UNIDAD 2

Programación multihilo

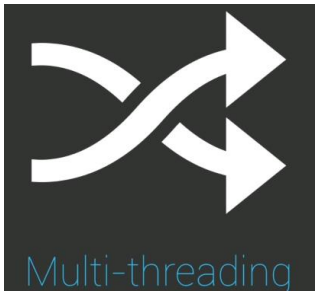
CONTENIDOS

2.1. Fundamentos de la programación multihilo
2.2. Programación multihilo: clases y librerías
2.3. Programación asíncrona
2.4. Problemas y soluciones de la programación concurrente. Sincronización

2.1. Fundamentos de la programación multihilo

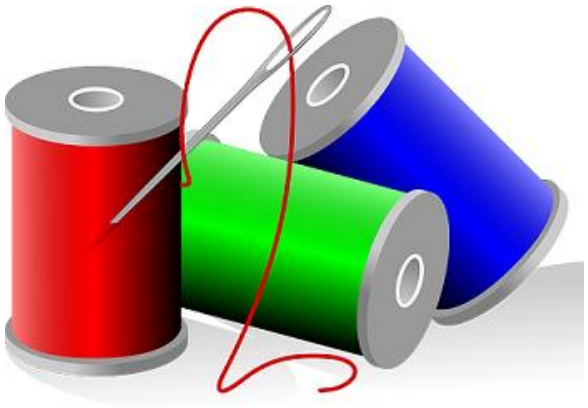
- Programación secuencial o de único hilo
- Programación concurrente o multihilo
- Estados de los hilos
- Introducción a los problemas de concurrencia

La promoción multihilo, es una técnica utilizada para conseguir procesamiento simultaneo.



El sistema operativo se encarga de hacer convivir los diferentes procesos y de repartir los recursos entre sí, por lo que cuando se programa no hay que tener en cuenta aspectos relacionados con cómo se van a gestionar los procesos o cómo estos van a tener acceso a los recursos. Salvo el optimizar el uso de memoria y conseguir que los algoritmos sean eficientes, los programadores no tienen que preocuparse del resto: el sistema operativo se encarga de la concurrencia a nivel de proceso.

Es dentro del interior de los procesos donde la programación tiene algo que decir con respecto a la concurrencia, mediante la programación multihilo.



Un **hilo,** también conocido como ***thread,*** es una pequeña unidad de computación que se ejecuta dentro del contexto de un proceso. Todos los programas utilizan hilos.

En el caso de un programa absolutamente secuencial, el hilo de ejecución es único, lo que provoca que cada sentencia tenga que esperar que la sentencia inmediatamente anterior se ejecute completamente antes de comenzar su ejecución.

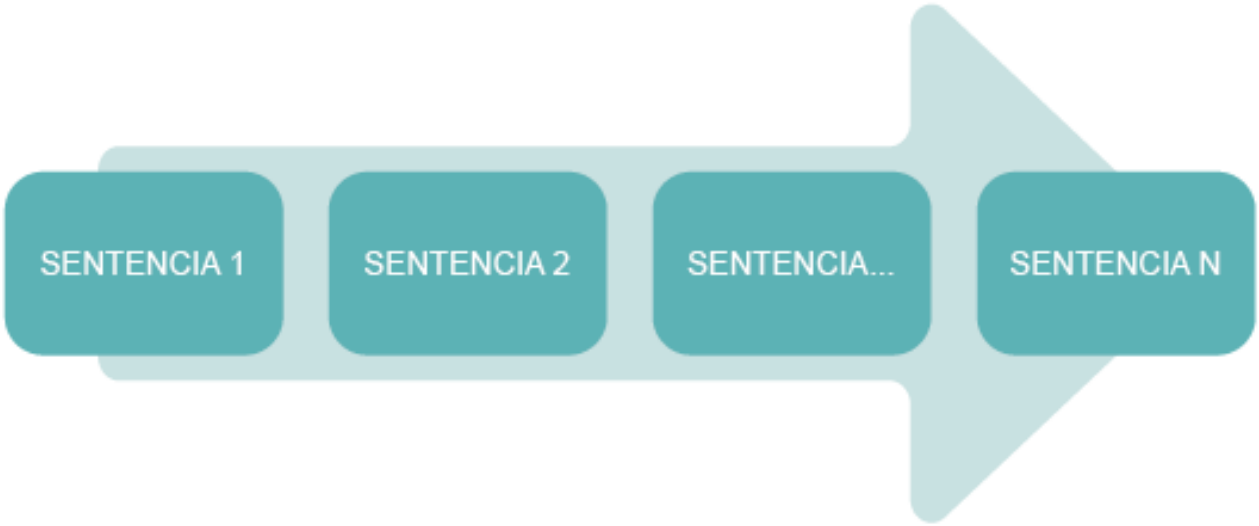


Figura 2.1. En los programas de un único hilo, la ejecución de las sentencias es secuencial.

En cambio, en un programa multihilo, algunas de las sentencias se ejecutan simultáneamente, ya que los hilos creados y activos en un momento dado acceden a los recursos de procesamiento sin necesitar esperar a que otras partes del programa terminen.

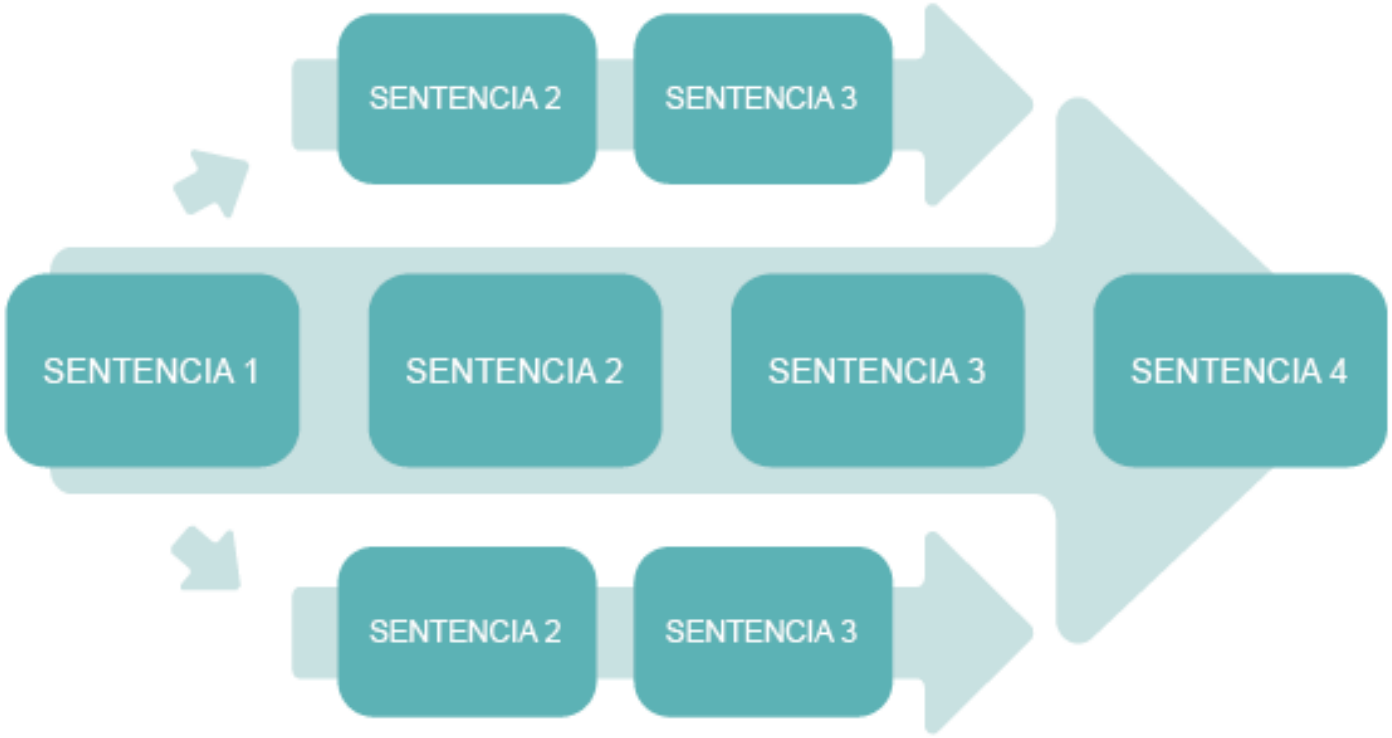
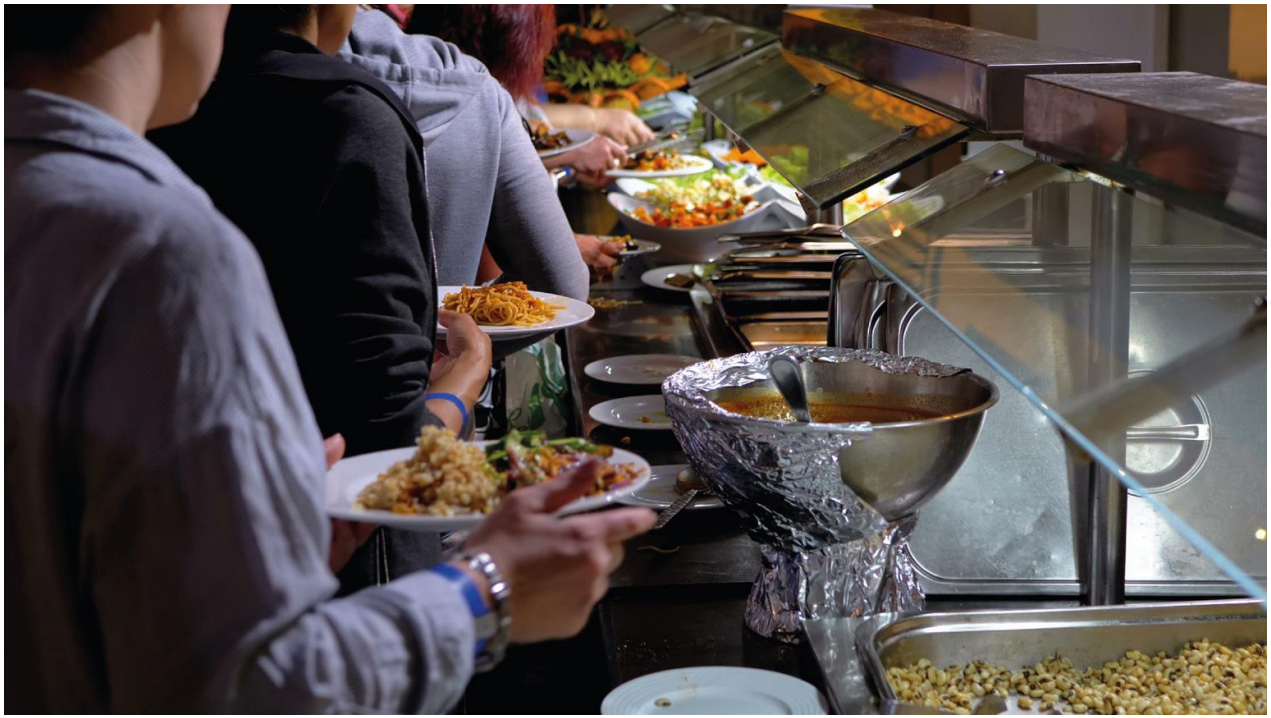


Figura 2.2. En los programas de múltiples hilos, algunas sentencias se ejecutan de manera concurrente.

Muchos programas funcionan de manera secuencial bien porque no es posible una ejecución concurrente, o bien porque no se ha considerado oportuno que así sea. Sea como sea, los programas de ejecución secuencial no aprovechan los recursos del sistema para reducir los tiempos de proceso.



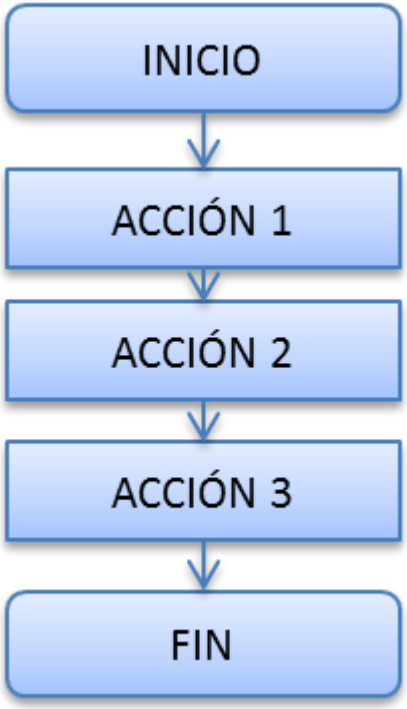


Figura 2.3. En la programación secuencial, las sentencias se ejecutan una detrás de otra.

Figura 2.4. En la programación concurrente varias sentencias se pueden ejecutar simultáneamente.

No todos los programas se pueden ejecutar en entornos multihilo. Es importante hacer un análisis previo de los procesos y de los algoritmos para determinar si es o no viable utilizar la **concurrencia** en la solución.

Características de los Hilos

A continuación, se citan algunas de las características que tienen los hilos:

- **Dependencia del proceso.** No se pueden ejecutar independientemente. Siempre se tienen que ejecutar dentro del contexto de un proceso.
- **Ligereza.** Al ejecutarse dentro del contexto de un proceso, no requiere generar procesos nuevos, por lo que son óptimos desde el punto de vista del uso de recursos. Se pueden generar gran cantidad de hilos sin que provoquen pérdidas de memoria.
- **Compartición de recursos.** Dentro del mismo proceso, los hilos comparten espacio de memoria. Esto implica que pueden sufrir colisiones en los accesos a las variables provocando errores de concurrencia.
- **Paralelismo.** Aprovechan los núcleos del procesador generando un paralelismo real, siempre dentro de las capacidades del procesador.
- **Concurrencia.** Permiten atender de manera concurrente múltiples peticiones. Esto es especialmente importante en servidores web y de bases de datos, por ejemplo.

Para ilustrar cómo se comporta un programa monohilo vs un programa multihilo, se puede hacer la analogía del trabajo de un camarero en una cafetería

2.1.1. Programación secuencial o de único hilo

Antes de continuar con los Hilos en java, vamos a preparar nuestro entorno de trabajo

Realizar Programa en java UD2_Actividad_1_1.
(Ratón Comelón)
(Solo clases y propiedades)

Este programa se realizará de la forma tradicional, sin la utilización de hilos. Más adelante lo iremos Modificando, hasta convertirlo en un programa Multihilos.

Enunciado del Ejercicio

Tarea: Simulador de Laboratorio de Ratones
Objetivo: Desarrollar un simulador básico que modele el comportamiento de ratones en un laboratorio utilizando clases, propiedades y métodos en Java.

Clase Raton:

Propiedades :
nombre: El nombre del ratón.
tiempoAlimentacion: El tiempo que tarda el ratón en alimentarse.
peso: Representará el peso del ratón
Métodos :
comer(): Simula al ratón comiendo durante un período de tiempo.
ejercitar(): Simulará al ratón ejercitándose y disminuirá su peso.
dormir(): Simulará al ratón durmiendo y aumentará su peso.

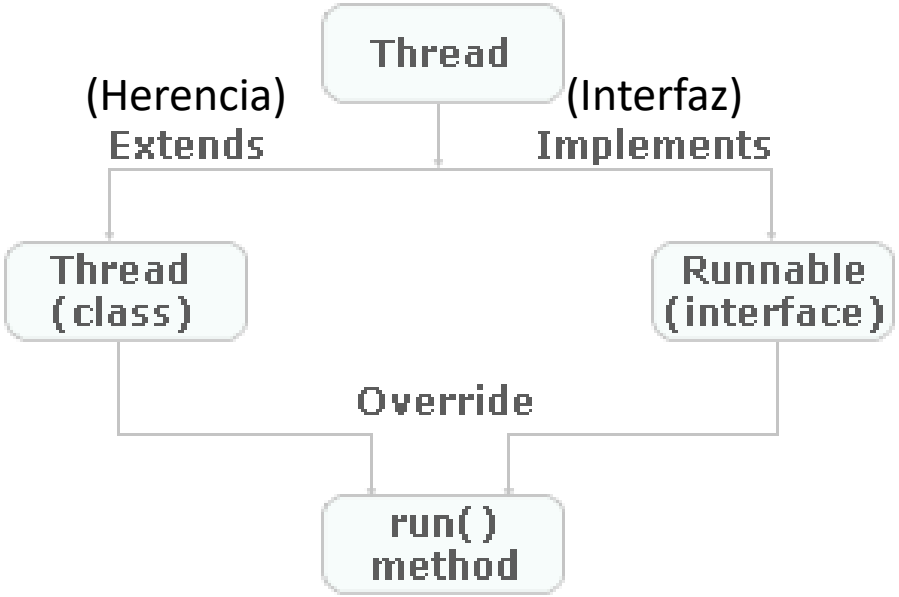
Clase Laboratorio:

Propiedades:
ratones: Una lista de todos los ratones en el laboratorio.
Métodos:
añadirRaton(Raton raton): Añadirá un ratón a ratones.
eliminarRaton(Raton raton): Eliminará un ratón de ratones.
simularDia(): Simulará un día en el laboratorio donde todos los ratones comen, se ejercitan y duermen

Hilos en Java

En Java hay dos formas de especificar hilos, extendiendo la clase Thread o implementando la interfaz Runnable.

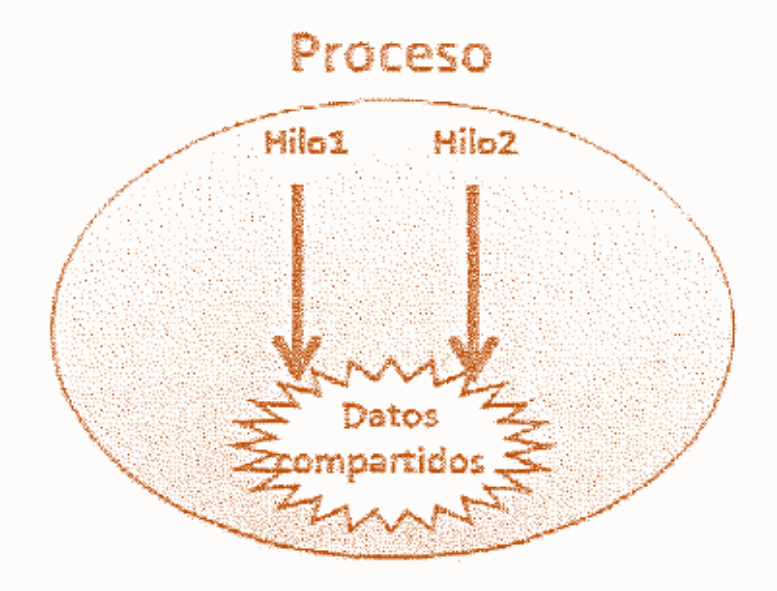
- La interfaz **Runnable** proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando dicha interfaz. ***La interfaz Runnable debería ser utilizada si la clase solamente va a utilizar la funcionalidad run de los hilos.***
- La clase **Thread** es responsable de producir hilos funcionales para otras clases e implementa la interfaz Runnable
- Tanto para la interfaz como la clase, habrá que *sobreescribir* el método ***run(),*** que es el cual ejecutará el código que tenga dentro de forma concurrente.



Para programar concurrentemente podemos dividir nuestro programa en hilos. Java proporciona la construcción de programas concurrentes mediante la clase **Thread** (hilo o hebra). Esta clase permite ejecutar código en un hilo de ejecución independiente.

En Java existen dos formas de utilizar o crear un hilo:

- Creando una clase que herede de la clase **Thread** y sobrecargando el método *run()*.
- Implementando la interface **Runnable**, y declararando el método *run()*.



El siguiente ejemplo crea un hilo de nombre *HiloSimple* heredando de la clase **Thread**. En el método *run()* se indican las líneas de código que se ejecutarán simultáneamente con las otras partes del programa. Cuando se termina la ejecución de ese método, el hilo de ejecución termina también:

```
public class HiloSimple extends Thread {
    public void run() {
      for (int i=0; i<5;i++)
        System.out.println("En el Hilo... ");
    }
}
```

Para usar el hilo creo la clase *UsaHilo*:

```
public class UsaHilo {
     public static void main(String[] args) {
       HiloSimple hs = new HiloSimple();
       hs.start();
       for (int i=0; i<5;i++)
        System.out.println("Fuera del hilo..");
    }
}
```

Desde esta clase se arranca el hilo: primero se invoca al operador *new* para crear el hilo y luego al método *start()* que invoca al método *run()*. La compilación y ejecución muestra la siguiente salida, en la que se puede observar que se intercala las instrucciones del hilo y de fuera del hilo.

Las 2 clases anteriores implementando la interfaz **Runnable** quedarían así:

```
public class HiloSimple2 implements Runnable{
   public void run() {
      for (int i=0; i<5;i++)
         System.out.println("En el Hilo...");
   }
}
public class UsaHilo2 {
    public static void main(String[] args) {
      HiloSimple2 hs = new HiloSimple2();
      Thread t = new Thread(hs);
      t.start();
      for (int i=0; i<5;i++)
        System.out.println("Fuera del hilo..");
   }
}
```

Recordando un poco…

La implementación de la *interface Runnable* obliga a programar el método sin argumentos *run*.

```
1  package es.paraninfo.multihilos;
2  public class HiloViaInterface implements Runnable {
3      @Override
4      public void run() {
5
6      }
7  }
```

Heredando de la clase *Thread* esto no es obligatorio porque dicha clase ya es una implementación de la *interface Runnable*.

```
1  package es.paraninfo.multihilos;
2
3  public class HiloViaHerencia extends Thread {
4
5  }
```

¿Qué implementación usar?

Como Java no permite herencia múltiple, usar la interfaz tiene ventajas ya que permite diseñar software teniendo en cuenta superclases.

Además, si usamos Runnable, podemos lanzar varias instancias de un hilo sobre un mismo objeto, y Thread creará un hilo por cada objeto.

El hecho de implementar interfaces favorece la composición (las clases tienen instancias de otras clases) y no la herencia, uno de los principales errores que conllevan software mal diseñado. https://en.wikipedia.org/wiki/Composition_over_inheritance

En términos de funcionalidad, ambas implementaciones hacen lo mismo: ejecutan un bucle que imprime un mensaje desde el hilo creado y otro mensaje desde el hilo principal. Sin embargo, la forma en que crean y gestionan los hilos difiere según la estrategia que elijan: ***extender*** **Thread** o ***implementar*** **Runnable**.

En la práctica, es más común y recomendable usar la interfaz ***Runnable*** para definir la tarea que se ejecutará en un hilo, ya que ofrece una mayor flexibilidad y evita las limitaciones de la herencia única de Java

Métodos de la clase **Thread**

Métodos	Descripción
start()	Hace que el hilo comience la ejecución; la máquina virtual de Java llama al método run() de este hilo
boolean isAlive()	Comprueba si el hilo está vivo
sleep(long mils)	Hace que el hilo en ejecución se duerma temporalmente durante el número de milisegundos indicado. Puede lanzar la excepción InterruptedException
run()	Forma el cuerpo del hilo. Es llamado por el método start() después de que el hilo se haya incilaizado.
String toString()	Devuelve una representación en formado cadena de este hilo. Nombre, prioridad y el grupo de hilos
long getId()	Devuelve el identificador del hilo
void join()	Espera que el hilo se muera
void yield()	Pare el hilo en ejecución para ejecutar otros hilos
String getName()	Devuelve el nombre del hilo
setName(String name)	Cambia el nombre del hilo

2.1.2. Programación concurrente o multihilo

El ejemplo del laboratorio de ratones se puede programar de manera concurrente, de forma sencilla, ya que no hay ningún recurso compartido.

Para ello, basta con convertir cada objeto de la clase *Raton* en un hilo (*thread*) y programar aquello que se quiere que ocurra concurrentemente dentro del método *run*. Una vez creadas las instancias, se invoca al método *start* de cada una de ellas, lo que provoca la ejecución del contenido del método *run* en un hilo independiente.

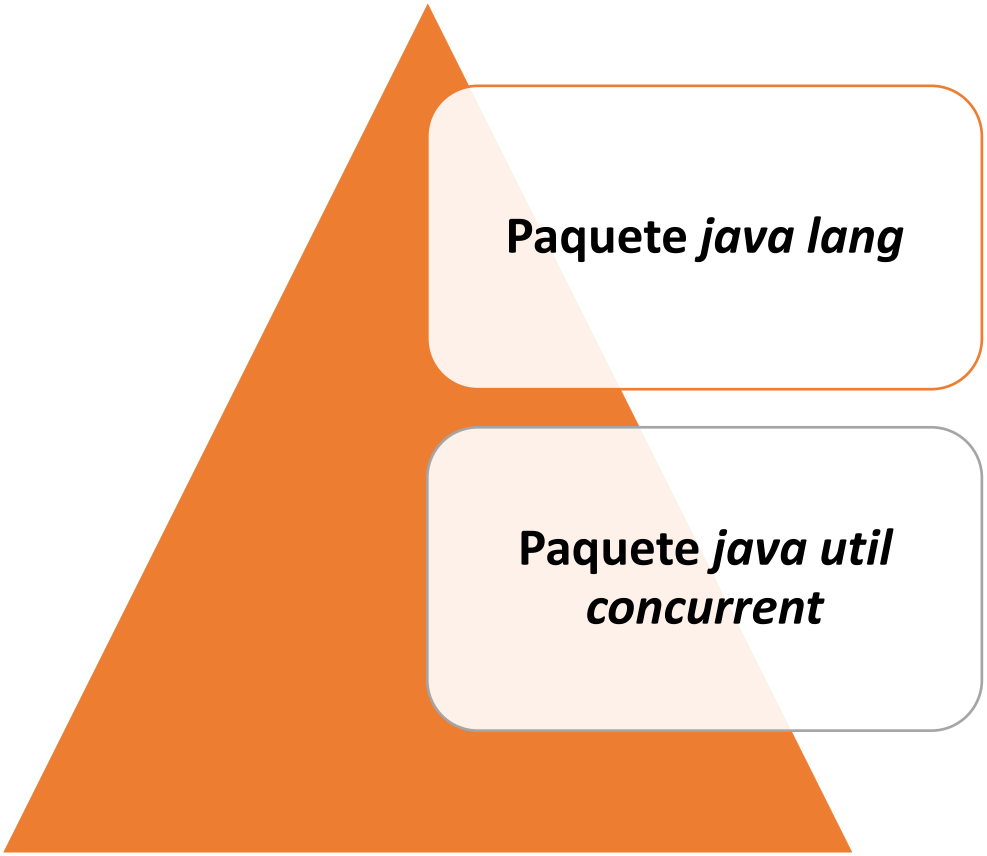
En Java existen dos formas para la creación de hilos:

- Implementando la *interface* java.lang.Runnable.
- Heredando de la clase java.lang.Thread.

Actividad: UD_2 Actividad_1_3

Actividad Propuesta 2.2
(Generación de problemas de concurrencia)

2.2. Programación multihilo: clases y librerías



Estados de un Thread

Estado	Valor en Thread.State	Descripción
Nuevo	NEW	El hilo está creado, pero aún no se ha arrancado.
Ejecutable	RUNNABLE	El hilo está arrancado y podría estar en ejecución o pendiente de ejecución.
Bloqueado	BLOCKED	Bloqueado por un monitor.
Esperando	WAITING	El hilo está esperando a que otro hilo realice una acción determinada.
Esperando un tiempo	TIME_WAITING	El hilo está esperando a que otro hilo realice una acción determinada en un período de tiempo concreto.
Finalizado	TERMINATED	El hilo ha terminado su ejecución.

Tabla 2.1. Estados de los hilos y su correspondencia en Java

Actividad: UD_2 Actividad_1_3_1(estado de los hilos)

Actividad: UD_2 Actividad_1_3_2(estado de los hilos Varios Ratones)

Actividad: UD_2 Actividad_1_3_3(GUI estado de los hilos Varios Ratones)

2.2.1. Paquete *java.lang*

En el paquete paquete *java.util.concurrent* se incluyen un interesante conjunto de clases y herramientas relacionadas con la programación concurrente.

Figura 2.5. Java dispone de una extensa biblioteca de clases dedicadas a la concurrencia.

Nombre	Tipo	Descripción
Runnable	Interface	Esta *interface* debe implementarse por aquellas clases que se quieran ejecutar como un *thread*. Define un método sin argumentos *run*.
Thread	Clase	Esta clase implementa la *interface Runnable*, siendo un hilo en sí misma. Una clase que hereda de *Thread* y que sobrescribe el método *run* se ejecuta de manera concurrente si se invoca al método *start*. Con la clase *Thread* se puede envolver un objeto que implemente *Runnable* provocando que se ejecute en un hilo independiente.
Timer	Clase	Permite la programación de la ejecución de tareas en diferido y de forma repetitiva mediante el método *schedule*. Este método espera tareas programables, representadas por objetos de la clase *TimerTask*, así como información del momento de inicio de la tarea y del tiempo que debe transcurrir entre cada una de las ejecuciones de la misma.
TimerTask	Clase abstracta	Heredando de esta clase y sobrescribiendo el método *run* se puede crear una tarea programable con la clase *Timer*.

Tabla 2.2. Clases del paquete *java.lang* implicadas en la programación concurrente

Executor

Es una *interface* para la definición de sistemas multihilo. Permite ejecutar tareas de tipo *Runnable*. Algunas de sus interfaces derivadas, así como clases directamente relacionadas se muestran en la Tabla 2.3.

Nombre	Tipo	Descripción
ExecutorService	Interface	*Subinterface* de *Executor*, permite gestionar tareas asíncronas.
ScheduledExecutorService	Interface	Permite la planificación de la ejecución de tareas asíncronas.
Executors	Clase	Fábrica de objetos *Executor*, *ExecutorServices*, *SchedulledExecutorService*, *ThreadFactory* y *Callable*.
TimeUnit	Enumeración	Proporciona representaciones de unidades de tiempo con distinta granularidad, desde días hasta nanosegundos.

Tabla 2.3. Componentes relacionados con la interfaz *Executor*

Executors:
Es una utilidad de **alto nivel** que facilita la gestión de hilos en Java.

- Permite la creación de grupos de hilos llamados "pools de hilos" (thread pools).
- Los ***Executors*** proporcionan métodos de fábrica para crear diferentes tipos de ***ExecutorService***, como ***newFixedThreadPool***, ***newCachedThreadPool***, etc.
- Es más **escalable** y se recomienda sobre el uso directo de la clase **Thread** cuando se gestionan muchas tareas en paralelo.

Ejemplo Executors:

ExecutorService executor = Executors.newFixedThreadPool(4); // Crea un pool con 4 hilos.
executor.execute(tarea); // Ejecuta la tarea utilizando uno de los hilos del pool.
executor.shutdown(); // Cierra el ExecutorService y sus hilos asociados.

Resumen implementación de hilos en Java:

- Usar **Runnable** y **Thread** directamente puede ser adecuado para ***tareas simples*** y cuando solo se necesitan unos pocos hilos.
- Para aplicaciones más ***grandes o con muchas tareas concurrentes***, se recomienda usar **Executors** debido a su facilidad de gestión y escalabilidad.
- **Runnable** es solo una definición de una tarea, mientras que **Thread** y **Executors** proporcionan mecanismos para ejecutar dichas tareas en hilos.

Actividad: UD2_Actividad_2_2_00_Executor (Básico)

Creación y gestión de **pools** de **hilos** usando ***ExecutorService***

Define 5 tareas que serán enviadas y ejecutadas por el pool de hilos. Cada tarea debe:
Imprimir un mensaje indicando el inicio de la tarea y el nombre del hilo en el que se está ejecutando.
Dormir durante 2 segundos para simular algún procesamiento.
Imprimir un mensaje indicando el final de la tarea y el nombre del hilo en el que se ejecutó.

PROPUESTA: **Actividad: UD2_Actividad_2_2_01_Executor (Dual Pool)**

Modifica el ejercicio anterior para que se utilizan 2 pools de hilos y se ejecutan múltiples tareas en cada pool.

Realización de actividades en el aula

Actividad: UD2_Actividad_2_2_1_Regadio

A. **SistemaRiego.java:**
Utiliza **TimerTask** (***Extends***) y **Timer,** para programar la ejecución repetida de una tarea.

A. **SistemaRiego02.java:**
Utiliza Runnable (implements) y se importan diversas clases y interfaces necesarias para la programación concurrente en Java, como Executors, ScheduledExecutorService y TimeUnit.

Descubre la funcionalidad de los métodos:
- ***newSingleThreadScheduledExecutor***
- ***scheduleAtFixedRate***

Ambos códigos logran el mismo resultado, pero mediante diferentes enfoques:
- **ScheduledExecutorService** es más moderno y ofrece una mayor flexibilidad.
- **Timer** y **TimerTask** pueden ser adecuados para tareas más simples.

Actividad: UD2_Actividad_2_2_2_Regadio (Executors STOP)

SistemaRiego.java_STOP:

Modifica el ejercicio SistemaRiego02.java:
Que utiliza **Runnable** (implements) y **Executors**, para que el proceso se detenga con una combinación del teclado.

Actividad: UD2_Actividad_2_2_3_Regadio (BIZONA)

SistemaRiego.java_BIZONA:

Modifica el ejercicio anterior , para que sea capaz de regar dos zonas al mismo tiempo, y se pueda iniciar y detener cada proceso por separado, a través de comandos lanzados por el usuario. **Exit**, para salir del programa inicial.

Actividad Propuesta 2.4
Planificación de ejercicio de estiramiento.

Java proporciona componentes que resuelven todo el espectro de necesidades relacionado con las colas para crear entornos seguros de ejecución de soluciones de mensajería, gestión de colas de trabajo y sistemas basados en esquemas productor-consumidor en entornos concurrentes.

Nombre	Tipo	Descripción
ConcurrentLinkedQueue	Clase	Una cola enlazada resistente a hilos.
ConcurrentLinkedDeque	Clase	Una cola doblemente enlazada resistente a hilos.
BlockingQueue	Interface	Una cola que incluye bloqueos de espera para la gestión del espacio. Algunas de sus implementaciones son LinkedBlockingQueue, ArrayBlockingQueue, SynchronousQueue, PriorityBlockingQueue y DelayQueue.
TransferQueue	Interface	Un tipo de BlockingQueue especialmente diseñado para mensajería.
BlockingDeque	Interface	Una cola doblemente enlazada que incluye bloqueos de espera para la gestión del espacio. Dispone de una implementación en la clase LinkedBlockingDeque.

Tabla 2.4. Clases e interfaces para la implementación de colas que admiten concurrencia

Listas en Multihilo

Si se espera que una cola sea accedida por múltiples hilos simultáneamente, es crucial elegir una estructura de datos que sea segura frente a la concurrencia para evitar errores inesperados.

Normalmente emplearíamos una estructura de datos ***LinkedList***. Dado que esta estructura no es segura para operaciones concurrentes (es decir, no está diseñada para manejar accesos simultáneos de múltiples hilos), su uso en un contexto multihilo puede llevar a errores.

La solución óptima a la hora de usar listas para compartir multihilos, sería utilizar una estructura de datos ***ConcurrentLinkedDeque***. Esta estructura está diseñada específicamente para operaciones concurrentes, lo que significa que puede manejar accesos simultáneos de múltiples hilos sin causar errores.

Usando LinkedList:

```
import java.util.LinkedList;
import java.util.Queue;

public class LinkedListExample {
    public static void main(String[] args) {
        Queue<String> queue = new
LinkedList<>();

        // Agregar elemento a la cola
        queue.add("elemento");

        // Remover elemento de la cola
        String elemento = queue.poll();
    }
}
```

Usando ConcurrentLinkedDeque:

```
import
java.util.concurrent.ConcurrentLinkedDeque;

public class ConcurrentLinkedDequeExample {
    public static void main(String[] args) {
        ConcurrentLinkedDeque<String> queue =
new ConcurrentLinkedDeque<>();

        // Agregar elemento a la cola
        queue.add("elemento");

        // Remover elemento de la cola
        String elemento = queue.poll();
    }
}
```

Actividad comparar ejecución **LinkedList y ConcurrentLinkedDeque en** Colas

Actividad: UD_2_Acividad_2_2_3_QUEUES:

- ColaNoConcurrente.java
- ColaConcurrente.java
- ConcurrentLinkedQueueExample
- ConcurrentLinkedDequeExample.java

Con estos ejemplos veremos como trabajar con estructuras de listas concurrentes

El paquete *java.util.concurrent* proporciona cinco clases específicas para conseguir que la concurrencia entre hilos se desarrolle de manera correcta.

Nombre	Tipo	Descripción
Semaphore	Clase	Proporciona el mecanismo clásico para regular el acceso de los hilos a recursos de uso limitado.
CountDownLatch	Clase	Proporciona una ayuda para la sincronización cuando los hilos deben esperar hasta que se realicen un conjunto de operaciones.
CyclingBarrier	Clase	Proporciona una ayuda para la sincronización cuando unos hilos deben esperar hasta que otros hilos alcancen un punto de ejecución determinado.
Phaser	Clase	Proporciona una funcionalidad similar a *CountDownLatch* y a *CyclingBarrier* a través de un uso más flexible.
Exchanger	Clase	Permite intercambiar elementos entre dos hilos.

Tabla 2.5. Clases del paquete *java.util.concurrent* que permiten sincronizar hilos

Actividad: UD_2_Acividad_2_2_3_QUEUES (ReentrantLock() Cajeros Automáticos)

Actividad: UD_2_Acividad_2_2_3_QUEUES (**ConcurrentLinkedDeque** () Cajeros Automáticos)

Synchronized()

Variables volatile

En Java, una variable **volatile** es una indicación al compilador y a la máquina virtual de Java (JVM) de que la variable puede ser modificada por diferentes **hilos** que están ejecutándose en el programa. La palabra clave volatile se utiliza para marcar una variable de instancia para garantizar la visibilidad de cambios entre hilos.

```
public class Variables_Volatile {
    // La variable 'flag' es declarada como volatile.
    volatile boolean flag = true;

    public void stopRunning() {
        flag = false;
// Un hilo puede llamar a este método para cambiar el valor de 'flag' a 'false'.
    }

    public void run() {
        while (flag) {
            // El cuerpo del hilo se ejecuta mientras 'flag' sea 'true'.
            // ...
        }
    }
}
```

Join()

El método join() en Java es utilizado para que un hilo espere a que otro hilo termine su ejecución. Cuando un hilo A ejecuta B.join(), A se detendrá hasta que el hilo B termine su ejecución. Si B ya ha terminado, A continuará sin detenerse. Este método es útil cuando la lógica de tu programa requiere que ciertas tareas deben completarse antes de que otras puedan comenzar.

```
public class Join_Example {
    public static void main(String[] args) {
        Thread thread1 = new Thread(() -> {
            System.out.println( x:"Thread 1 está corriendo.");
            // Realiza alguna tarea...
        });

        Thread thread2 = new Thread(() -> {
            try {
                thread1.join(); // Espera a que thread1 termine
                System.out.println( x:"Thread 1 ha terminado, ahora Thread 2 está corriendo.");
                // Realiza alguna tarea después de que thread1 ha terminado...
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        thread1.start();
        thread2.start();
    }
}
```

Synchronized()

Synchronized en Java es un mecanismo para la sincronización de hilos, que se utiliza para controlar el acceso a recursos compartidos en un entorno de programación multihilo. Aquí están los puntos clave sobre cómo funciona y su importancia:

Uso con Métodos y Bloques: ***synchronized*** se puede utilizar para sincronizar un método completo o un **bloque** específico de instrucciones.

- **Métodos Sincronizados**: Al declarar un método con la palabra clave ***synchronized***, el hilo que llama al método adquiere el bloqueo del objeto (si es un método de instancia) o de la clase (si es un método estático) antes de ejecutar el método.
- **Bloques Sincronizados**: Dentro de un método, se puede sincronizar un bloque específico de

Bloqueo Intrínseco: ***synchronized*** proporciona un bloqueo intrínseco (también conocido como bloqueo de monitor) que asegura que solo un hilo pueda ejecutar una sección de código crítico a la vez. Esto previene la condición de carrera, donde múltiples hilos acceden y modifican datos compartidos de manera concurrente, lo que puede llevar a estados inconsistentes o impredecibles.

```java
public class Synchronized_Basico {

 private int count = 0;

    // Método sincronizado para incrementar
    public synchronized void increment() {
        count++;
    }

    // Método sincronizado para decrementar
    public synchronized void decrement() {
        count--;
    }

    // Método sincronizado para obtener el valor actual
    public synchronized int getCount() {
        return count;
    }
}
```

ReentrantLock

ReentrantLock es una herramienta de sincronización utilizada en programación Java que permite a los hilos bloquear el acceso a un recurso compartido. Es parte del paquete java.util.concurrent.locks y proporciona capacidades de bloqueo que son similares a las del uso de la palabra clave synchronized, pero con más funcionalidades y flexibilidad.

```
public class ReentrantLock_Example {
    private final ReentrantLock lock = new Reentrant
    private int count = 0;

    public void increment() {
        lock.lock(); // Bloquea para asegurar que so
        try {
            count++;
        } finally {
            lock.unlock(); // Asegura que el bloqueo
        }
    }

    public int getCount() {
        return count;
    }
}
```

Este ejemplo muestra cómo varios hilos pueden interactuar de manera segura con un objeto compartido utilizando métodos sincronizados y cómo la comunicación entre hilos se maneja con wait() y notify().

UD_2_Acividad_2_2_4_Sincronizadores

- ThreadSafeCounter.java
- CounterTest.java

Exchanger

El uso de la clase **Exchanger** en Java, permite intercambiar información entre dos hilos.
Por ejemplo: Las clases **TareaA** y **TareaB** implementan Runnable y utilizan una instancia de **Exchanger** para intercambiar información entre sí.

Actividad: UD_2_Acividad_2_2_4_Sincronizadores
- Exachanger.java

Al ejecutar el código, las clases **TareaA** y **TareaB** intercambiarán sus mensajes utilizando la instancia de **Exchanger**. Es posible que el orden exacto de los mensajes varíe debido a la concurrencia, pero ambas tareas intercambiarán información.

Estructuras de datos concurrentes

La clase *Collections* de Java proporciona métodos estáticos para convertir estructuras de datos en seguras respecto a hilos (*thread-safe*), como *synchronizedList*, *synchronizedMap* o *synchronizedSet* entre otros, pero no son las alternativas más eficientes en todos los casos.

Nombre	Tipo	Descripción
ConcurrentHashMap	Clase	Equivalente a un HashMap sincronizado.
ConcurrentSkipListMap	Clase	Equivalente a un TreeMap sincronizado.
CopyOnWriteArrayList	Clase	Equivalente a un ArrayList sincronizado.
CopyOnWriteArraySet	Clase	Equivalente a un Set sincronizado.

Tabla 2.6. Estructuras de datos sincronizadas (seguras en entornos multihilo)del paquete *java.util.concurrent*

CopyOnWriteArraySet en lugar de ArrayList

Java dispone de interfaces y clases para almacenar información con prácticamente cualquier tipo de estructura de datos. *Interfaces* heredadas de la *interface Collection,* como *List*, *Map*, *Set*, *Qeue* o *Deque,* son la base de una serie de clases que, implementando dichas *interfaces,* proporcionan el soporte necesario para todo tipo de estructuras de datos.

Desde el punto de vista de la concurrencia, estas implementaciones no están diseñadas para soportar que múltiples hilos lean y escriban simultáneamente en ellas, pudiendo provocar errores.

Ejemplo usando la estructura **ArrayList** en forma **concurrente**

```
private static List<String> palabras = new ArrayList<String>();
for (int i=0; i<100; i++) {
        new LectorEscritorNoSeguro().start();
    }
```

UD_2_Acividad_2_2_5_EstructDatosConcurrentes LectorEscritorNoSeguro.java

```
77
Exception in thread "Thread-63" Exception in thread "Thread-86" 71
java.util.ConcurrentModificationException
79
        at java.base/java.util.ArrayList$Itr.checkForComodification(ArrayList.java:1013)
```

Demostramos el uso seguro de Estructuras de Datos en entornos concurrentes

Estructura de Datos Segura para Concurrencia: **CopyOnWriteArrayList** es una implementación de la interfaz **List** diseñada para ser utilizada en entornos ***multi-hilo***. Cada vez que se modifica la lista (por ejemplo, mediante ***add***), se crea una copia fresca de la lista subyacente. Las operaciones de lectura se realizan en la lista actual, por lo que no hay interferencia entre las operaciones de lectura y escritura.

CopyOnWriteArrayList: Evita ***Condiciones de Carrera***: Las operaciones de modificación (como ***add***) son automáticamente atómicas, lo que evita problemas de condiciones de carrera. Evita ***ConcurrentModificationException***: Durante la iteración, nunca se lanzará ***ConcurrentModificationException***, incluso si la lista se modifica simultáneamente.

Desventajas de **CopyOnWriteArrayList**:
Costo de Copia: La principal desventaja es el costo asociado con la copia de la lista subyacente para cada operación de modificación. Esto puede ser ineficiente para listas grandes o para situaciones donde las modificaciones son frecuentes.

UD_2_Acividad_2_2_5_EstructDatosConcurrentes LectorEscritorSeguro.java

Las interfaces *ExecutorService, Callable* y *Future*

Usadas de manera conjunta, estas tres interfaces proporcionan mecanismos para ejecutar el código de manera asíncrona.

ExecutorService proporciona el marco de ejecución de código asíncrono contenido en objetos de las *interfaces Callable* y *Runnable*. Sus principales métodos se muestran en la Tabla 2.7.

Método	Descripción
awaitTermination	Bloquea el servicio cuando se recibe una petición de apagado hasta que todas las tareas que tenía asignadas han terminado o se ha alcanzado el tiempo límite de espera (*timeout*) o ha habido una interrupción.
invokeAll	Permite lanzar una colección de tareas y recoger una lista de objetos *Future*.
invokeAny	Permite lanzar una colección de tareas y recoger el resultado de la que termine satisfactoriamente (si alguna lo hace).
shutdown()	Hace que el servicio deje de aceptar nuevas tareas, espera a que terminen las que están en ejecución y finaliza.
shutdownNow()	Finaliza el servicio sin esperar a que las tareas que tiene en ejecución lo hagan.
submit()	Envía a ejecución una tarea *Runnable* o *Callable*.

Tabla 2.7. Principales métodos de *ExecutorService*

Las interfaces *ExecutorService, Callable* y *Future*

La construcción de instancias de *ExecutorService* se realiza a través de una serie de métodos estáticos de la clase *Executors*. Estos métodos permiten indicar la estrategia de hilos que desea que siga el *ExecutorService*. Algunos de los métodos estáticos de *Executors* para crear objetos de la interfaz *ExecutorService* son:

- *Executors.newCachedThreadPool()* → Crea un *ExecutorService* con un *pool* de hilos con todos los que sean necesarios, reutilizando aquellos que están libres.
- *Executors.newFixedThreadPool(int nThreads)* → Crea un *ExecutorService* con un *pool* con un número determinado de hilos.
- *Executors.newSingleThreadExecutor()* → Crea un *ExecutorService* con un único hilo.

Por su parte, **la interfaz *Callable*** es una interfaz que funciona similar a *Runnable*, pero con la diferencia de que puede devolver un retorno y lanzar una excepción. Es una interfaz funcional, ya que solo tiene el método *call*, por lo que se puede utilizar en expresiones lambda. El código asíncrono de un objeto *Callable* se ejecuta a través del método *submit* de *ExecutorService*.

La interfaz *Future* representa un resultado futuro generado por un proceso asíncrono. De alguna manera, es un sistema que permite dejar en suspenso la obtención de un resultado hasta que este está disponible. El resultado de invocar al método *submit* de un *ExecutorService* es un objeto *Future*.

Método	Descripción
cancel	Intenta cancelar la ejecución de la tarea.
get()	Espera a que termine la tarea y obtiene el resultado. En una de sus formas admite un *timeout* para limitar el tiempo de espera.
isCancelled()	Indica si la tarea se ha cancelado antes de terminar.
isDone()	Indica si la tarea ha terminado.

Tabla 2.8. Principales métodos de *Future*

En resumen, la relación entre estas tres interfaces es la siguiente: *ExecutorService* ejecuta un objeto *Callable* (o *Runnable*) con una estrategia multihilo determinada y deposita en un objeto *Future* el retorno futuro de la tarea.

El siguiente ejemplo utiliza las interfaces *Callable*, *ExecutorService* y *Future*, así como la clase *Executors*. El objetivo del ejemplo es lanzar una tarea que se realiza de manera asíncrona y dejar en espera el tratamiento de la respuesta. El proceso completo consiste en construir un objeto que implemente *Callable* y entregarlo a una instancia de *ExecutorService*. El resultado se almacena en un objeto *Future* mediante el método *get*, pudiendo evaluar si el proceso ha terminado correctamente o ha sido cancelado, mediante los métodos *isDone* o *isCanceled*. Para finalizar, el *ExecutorService* es apagado utilizando el método *shutdown*, ya que de lo contrario quedaría a la espera de nuevas tareas.

UD_2_Acividad_2_2_6_IntroAsincrona

Se repiten muchas cosas

- La interfaz *Runnable*
- La clase *Thread*
- Suspensión de ejecución: el método *sleep*
- Interrupciones
- Compartición de información

Tabla 2.9. Métodos principales de la clase *Thread*

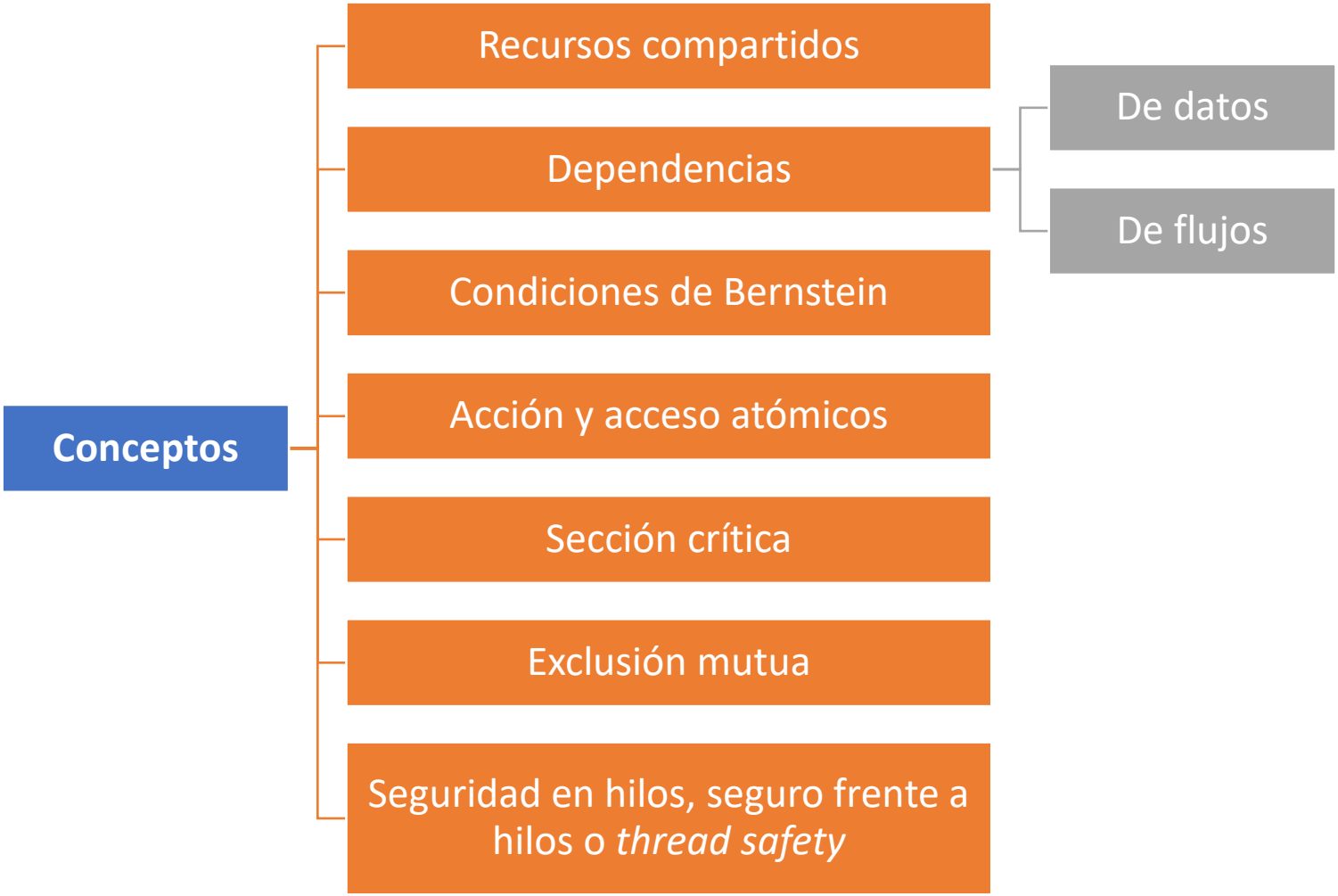
Método	Descripción
start	Método de inicio del bloque asíncrono de la clase.
run	Método en el que se programa el bloque asíncrono. Se ejecuta cuando se invoca el método *start*.
join	Bloquea el hilo hasta que termina el hilo referenciado.
sleep	Método estático que detiene temporalmente la ejecución del hilo.
getId	Devuelve el identificador del hilo. Es un *long* positivo generado cuando se crea el hilo.
getName	Devuelve el nombre del hilo, asignado en algunas de las formas del constructor.
getState	Devuelve el estado del hilo como un valor de la enumeración *Thread.State*.
interrupt	Interrumpe la ejecución del hilo.
interrupted	Método estático que comprueba si el hilo actual ha sido interrumpido.
isInterrupted	Comprueba si el hilo en el que se encuentra ha sido interrumpido.
isAlive	Comprueba si el hilo está vivo.
setPriority	Cambia la prioridad del hilo. El valor debe estar entre las constantes MIN_PRIORITY y MAX_PRIORITY de la propia clase *Thread*. El uso que se haga de las prioridades establecidas viene determinado por el sistema operativo.
setDaemon	Permite marcar el hilo como un hilo demonio.
isDaemon	Determina si un hilo es un hilo demonio.
yield	Método estático que indica al planificador que está dispuesto a ceder su uso actual de procesador. El planificador decide si atiende o no esta sugerencia.

El método estático *sleep* de la clase *Thread* permite suspender la ejecución del hilo desde el que se invoca.

Figura 2.6. El método estático *sleep* permite suspender temporalmente la ejecución de un hilo.

2.4. Problemas y soluciones de la programación concurrente. Sincronización

- Conceptos
- Problemas de la programación concurrente
- Sincronización básica: variables *volatile*
- Sincronización básica: *wait, notify* y *notifyAll*
- Sincronización básica: el método *join*
- Sincronización básica: estructuras de datos resistentes a hilos
- Sincronización avanzada: exclusión mutua, *synchronized* y monitores
- Sincronización avanzada: semáforos


Conceptos
Recursos compartidos
Dependencias
De datos
De flujos
Condiciones de Bernstein
Acción y acceso atómicos
Sección crítica
Exclusión mutua
Seguridad en hilos, seguro frente a hilos o *thread safety*

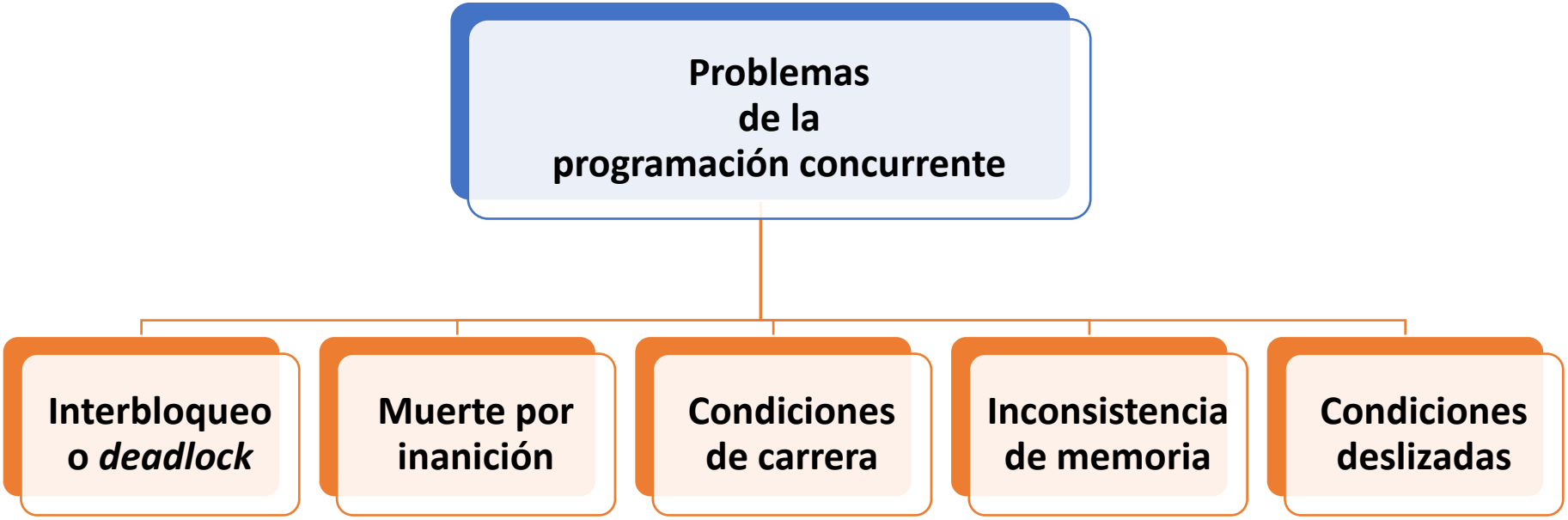

Problemas de la programación concurrente
Interbloqueo o *deadlock*
Muerte por inanición
Condiciones de carrera
Inconsistencia de memoria
Condiciones deslizadas

Tabla 2.10. Métodos principales de la clase *Semaphore*

Método	Descripción
acquire	Adquiere uno o más permisos del semáforo si hay alguno disponible. En caso contrario el hilo queda a la espera.
release	Libera uno o más permisos concedidos previamente.
tryAcquire	Intenta obtener uno o más permisos, pudiendo quedar en espera durante un tiempo limitado.