Introducción a la Programación

Tema 19. Introducción a la programación concurrente

1. Introducción

La programación que se lleva a cabo actualmente, debido a la disponibilidad de redes de ordenadores, usa en una proporción importante un estilo de programación denominado

Programación Concurrente. Este estilo de programación permite diseñar varias actividades para que se lleven a cabo simultáneamente y cooperen entre ellas. Podemos distinguir dos tipos de *Programación Concurrente*: *Programación Distribuida* que hace referencia a actividades concurrentes que no comparten una memoria común y *Programación Paralela* que hace referencia a actividades concurrentes que comparten una memoria común.

Un concepto es importante en este estilo de programación: el concepto de hebra (**thread**). Una hebra es una actividad secuencial que puede interactuar con otras actividades que ocurren de forma simultánea.

Un **proceso** es una unidad de ejecución gestionada por el sistema operativo. Cada proceso tiene su propio espacio de direcciones al cual no pueden acceder otros procesos. Una hebra (**thread)** es un flujo de control secuencial que se ejecuta dentro de un proceso. Todas las hebras dentro de un proceso comparten el mismo espacio de direcciones y la misma memoria. Una hebra es denominada un *proceso ligero*.

En este capítulo veremos los elementos de la *Programación Concurrente* (independientemente de que sea Paralela o Distribuida) con los detalles específicos del lenguaje Java. En este contexto el concepto fundamental es el de hebra (**thread).** Java proporciona, tal como veremos más adelante una clase (la clase Thread), con la que podemos crear hebras y gestionarlas.

Las ideas que vamos a explicar queremos combinar la *Programación Concurrente* con la *Programación Orientada a Objetos* que hemos visto en capítulos anteriores. Distinguiremos entonces de ahora en adelante objetos activos y objetos pasivos. Los segundos son los que hemos visto en los capítulos anteriores.

De una manera muy general los objetos pasivos son los que hemos visto en los capítulos anteriores. Este tipo de objetos son usados generalmente para almacenar información, que puede ser compartida por otros objetos, junto con la funcionalidad para gestionarla. Tienen que esperar a otro objeto llame a uno de sus métodos. En muchos casos un objeto pasivo puede imponer la restricción de ser accedido secuencialmente. En este caso el objeto pasivo puede retardar el acceso de un nuevo objeto hasta que el que está accediendo en ese momento termine.

Un objeto activo, por el contrario, tiene asociada su propia hebra. Llevan a cabo una actividad pudiendo acceder a los demás objetos, esperar hasta que se le conceda permiso para ello, y pueden compartir recursos con otros objetos activos.

En programas secuenciales el único objeto activo el programa principal. Aquel que empezó ejecutando el método main(…). Los objetos pasivos sirven como datos del programa.

Un *programa concurrente* es un conjunto de objetos activos (cada uno llevando a cabo su actividad) y un conjunto de objetos pasivos a los que pueden acceder.

En java los objetos activos deben implementar el interface *Runnable* y disponer internamente de una hebra que mantiene el estado de su actividad y ofrece métodos para esperar, iniciar, etc.

Los objetos pasivos en Java pueden añadir *synchronize* a alguno o a todos sus métodos para indicar que en cada momento sólo se permitirá acceder a un objeto activo.

Un objeto activo sólo puede hacer una cosa en cada momento. La mayoría de las acciones que lleva a cabo son reactivas. Es decir acciones en respuesta a mensajes de otros objetos. Aunque también puede llevar a cabo acciones de forma autónoma.

2. Problemas en el diseño de programas concurrentes

Un primer tema es la seguridad de los estados de un objeto. Un objeto pasivo podría ser accedido por varios objetos activos a la vez. Estos objetos que acceden a la vez pueden intentar llevar a cabo acciones conflictivas entre sí. Un objeto pasivo puede protegerse a sí mismo de esas actividades conflictivas. Hablamos, entonces, de objetos seguros: aquéllos que sólo permiten acciones que produzcan estados consistentes (estados permitidos o válidos).

Además un objeto puede poner restricciones para ser accedido por uno sólo objeto activo (hebra) en cada momento. Toda esta problemática, asegurar la consistencia de los estados de los objetos pasivos, la englobamos con el nombre de Seguridad (Safety): requisitos de integridad del sistema. O dicho de otra manera: el sistema siempre debe permanecer en un estado correcto (deben cumplirse los invariantes, etc…). Ejemplos de violaciones de Seguridad son:

- Conflictos de lectura/escritura
- Violaciones de los invariantes

En general si muchas hebras acceden al estado de un objeto de un tipo mutable la seguridad del objeto puede ser violada. Hay tres maneras, como iremos viendo, para solucionar este problema:

- No compartir el estado entre varias hebras
- Hacer que el estado del objeto sea de un tipo inmutable
- Diseñar la sincronización adecuada para acceder al estado del objeto

Para conseguir lo anterior un buen diseño orientado a objeto es de una gran ayuda. Es importante la encapsulación del estado (haciendo privados los atributos de las clases), el diseño de tipo inmutables cuando sea posible y la especificación de los invariantes de los tipos y las precondiciones y postcondiciones de los métodos.

Podemos decir que una **clase es segura** en un uso concurrente cuando se comporta según la especificación aun cuando pueda ser accedida por varias hebras a la vez. El diseño concurrente introducirá restricciones de sincronización para que no se viole la especificación de la clase cuando es accedida simultáneamente por varias hebras. Con esta definición hay varias cuestiones a tener en cuenta:

- Los tipos inmutables son seguros.
- Operaciones atómicas. Son una secuencia de operaciones que deben ser ejecutadas secuencialmente por una hebra sin entrelazarse en su ejecución con cualquier otra. Más adelante veremos técnicas para diseñar una operación atómica. El API de *Java* y el de *Guava* ya ofrecen tipos cuyas operaciones son atómicas.
- Java proporciona un mecanismo para conseguir atomicidad: la cláusula *synchronized* que puede cualificar a un método o a un trozo de código.
- Hay que diseñar mecanismos para asegurar que se mantienen los invariantes especificados. Esto se puede conseguir mediante guardas específicas. Es decir restricciones que harán esperar a algunas hebras hasta que se le permita acceder al objeto compartido.

Por otra existe el concepto de **Viveza (Liveness):** cada actividad de un objeto activo debe progresar hasta completarse. Ejemplos de problemas de Viveza son:

- Un objeto activo se bloquea esperando un evento, mensaje o condición que debería ser per no es producido por otro objeto activo
- Adjudicación de recursos insuficiente o no justa
- Fallos y/o errores de diferentes clases

La Seguridad y la Viveza son dos fuerzas que presionan en sentidos opuestos a la hora de diseñar un sistema concurrente. Junto a ellas tenemos que considerar, en el diseño del sistema, principios generales de diseño de software: **Eficiencia** y **Reusabilidad.**

3. Diseño de objetos activos y hebras

La clase **Thread** puede ser usada para representar el estado de una actividad independiente. Es decir para dotar a un objeto activo de actividad. Cada objeto de esta clase es un miembro de un **ThreadGroup**. Cada objeto activo implementa, además, la interfaz

```
interface Runnable {
      public void run();
}
```

La implementación del método *run()* especificará la actividad a realizar por el objeto activo.

La clase **Thread** ofrece la siguiente funcionalidad.

Constructores:

- *Thread(Runnable r)* construye una hebra pasándole como parámetro un objeto que implementa *Runnable*.
- *Thread()* construye una hebra.

Métodos principales:

- *void start():* comienza la ejecución de la hebra y vuelve al método que hizo la llamada. La actividad asociada a una hebra termina cuando acabe el método *run().*
- *void run():* método que es llamado por start() si al constructor de la hebra se le pasó un objeto que implementaba *Runnable.* En otro caso no hace nada.
- *boolean isAlive():* devuelve verdadero si la hebra empezó pero no terminó.
- *void join() throws InterruptedException:* la hebra invocante espera a que termine ala invocada.
- *void join(long ms) throws InterruptedException:* la hebra invocante espera a que termine ala invocada o pase el tiempo indicado.
- *void interrupt():* interrumpe la hebra. Si la hebra está bloqueada por una llamada previa a *wait(…), join(…), sleep(…)* entonces se desbloquea y el método correspondiente recibe la excepción *InterruptedException*.

Métodos estáticos:

- *Thread currentThread()*: devuelve la hebra actual
- *void sleep(long ms) throws InterruptedException*: suspende la hebra durante (al menos) ms milisegundos

A partir de la clase **Thread** hay dos maneras de diseñar clases adecuadas para crear un objeto activo. La primera es mediante herencia: diseñamos una clase que herede de *Thread* y re-implementamos el método *run().* Por ejemplo un objeto activo que calcula primos mayores que un entero dado:

```
class PrimeThread extends Thread {
         long minPrime;
         PrimeThread(long minPrime) {
             this.minPrime = minPrime;
         }

         public void run() {
             // calcula e imprime los sucesivos primos mayores que
             // que minPrime
              . . .
         }
}
```

Con ese diseño la forma de crear el objeto e iniciar su actividad sería:

```
PrimeThread p = new PrimeThread(143);
p.start();
```

La segunda es diseñar una clase que implemente Runnable:

```
class PrimeRun implements Runnable {
          long minPrime;
          PrimeRun(long minPrime) {
              this.minPrime = minPrime;
          }

          public void run() {
              // compute primes larger than minPrime
               . . .
          }
}
```

Con ese diseño la forma de crear el objeto e iniciar su actividad sería crear una instancia de la clase que implementa *Runnable*, pasarla como parámetro al constructor de *Thread* y posteriormente iniciar su actividad.

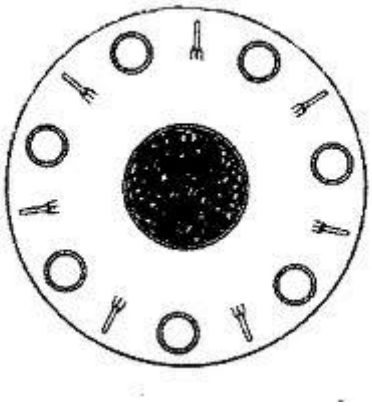
```
PrimeRun p = new PrimeRun(143);
new Thread(p).start();
```

Una tercera forma es unificar las dos anteriores: diseñar una clase que implemente *Runnable* y posiblemente otros interfaces y que reutilice *Thread* mediante delegación. En general un objeto activo estará en varios estados posibles. Su actividad consistirá en entrar en un estado hacer una tarea y salir cuando la termine o entrar a un estado (o salir de otro) cuando consiga un recurso o reciba un determinado mensaje. Un objeto activo suele ofrecer métodos reutilizados *Thread* como son:

```
void start();
void stop()
```

El primero inicia la actividad del objeto usualmente llamando al método *run().* El segundo libera los recursos que ha adquirido y detiene la actividad.

Como ejemplo de objeto activo veamos a forma de modelar un filósofo en el problema conocido como problema de Los Filósofos Comensales. Esencialmente se trata de *n* filósofos sentados a una mesa donde hay *n* platos y *n* tenedores. Cada filósofo tiene que conseguir, cuando estén libres, los dos tenedores que tiene cerca. Su actividad consiste en un ciclo: intenta coger los dos tenedores, come durante un tiempo cuando los consiga, los libera y piensa durante un tiempo y vuelve a comenzar el ciclo.



Los estados posibles en los que puede estar pueden ser modelados mediante un tipo enumerado.

```
enum Estado {Pensando,Comiendo,Esperando,Fin};
```

La actividad del filósofo la implementamos en el método run:

```
public void run() {
            while(!fin){
                  esperar();
                  try {
                        Tenedores.obtenerTenedores(i);
                        comer();
                  } catch (InterruptedException e) {
                        e.printStackTrace();
                  } finally {
                        Tenedores.liberarTenedores(i);
                  }
                  pensar();
            }
            setEstado(Estado.Fin);
}
```

Las actividades de pensar y comer se modelan como una espera de duración aleatoria. El filósofo esperará hasta conseguir los dos tenedores. Como veremos posteriormente la forma de conseguir recursos, usarlos (en este caso la actividad comer) y posteriormente liberarlos tiene la estructura:

```
try {
      conseguir recursos;
      usarlos;
} catch (InterruptedException e) {
      gestionar la possible excepción. En este caso no existe;
} finally {
      liberarlos;
}
```

```
public void comer(){
            setEstado(Estado.Comiendo);
            try {
               Thread.sleep(escala*Math2.getEnteroAleatorio(0, 1000));
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
      }

      public void pensar(){
            setEstado(Estado.Pensando);
            try {
               Thread.sleep(escala*Math2.getEnteroAleatorio(0, 1000));
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
      }
```

Los métodos start y stop:

```
public void start(){
            this.threat = new Thread(this);
            this.threat.start();
}
public void stop(){
            fin = true;
}
```

La clase que implementa el filósofo sería de la forma:

```
public class Filosofo implements Runnable {

      public enum Estado {Pensando,Comiendo,Esperando,Fin};

      public static Long escala = 10L;

      private Estado estado;
      private boolean fin = false;

      int i;
      private Thread threat;

      public Filosofo(int i) {
            this.estado = Estado.Esperando;
            this.i = i;
      }
      public Estado getEstado() {
            return estado;
      }
```

```
	public void setEstado(Estado estado) {
		this.estado = estado;
	}
	…
}
```

Podemos diseñar una factoría de filósofos con métodos para inicializar el conjunto, poner sus actividades en marcha o detenerlos.

```
public class Filosofos {

	public static int numeroDeFilosofos;
	private static Filosofo[] filosofos;

	public static void init(int n) {
		numeroDeFilosofos = n;
		filosofos = new Filosofo[numeroDeFilosofos];

		for(int i =0;i <numeroDeFilosofos; i++){
			filosofos[i] = new Filosofo(i);
		}
	}

	public static void start() {
		for(int i =0;i <numeroDeFilosofos; i++){
			filosofos[i].start();
		}
	}

	public static void stop() {
		for(int i =0;i <numeroDeFilosofos; i++){
			filosofos[i].stop();
		}
	}

	public static Filosofo[] getFilosofos() {
		return filosofos;
	}
}
```

4. Diseño de objetos pasivos: monitores y synchronized

Los objetos pasivos pueden ser accedidos por varios objetos activos. Por eso tienen que imponer reglas de acceso que garanticen estar en estados consistentes y hagan esperar a los objetos activos que llevarían al objeto pasivo a un estado inconsistente. Para hacer eso posible disponemos de la clase *Monitor* ofrecida por *Guava*.

En el problema anterior de los filósofos hemos diseñado previamente cada filósofo como un objeto activo. Este objeto tiene que obtener los recursos que necesite para su actividad y esperar mientras que no los consiga. Cada filósofo necesita sus dos tenedores adyacentes para comer. Cuando los necesite intentará obtenerlos. Si alguno de ellos está ocupado el filósofo deberá esperar. Para implementar estas ideas los tenedores los diseñamos como un conjunto de objetos pasivos y para gestionarlos usamos un conjunto de *Monitores*.

Los monitores son objetos que permiten gestionar el acceso concurrente por varias hebras a recursos compartidos. Los monitores proporcionan mecanismos para dar a una hebra acceso temporal exclusivo a un conjunto de recursos, o hacerla esperar hasta que se cumpla una condición sobre los mismos. Los monitores también proporcionan mecanismos para anunciar a algunas hebras que se cumplen determinadas condiciones sobre los recursos compartidos considerados.

Vamos a ver aquí la parte relevante de la funcionalidad ofrecida por la clase *Monitor* ofrecida por *Guava*. Un monitor ofrece métodos que permiten a una hebra ocupar el monitor (métodos *enter…*) o dejarlo libre (método *leave*). Una hebra se dice que ocupa el monitor si ha entrado pero todavía no ha salido. En cada momento sólo una hebra puede ocupar el monitor.

Asociado a cada monitor puede haber una o varias guardas. Una guarda es un objeto asociado a un monitor que evalúa una condición sobre los recursos controlados por el mismo. Una guarda es un objeto de tipo *Guard*. Los objetos de este tipo tienen un único método:

- *abstract boolean isSatisfied();*

Este método debe ser implementado para cada guarda concreta.

Algunos métodos de la clase *Monitor* son:

- *void enter()*
- *void enterInterruptibly() throws InterruptedException*
- *boolean enter(long time,TimeUnit unit)*
- *boolean enterInterruptibly(long time,TimeUnit unit)*
  throws InterruptedException
- *void enterWhen(Monitor.Guard guard) throws InterruptedException*
- *void enterWhenUninterruptibly(Monitor.Guard guard)*
- *boolean enterWhen(Monitor.Guard guard,long time, TimeUnit unit)*
  throws InterruptedException
- *boolean enterIf(Monitor.Guard guard)*
- *boolean enterIfInterruptibly(Monitor.Guard guard)*
  throws InterruptedException
- *boolean enterIf(Monitor.Guard guard,long time,TimeUnit unit)*
- *boolean enterIfInterruptibly(Monitor.Guard guard,long time,*
  TimeUnit unit) throws InterruptedException
- *boolean tryEnter()*
- *boolean tryEnterIf(Monitor.Guard guard)*
- *void waitFor(Monitor.Guard guard)throws InterruptedException*
- *void waitForUninterruptibly(Monitor.Guard guard)*
- *boolean waitFor(Monitor.Guard guard,long time,TimeUnit unit)*
  throws InterruptedException
- *boolean waitForUninterruptibly(Monitor.Guard guard,long time,*
  TimeUnit unit)

- *void leave()*

Como vemos hay tres tipos de métodos *enter, enterIf, enterEhen, waitFor*. Los métodos *enter…* intentar ocupar el monitor. Si ya está ocupado deben esperar. Adicionalmente una hebra puede tener que esperar a que se cumpla una condición establecida por la guarda que recibe como parámetro. Las esperas para ocupar el monitor pueden ser limitadas por tiempo, y ser o no interrumpidos. El método *leave* libera el monitor.

El método *tryEnter* entra al monitor si es posible hacerlo inmediatamente. El método *tryEnterIf* entra al monitor si es posible hacerlo inmediatamente y la guarda se cumple. En otro caso no esperan.

Los métodos *enterWhen* esperarán para entran al monitor hasta que no esté ocupado y la guarda sea satisfecha.

Los métodos *enterIf* esperaran para entrar al monitor sólo si la guarda es satisfecha y el monitor está ocupado.

Los métodos *waitFor* esperan a que se cumpla una condición una vez que se está dentro del monitor.

Para usar los métodos anteriores es conveniente hacerlo dentro de unos esquemas. Estos esquemas son importantes para asegurar que se libera el monitor cuando la hebra ha terminado de usarlo.

Los métodos que no devuelven nada:

```
monitor.enter();
try{
      … usar el monitor;
} finally {
     monitor.leave();
}
```

Los métodos que devuelven *boolean*:

```
if(monitor.enterIf(guard){
      try{
            … usar el monitor;
      } finally {
            monitor.leave();
      }
} else{
      monitor no disponible;
}
```

Los métodos que devuelven *boolean* y pueden ser interrumpidos pueden gestionar la excepción de la forma:

```
if(monitor.enterIf(guard){
      try{
            … usar el monitor;
```

```
      } catch(InterruptedException e){
      gestionar excepción;
      }finally {
            monitor.leave();
      }
} else{
      monitor no disponible;
}
```

La clase que se muestra a continuación tiene un recurso (el atributo privado *value*) que debe ser gestionado. La clase usa un monitor para ello y define dos guardas según que el atributo se o no *null*.

```
public class SafeBox<V> {
     private final Monitor monitor = new Monitor();
     private final Monitor.Guard valuePresent =
           new Monitor.Guard(monitor) {
       public boolean isSatisfied() {
         return value != null;
       }
     };
     private final Monitor.Guard valueAbsent =
           new Monitor.Guard(monitor) {
       public boolean isSatisfied() {
         return value == null;
       }
     };
     private V value;

     public V get() throws InterruptedException {
       monitor.enterWhen(valuePresent);
       try {
         V result = value;
         value = null;
         return result;
       } finally {
         monitor.leave();
       }
     }

     public void set(V newValue) throws InterruptedException {
       monitor.enterWhen(valueAbsent);
       try {
         value = newValue;
       } finally {
         monitor.leave();
       }
     }
   }
```

En Java existe el modificador *synchronized* para los métodos. Si un objeto tiene varios métodos cualificados con *synchronized* se comporta como un monitor. Cada hebra que intente invocar

uno de esos métodos tendrá que esperar hasta que el monitor asociado al objeto esté desocupado.

La clase *Object* (y por lo tanto todos los objetos) ofrece métodos para gestionar las esperas en el monitor asociado a un objeto y poder emular las guardas que hemos comentado previamente en la clase Monitor. Esto métodos son:

- *wait()*
- *wait(ms)*
- *notify()*
- *notifyAll()*

Estos métodos heredados de *Object*, gestionan la suspensión de una actividad o su reanudación.

- *void wait() throws InterruptedException:* la hebra espera, queda bloqueada, hasta que otro objeto invoque *notify()* o *notifyAll()* sobre este objeto.
- *void wait(long ms) throws InterruptedException:* la hebra espera, queda bloqueada, hasta que otro objeto invoque *notify()* o *notifyAll()* sobre este objeto o pase la cantidad indicada de tiempo en milisegundos
- *void notify():* despierta una de las hebras de las que están esperando en este objeto
- *void notifyAll():* despierta todas las hebras que están esperando en este objeto

Con los métodos anteriores es posible simular la semántica de las guardas que ofrece la clase Monitor y otros aspectos de la misma. La clase anterior implementada sin usar el Monitor proporcionado por *Guava* es:

```
public class SafeBox<V> {
     private V value;

     public synchronized V get() throws InterruptedException {
       while (value == null) {
         wait();
       }
       V result = value;
       value = null;
       notifyAll();
       return result;
     }

     public synchronized void set(V newValue) throws
              InterruptedException {
       while (value != null) {
         wait();
       }
       value = newValue;
       notifyAll();
     }
   }
```

Usar la clase *Monitor* o el modificador *synchronized* dependerá del problema en particular. En general usaremos *synchronized* si el objeto es usado simplemente como una llave que puede ser poseída o liberada. En casos más complejos donde puede haber guardas y deferentes formas de gestionarlas es más sencillo usar la clase Monitor porque la programación con los métodos *wait, notify*, etc. puede ser compleja.

Con las ideas anteriores podemos diseñar el tipo *Tenedor* y el gestor de recursos *Tenedores*. Son los recursos que usarán los objetos activos *Filósofos* de la sección anterior. Para gestionar el conjunto de tenedores diseñamos un conjunto de monitores, cada monitor con una guarda. La guarda del monitor *i* será satisfecha si los tenedores *i* e *i+1* están disponibles.

```
public class Tenedores {

      private static int nt = Filosofos.numeroDeFilosofos;
      private static Tenedor[] tenedores;
      private static Monitor[] monitores;
      private static Monitor.Guard[] disponibles;

      public static void init() {
        tenedores = new Tenedor[nt];
        monitores = new Monitor[nt];
        disponibles = new Monitor.Guard[nt];

        for(int i =0;i <nt; i++){
            tenedores[i] = new Tenedor(i);
        }

        for(int i =0;i <nt; i++){
            monitores[i] = new Monitor();
        }

        for(int i =0;i < nt; i++){
            disponibles[i] = newGuarda(i);
        }
      }

      private static Guard newGuarda(final int i) {
        return new Monitor.Guard(monitores[i]) {
            @Override
            public boolean isSatisfied() {
              return tenedores[i].isDisponible() &&
                     tenedores[(i+1)%nt].isDisponible();
            }
        };
      }

      public static int getNumeroDeTenedores() {
        return nt;
      }
```

```
      public static Tenedor[] getTenedores() {
        return tenedores;
      }

      public static void obtenerTenedores(int i) {
        Preconditions.checkElementIndex(i, nt);

        monitores[i].enterWhenUninterruptibly(disponibles[i]);
                  tenedores[i].setEstado(Estado.Ocupado);
                  tenedores[(i+1)%nt].setEstado(Estado.Ocupado);
      }

      public static void liberarTenedores(int i){
        Preconditions.checkElementIndex(i, nt);
        monitores[i].leave();
        tenedores[i].setEstado(Estado.Libre);
        tenedores[(i+1)%nt].setEstado(Estado.Libre);
      }
}
```

Un tenedor es un objeto pasivo con estado disponible o no disponible.

```
public class Tenedor {

      public enum Estado {Libre, Ocupado};

      private Estado estado = Estado.Libre;
      private int i;
      public Tenedor(int i) { this.i = i; }

      public void setEstado(Estado estado) {
            this.estado = estado;
      }

      public Estado getEstado() { return estado; }

      public boolean isDisponible(){return estado == Estado.Libre;}

      public int getNumero() { return i; }

}
```

5. Diseño de objetos para la concurrencia: patrones

Como hemos visto anteriormente el programa concurrente puede ser considerado como un conjunto de objetos activos que acceden a otro conjunto de objetos pasivos. Ambos tienen que ser diseñados siguiendo dos políticas que en muchos casos son contradictorias entre sí:

- Seguridad: Todos los objetos deben estar solamente en estados válidos

- Viveza: La actividad de cada uno de los objetos activos debe progresar.

Ambos requerimientos son incompatibles en muchos casos. Para conseguir el mejor diseño posible es conveniente usar algunos patrones que ha ido depurando la experiencia. Algunos de estos patrones son, también, adecuados para conseguir u código más estructurad y mantenible. Veamos algunos.

5.1 Inmutabilidad

Siempre que sea posible usar objetos inmutables. Estos objetos nunca cambian de estado. Las acciones sobre objetos inmutables son siempre seguras y hacen que avance la actividad del objeto activo involucrado.

Los objetos inmutables tienen muchas aplicaciones: servicios sin estado, objetos que representan valores, etc. El uso de objetos inmutables simplifica el diseño concurrente porque pueden ser usados por más de un objeto activo y por lo tanto no tienen que establecer restricciones específicas. Pueden ser usados en el código como una constante. Esto favorece el avance de la actividad de los objetos activos (viveza).

Guava ofrece versiones inmutables de todas las colecciones*: ImmutableSet, ImmutableList, ImmutableMap, etc.*

Junto a las colecciones inmutables ofrecidas por *Guava* podemos usar las vistas no modificables de las colecciones ofrecidas en el API de Java. Muchas se consiguen usando métodos de la clase *Collections* como

- *static <T> Set<T> unmodifiableSet(Set<? extends T> s)*
- *static <T> List<T> unmodifiableList(List<? extends T> s)*
- *static <K,V> Map<K,V > unmodifiableMap(Map<? Extends K,? extends V> s)*
- *etc.*

5.2 Variables atómicas

En el paquete *java.util.concurrent.atomic* de java se ofrecen clases que soportan operaciones atómicas sobre las variables de ese tipo. Es decir ofrecen operaciones que se ejecutan como un todo. Ejemplos son: *AtomicInteger, AtomicIntegerArray, AtomicBoolean, AtomicLong, AtomicLongArray*, etc.

Un método típico de la clase AtomicInteger es:

- *public final int getAndSet(int newValue)*

Este método actualiza el valor al nuevo valor y devuelve la antiguo. Todo ello atómicamente.

5.3 Colecciones concurrentes

Java ofrece vistas concurrentes para cada una de las colecciones del API. Una colección concurrente es aquella que puede ser accedida en cada momento por una sola hebra. La mayoría están disponibles como métodos de la clase *Collections* como:

- *static <T> Set<T> synchronizedSet(Set<? extends T> s)*
- *static <T> List<T> synchronizedList(List<? extends T> s)*
- *static <K,V> Map<K,V > synchronizedMap(Map<? Extends K,? extends V> s)*
- *etc.*

Además Java ofrece otros tipos diseñados específicamente para su uso concurrente. Estos tipos, que no detallaremos aquí, ofrecen métodos adicionales a los tipos de los que derivan. Algunos de estos tipos son:

- *BlockingQueue:* Una cola acotada donde las hebras esperan si quieren añadir y la cola está vacía o quieren sacar y la cola está vacía.
- *ConcurrentMap:* Un *Map* con métodos añadidos para llevar a cabo algunas operaciones atómicas.
- *ConcurrentNavigableMap:* Un *NavigableMap* con métodos añadidos para llevar a cabo algunas operaciones atómicas.

Guava proporciona, además, el tipo

- *ConcurrentHashMultiset<E>:* Un Multiset en el que algunos métodos se ejecutan de forma atómica en entornos concurrentes.

Guava también proporciona vistas no modificables y sincronizadas de las colecciones que ofrece.

5.4 Tratamiento de las precondiciones y postcondiciones

Cuando diseñamos un tipo nuevo aparecen precondiciones, postcondiciones e invariantes en algunos de sus métodos. Las precondiciones deben ser respetadas para que el método correspondiente entre en un estado válido. Las postcondiciones y los invariantes deben ser válidos al final de la ejecución del método. En programación secuencial la implementación del método disparará una excepción si se intenta ejecutar sin que se cumpla la precondición. En programación concurrente hay otras alternativas como hacer esperar a la hebra correspondiente hasta que se verifique la condición. Por cada precondición tenemos, por tanto, algunas de las siguientes alternativas:

- Disparar una excepción si no se cumple la condición
- Hacer esperar la hebra correspondiente hasta que se cumpla la condición

- Hacer esperar la hebra correspondiente durante un tiempo hasta que se cumpla la condición. Si se agota el tiempo sin que se cumpla la condición disparar una excepción o devolver un valor adecuado.
- Ignorar la petición si no se cumple la precondición devolviendo el valor adecuado
- Intentar ejecutar el método y comprobar si se cumple la condición y el invariante. Si no se cumple deshacer los cambios. Este diseño requiere la implementación de mecanismos para deshacer los cambios o confirmarlos.
- Etc.

Todas esas posibilidades pueden ser implementadas usando los diversos métodos de la clase *Monitor.*

6. Tareas, ejecutores, servicios y futuros

En muchos casos es conveniente separar la tarea en sí (objeto que implementa *Runnable*) de la hebra que añade la actividad a ese objeto.

6.1 Servicios de ejecución

Un ejecutor es un objeto que tomando como parámetro una tarea (objeto que implementa *Runnable*) lo ejecuta dotándolo de una hebra. En Java un ejecutor es un objeto que implementa el interface *Executor*. Este interface tiene un solo método:

- *void execute(Runnable tarea):* Ejecuta la tarea en algún momento futuro, ya en una nueva hebra, en un pool de hebras o de alguna forma alternativa dependiendo de la implementación.

Un refinamiento del tipo *Executor* anterior es *ExecutorService.* Este proporciona un método *submit* que permite a tarea ejecutada devolver un valor. El método *submit* acepta un objeto de tipo *Runnable* o de tipo *Callable* y devuelve un objeto de tipo *Future* adecuado para gestionar la respuesta de la tarea cuando esté disponible. Los métodos *submit* del tipo *ExecutorService* tienen la siguiente signatura:

- *Future<?> submit(Runnable tarea)*
- *<T> Future<T> submit(Runnable tarea, T result)*
- *<T> Future<T> submit(Callable<T> tarea)*
- *<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tareas)*
- *<T> List<Future<T>> invokeAll(*
  Collection<? extends Callable<T>> tareas, long timeout, TimeUnit unit)
- *<T> T invokeAny(Collection<? extends Callable<T>> tareas)*
- *<T> T invokeAny(Collection<? extends Callable<T>> tareas,*
  long timeout, TimeUnit unit)

Como vemos el tipo *ExecutorService* dispone, también de métodos, para ejecutar colecciones de tareas. Los métodos *invokeAll* ejecutan las tareas, posiblemente con una limitación de tiempo, y devuelven una lista de objetos de tipo *Future*. Los métodos *invokeAny* ejecutan las tareas y devuelven el resultado de una que haya terminado con éxito (posiblemente con restricciones de tiempo.

El tipo *ScheduledExecutorService* refina a *ExecutorService* para ofrecer métodos similares a *submit* (ahora denominados *schedule*) pero adecuados para ejecutar tareas con retardos programados. Algunos de sus métodos son:

- *<T> ScheduledFuture<T> schedule(Runnable tarea, , long delay, TimeUnit unit)*
- *<T> ScheduledFuture<T> schedule(Callable<T> tarea, long delay, TimeUnit unit)*

6.2 Futuros

El tipo *Future<T>* ofrece métodos para gestionar los resultados de una tarea o incluso para cancelarla. Estos métodos son:

- *<T> T get():* La hebra que invoca el método espera hasta que la correspondiente tarea acabe y luego consigue el resultado.
- *<T> T get(long timeout, TimeUnit unit):* La hebra que invoca el método espera un máximo de *timeout* hasta que la correspondiente tarea acabe y luego consigue el resultado si está disponible.
- *boolean cancel(boolean mayInterruptIfRunning*): Intenta cancelar la tarea.
- *boolean isCancelled():* Si la tarea está cancelada
- *boolean isDone():* Si la tarea ha terminado.

Los objetos de tipo *Callable<T>* ofrecen el único método:

- *V call() throws Exception:* El método calcula un resultado y si no puede dispara una excepción.

Junto a los tipos anteriores disponemos de *RunnableFuture<T>* que es un subtipo de *Future<T>* y de *Runnable.* Una implementación de este tipo viene dada por la clase *FutureTask<V>* con los constructores:

- *FutureTask(Callable<V> callable)*
- *FutureTask(Runnable runnable, V result)*

Los objetos de este tipo ofrecen los métodos de los dos interfaces que implementan. Los objetos de tipo *FutureTask* son capaces de tomar una tarea (una implementación de *Runnable* o *Callable*) y ejecutarla. Esta tarea puede ser cancelada o puede proporcionar el resultado cuando termine (mediante el método *get*). En definitiva los objetos de este tipo proporcionan

una hebra para ejecutar una tarea y los métodos del tipo *Future* para conseguir el resultado o cancelar la tarea.

Junto a la posibilidad anterior (buscar una hebra y asignarla para ejecutar una tarea) tenemos la posibilidad de usar factorías de las mismas capaces de ejecutar una o varias tareas mediante el uso de una hebra o un pool de hebras. Son objetos del tipo *Executors.*

La clase *Executors* ofrece métodos para crear objetos de tipos *ExecutorService* y similares y adaptadores entre objetos de tipo *Runnable* y *Callable*. Algunos métodos son:

- *static Callable<Object> callable(Runnable task)*
- *static <T> Callable<T> callable(Runnable task, T result)*
- *static ThreadFactory privilegedThreadFactory():* Devuelve una factoría para crear hebras con los mismos privilegios que las actual.
- *static ExecutorService newCachedThreadPool():* Crea un pool de hebras que va creando hebras cuando las necesita y reúsa las que ya tiene cuando están disponibles.
- *static ExecutorService newCachedThreadPool(ThreadFactory threadFactory):* Crea un pool de hebras que va creando hebras cuando las necesita y reúsa las que ya tiene cuando están disponibles.
- *static ExecutorService newFixedThreadPool(int nThreads):* Crea un pool de hebras con un número fijo de las mismas.
- *static ExecutorService newFixedThreadPool(int nThreads,ThreadFactory threadFactory)*
- *static ExecutorService newSingleThreadExecutor():* Crea un servicio para ejecutar tareas con una sola hebra.
- *static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)*
- *static ScheduledExecutorService newSingleThreadScheduledExecutor()*
- *static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory)*
- *static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)*
- *static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)*

Guava ofrece el tipo de mucha utilidad *ListenableFuturo<T>* que refina el tipo *Future<T>*. También añade métodos para transformar los objetos de tipo *Future<T>* y diversos adaptadores. Veamos algunos detalles.

El tipo *ListenableFuturo<T>* refina *Future<T>* y añade el método:

- *void addListener(Runnable listener, Executor executor):* Añade la tarea listener para ser ejecutada en el executor cuando la tarea asociada al objeto de tipo *ListenableFuture<T>* haya terminado.

El tipo *ListenableFuture* se comporta como el tipo *Future* pero ofrece además la posibilidad de añadir una tarea que se ejecute sobre un *Executor* cuando la tarea original acabe.

```
...
ListenableFuture futureTask = executorService.submit(callableTask);
futureTask.addListener(new Runnable() {
       @Override
       public void run() {
          …
       }
     }, executorService);
```

El tipo *FutureCallback<V>* ofrecido por *Guava* tiene los métodos:

- *void onSuccess(V result):* Código a ejecutar si la tarea asociada tiene éxito devolviendo result.
- *void onFailure(Throwable t):* Código a ejcutar si la tarea asociada falla disparando la excepción *t*.

El tipo *AsyncFunction<I,O>* es similar al tipo *Function<I,O>* pero devuelve un *ListenableFuture<O>* en vez de un valor. Es decir obtiene un valor a partir de otro pero posiblemente de forma asíncrona. Tiene el único método:

- *ListenableFuture<O> apply(I input) throws Exception*

En la clase *Futures* de *Guava* se ofrecen métodos para transformar objetos de tipo *ListenableFuture* mediante una función, asociar *FutureCallBack* a *ListenableFuture*, etc. Algunos métodos de esta clase son:

- *static <I,O> ListenableFuture<O> transform(ListenableFuture<I> input, AsyncFunction<? super I,? extends O> function)*
- *static <I,O> ListenableFuture<O> transform(ListenableFuture<I> input, Function<? super I,? extends O> function)*
- *static <V> void addCallback(ListenableFuture<V> future, FutureCallback<? super V> callback)*

El tipo *ListeningExecutorService* refina *ExecutorService* del API de Java, visto más arriba. En este subtipo los métodos *submit* devuelven *ListenableFuture<T>.*

- *ListenableFuture<?> submit(Runnable tarea)*
- *<T> ListenableFuture<T> submit(Runnable tarea, T result)*
- *<T> ListenableFuture<T> submit(Callable<T> tarea)*
- *<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tareas)*
- *<T> List<Future<T>> invokeAll(*
  Collection<? extends Callable<T>> tareas, long timeout, TimeUnit unit)

En los métodos *invokeAll* cada *Future<T>* de la lista es una instancia de *ListenableFuture<T>. Guava* ofrece un decorador para obtener un *ListeningExecutorService* a partir de un *ExecutorService*.

De forma parecida Guava ofrece el tipo *ListeningScheduledExecutorService* similar al *ScheduledExecutorService* pero devolviendo *ListenableFuture<T>.*

En la clase *MoreExecutors* se ofrecen métodos para crear objetos de tipo *ExecutorService* y similares y adaptadores para obtener *ListeningExecutorService* y similares a partir de los anteriores. Algunos de sus métodos son:

- *static ListeningExecutorService listeningDecorator(ExecutorService delegate):* Crea un servicio de ejecución que devuelve *ListenableFuture*.
- *static ListeningScheduledExecutorService listeningDecorator(ScheduledExecutorService delegate)*
- *static ListeningExecutorService sameThreadExecutor():* Crea un servicio de ejecución que ejecuta cada tarea en la hebra que invoca *submit/execute*.

Una posibilidad de uso es crear un servicio de ejecución y luego añadirle un decorador.

```
ExecutorService executorService =
MoreExecutors.listeningDecorator(Executors.newCachedThreadPool());
```

Por último *Guava* ofrece el tipo *Service* y varias implementaciones del mismo. Un servicio es un objeto activo adecuado para llevar a cabo una tarea con métodos:

- *ListenableFuture<Service.State> start()*
- *ListenableFuture<Service.State> stop()*
- *Service.State state()*
- *boolean isRunning()*

Donde los estados del servicio son: *NEW, STARTING, RUNNING, STOPPING, TERMINATED*

6.3 Clases de ayuda para la sincronización

En programación concurrente es muy importante disponer de mecanismos para sincronizar el trabajo de varias hebras. Un sincronizador es un objeto que puede ayudarnos a coordinar a otras hebras. Veamos algunos de estos sincronizadores y su utilidad.

Para la sincronización de objetos activos que producen resultados con otros que los consumen se usan colas con capacidad de bloqueo. La clase *BlockingQueue* construye objetos de este tipo.

Otros sincronizadores incluyen semáforos, barreras y seguros. Veamos detalles de los mismos.

Un *seguro* es un sincronizador que puede retardar el progreso de unja hebra hasta que el seguro esté en un estado dado. Un seguro actúa como una puerta que está cerrada hasta que el seguro alcanza un estado. Una vez que el estado ha sido alcanzado la puerta permanece abierta para siempre. La clase *CountDownLatch,* ofrecida en el API de Java, ofrece una implementación de un seguro. Los objetos de este tipo tienen asociado un contador que es inicializado en el constructor. Las hebras que invocan el método *await* esperan hasta que el contador asociado sea cero. Los métodos que ofrece son:

- *void await() throws InterruptedException:* Espera hasta que el valor asociado ser cero.
- *public boolean await(long timeout, TimeUnit unit) throws InterruptedException:* Espera hasta que el valor asociado ser cero o pase el tiempo indicado.
- *public void countDown():* Disminuye el valor del contador en uno
- *public long getCount():* Devuelve el valor del contador

Con un objeto de este tipo inicializado a uno podemos asegurar que una hebra no comience hasta que los recursos que necesite estén disponibles (cuando el contador llegue a cero). Con un objeto de este tipo inicializado a un entero podemos conseguir que un número conocido de participantes no inicien su actividad hasta que todos estén listos para ello.

La clase *FutureTask* vista anteriormente puede actuar como un seguro. La invocación del método *get* hace esperar a la hebra correspondiente hasta que la tarea asociada acabe.

Los semáforos son usados para controlar el número de hebras que pueden acceder a un recurso a la vez. Los semáforos pueden ser usados para imponer restricciones sobre el número de elementos en una colección. El semáforo tiene asociado un número entero que es inicializado en el constructor y representa el número de permisos disponibles. Ofrece los métodos:

- *void acquire() throws InterruptedException:* Disminuye en uno el número de permisos disponibles. Bolque hasta que haya permisos disponibles.
- *int availablePermits():* Número de permisos disponibles.
- *void release():* Aumenta en uno el número de permisos disponibles.
- *…*

Otras clases útiles para sincronizar hebras son *CyclicBarrier.* Son objetos diseñados para sincronizar un número fijo de hebras que se proporciona en el constructor. Ofrece dos constructores:

- *CyclicBarrier(int parties):* Los parámetros son el número de hebras a sincronizar
- *CyclicBarrier(int parties, Runnable barrierAction):* Los parámetros son el número de hebras a sincronizar y la actividad a poner en marcha cunado la última de ellas entre en la barrera.

Los métodos son:

- *int await() throws InterruptedException, BrokenBarrierException:* Espera hasta que la última hebra haya invocado a este método. Devuelve el orden de invocación de la tarea actual. La última devuelve cero.
- *int getParties():* Devuelve el número de participantes establecidos en el constructor.
- *int getNumberWaiting():* Número de hebras esperando
- *void reset():* Sitúa la barrera en el estado inicial. Las hebras que estuvieran esperando reciben la excepción *BrokenBarrierException.*

7. Documentación

En el libro *Java Concurrency in Practice,* además de muchos ejemplos que veremos posteriormente, se proponen varias anotaciones que son adecuadas para documentar las decisiones de diseño concurrente. Estas anotaciones son: *@GuardedBy, @ThreadSafe, @Immutable y @NotThreadSafe.*

La anotación @Immutable indica que la clase es inmutable y por lo tanto segura ante el acceso de varias hebras. *@ThreadSafe* indica que la clase es segura ante el acceso de varias hebras.

La anotación *@GuardedBy* indica la llave que guarda el acceso a un campo o método.

8. Algunos ejemplos

8.1 Diseño de clases seguras

Como primer ejemplo veamos el diseño de clases mutables seguras. Para ello seguiremos los siguientes pasos:

- Identificar el estado de los objetos de la clase
- Identificar los invariantes
- Establecer una política de acceso al estado del objeto

Como primer ejemplo veamos una clase que implementa un contador. Tiene sólo un atributo de tipo long (lo cual implica el invariante Long.MIN_VALUE <= value <= Long.MAX_VALUE). Además la operación de actualización debe ser atómica. Una posible solución es:

```
@ThreadSafe
public final class Counter {
      @GuardedBy("this")
      private long value = 0;

      public synchronized long getValue() {
```

```
            return value;
      }

      public synchronized long increment() {
            if (value == Long.MAX_VALUE)
                throw new IllegalStateException("counter overflow");
            return ++value;
      }
}
```

El incumplimiento del invariante se ha gestionado con el disparo de una excepción. Para garantizar la atomicidad de *increment()* se ha usado la cláusula *synchronized.*

Una segunda solución es usar una variable atómica que ofrece las operaciones atómicas que necesitamos.

```
@ThreadSafe
public class Counter2 {

      private AtomicLong value = new AtomicLong();

      public long getValue() {
             return value.longValue();
      }

      public long increment() {
            if (value.longValue() == Long.MAX_VALUE)
                  throw new IllegalStateException("counter overflow");
            return value.incrementAndGet();
      }
}
```

Esta segunda solución es preferible a la primera. Un esquema similar podemos adoptar para diseñar acumuladores concurrentes que soporten la interacción de varias hebras.

Como segundo ejemplo veamos el diseño de un buffer acotado. Es un tipo muy útil en programación concurrente. Diseñamos una clase base capaz de ofrecer la funcionalidad básica. Esta clase no tiene en cuenta precondiciones, postcondiciones o invariantes.

```
public class BaseBoundedBuffer<V> {

      private final V[] buf;
      private int tail = 0;
      private int head = 0 ;
      private int count = 0;

      @SuppressWarnings("unchecked")
      public BaseBoundedBuffer(int capacity) {
            this.buf = (V[]) new Object[capacity];
      }
```

```
	protected void doPut(V v){
		buf[tail] = v;
		if (++tail == buf.length)
			tail = 0;
		++count;
	}

	protected V doTake(){
		V v = buf[head];
		buf[head] = null;
		if (++head == buf.length)
			head = 0;
		--count;
		return v;
	}

	public boolean isFull(){
		return count == buf.length;
	}

	public boolean isEmpty(){
		return count == 0;
	}
}
```

Sobre la clase base anterior diseñamos dos nuevas clases con dos políticas diferentes para hacer la clase segura y tratar con la concurrencia. La primera opción es disparar una excepción cuando no se cumplen las precondiciones. Junto con lo anterior se permite una hebra a la vez en el buffer. Es lo que se denomina política de poner obstáculos (*balking)*.

```
@ThreadSafe
public class BalkingBoundedBuffer<V> extends BaseBoundedBuffer<V> {

	public BalkingBoundedBuffer(int capacity) {
		super(capacity);
			// TODO Auto-generated constructor stub
	}

	@GuardedBy("this")
	public synchronized void put(V v) throws BufferFullException {
		if (isFull())
			throw new BufferFullException();
		doPut(v);
	}

	@GuardedBy("this")
	public synchronized V take()throws BufferEmptyException {
		if (isFull())
			throw new BufferEmptyException();
		return doTake();
	}
```

```
	...
}
```

La siguiente posibilidad es gestionar las precondiciones con guardas asociadas a un monitor. Esta política hace esperar a las hebras hasta que se cumpla la condición. Es una política más flexible que la anterior.

```
@ThreadSafe
public class BoundedBufferMonitor<V> extends BaseBoundedBuffer<V> {

	private Monitor monitor;
	private Monitor.Guard notFull;
	private Monitor.Guard notEmpty;

	public BoundedBufferMonitor(int capacity) {
		super(capacity);
		// TODO Auto-generated constructor stub
		monitor = new Monitor();

		notFull = new Monitor.Guard(monitor) {
			@Override
			public boolean isSatisfied() {
				// TODO Auto-generated method stub
				return !isFull();
			}
		};

		notEmpty = new Monitor.Guard(monitor) {
			@Override
			public boolean isSatisfied() {
				// TODO Auto-generated method stub
				return !isEmpty();
			}
		};
	}

	public void put(V v) throws InterruptedException {
		monitor.enterWhen(notFull);
		try {
			super.doPut(v);
		} finally{
			monitor.leave();
		}

	}

	public V take()throws InterruptedException {
		V r;
		monitor.enterWhen(notEmpty);
		try {
			r = super.doTake();
		} finally{
```

```
                    monitor.leave();
                }
                return r;
        }
        ...
}
```

Otra implementación alternativa usando semáforos es:

```
@ThreadSafe
public class BoundedBufferSemaphore<V> extends BaseBoundedBuffer<V> {
        private final Semaphore availableItems, availableSpaces;

        public BoundedBufferSemaphore(int capacity) {
            super(capacity);
            availableItems = new Semaphore(0);
            availableSpaces = new Semaphore(capacity);
        }

        @Override
        public boolean isEmpty() {
            return availableItems.availablePermits() == 0;
        }

        @Override
        public boolean isFull() {
            return availableSpaces.availablePermits() == 0;
        }

        public void put(V x) throws InterruptedException {
            availableSpaces.acquire();
            synchronized (this) {
                super.doPut(x);
            }
            availableItems.release();
        }

        public V take() throws InterruptedException {
            availableItems.acquire();
            V item;
            synchronized(this){
                item = super.doTake();
            }
            availableSpaces.release();
            return item;
        }
}
```

El problema de productor y consumidor puede implementarse usando una cola que bloquea (con alguna de las implementaciones anteriores o la proporcionada por el API de Java).

```
class Producer implements Runnable {
        private final BlockingQueue queue;
        Producer(BlockingQueue q) { queue = q; }
        public void run() {
           try {
             while (true) { queue.put(produce()); }
           } catch (InterruptedException ex) {
               ... handle ...
           }
         }
         Object produce() { ... }
}

class Consumer implements Runnable {
        private final BlockingQueue queue;
        Consumer(BlockingQueue q) { queue = q; }
        public void run() {
            try {
              while (true) { consume(queue.take()); }
            } catch (InterruptedException ex) {
                ... handle ...
            }
        }
        void consume(Object x) { ... }
}

class Setup {
        void main() {
           BlockingQueue q = new BufferBounded(10);
           Producer p = new Producer(q);
           Consumer c1 = new Consumer(q);
           Consumer c2 = new Consumer(q);
           new Thread(p).start();
           new Thread(c1).start();
           new Thread(c2).start();
         }
}
```

8.2 Gestión de recursos: bloqueos

El bloqueo (*deadlock*) es una situación en la que varias hebras no pueden progresar porque cada una está esperando recursos que han sido reservados por la otra. Un ejemplo de posible bloque ocurre en el ejemplo de los filósofos que vimos anteriormente. Cada filósofo necesita dos tenedores: el que tiene a la izquierda y el que tiene a la derecha. Si cada filósofo reserva los tenedores uno a uno puede darse una situación de bloqueo entre dos filósofos vecinos *i* e *i+1*. El bloqueo aparece si cada filósofo adquiere su tenedor izquierdo. En ese caso todos los filósofos se quedan esperando a conseguir su tenedor derecho.

En general este problema se puede resolver proporcionando un orden global a los recursos disponibles. Si disponemos de ese orden global entonces se trata de que cada hebra obtenga todos los recursos que necesite en ese orden. Siempre se puede inducir un orden global basado en el *hashCode* de cada objeto e introducir una llave adicional para el caso de que los *hashCode* sean iguales.

En el caso de los tenedores el orden global viene proporcionado por el número del tenedor. En el caso de cuentas de un banco por la clase de la cuenta que debe ser Comparable o si no existe por el código hash de la misma.

Otra solución al problema viene es el uso de monitores y guardas asociadas. Los recursos ofrecen guardas que devolverán true cuando el conjunto de recursos que sean necesarios para que una hebra pueda hacer su trabajo estén disponibles.

Veamos fragmentos de cada una de estas soluciones. En el caso de uso de un orden global para acceder a los recursos (*FilosofoSync, FilosofosSync, TenedoresSync*) los tenedores no impondrían ninguna restricción. Es el filósofo el que reserva las llaves asociadas a cada tenedor en el orden global establecido.

```
...
public FilosofoSync(int i) {
            this.estado = Estado.Esperando;
            this.i = i;
            int i1 = i;
            int i2 = (i+1)% FilosofosSync.numeroDeFilosofos;
            if(i1 < i2){
                  this.t1= TenedoresSync.getTenedor(i1);
                  this.t2 = TenedoresSync.getTenedor(i2);
            } else{
                  this.t1= TenedoresSync.getTenedor(i2);
                  this.t2=TenedoresSync.getTenedor(i1);
            }
}
...

@Override
public void run() {
      while(!fin){
            esperar();
            synchronized(t1){
                  synchronized(t2){
                        t1.setEstado(Tenedor.Estado.Ocupado);
                        t2.setEstado(Tenedor.Estado.Ocupado);
                        comer();
                        t1.setEstado(Tenedor.Estado.Libre);
                        t2.setEstado(Tenedor.Estado.Libre);
                  }
            }
            pensar();
      }
```

```
        setEstado(Estado.Fin);
}
...
```

Frente a esta solución podemos considerar la solución vista más arriba (*Filosofo, Filosofos, Tenedores*) donde para acceder a los recursos (*Tenedores*) se establecen monitores. Y por cada monitor una guarda que devuelve *true* cuando están libres los tenedores *i* e *(i+1)%numeroDeFilosofos*. La clase tenedores, en ese caso, ofrece un método para coger los dos tenedores necesarios a la vez. La guarda establecida hace esperar a las hebras hasta que tienen todos los recursos necesarios.

Si para los recursos no estuviera disponible un orden global entonces podemos usar el *hashCode* de cada objeto y un objeto adicional para el caso de *hashCode* iguales. Así se implementan en la clase siguiente las transferencias de fondos:

```
public class InduceLockOrder {

      private static final Object tieLock = new Object();

      public void transferMoney(final Account fromAcct,
       final Account toAcct,final DollarAmount amount)
           throws InsufficientFundsException {

            int fromHash = System.identityHashCode(fromAcct);
            int toHash = System.identityHashCode(toAcct);

            if (fromHash < toHash) {
                  synchronized (fromAcct) {
                     synchronized (toAcct) {
                        new Helper().transfer(fromAcct,toAcct,amount);
                     }
                  }
            } else if (fromHash > toHash) {
                  synchronized (toAcct) {
                    synchronized (fromAcct) {
                        new Helper().transfer(fromAcct,toAcct,amount);
                    }
                  }
            } else {
                  synchronized (tieLock) {
                    synchronized (fromAcct) {
                      synchronized (toAcct) {
                        new Helper().transfer(fromAcct,toAcct,amount);
                      }
                    }
                  }
            }
      }

      interface DollarAmount extends Comparable<DollarAmount> {
      }
```

```
      interface Account {
            void debit(DollarAmount d);

            void credit(DollarAmount d);

            DollarAmount getBalance();

            int getAcctNo();
      }

      class InsufficientFundsException extends Exception {
      }

      class Helper {

        public void transfer(final Account fromAcct,
           final Account toAcct,final DollarAmount amount)
             throws InsufficientFundsException {

            if (fromAcct.getBalance().compareTo(amount) < 0)
                  throw new InsufficientFundsException();
            else {
                  fromAcct.debit(amount);
                  toAcct.credit(amount);
            }
        }

      }
}
```

8.3 Diseño de una función con memorización

En muchos casos en importante disponer de una función que pueda calcular valores a partir de argumentos y tenga en cuenta cálculos previamente hechos (tenga memoria en definitiva).

Por concretar un cálculo puede ser cualquier implementación del tipo:

```
public interface Computable<A, V> {
      V compute(A arg) throws InterruptedException;
}
```

Diseñamos la implementación para tener en cuenta que el cálculo puede durar un tiempo no previsto de antemano y por lo tanto es necesario prever que el resultado puede estar disponible no en el momento. Cada tarea, por lo tanto, podemos ponerla en marcha construyendo un objeto de tipo *FutureTask*. Para recordar las tareas que se pusieron en marcha previamente usamos un objeto de tipo *ConcurrentMap<A,Future<V>>.*

La implementación tiene en cuenta que debe ser usada en un entorno concurrente, que las tareas pueden ser canceladas y que añadir un nuevo para argumento-resultado futuro debe ser ejecutado como una acción atómica. Esto puede conseguirse con el método *pulfAbsent* del tipo *ConcurretMap*.

```
public class Memorizer<A, V> implements Computable<A, V> {
      private final ConcurrentMap<A, Future<V>> cache
            = new ConcurrentHashMap<A, Future<V>>();
      private final Computable<A, V> c;

      public Memorizer(Computable<A, V> c) {
            this.c = c;
      }

      public V compute(final A arg) throws InterruptedException {
            while (true) {
                  Future<V> f = cache.get(arg);
                  if (f == null) {
                        Callable<V> eval = new Callable<V>() {
                              public V call()
                                  throws InterruptedException {
                                     return c.compute(arg);
                              }
                        };
                        FutureTask<V> ft = new FutureTask<V>(eval);
                        f = cache.putIfAbsent(arg, ft);
                        if (f == null) {
                              f = ft;
                              ft.run();
                        }
                  }
                  try {
                        return f.get();
                  } catch (CancellationException e) {
                        cache.remove(arg, f);
                  } catch (ExecutionException e) {
                        throw LaunderThrowable.launderThrowable(
                                      e.getCause());
                  }
            }
      }
}
```

8.4 Ejecución de tareas

La ejecución de tareas puede ser hecha usando objetos de tipo *Executor*. Estos objetos están diseñados con la filosofía productor consumidor. Las actividades (productores) envían al ejecutor tareas para ser ejecutadas. El ejecutor dispone de una o un conjunto de hebras (consumidores) que aceptan las tareas y las ejecutan.

Hay muchas implementaciones posibles para un *Executor* tal como hemos visto más arriba. La clase *Executors* actúa como una factoría para crear objetos de tipo *Executor* (o alguno de sus subtipos como *ExecutorService*) con diferentes implementaciones. El subtipo *ExecutorService* dispone de métodos que devuelven objetos de tipo *Future* cuando se envía una tarea a ejecución.

Por último los objetos del subtipo *ScheduledExecutorService* pueden aceptar tareas para ser ejecutadas tras un retardo o incluso ser ejecutadas periódicamente.

Una de las posibles implementaciones para los servicios de ejecución son los pools de hebras. Un problema puede aparecer para escoger el número de hebras más adecuado. Es un problema difícil. Pero algunas recomendaciones se pueden hacer (ver *Java Concurrency in Practice*). Suponiendo que dispongamos de *N* cpus si las tareas son de cálculo intensivo y tenemos *n* tareas entonces el número de hebras *T* podría ser *T = N+1*. Pero en general hará falta una fase de ajuste del número de hebras del pool.

8.5 Paralelización de algoritmos recursivos

En esta sección veremos distintos patrones para paralelizar algoritmos recursivos.

El primer patrón es transformar una ejecución secuencial de una lista de tareas independientes en una paralela de las mismas. La idea es usar un objeto de tipo *Executor* y enviarle a lista de tareas. El equivalente paralelo al método *processSequentially* es el método *processInParallel.*

El segundo patrón es transformar un algoritmo recursivo que hace una búsqueda en profundidad en un grafo implícito y donde el resultado no depende del resultado de la llamada recursiva. Los resultados calculados en todos los nodos son devueltos en una colección como en el caso anterior. El grafo implícito es modelado por el tipo *Node<T>*. Este tipo tiene como propiedad una lista de hijos y un método para calcular un resultado asociado al nodo. Un algoritmo de este tipo tiene el esquema del método *SequentialRecursive*. El equivalente, *parallelRecursive*, también deja los resultados en una colección pero en vez de calcular el resultado poner marcha una tarea y calculará el resultado. Para ejecutar todas las tareas que van apareciendo usa un *Executor*. Para usar esta segunda versión hay que crear un *ExecutorService* y una cola para mantener los resultados cuando sean producidos. El *ExecutorService* construido tiene métodos específicos para ejecutar todas las tareas que se han enviado a ejecutar y esperar hasta que terminen. Son los métodos:

- *void shutdown()*
- *boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException*

El primero inicia el cierre del servicio de ejecución y las tareas previamente enviadas y el segundo espera hasta que acabe. Esas ideas se aplican en el método *getParallelResults.*

Todos los métodos anteriores están agrupados en la clase abstracta *TransformingSequential.* Para obtener clases concretas hay que implementar el método que hace el procesamiento asociado a un elemento de una lista (método *process*) y una implementación del tipo *Node.*

```
public abstract class TransformingSequential {

      public <T> void processSequentially(List<T> elements) {
      for (T e : elements)
           process(e);
      }

      public <T> void processInParallel(Executor exec,
                                        List<Element> elements) {
           for (final T e : elements)
           exec.execute(new Runnable() {
                public void run() {
                     process(e);
                }
           });
      }

      public abstract <T> void process(T e);

      public <T> void sequentialRecursive(List<Node<T>> nodes,
                                          Collection<T> results) {
            for (Node<T> n : nodes) {
                  results.add(n.compute());
                  sequentialRecursive(n.getChildren(), results);
            }
      }

      public <T> void parallelRecursive(final Executor exec,
                  List<Node<T>> nodes, final Collection<T> results) {
            for (final Node<T> n : nodes) {
                  exec.execute(new Runnable() {
                      public void run() {
                          results.add(n.compute());
                      }
                  });
                  parallelRecursive(exec, n.getChildren(), results);
            }
       }

       public <T> Collection<T> getParallelResults(
                    List<Node<T>> nodes) throws InterruptedException {
              ExecutorService exec = Executors.newCachedThreadPool();
              Queue<T> resultQueue = new ConcurrentLinkedQueue<T>();
              parallelRecursive(exec, nodes, resultQueue);
              exec.shutdown();
              exec.awaitTermination(Long.MAX_VALUE, TimeUnit.SECONDS);
              return resultQueue;
       }
```

```
        interface Node <T> {
                T compute();
                List<Node<T>> getChildren();
        }
}
```

En el código anterior podemos ver que cuando usamos la factoría Executors para construir nuevos servicios de ejecución tenemos diferentes posibilidades según vimos anteriormente:

- *static ExecutorService newCachedThreadPool():* Crea un pool de hebras que va creando hebras cuando las necesita y reúsa las que ya tiene cuando están disponibles.
- *static ExecutorService newFixedThreadPool(int nThreads):* Crea un pool de hebras con un número fijo de las mismas.
- *static ExecutorService newSingleThreadExecutor():* Crea un servicio para ejecutar tareas con una sola hebra.

8.6 Problema del puzzle concurrente

Una aplicación de las ideas de paralelización de algoritmos vista antes es la solución concurrente al problema del puzle.

Un puzle puede ser modelado como un grafo implícito dónde sus nodos son de la forma general:

```
public interface Puzzle<P, M> {
      P initialPosition();
      boolean isGoal(P position);
      Set<M> legalMoves(P position);
      P move(P position, M move);
}
```

Es decir existen una serie de posibles posiciones (P) y un conjunto de posibles movimientos dada una posición (M). Hay una posición inicial, una posición objetivo y de una posición a otra se puede ir mediante un movimiento legal.

El puzle se podría modelar alternativamente como un grafo tal como vimos en capítulos anteriores.

En el problema del puzle (y en otros similares) necesitaremos guardar las posiciones ya alcanzadas y el camino hasta ellas. Las soluciones del problema podemos concretarlas como una secuencia de movimientos desde el nodo inicial. Una posibilidad es con una implementación de la forma:

```
@Immutable
public class PuzzleNode<P, M> {
      final P pos;
      final M move;
```

```
        final PuzzleNode<P, M> prev;

        public PuzzleNode(P pos, M move, PuzzleNode<P, M> prev) {
                this.pos = pos;
                this.move = move;
                this.prev = prev;
        }

        List<M> asMoveList() {
                List<M> solution = new LinkedList<M>();
                for (PuzzleNode<P, M> n = this; n.move != null; n = n.prev)
                        solution.add(0, n.move);
                return solution;
        }
}
```

Con los tipos anteriores podemos diseñar un algoritmo secuencial para resolver el problema del puzle. Es decir encontrar un camino desde el nodo inicial al final. El algoritmo no está pensado para encontrar el camino mínimo. Para ello usa un conjunto de posiciones ya vistas. Va alcanzando consecutivamente las posiciones no encontradas hasta alcanzar el nodo final si es posible. El algoritmo va haciendo una búsqueda en profundidad del nodo objetivo.

```
public class SequentialPuzzleSolver<P, M> {
        private final Puzzle<P, M> puzzle;
        private final Set<P> seen = new HashSet<P>();

        public SequentialPuzzleSolver(Puzzle<P, M> puzzle) {
                this.puzzle = puzzle;
        }

        public List<M> solve() {
                P pos = puzzle.initialPosition();
                return search(new PuzzleNode<P, M>(pos, null, null));
        }

        private List<M> search(PuzzleNode<P, M> node) {
                if (!seen.contains(node.pos)) {
                        seen.add(node.pos);
                        if (puzzle.isGoal(node.pos))
                                return node.asMoveList();
                        for (M move : puzzle.legalMoves(node.pos)) {
                                P pos = puzzle.move(node.pos, move);
                                PuzzleNode<P, M> child =
                                    new PuzzleNode<P, M>(pos, move, node);
                                List<M> result = search(child);
                                if (result != null)
                                        return result;
                        }
                }
                return null;
        }
```

```
}
```

Igualmente podemos implementar una versión concurrente de ese algoritmo. La versión concurrente hace una búsqueda en anchura. El algoritmo lo que hace es asociar una tarea a cada nodo alcanzado y mandarla a ejecutar. Las tareas asociadas a cada nodo se construyen mediante la clase *SolverTask.*

El algoritmo dispone de un servicio de ejecución (*exec*) que se ha construido como un pool de hebras. También dispone de un conjunto de posiciones ya visitadas que se ha implementado como un *ConcurrentMap<P, Boolean>.* Es la variable *seen*. La solución será del tipo *List<M>* y estamos buscando una solución solamente. Es decir una lista de movimientos para alcanzar el nodo objetivo desde el nodo inicial. Es la variable *solution* de tipo *ValueLatch<T >* con *T* instanciado a *PuzzleNode<P, M>.* Los detalles de esta clase los veremos más abajo pero adelantamos que tiene los métodos:

- *boolean isSet():* Verdadero si ya existe una solución
- *void setValue(T newValue):* Actualiza la solución con el nuevo valor
- *T getValue():* La hebra invocante espera hasta que haya una solución disponible.

Lo primero es diseñara una tarea asociada a cada *Node* que se vaya encontrando. Es la clase *SolverTask* que hereda de la clase *PuzzleNode<P,M>* que hemos diseñador previamente e implementa *Runnable*. La tarea acaba si ya se ha encontrado una solución *(solution.isSet())* o si la posición ha sido alcanzada previamente *(seen.putIfAbsent(pos, true) != null)*. También se acaba si se encuentra el nodo objetivo. En este caso se actualiza la solución *(solution.setValue(this)).* En otro caso se crea una nueva tarea y se manda a ejecutar con las nuevas posiciones alcanzadas con los movimientos permitidos desde la posición actual *(exec.execute(newTask(puzzle.move(pos, m), m, this));*

Con los elementos anteriores el método *solve* resuelve el problema. Crea la tarea asociada al nodo inicial y la envía a ejecución. Espera a que haya una solución disponible si la hay y la devuelve. Finalmente inicia el cierre del servicio de ejecución *(exec.shutdown()).*

Todas estas ideas están en la clase *ConcurrentPuzzleSolver.*

```
public class ConcurrentPuzzleSolver<P, M> {
      private final Puzzle<P, M> puzzle;
      private final ExecutorService exec;
      private final ConcurrentMap<P, Boolean> seen;
      protected final ValueLatch<PuzzleNode<P, M>> solution =
            new ValueLatch<PuzzleNode<P, M>>();

      public ConcurrentPuzzleSolver(Puzzle<P, M> puzzle) {
            this.puzzle = puzzle;
            this.exec = initThreadPool();
            this.seen = new ConcurrentHashMap<P, Boolean>();
            if (exec instanceof ThreadPoolExecutor) {
                  ThreadPoolExecutor tpe = (ThreadPoolExecutor) exec;
                  tpe.setRejectedExecutionHandler(
                     new ThreadPoolExecutor.DiscardPolicy());
```

```
            }
      }

      private ExecutorService initThreadPool() {
            return Executors.newCachedThreadPool();
      }

      public List<M> solve() throws InterruptedException {
            try {
                  P p = puzzle.initialPosition();
                  exec.execute(newTask(p, null, null));
                  // bloquea hasta encontrar la solución
                  PuzzleNode<P, M> solnPuzzleNode =
                      solution.getValue();
                  return (solnPuzzleNode == null) ? null :
                            solnPuzzleNode.asMoveList();
            } finally {
                  exec.shutdown();
            }
      }

      protected Runnable newTask(P p, M m, PuzzleNode<P, M> n) {
            return new SolverTask(p, m, n);
      }

      protected class SolverTask
               extends PuzzleNode<P, M>  implements Runnable {

            SolverTask(P pos, M move, PuzzleNode<P, M> prev) {
                  super(pos, move, prev);
            }

            public void run() {
                  if (solution.isSet()
                        || seen.putIfAbsent(pos, true) != null)
                      return; // ya resuelto o posición alcanzada
                  if (puzzle.isGoal(pos))
                      solution.setValue(this);
                  else
                      for (M m : puzzle.legalMoves(pos))
                           exec.execute(newTask(
                                       puzzle.move(pos, m), m, this));
            }
      }
}
```

La solución anterior no acaba si el nodo objetivo no es alcanzable. Claramente si el nodo objetivo no es alcanzable la solución debe ser *null* pero el diseño anterior queda esperando en el método *solution.getValue().* La clase siguiente, *PuzzleSolver*, refina la anterior. Para conseguir que el algoritmo acabe cuanta las tareas que están activas y pone la solución a *null*

cuando el contador se hace cero. El contador debe ser implementado con una variable de tipo *AtomicInteger*.

```
public class PuzzleSolver<P, M> extends ConcurrentPuzzleSolver<P, M> {

      PuzzleSolver(Puzzle<P, M> puzzle) {
            super(puzzle);
      }

      private final AtomicInteger taskCount = new AtomicInteger(0);

      protected Runnable newTask(P p, M m, PuzzleNode<P, M> n) {
            return new CountingSolverTask(p, m, n);
      }

      class CountingSolverTask extends SolverTask {

           CountingSolverTask(P pos, M move, PuzzleNode<P, M> prev) {
                  super(pos, move, prev);
                  taskCount.incrementAndGet();
           }

            public void run() {
                  try {
                        super.run();
                  } finally {
                        if (taskCount.decrementAndGet() == 0)
                              solution.setValue(null);
                  }
            }
      }
}
```

La clase *ValueLatch* implementa un seguro que permite hacer esperar a hebras hasta que haya un primer valor para la variable *value*. Esto se consigue reutilizando la clase *CountDownLatch.*

```
@ThreadSafe
public class ValueLatch<T> {
      @GuardedBy("this")
      private T value = null;
      private final CountDownLatch done = new CountDownLatch(1);

      public boolean isSet() {
            return (done.getCount() == 0);
      }

      public synchronized void setValue(T newValue) {
            if (!isSet()) {
                  value = newValue;
                  done.countDown();
            }
      }
```

```
	public T getValue() throws InterruptedException {
		done.await();
		synchronized (this) {
			return value;
		}
	}
}
```

9. Testing de programas concurrentes

El *testing* de programas concurrentes tiene dos partes bien diferenciadas. En primer lugar el *testing* de la clase como si fuera secuencial. Es el *testing* para comprobar la *corrección* de la clase. Para ello debemos identificar invariantes, precondiciones y post-condiciones. A partir de esos requisitos es posible escribir los test. Si esos requisitos no están disponibles el proceso de diseño de los test se convierte, por lo general, en un proceso iterativo.

Los primeros test podemos diseñarlos para comprobar el estado de un buffer recién creado o el estado cuando hemos añadido un número de ítems igual a su capacidad.

El test adecuado para comprobar que una hebra se bloquea en los sitios diseñados es más complejo. Este es el test *testTakeBlocksWhenEmpty.* La idea es usar una hebra que debe bloquearse y si no lo hace se produce un fallo. La hebra principal la interrumpe tras un tiempo y espera que termine.

```
public class BoundedBufferTest {

	private static final long LOCKUP_DETECT_TIMEOUT = 1000;

	@Test
	public void testIsEmptyWhenConstructed() {
	//BoundedBufferMonitor<Integer> bb =
	//     new BoundedBufferMonitor<Integer>(10);
		BoundedBufferSemaphore<Integer> bb =
		 new BoundedBufferSemaphore<Integer>(10);
		assertTrue(bb.isEmpty());
		assertFalse(bb.isFull());
	}

	@Test
	public void testIsFullAfterPuts() throws InterruptedException {
	//BoundedBufferMonitor<Integer> bb =
	//      new BoundedBufferMonitor<Integer>(10);
		BoundedBufferSemaphore<Integer> bb =
		    new BoundedBufferSemaphore<Integer>(10);
		for (int i = 0; i < 10; i++)
		      bb.put(i);
		assertTrue(bb.isFull());
```

```
            assertFalse(bb.isEmpty());
      }

      @Test
      public void testTakeBlocksWhenEmpty() {
      //final BoundedBufferMonitor<Integer> bb =
      // new BoundedBufferMonitor<Integer>(10);
            final BoundedBufferSemaphore<Integer> bb =
               new BoundedBufferSemaphore<Integer>(10);
            Thread taker = new Thread() {
                  public void run() {
                        try {
                              @SuppressWarnings("unused")
                              int unused = bb.take();
                              fail(); // if we get here, it's an error
                        } catch (InterruptedException success) {
                        }
                  }
            };
            try {
                  taker.start();
                  Thread.sleep(LOCKUP_DETECT_TIMEOUT);
                  taker.interrupt();
                  taker.join(LOCKUP_DETECT_TIMEOUT);
                  assertFalse(taker.isAlive());
            } catch (Exception unexpected) {
                  fail();
            }

      }
}
```

10. Usos de ListenableFuture y FutureCallback

El Api de *Guava* aporta dos tipos de tienen gran utilidad: *ListenableFuture* y *FutureCalBack*. El primero, como vimos en apartados anteriores, admite nuevas tareas para ser ejecutadas cuando la tarea asociada acabe. Esto es posible con el método *addListener*. *Guava* ofrece un decorador que permite transformar un *ExecutorService* en un *ListeningExecutorService*. La diferencia está en que le primero produce *Future* objetos cuando acepta una tarea para ser ejecutada y el segundo *ListenableFuture* objetos. Estos últimos, como hemos visto anteriormente, pueden aceptar *listeners* capaces de ejecutar tareas, en servicios de ejecución dados, cuando la tarea a la que están asociados acabe.

De la misma forma a un objeto de tipo *ListenableFuture* le podemos asociar una acción simple para ser ejecutada cuando acabe. Esto se consigue con el método *Futures.addCallBack(listenableFuture, callBack).*

```
public class TestListenableFuture {

	public static void main(String[] args) {

		ListeningExecutorService exec =
			MoreExecutors.listeningDecorator(
				Executors.newSingleThreadExecutor());

		ListenableFuture<Integer> lf = exec.submit(
			new Callable<Integer>(){
			@Override
			public Integer call() throws Exception {
				Integer n = Math2.getEnteroAleatorio(0, 1000);
				System.out.println("Entero = "+n);
				Thread.sleep(1000);
				if(n%2==0){
					throw new Exception();
				} else {
					return n;
				}
			}});

		lf.addListener(new Runnable(){
			@Override
			public void run() {
				try {
					Thread.sleep(1000);
				} catch (InterruptedException e) {
					e.printStackTrace();
				}
				System.out.println("Ha estado haciendo otra tarea");
			}}, exec);

			Futures.<Integer>addCallback(lf,
			  new FutureCallback<Integer>(){
			    @Override
			    public void onFailure(Throwable e) {
			      System.out.println("Se ha producido una excepción");
			    }
			    @Override
			    public void onSuccess(Integer e) {
			      Long e2 = (long) e*e;
			      System.out.println("El cuadrado es = "+e2);
			   }});

			try {
				exec.awaitTermination(5,TimeUnit.SECONDS);
			} catch (InterruptedException e) {
				e.printStackTrace();
			}
			exec.shutdown();
	}
}
```

11. Diseño de interfaces de usuario

Cuando diseñamos interfaces usuario con la tecnología Swing es necesario usar ideas y conceptos de programación concurrente.

La primera idea es tener en cuenta las hebras que participan en la ejecución de una interfaz gráfica (si usamos la tecnología *Swing*). Son la hebra inicial (que hace la llamada para inicializar la interfaz de usuario), la hebra que gestiona los eventos y un conjunto de hebras adicionales creadas por el programador para conseguir que la interfaz de usuario no se quede congelada cuando hay que ejecutar tareas de larga duración. Estas hebras se crean por clases específicas de la tecnología Swing.

La clase *SwingUtilities* ofrece métodos para crear la hebra encargada de gestionar los eventos de la interfaz. Son los métodos *invokeLater* y *invokeAndWait*. Ambos reciben una tarea y la ponen en ejecución. La diferencia es que el método *invokeLater* pone en marcha la tarea y termina mientras el método *invokeAndWait* pone en marcha la tarea y espera que acabe. Así por ejemplo para inicializar la clase PuntoFrame2 que tiene varios componentes gráficos podemos hacerlo de la forma:

```
public static void initGrafico(){
      SwingUtilities.invokeLater(new Runnable() {
         public void run() {
            try {
                        PuntoFrame2 frame = new PuntoFrame2();
                        frame.setVisible(true);
            } catch (Exception e) {
                        e.printStackTrace();
            }
         }
      });
}
```

El método anterior puede ser invocado desde el programa principal o desde otra hebra. Los cálculos costosos deben hacerse en hebras adicionales para conseguir que la interfaz gráfica no se quede congelada. Las hebras adicionales es conveniente crearlas como instancias de la clase *SwingWorker<T,V>.* Esta clase implementa el tipo *Future<T>* y *Runnable*. Esta clase tiene, además de los métodos definidos por los tipos que implementa, los métodos:

- *void execute():* Inicializa la actividad y termina.
- *protected abstract T doInBackground():* Método que hay que implementar y dónde se describe la tarea a llevar a cabo.
- *protected void process(List<V> chunks):*

- *protected final void publish(V... chunks):* Método para ser usado dentro del método *doInBackground().* Envía datos para ser procesados por el método *process(…)* anterior.
- *T get():* Método asociado al tipo *Future<T>.* El invocante espera hasta que la actividad termine.
- *int getProgress(), void setProgress(int progress):* Métodos asociados la propiedad *Progress* (progreso de la actividad).

En la clase siguiente vemos los detalles de una tarea se supone costosas y que implementamos reutilizando la clase *SwingWorker*. La tarea calcula los primeros números primos. Para hacer más visible el funcionamiento hace que el cálculo de cada número primo tarde más haciendo que la hebra espere un tiempo.

La clase toma como parámetros un área de texto (donde se irán imprimiendo lo números primos calculados), un botón para imprimir mensajes que indiquen si se está calculando o la tarea ha sido interrumpida y el número de números primos a calcular.

Los objetos de tipo *SwingWorker* tienen una propiedad (*Progress*) que puede ser actualizada según el cálculo vaya avanzando. Esta propiedad puede ser mostrada en la interfaz gráfica mediante una barra de progreso.

El método *doInBackground* empieza su trabajo cuando se llama al método *execute*. Cuando termina este método termina la tarea y se ejecuta el método *done*. Dentro del código de este método ya podemos llamar al método *get* (sin que la hebra correspondiente tenga que esperar porque el resultado ya está disponible).

La secuencia de valores publicados (sucesivos parámetros de *publish*) son recogidos en forma de lista en el método *process.* En este caso se imprimen en el área de texto prevista para ello.

```
class PrimeNumbersTask extends SwingWorker<List<Long>, Long> {

	private int numbersToFind;
	private JTextArea textArea;
	private JButton mensajeButton;
	private List<Long> numbers = Lists.newArrayList();

	PrimeNumbersTask(JTextArea textArea, JButton mensajeButton,
			int numbersToFind) {
		this.textArea = textArea;
		this.numbersToFind = numbersToFind;
		this.mensajeButton = mensajeButton;
	}

	@Override
	public List<Long> doInBackground() {
		Long number = 1L;
		numbers.add(number);
		publish(number);
		while (numbers.size() < numbersToFind && ! isCancelled()) {
			number = nextPrimeNumber(number,numbers);
			publish(number);
```

```
                numbers.add(number);
                setProgress(100 * numbers.size() / numbersToFind);
          }
          return numbers;
    }

    private Long nextPrimeNumber(Long n, List<Long> primes) {
          Long r = null;
          for(Long i=n+1;true;i++){
                if(isPrime(i,primes)){
                      r = i;
                      break;
                }
          }
          try {
                Thread.sleep(1000);
          } catch (InterruptedException e) {
          }
          return r;
    }

    private boolean isPrime(Long nn, List<Long> primes){
          boolean r = true;
          for(Long e: primes){
                if(e==1){
                      continue;
                }
                if(nn%e==0){
                      r = false;
                      break;
                }
          }
          return r;
    }

    @Override
    protected void process(List<Long> chunks) {
          for (Long number : chunks) {
                textArea.append(number + "\n");
          }
    }

    @Override
    protected void done() {
          List<Long> r = null;
          try {
                r = get();
                System.out.println(r);
          } catch (Exception e) {
                mensajeButton.setText("Tarea Interumpida");
          }

    }
```

```
}
```

La tarea anterior tiene que ser llamada desde la encargada de construir los elementos gráficos, e iniciar y sincronizar las propiedades de unos. Esta clase es recomendable crearla con herramientas como la que hemos visto en el capítulo anterior. Algunos detalles son:

```
public class PrimeNumbers extends JFrame {

	private static final long serialVersionUID = 1L;
	private JPanel contentPane;
	private static JTextArea textArea;
	private JProgressBar progressBar;
	public static int n = 100;
	private static PrimeNumbersTask task;
	private JButton stopButton;
	private JButton mensajeButton;
	private JPanel panel;
	private JPanel panel_1;
	private JButton startButton;
	public PrimeNumbers() {

		setDefaultValues();

		……
		startButton = new JButton("START");
		startButton.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent arg0) {
				mensajeButton.setEnabled(true);
				mensajeButton.setText("Calculando...");
					task.execute();
			}
		});
		…
		stopButton.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent arg0) {
				task.cancel(true);
			}
		});
		…
		initDataBindings();
	}
	……
	protected void initDataBindings() {
		task = new PrimeNumbersTask(textArea, mensajeButton, n);

		BeanProperty<PrimeNumbersTask, Integer>
		  primeNumbersTaskBeanProperty =
		       BeanProperty.create("progress");
		BeanProperty<JProgressBar, Integer>
		  jProgressBarBeanProperty = BeanProperty.create("value");
		AutoBinding<PrimeNumbersTask, Integer,
```

```
                JProgressBar, Integer> autoBinding =
                  Bindings.createAutoBinding(UpdateStrategy.READ, task,
                     primeNumbersTaskBeanProperty, progressBar,
                        jProgressBarBeanProperty);
            autoBinding.bind();


      }
}
```