Programación de Servicios y Procesos

Bloque 2:
PROGRAMACIÓN MULTIHILO
U4-Sincronización

INDEX

1. OBJECTIVES

- Aprender a comunicar y sincronizar subprocesos.
- Comprender los problemas de programación concurrente con la programación multihilo
- Crear clases personalizadas con la habilidad de coordinarse y sincronizarse.
- Conocer los objetos de concurrencia de alto nivel necesarios en aplicaciones concurrentes masivas que explotan masivamente los sistemas multiprocesador y multinúcleo.
- Escribir aplicaciones multihilo.

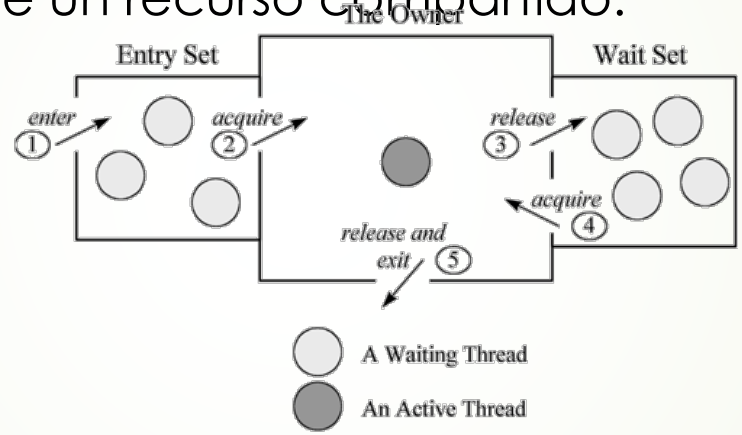
2. SINCRONIZACIÓN

- Un objeto o clase es thread-safe cuando sus datos están protegidos de accesos concurrentes descontrolados por más de un hilo (sobre todo cuando intentan escribir)
- Hacer que un objeto sea thread-safe requiere utilizar **sincronización** para **coordinar** el acceso a sus datos.
- Si no se realiza, puede acabar en corrupción de datos o en otras consecuencias imprevistas
- El mecanismo primario para la sincronización en Java es la palabara reservada **synchronized**, que prove de un **monitor** con **cierre exclusivo**.
  - Un monitor es un conjunto de métodos atómicos que proveen de un mecanismo siempre para assegurar la exclusion mutual para el acceso a un recurso compartido.

2. SINCRONIZACIÓN

¿Qué es un monitor?

- Un **monitor** es un mecanismo que asegura el bloqueo exclusive de un recurso compartido.



- En la **JVM**, cada objeto y clase tiene un monitor asociado.
- Para implementar la exclusion mútua, un lock (muchas veces llamado mutex (binary semaphore) se asocial con cada objeto y clase.
- Los locks (cerrojos) se implementan automáticamente en Segundo plano por la JVM

2. SINCRONIZACIÓN

Ejemplo de escenario de conflicto de recursos

*Imagina una aplicación que lanza **100 hilos**, cada uno de ellos añade **1 euro** a una cuenta compartida. La cuenta está inicialmente vacía.*

- ***¿Cuál será el importe total en la cuenta cuando finalicen los hilos?***

- *Encontrarás el código del ejemplo descrito anteriormente en la documentación de la unidad..*
- *Programa y ejecute el ejemplo tu IDE de Java.*
  - ***¿Obtuviste el resultado esperado?***
  - ***¿Qué ocurre?***

2. SINCRONIZACIÓN
Ejemplo de escenario de conflicto de recursos

- 100 hilos están modificando el saldo de la cuenta al mismo tiempo antes de que un subproceso haya finalizado su actualización.
- Consecuencia: el valor final **es diferente** para cada **ejecución** del programa.
- Esto se conoce como **condición de carrera** y, por lo tanto, la clase no es **thread-safe**.
  - Para **evitar** condiciones de carrera: solo se puede ejecutar un hilo en una **región crítica**: el fragmento de código que modifica el recurso compartido.
  - En el ejemplo, el método *deposit()* es la región crítica

2. SINCRONIZACIÓN

Métodos de instancia sincronizados

- Para sincronizar un método: agrega la palabra clave **synchronized** a la declaración de método.

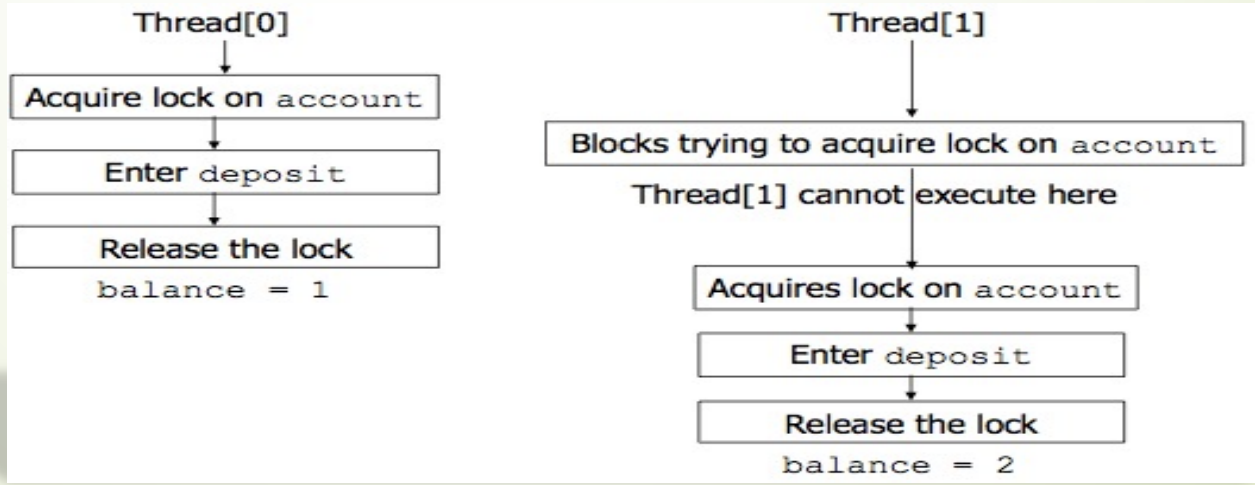
```
public synchronized void deposit(int amount) {
    int newBalance = balance + amount;

    try {
        Thread.sleep(1);     // sleep is deliberately added to make
                             // the data corruption problem more evident
    } catch (InterruptedException ex) {
    }

    balance = newBalance;    // finally, update the balance

}
```

2. SINCRONIZACIÓN

Métodos de instancia sincronizados

Cuando un método de instancia es **Synchronized:**

- Un hilo ha de obtener un **lock** de la instancia del objeto antes de empezar a ejecutarlo.
- Mientras un hilo ejecuta un método synchronized el resto de hilos que esperan a ejecutar el Código **se bloquean** y **esperan** (en una cola FIFO) hasta que el "lock" se libera.



2. SINCRONIZACIÓN
Métodos de instancia sincronizados

Lee con atención los siguientes ejercicios que se encuentran en la documentación y realiza las tareas.

Ejercicio 7: Sincronizar para evitar condiciones de carrera y la corrupción de los datos

Ejercicio 8: ¿Sincronizar campos estáticos y **final**?

2. Sincronización
Bloques sincronizadas

- *Un bloque sincronizado solo bloquea una sección del código*:
- Beneficios:
  - **La concurrencia aumenta** mientras que el tiempo que dura el lock puede disminuir.
  - **El rendimiento aumenta** porque la region crítica es menor.
- *La sincronización de bloques oblgiatoriamente ha de proveer el objeto que necesario para el "lock"*

2. SINCRONIZACIÓN

Bloques sincronizados

```
// deposit using a synchronized block
public void deposit(int amount) {
    synchronized (this) {
        int newBalance = balance + amount;

        try {
            Thread.sleep(1); // sleep is deliberately added to make
                             // the data corruption problem more evident
        }
        catch (InterruptedException ex) {
        }
        balance = newBalance;    // finally, update the balance
    }
}
```

2. SINCRONIZACIÓN
Cooperación entre hilos

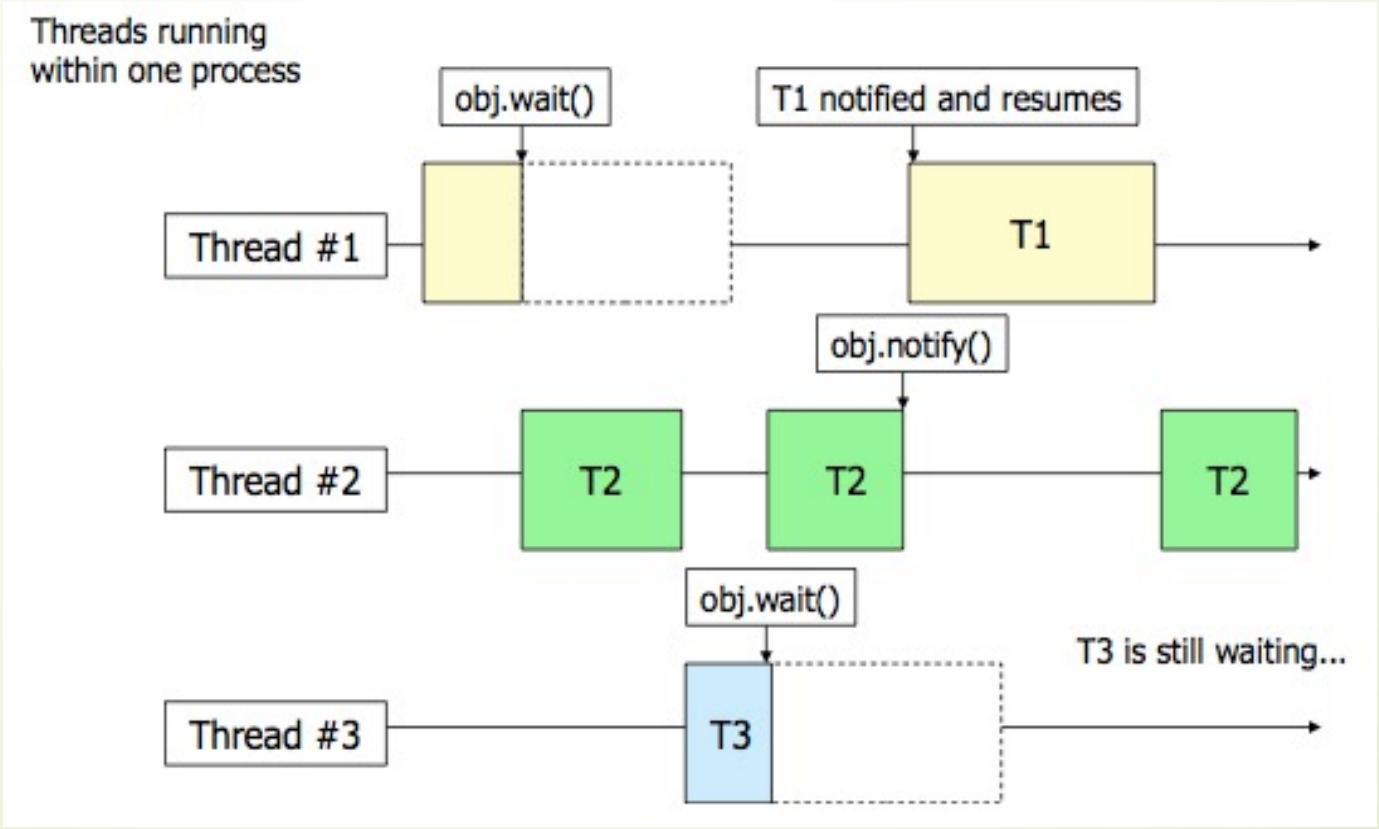
- La sincronización elimina las condiciones de Carrera.
- Problema: la sicronización puede introducir **contención de hilos**:
  - Dos o más hilos compitiendo por acceder al mismo recurso al mismo tiempo pueden causar que la JVM virtual los ejecute más lentamente o incluso que suspenda su ejecución.
- La contención de hilos puede acabar resultando en **inanición:**
  - Un hilo no es capaz obtener regulamente acceso a un recurso compartido y no pueve avanzar en su tarea.
- Para eliminar la inanción de se debe **coordinar** y **cooperar** entre hilos.

2. SINCRONIZACIÓN
Cooperación entre hilos

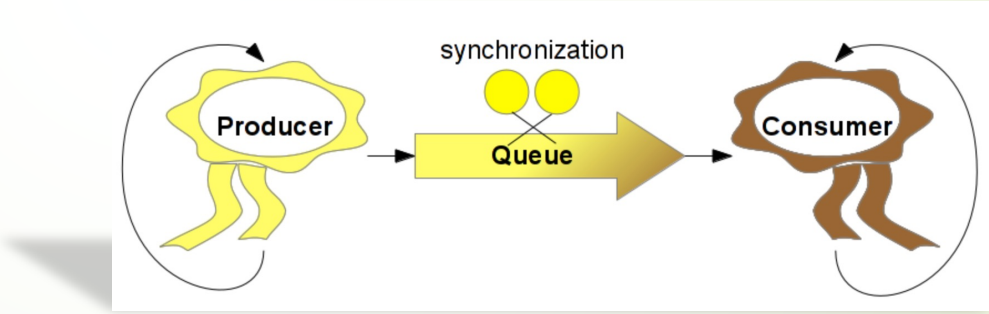
- En Java temenos tres métodos de objeto para implementar la coordinación entre hilos:
  - ***wait***() → hace que se bloquee un subproceso hasta que se llame a notify o notifyAll (o Interruption) en el objeto,
  - ***notify***() → despierta un hilo esperando en el objeto.
  - ***notifyAll***() → activa todos los subprocesos que esperan en el objeto (el planificador decide cuál recibe la notificación del primero).
- Estos 3 métodos deben llamarse siempre dentro de un método o bloque **synchronized**.
- ***wait/notify*** *permite que los hilos se coordinen y compartan datos de forma segura*.

2. SINCRONIZACIÓN
Cooperación entre hilos

- Ejemplo:



PRÁCTICA :
El problema del "Productor-consumidor"



Lee la práctica y realiza las tareas

3. OBJETOS DE ALTO NIVEL PARA LA CONCURRENCIA

- Las APIs de bajo nivel de sincronización son **adecuadas** para tareas **muy básicas.**
- Aplicaciones concurrentes avanzadas como aplicaciones concurrentes masivas, requiren estrucuras de un nivel más alto sobre todo cuando:
  - Los hilos trabajan con muchos recursos.
  - No es simple controlar las tareas al detalle.
- Desde la versión 5 de la plataforma **java.util.concurrent** el paquete contiene objetos de alto nivel para la gestion de la concurrencia.
- Existen además, nuevas estructuras cocnurrentes en **Java Collections** Framework

3. OBJETOS DE ALTO NIVEL PARA LA CONCURRENCIA

- **Executors** → definen una api de alto nivel para lanzar y gestionar threads, especialmente para aplicaciones a gran escala.
- **Colecciones concurrentes** → facilitan la gestion de coleciones grandes y reducen la necesidad de sincronización.
- **Variables atómicas**→ minimizan la necesidad de sincronización y y ayudan a reducir los errores de consistencia en memoria.
- **Numeros aletorios concurrentes → ThreadLocalRandom** (en JDK 7) Provee mecanismos para la generación eficiente de pseudo-números-aleatorios desde multiples hilos.
- **Lock objects** → ayudan a simplificar el Desarrollo de muchas aplicaciones concurrentes.
- **Semaphores** → útil en distintas situaciones cuando se ha de limitar el número de accesos concurrentes a ciertas partes del Código)

3.1 Executors

- Trabajar con la clase Thread puede resultar muy tedioso (como habréis podido comprobar)
- Esto se hace especialmente patente en aplicaciones a gran escala
- La API de Concurrencia introduce el Concepto de un **ExecutorService** que es un reemplazo de más alto nivel para Thread
  - Esta clase nos permite gestionar tareas asíncronas en grupos de hilos de una manera más eficiente.
  - Podemos gestionar grupos de hilos para distintas tareas.
  - Estos grupos de hilos pueden reutilizarse para todas las tareas de la App.

3.1 Executors
Interfaz Executor

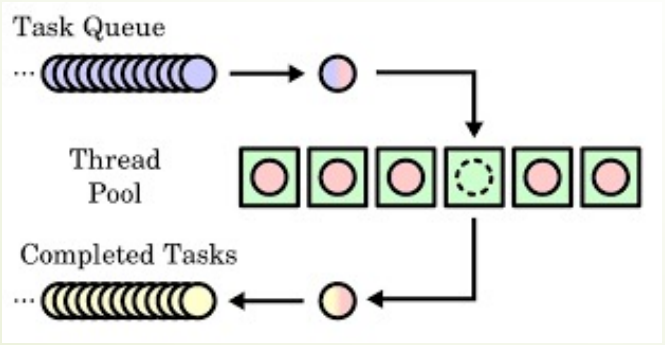
Las tres interfaces para Executor definidas en java.util.concurrent.

1. **Executor** -> Interfaz simple para lanzar nuevas tareas.
2. **ExecutorService** -> Subinterfaz de **Executor** que es capaz de manejar el ciclo de vida de las tareas individuales y del Ejecutor.
3. **ScheduledExecutorService** -> Subinterfaz de ExecutorService que soporta tareas futuras o repetivas.

Normalmente las variables utilizadas para trabajar con estos objetos son del tipo interfaz.

3.1 Executor
Thread Pool – Grupo de Hilos

- La mayora de las implementaciones de Executor (transparentes para nosotros) utilizan grupos de hilos formdos por (worker threads). **Se reduce la sobrecarga**.
- En aplicaciones a gran escala la sobrecarga es un problema, tanto a nivel computacional, como de comprensión del código.
- Fixed Thread Pool (grupo fijo de hilos)



¡Ejercicio 10!

3.1 Executors Thread Pools

Como creamos Pools, pues utilizando los métodos estáticos de Exectors.

- newFixedThreadPool(n): Crea un grupo fijo de hilos.

```
ExecutorService executorService2 = Executors.newFixedThreadPool(10);
```

- newSingleThreadExecutor(): Sólo ejecuta una tarea.

```
ExecutorService executorService1 = Executors.newSingleThreadExecutor();
```

- newScheduledThreadPool(): Crea un grupo de threads que pueden ejecutarse después de un tiempo o periódicamente.

```
ExecutorService executorService3 = Executors.newScheduledThreadPool(10);
```

3.1 Executor
Executor Service - Ejemplo

En este ejemplo, vamos a ver un ejemplo sencillo a su vez muy clarificador acerca del funcionamiento de este modelo:

1. Primero creamos un **ExecutorService** que nos permita ejecutar tareas.
2. Ejecutamos las tareas que necesitamos mediante el método ***execute(*** *Runnable task* ***);*** (Cualquier cosa que implemente Runnable
3. Detenemos el ExecutorService para que la máquina virtual pueda parar: **shutdown()** y **shutdownNow()**

3.1 Executors ExecutorService

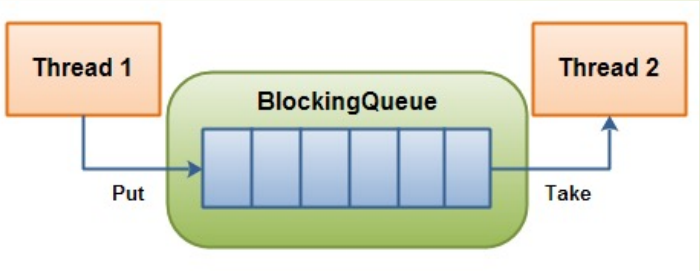
```java
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class U2ExecutorsExample1 {

    public static void main(String[] args) {

        // create an executor service with a fixed pool of threads = availableProcessors
        ExecutorService executorService = Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors());

        // create an anonymous Runnable object that is executed by one of the threads in the pool
        executorService.execute(new Runnable() {
            @Override
            public void run() {
                System.out.println("This is an Asynchronous task");
            }
        });

        // terminate all threads running in the ExecutorService
        // unless, the JVM process keeps running
        executorService.shutdown();
    }

}
```

3.1 Executor
Executor Service

- isTerminated() permite verificar si se han completado todas las tareas.
- awaitTermination(timeout, timeUnit) este método se bloquea hasta que:
  - Todas las tareas han completado su ejecución, o
  - El timeout ocurre, o
  - El hilo actual se interrumpe.

3.2 Colecciones Concurrentes

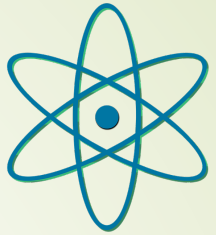
- El paquete ***java.util.concurrent*** incluye colecciones que ayudan a evitar los errores de incosistencia de memoria (se producen cuando diferentes subprocesos tienen vistas incoherentes de lo que deberían ser los mismos datos)
- Interfaces: ***BlockingQueue*** implementaciones:
  - ArrayBlockingQueue
  - DelayQueue
  - LinkedBlockingQueue
  - PriorityBlockingQueue
  - SynchronousQueue



3.2 Colecciones Concurrentes

- Interfaz ConcurrentMap implementaciones:
  - ConcurrentHashMap
- Interfaz ConcurrentNavigableMap implementaciones
  - ConcurrentSkipListMap

3.3 Atomic Variables



- Clases especiales para definir variables de los tipos básicos de datos, de forma que se puede realizar determinadas operaciones sobre ellas de una forma thread-safe.
- No es la panacea pero ayuda a que nuestro código sea threadsafe y más eficiente
  - Ej: AtomicLong, AtomicInt…
- ¿Para que sirve volatile?

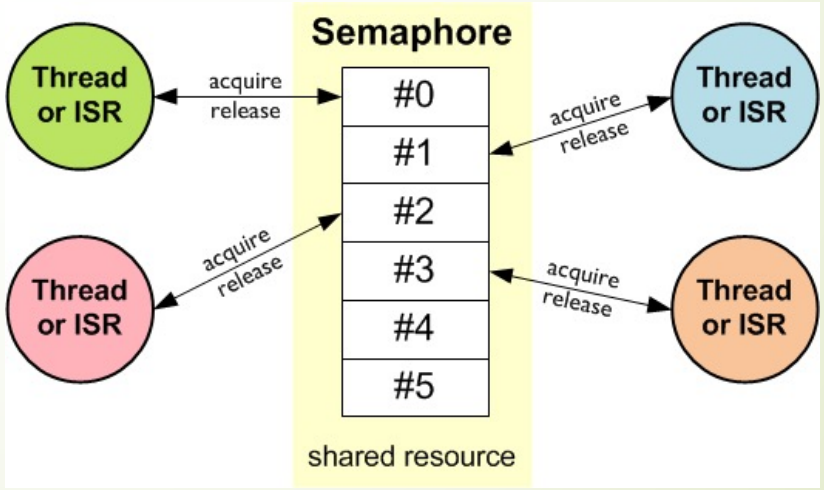
3.4 Concurrent Random Numbers

- En JDK 7 se introduce ThreadLocalRandom que consigue mejores resultados y es más eficiente que Math.random(),
- Ej:
  - int randomNumber = ThreadLocalRandom.current().nextInt(50, 90);

3.5 Semáforos

- Es una estructura de datos que nos habilita para controlar simúltanemente el número de hilos que acceden a una zona concurrente.
- syncronized (this) es como un semáforo (Semafore) inicializado a 1.
- Ej:

```
Semaphore mySemaphore = new Semaphore(1);// only one thread can take permit


//critical section
mySemaphore.acquire();

    ...  // do critical job


mySemaphore.release();
```



3.6 Locks (candados)

- *java.util.concurrent.locks. Lock es una interfaz utilizada como mecanismo de sincronización de hilos similar a bloques sincronizados.*
- *Sin embargo, un Lock es más flexible y sofisticado que un bloque sincronizado;*

3.6 Locks Características

- Los hilos en espera de entrar en un bloque sincronizado no tienen garantías sobre la secuencia seguida para conceder el acceso.
- En un bloque sincronizado no podemos establecer un tiempo de espera al intentar obtener acceso.
- El bloque sincronizado debe estar completamente contenido dentro de un solo método. Un Lock, en cambio puede tener sus llamadas a lock() y unlock() en métodos separados
- **Mátodos**: lock(), lockInterruptibly(), tryLock(), tryLock(long timeout, TimeUnit timeUnit), unlock().

3.7 Lock Ejemplo

Lock lock = new ReentrantLock(); // ReentrantLock implementa la interfaz lock.

lock.lock(); // Bloquea el hilo si no está disponible.

….. //sección crítica.

lock.unlock(); // Solo el thread que posee el lock es el único capaz de llamarlo, si no obtendríamos RuntimeException