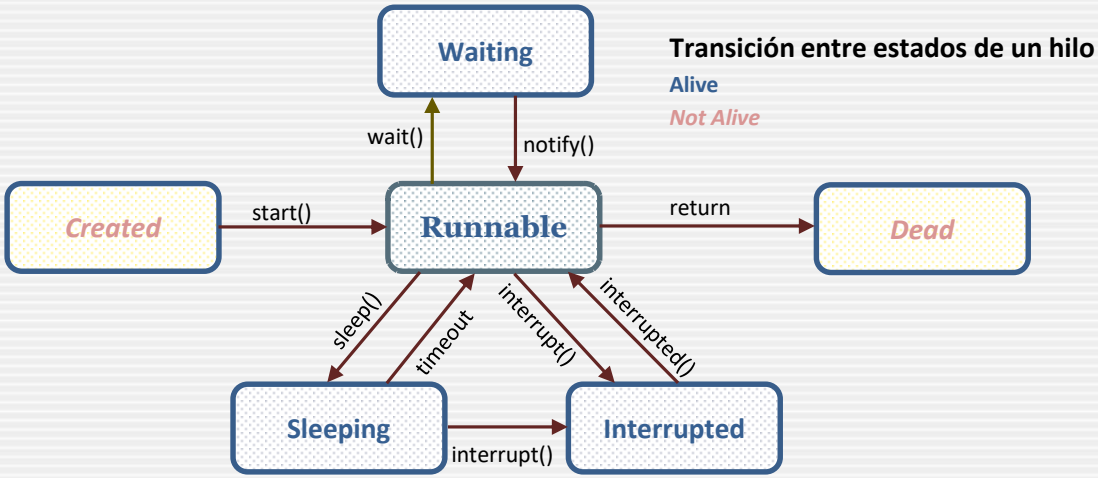
wait() fuera de synchronized

```
public class Error
{
    int i = 0;
    public static void main(String argv[])
    {
        Error w = new Error();
        w.unMetodo();
    }
    public void unMetodo()
    {
        while (true)
        {
            try {
                wait();
            } catch (InterruptedException e) { }
            i++;
        }
    }
}
```

¿Qué hace este programa?

```
run:
Exception in thread "main"
java.lang.IllegalMonitorStateException
at java.lang.Object.wait(Native Method)
at java.lang.Object.wait(Object.java:502)
at error.Error.unMetodo(Error.java:21)
at error.Error.main(Error.java:16)
Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

Estados de un hilo en Java



- El **cambio** entre estados de un hilo Java se puede **controlar**
  - Y el **estado** se puede **verificar** desde el mismo y otros hilos
- Tenemos métodos para:
  - Esperar a la finalización y verificarla
  - Interrumpir y verificar la interrupción
  - Dormir un hilo

Control de finalización de un hilo

- Métodos para **verificar la vivacidad** de un hilo y **esperar hasta la finalización** de un hilo
  - `final boolean isAlive()`
    - Devuelve true si el hilo se encuentra 'vivo', es decir, ya ha comenzado y aún no ha terminado
  - `final void join() throws InterruptedException`
    - Suspende el hilo que invoca hasta que el hilo invocado haya terminado
  - `final void join(long milliseg) throws InterruptedException`
    - Suspende el hilo que invoca hasta que el hilo invocado haya terminado, o hasta que hayan transcurrido los milisegundos

Interrupción de hilos

- Métodos para **interrumpir** y **verificar** si un hilo está interrumpido
  - `void interrupt()`
    - El hilo pasa a estado *Interrupted* (**pone flag=1**)
    - **Si está esperando** en *wait()*, *join()* o *sleep*() o llega a uno de estos métodos, termina y lanza una ***InterruptedException*** (y **pone flag=0**)
    - **Si está** en estado *Runnable*, continúa **ejecutándose**, aunque cambia su estado a *Interrupted* (**pone flag=1**)
  - `static boolean interrupted()` //Método estático
    - Devuelve *true* si el hilo que lo invoca se encuentra en estado *Interrupted* (si flag=1). Limpia el estado *Interrupted* (**pone flag=0**)
    - Si estuviese en el estado *Interrupted*, lo pasa al estado *Runnable*
  - `boolean isInterrupted()`
    - Devuelve *true* si el hilo en que se invoca se encuentra en el estado *Interrupted* (si flag=1)
    - La ejecución de este método no cambia el estado del hilo (**flag=su valor**)

Ejemplo 1: comprobar flag con isInterrupted()

```
public class Ejemplo1
{
    public static void main(String[] args)
    {
        Thread t = Thread.currentThread();
        System.out.println("A:t.isInterrupted()=" + t.isInterrupted());
        t.interrupt(); //Pone flag=1
        System.out.println("B:t.isInterrupted()=" + t.isInterrupted());
        System.out.println("C:t.isInterrupted()=" + t.isInterrupted());
        try {
            Thread.sleep(2000);
            System.out.println("No ha sido interrumpida");
        } catch (InterruptedException e) {
            System.out.println("Sí ha sido interrumpida");
        }
        System.out.println("D:t.isInterrupted()=" + t.isInterrupted());
    }
}
```

Resultado:

```
run:
A:t.isInterrupted()=false
B:t.isInterrupted()=true
C:t.isInterrupted()=true
Sí ha sido interrumpida
D:t.isInterrupted()=false
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejemplo 2: finalización por InterruptedException

```
public class Ejemplo2 {
   public static void main(String[] args) {
       Thread hilo= new Hilo2();
       hilo.start();
       try {
            Thread.sleep(1000);
        } catch (InterruptedException e){};
       hilo.interrupt(); //Pone flag=1
   }
}
```

Resultado:

```
run:
Ejecuto. Flag=false
Termino sleep. Flag:false
Ejecuto. Flag=true
Me despiertan. Flag=false
BUILD SUCCESSFUL (total time: 1 second)
```

```
public class Hilo2 extends Thread {
   public void run() {
      boolean salir = false;
      while (!salir) {
         System.out.println("Ejecuto. Flag="+this.isInterrupted());
         try {
            Thread.sleep(1000);
            System.out.println("Termino sleep. Flag:"+this.isInterrupted());
         } catch (InterruptedException e){
            System.out.println("Me despiertan. Flag="+this.isInterrupted());
            salir=true;
          }
       }
   }
}
```

Ejemplo 3: uso de interrupt() e interrupted()

```
public class Ejemplo3
{
   public static void main(String[] x)
   {
      Thread hilo=new Hilo3();
      hilo.start();
      try {
         Thread.sleep(1000);
      } catch (InterruptedException e) {};
      hilo.interrupt();
   }
}
```

```
public class Hilo3 extends Thread
{
    public void run()
    {
        while (!Thread.interrupted())
        {
            System.out.println("Ejecuto. Flag=" + this.isInterrupted());
        }
        System.out.println("Termino. Flag=" + this.isInterrupted());
    }
}
```

Resultado:

```
run:
Ejecuto. Flag=false
Ejecuto. Flag=false
Ejecuto. Flag=false
...
Ejecuto. Flag=false
Termino. Flag=false
BUILD SUCCESSFUL (total time: 1 second)
```

¿Por qué siempre se imprime "Flag=false"?

Ejemplo 4: interrupción durante sleep()

```
public class Ejemplo4 implements Runnable
{
    public void run()
    {
        try {
            System.out.println("En run(): me duermo 20 s");
            Thread.sleep(20000);
            System.out.println("En run(): me despierto");
        } catch (InterruptedException e) {
            System.out.println("En run(): despertado");
            return;
        }
        System.out.println("En run(): fin normal");
    }
    public static void main(String[] args)
    {
        Ejemplo4 ej = new Ejemplo4();
        Thread t = new Thread(ej);
        t.start();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        };
        System.out.println("En main(): Interrumpo a t");
        t.interrupt();
        System.out.println("En main(): termino");
    }
}
```

¿Cuánto tiempo se duerme el hilo que ejecuta el run()?

Resultado:

```
run:
En run(): me duermo 20 s
En main(): Interrumpo a t
En main(): termino
En run(): despertado
BUILD SUCCESSFUL (total time: 1 second)
```

Ejemplo 5: interrupción o no durante sleep()

```
public class Ejemplo5
{
    public static void main(String[] args)
    {
        if (args.length>0) //Si tiene argumentos
        {
               Thread.currentThread().interrupt();
        }
        long tiempoInicial=System.currentTimeMillis();

        try
        {
              Thread.sleep(2000);
              System.out.println("No es interrumpida");
        }
        catch (InterruptedException e) {
               System.out.println("Es interrumpida");
        }
        System.out.println("Tiempo gastado: "+ (System.currentTimeMillis()-tiempoInicial));
    }
}
```

Resultado de ejecutar > java Ejemplo5

```
No es interrumpida
Tiempo gastado: 2000
```

Resultado de ejecutar > java Ejemplo5 no

```
Es interrumpida
Tiempo gastado: 0
```

Tareas y Threads

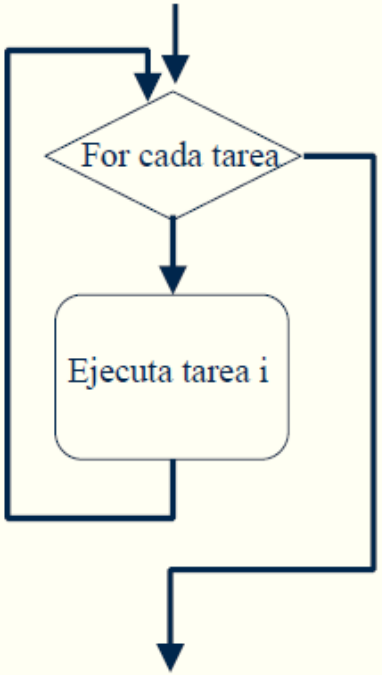
- Hasta ahora:
  - Programa concurrente = varias actividades concurrentes
  - Actividad concurrente = hilo nuevo
- Nueva posibilidad:
  - Una actividad concurrente es una **tarea** independiente de las otras
    - Definida como una unidad de trabajo abstracta y discreta
      - Ejemplo: cada cliente y respuesta del servidor es una tarea
- Concepto útil en comunicación/sincronización:
  - Incrementa la capacidad de prestar servicios (rendimiento o *throughput*)
  - Mejora tiempos de respuesta (*responsiveness*) con carga normal
  - Mejora el ritmo de degradación de prestaciones con cargas elevadas

Estrategias de ejecución

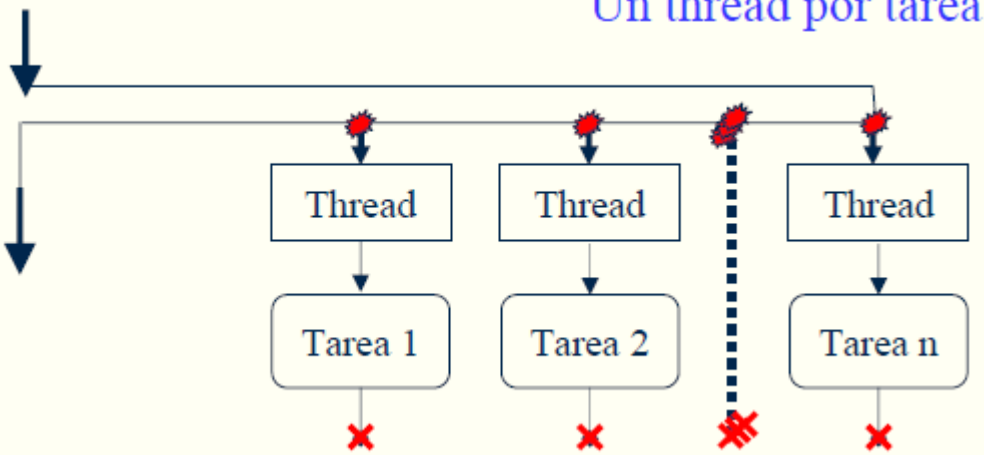
- Estrategias para ejecutar tareas:
  - **Secuencial**
    - Bajo rendimiento
    - Tiempo de respuesta malo
  - **Un hilo por tarea**
    - Mejor uso de recursos
    - Para tareas breves, la creación/destrucción de hilos es sobrecarga
    - Con carga baja, buena respuesta
    - Con carga alta, peligro de agotamiento
  - Un grupo de **hilos a reutilizar** (***pool***)
    - Se elimina sobrecarga por creación/destrucción
    - Se limita el acceso a los recursos reservados
    - Se alcanza estabilidad en carga alta

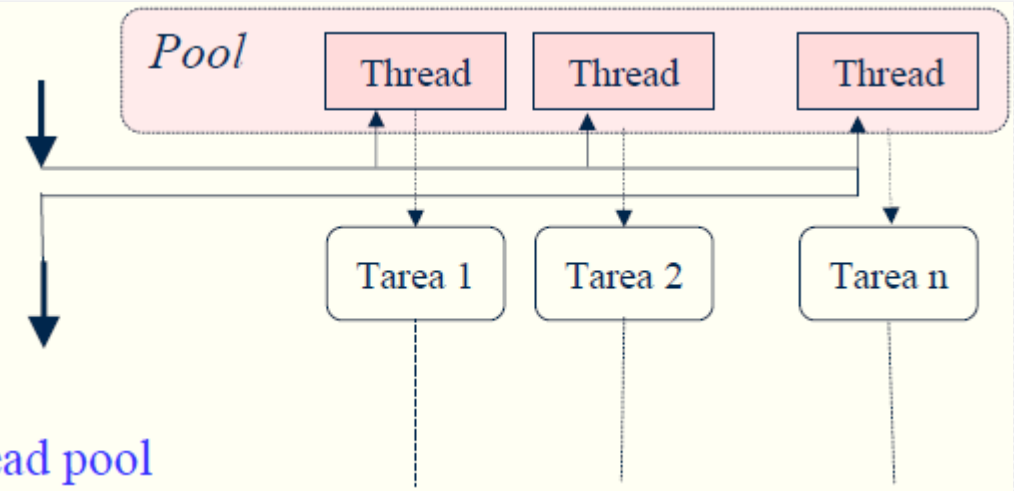
Estrategias de ejecución

Thread simple



Un thread por tarea



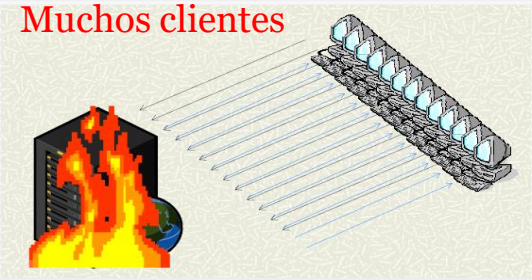


Thread pool

Estrategias erróneas

- Ejemplo: Web server que acepta conexiones por sockets en el puerto 80
  - Para cada petición crea un hilo nuevo:

```
class UnreliableWebServer
{
    public static void main(String[] args)
    {
        ServerSocket socket = new ServerSocket(80);
        while (true)
        {
            final Socket connection = socket.accept();
            Runnable r = new Runnable()
            {
                public void run() {  handleRequest(connection); }
            };
            //¡No hacer esto!
            new Thread(r).start();
        }
    }
}
```



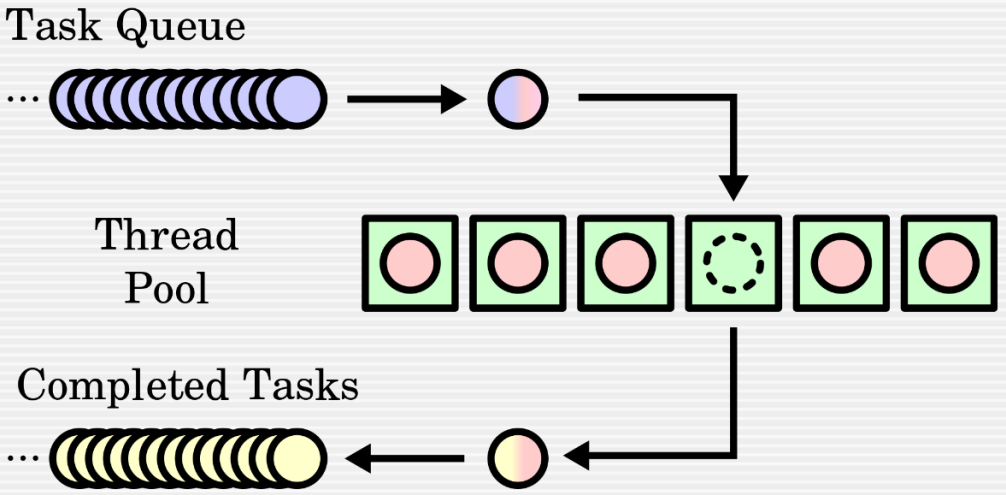
  - Sólo será capaz de atender a un número reducido de clientes

Estrategia eficaz: pool de hilos

- Pool: estructura superior al hilo que agrupa y gestiona varios hilos
  - Ejecuta **tareas** organizadas en una **cola**
    - Habrá más tareas que hilos
  - **Cada hilo** puede ejecutar **una tarea** cada vez
  - El número de hilos puede variar según necesidades
  - Se pueden repartir los pools por granjas de servidores

Algoritmo de comportamiento de cada hilo

```
while (true)
{
    if (no tasks) wait for a task;
    execute the task;
}
```



Pool de hilos en Java

- Implementado mediante el framework Executor:
  - Conjunto de interfaces
  - Ejecución concurrente y asíncrona de tareas
- La base del framework es la interfaz **Executor**
  - Describe un objeto para ejecutar Runnables

```
public interface Executor
{
    void execute(Runnable command); //Manda la tarea al pool
}
```

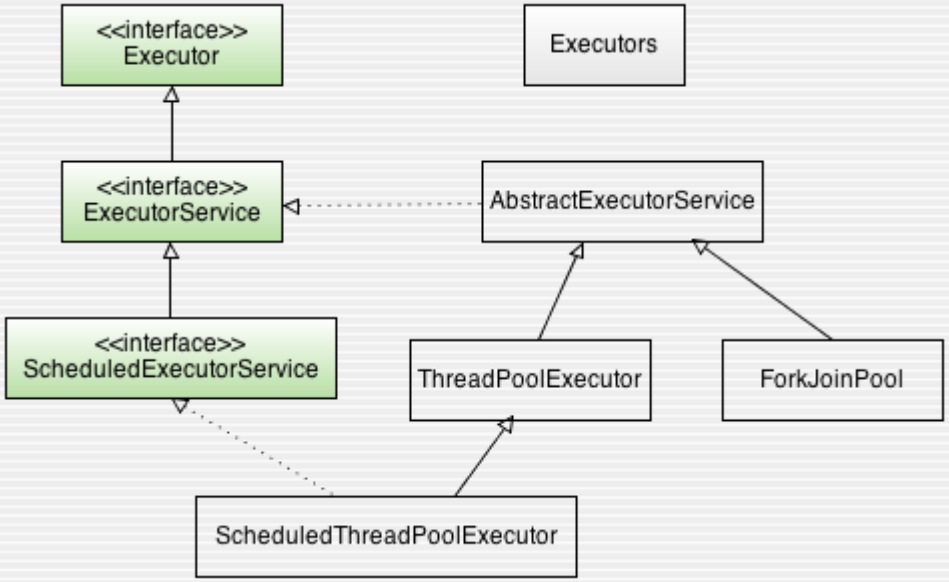
- Flexibiliza el mantenimiento
  - Permite cambios de las políticas de ejecución sin cambiar el código

Pool de hilos en Java

- En el framework Executor podemos aplicar políticas de ejecución
  - Una política de ejecución incluye:
    - Definir **en qué hilo** se ejecutan las tareas
    - Definir el **orden** en el que se ejecutan las tareas
    - El **número de tareas** que se permiten **ejecutar concurrentemente**
    - El **número de tareas** que pueden encolarse **en espera** de ejecución
    - Selección de la tarea que debe ser **rechazada** si hay sobrecarga
    - Definir **acciones** a ejecutar **antes y después** de una tarea
  - Una política es una herramienta de **gestión** de recursos
    - La gestión óptima depende de:
      - Los **recursos** disponibles
      - La **calidad** de servicio (*throughput*) que se requiere

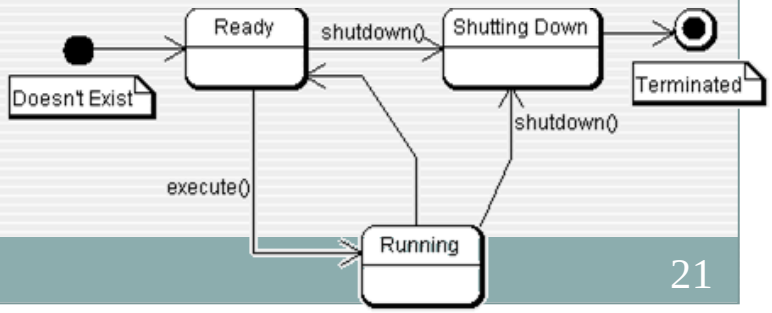
Pool de hilos en Java

- Interfaces y clases para crear pools
  - **Executor**: interfaz general que permite ejecutar tareas en un pool
  - **ExecutorService**: interfaz que hereda de la anterior y permite gestionar el ciclo de vida de un pool
  - **Executors**: factoría para crear implementaciones de ExecutorService



Pool de hilos en Java

- La interfaz **ExecutorService**
  - Ofrece servicios para controlar **ciclo de vida** de los pools de hilos
  - Un pool de hilos sigue un ciclo de vida con tres **estados**:
    - **Running**: Cuando se crea y se encuentra en régimen normal de funcionamiento
      - Admite tareas y cuando tiene hilos disponibles las ejecuta
    - **Shutting down**: Terminando
      - De manera **gradual**: no acepta nuevas tareas, pero las que están en cola se ejecutan (aunque no hayan iniciado su ejecución)
      - De manera **forzada**: no acepta nuevas tareas, no ejecuta las que están en cola, e intenta detener las que están ejecutándose
    - **Terminated**: Terminado
      - Todas las tareas han sido completadas



Pool de hilos en Java

- public interface **ExecutorService** extends Executor
  - Métodos para **manejar el ciclo de vida** de un pool

```
void shutdown();        //Inicia un apagado gradual: se ejecutan las tareas ordenadas
                        //anteriormente, pero no se aceptan nuevas tareas

List<Runnable> shutdownNow(); //Intenta detener las tareas que se están ejecutando,
                              //detiene el procesamiento de las que esperan y
                              //devuelve una lista de las que esperaban

boolean isShutdown();  //Devuelve true si se está apagando

boolean isTerminated();  //Devuelve true si todas las tareas se han completado
                         //después del apagado

boolean awaitTermination(long timeout, TimeUnit unit); //Espera hasta que el
          //sistema  haya terminado, o pase el timeout o el hilo actual sea interrumpido

//Otros métodos para iniciar tareas (se verán más adelante)
```

Pool de hilos en Java: ThreadPoolExecutor

- class **ThreadPoolExecutor** implements ExecutorService
  - Tiene 4 constructores, con los siguientes parámetros en común:

```
int corePoolSize            // Número inicial (y mínimo) de hilos que se crean
int maximumPoolSize         // Número máximo de hilos que se pueden crear
long keepAliveTime, TimeUnit unit // Tiempo para destruir hilos que sobran hasta corePoolSize
BlockingQueue<Runnable> workQueue  // Tipo de cola de tareas
```

  - Además podemos tener: ningún parámetro más, uno de los dos siguientes, o los dos:

```
ThreadFactory threadFactory      // Política con la que se crean los hilos dinámicamente
                                 // (Prioridad, Grupo, naturaleza)
RejectedExecutionHandler handler // Define la política con la que se descartan las tareas:
                                  // tareas cuya ejecución es rechazada, bien por shutdown
                                 // o por cola de tareas limitada
```

Pool de hilos en Java: factoría Executors

- Crear pools con ThreadPoolExecutor es complejo
- Es recomendable usar los métodos de factoría:
  - Executors.newCachedThreadPool()
  - Executors.newFixedThreadPool(int n)
  - Executors.newSingleThreadExecutor()
  - Executors.newScheduledThreadPool(int n)
- Estos métodos:
  - Tienen políticas ya establecidas
  - Devuelven un objeto de tipo ExecutorService
    - Podemos manejar su ciclo de vida
    - Podemos ejecutar tareas en el pool

Executors.newCachedThreadPool()

- Pool de hilos que **no está limitado** en tamaño
  - Puede **reutilizar** hilos creados previamente cuando se quedan libres
  - **Si no hay** ningún hilo **disponible** para una tarea nueva, **se crea** uno
  - Los hilos que **no** han sido **utilizados** durante un minuto, **se destruyen**

Qué pasaría si...:

1. Sistema inactivo
2. Llegan 10 tareas simultáneamente
3. Pasan 10 segundos
4. Llegan 5 tareas simultáneamente
5. Pasan 5 minutos
6. Llegan 20 tareas simultáneamente

Tareas de entre 5 y 15 segundos para procesarse

Executors.newFixedThreadPool(int n)

- Pool que reutiliza un **conjunto fijo** de hilos
  - Con una cola compartida de tareas de tamaño no limitado
  - En un momento dado, habrá **máximo *n* hilos ejecutando** tareas
  - **Si** termina algún hilo debido a un **fallo** durante la ejecución, se **vuelve a crear uno** nuevo para sustituirlo
  - Los hilos **existen hasta** que se hace el **shutdown**

Qué pasaría si...:

1. Sistema inactivo
2. Llegan 10 tareas simultáneamente
3. Pasan 10 segundos
4. Llegan 5 tareas simultáneamente
5. Pasan 5 minutos
6. Llegan 20 tareas simultáneamente

Tareas de entre 5 y 15 segundos para procesarse

Executors.newSingleThreadExecutor()

- Crea un pool con **un único hilo**
  - Con una cola de tareas de tamaño no limitado
  - Es el modelo seguido por el Swing event thread
  - Garantiza que **las tareas se ejecutan secuencialmente**
  - Garantiza que haya **solamente una activa** en cualquier momento dado
  - **Si** el hilo **falla** antes del shutdown, **se generará otro**

Qué pasaría si...:

1. Sistema inactivo
2. Llegan 10 tareas simultáneamente
3. Pasan 10 segundos
4. Llegan 5 tareas simultáneamente
5. Pasan 5 minutos
6. Llegan 20 tareas simultáneamente

Tareas de entre 5 y 15 segundos para procesarse

Executors.newScheduledThreadPool(int n)

- Crea un pool que va a ir ejecutando **tareas programadas**
  - Puede ser: en un **instante** dado o de manera **repetitiva**
  - Es parecido a un Timer
    - Con la diferencia de que puede tener varios hilos para poder realizar varias tareas programadas simultáneamente
  - Mantiene *n* hilos en el pool, aunque estén *idle*

Qué pasaría si...:

1. Sistema inactivo
2. Llegan 10 tareas simultáneamente
3. Pasan 10 segundos
4. Llegan 5 tareas simultáneamente
5. Pasan 5 minutos
6. Llegan 20 tareas simultáneamente

Tareas de entre 5 y 15 segundos para procesarse

Gestión de tareas correcta

- Ejemplo de un web server que acepta conexiones por sockets en el puerto 80
  - Dispone de un pool de 7 hilos para atender clientes

```
class ReliableWebServer
{
   ExecutorService pool = Executors.newFixedThreadPool(7);
   public static void main(String[] args)
   {
       ServerSocket socket = new ServerSocket(80);
       while (true)
       {
            final Socket connection = socket.accept();
            Runnable r = new Runnable() {
                public void run() { handleRequest(connection); }
            };
            pool.execute(r);
       }
    }
}
```

Ejemplo de finalización de las tareas del pool

- Suponiendo que tuviéramos un botón en la interfaz gráfica que controla el Web Server:

```
private void jButtonApagar (ActionEvent  evt) {
   pool.shutdown(); // Finaliza gradualmente: No deja entrar nuevas tareas
   try {            // Espera durante 1 min. para  ver si han terminado ya
        if (!pool.awaitTermination(60, TimeUnit.SECONDS))
        {
             pool.shutdownNow(); // Fuerza la terminación de las tareas
             // Espera otro minuto para ver si el pool ya ha terminado
             if (!pool.awaitTermination(60, TimeUnit.SECONDS))
                  System.err.println("El pool no terminó");
        }
     } catch (InterruptedException ie) { // (Re-)Cancela el pool nuevamente
             pool.shutdownNow();
             Thread.currentThread().interrupt();  // Preserva el flag interrupt
     }
}
```

Ejemplo sencillo de ExecutorService

```
public class Tarea implements Runnable {
   private int sleepTime;
   private String name;
   public Tarea(String name) {
       this.name = name; //Asignamos el nombre a la tarea
       sleepTime = 1000;
   }
   public void run()
   {
       try
       {
           System.out.println("El hilo de la tarea
"+this.name+" va a dormir durante "+sleepTime+" ms");
           Thread.sleep(sleepTime); //Duerme 1 segundo
       } catch(InterruptedException exception){
            exception.printStackTrace();
       }
       System.out.println("Este hilo ha despertado");
   }
}
```

```
public class EjemploThreadPool {
    public static void main(String args[]) {
        System.out.println("Comienza la ejecución");
        ExecutorService ex =
Executors.newFixedThreadPool(10);  //Se crea un pool de
                                   //hilos de tamaño 10
        Tarea t;
        for (int i = 0; i < 200; i++) {
            t = new Tarea("" + i);
            ex.execute(t); //Envía a ejecutar cada tarea
        }
        ex.shutdown(); //Se indica que finalice el pool
    }
}
```

```
run:
Comienza la ejecución
El hilo de la tarea 1 va a dormir durante 1000 ms
El hilo de la tarea 7 va a dormir durante 1000 ms
...
Este hilo ha despertado
El hilo de la tarea 10 va a dormir durante 1000 ms
...
Este hilo ha despertado
BUILD SUCCESSFUL (total time: 20 seconds)
```

Tareas con resultados demorados

- En el framework Executor:
  - Las tareas son objetos que implementan la interfaz Runnable
  - Su ejecución es el método public void run(){…}
  - Con algunas limitaciones:
    - **No** puede retornar valores
    - **No** puede lanzar excepciones
  - No se puede saber si una tarea se ha llegado a ejecutar o si ha finalizado
- Se añadieron las interfaces **Callable** y **Future** porque:
  - Muchas tareas representan una computación aplazada (Future)
  - Para conocer si una tarea ha sido ejecutada ya correctamente o no (Future)
  - Para poder retornar valores con un hilo (Callable)

Interfaz Callable

- public interface **Callable**<V>
  - Método:
    - `V call() throws Exception`

- Los objetos que implementan esta interfaz son Runnables, pero:
  - Deben implementar la interface **Callable**, en lugar de Runnable
  - El código a ejecutar está en el método **call**(), no en run()
  - **Devuelven un resultado** de tipo V
  - Pueden lanzar una excepción

Interfaz Future

- La interfaz **Future** permite representar una tarea que puede estar:
  - Completada
  - En proceso de ejecución
  - No haber comenzado su ejecución

- Mediante Future, en un pool se puede:
  - **Cancelar** una tarea que no haya terminado
  - **Preguntar si** una tarea **ha terminado o ha sido cancelada**
  - **Obtener** (o esperar por) **el resultado** de una tarea Callable

Interfaz Future

- Métodos
  - `boolean cancel(boolean mayInterruptIfRunning)` //Cancela una tarea
  - `boolean isCancelled()` //Pregunta si una tarea está cancelada
  - `boolean isDone()` //Pregunta si una tarea ha terminado
  - `V get() throws InterruptedException, ExecutionException, CancellationException` //Obtiene el resultado de una tarea Callable. Si al llamarlo //la tarea no ha terminado, el hilo llamante se bloquea
  - `V get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, CancellationException, TimeoutException`

    //Obtiene el resultado de una tarea Callable con un tiempo máximo (el hilo llamante se //bloquea hasta que termine la tarea o hasta que expire el timeout)

Otros métodos de ExecutorService

- public interface ExecutorService extends Executor

//Para **enviar** una tarea y **controlar su resultado y su estado** mediante Future<T>:

- `Future<T> submit(Callable<T> task)`

//Para **enviar** una tarea Runnable y **controlar su estado**:

- `Future<?> submit(Runnable task)`

//**Recibe** una colección de tareas, las **ejecuta** y **devuelve** una lista de objetos de tipo Future con **sus resultados**:

- `List<Future<T>> invokeAll(Collection<Callable<T>> tasks)`

//**Recibe** una colección de tareas, luego las **ejecuta** y **devuelve el resultado de la primera tarea que termina** sin lanzar una excepción. Las tareas que no han finalizado se cancelan

- `<T> invokeAny(Collection<Callable<T>> tasks)`

//etc...

Ejemplo de Callable y Future

- Objetos Callable que calculan la **multiplicación de dos números** y se ejecutan **en un pool**

```
public class CalculadorMultiplicacion implements Callable<Integer>
{
    private int operador1;
    private int operador2;

    //Constructor que recibe dos parámetros
    public CalculadorMultiplicacion(int operador1, int operador2)
    {
        this.operador1 = operador1;
        this.operador2 = operador2;
    }

    //Método call() en lugar de run(): retorna un valor, a diferencia de run()
    public Integer call() throws Exception
    {
        return operador1 * operador2;
    }
 }
```

Ejemplo de Callable y Future

```
public class Main {
    public static void main(String argv[]) {
        ExecutorService executor = Executors.newFixedThreadPool(2); //Con 2 hilos
        List<Future<Integer>> listaResultados = new ArrayList<Future<Integer>>(); //Lista de Futures de tipo Integer
        for (int i = 0; i < 10; i++) {
            CalculadorMultiplicacion calculador=new CalculadorMultiplicacion((int)(Math.random()*10), (int)(Math.random()*10));
            Future<Integer> resultado = executor.submit(calculador);  //Las tareas se mandan a ejecutar
            listaResultados.add(resultado);
        }
        for (int i = 0; i < listaResultados.size(); i++) {
            Future<Integer> resultado = listaResultados.get(i); //Se obtiene el elemento correspondiente a la tarea finalizada
            try { //Se imprime el resultado de cada tarea finalizada:
                System.out.println("El resultado de la tarea " + i + " es:" + resultado.get());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        executor.shutdown(); //Apagamos gradualmente el pool
    }
}
```

```
run:
El resultado de la tarea 0 es:0
El resultado de la tarea 1 es:0
El resultado de la tarea 2 es:2
El resultado de la tarea 3 es:27
El resultado de la tarea 4 es:4
El resultado de la tarea 5 es:6
El resultado de la tarea 6 es:6
El resultado de la tarea 7 es:54
El resultado de la tarea 8 es:30
El resultado de la tarea 9 es:6
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejemplo de uso de invokeAny()

- Obteniendo el resultado de la **primera** tarea en terminar:

```
public class Main {
    public static void main(String argv[]) {
        ExecutorService executor = Executors.newCachedThreadPool();
        List<CalculadorMultiplicacion> listaTareas = new ArrayList<CalculadorMultiplicacion>();
        for (int i = 0; i < 10; i++) { //Creamos las tareas y las añadimos a la lista
            CalculadorMultiplicacion calculador=new CalculadorMultiplicacion((int)(Math.random()*10), (int)(Math.random() * 10));
            listaTareas.add(calculador);
        }
        try { //Ejecutamos cualquiera de las tareas (devolverá el resultado de la primera en ejecutarse):
            Integer resultado = executor.invokeAny(listaTareas);
            System.out.println("El resultado de la primera tarea en terminar es:" + resultado);
        } catch (Exception e) { }
        executor.shutdown(); //Termina el Executor
    }
}
```

```
run:
El resultado de la primera tarea en terminar es:10
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejemplo de uso de invokeAll()

- Obteniendo **todos** los resultados:

```
public class Main {
    public static void main(String argv[]) {
        ExecutorService executor = Executors.newCachedThreadPool();
        List<CalculadorMultiplicacion> listaTareas = new ArrayList<CalculadorMultiplicacion>();
        for (int i = 0; i < 10; i++) { //Crea las tareas y las añade a la lista
            CalculadorMultiplicacion calculador=new CalculadorMultiplicacion((int)(Math.random()*10), (int)(Math.random()*10));
            listaTareas.add(calculador);
        }
        List<Future<Integer>> listaResultados = null;
        try { //Manda las tareas a ejecutar
            listaResultados = executor.invokeAll(listaTareas);
        } catch (Exception e) { }
        executor.shutdown();
        for (int i = 0; i < listaResultados.size(); i++) {
            Future<Integer> resultado = listaResultados.get(i);
            try { //Obtiene el resultado (o se bloquea hasta que esté disponible):
                System.out.println("El resultado de la tarea " + i + " es:" + resultado.get());
            } catch (Exception e) { }
        }
    }
```

```
run:
El resultado de la tarea 0 es:1
El resultado de la tarea 1 es:63
El resultado de la tarea 2 es:48
El resultado de la tarea 3 es:54
El resultado de la tarea 4 es:0
El resultado de la tarea 5 es:18
El resultado de la tarea 6 es:8
El resultado de la tarea 7 es:45
El resultado de la tarea 8 es:9
El resultado de la tarea 9 es:36
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ejercicios (1/2)

- 1.- Ejecutar los códigos utilizados en esta presentación.
- 2.- Pensar qué ocurriría en los supuestos planteados en las transparencias 26 a 29 en cada uno de los tipos de pools.
- 3.- Crear un programa que maneje un pool de hilos que se encargue de **calcular e imprimir** por pantalla todos los números primos existentes entre el 1 y los 10.000.000.
  - Cada tarea hará el cálculo de un rango de 100.000 números e imprimirá por pantalla los números que vaya encontrando.
  - Crear un pool de tamaño 10.

Ejercicios (2/2)

- 4.- Crear un programa que maneje un pool de hilos que se encargue de **calcular la suma** de todos los números primos existentes entre el 1 y los 10.000.000.
  - Cada tarea hará el cálculo de un rango de 100.000 números y devolverá la suma parcial a través de un objeto Future.
  - Crear un pool de tamaño 10.