

CONCURRENCIA

Se dice que dos tareas se ejecutan concurrentemente cuando se ejecutan "a la vez".



¿Por qué necesitamos la concurrencia?

- Hace posible compartir recursos y subsistema complejos.
- En sistemas monoprocesador permite optimizar el uso de los recursos.
- La concurrencia permite la ejecución simultánea o intercalada de múltiples procesos o tareas en un sistema.
- La concurrencia mejora el rendimiento y la eficiencia de los sistemas informáticos.
- Es esencial para la programación de aplicaciones en tiempo real y para proporcionar una buena experiencia de usuario en aplicaciones interactivas.

Cambios de contexto

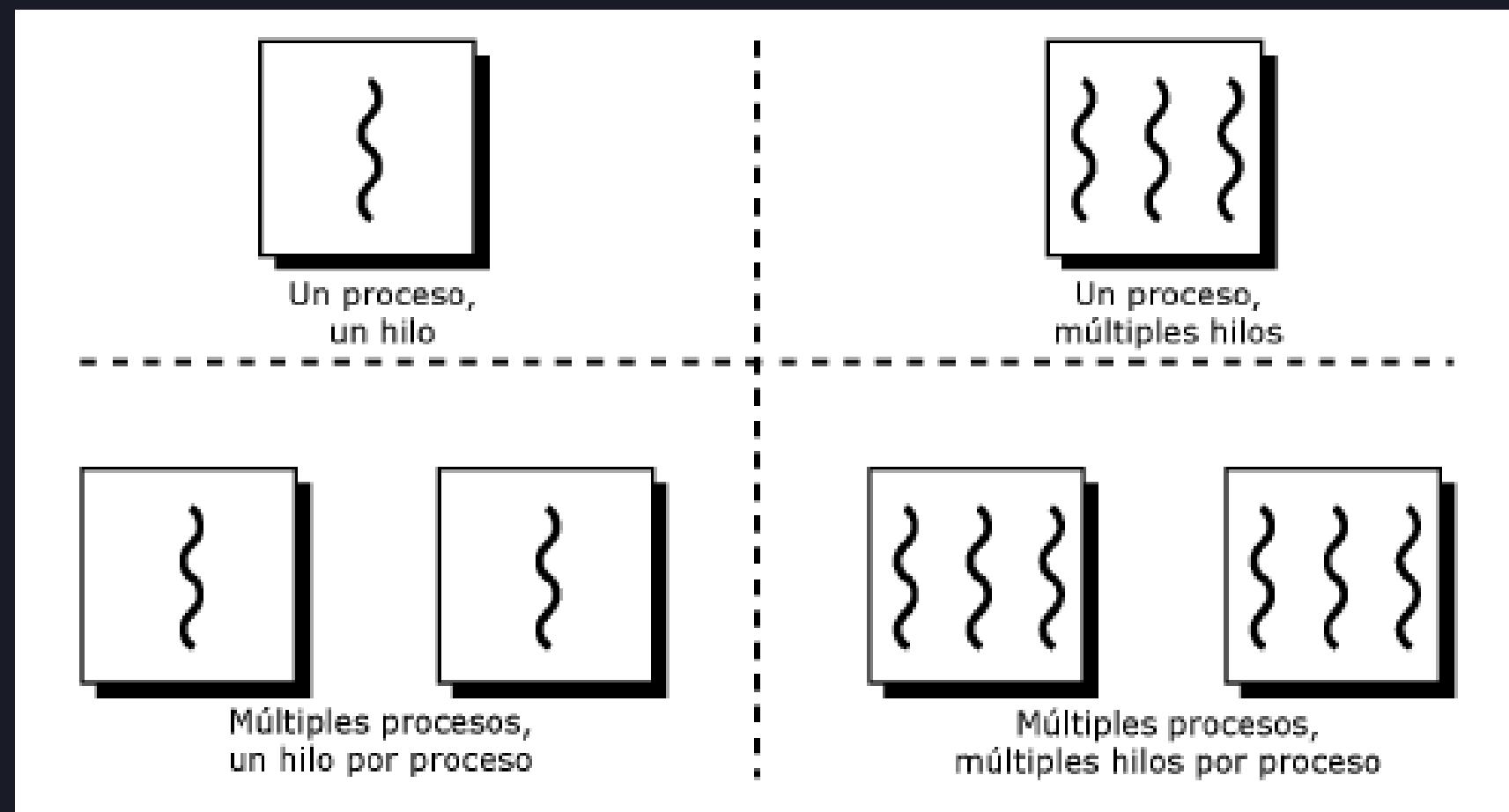
Es cuando el procesador interrumpe la ejecución de un proceso para permitir que otro proceso se ejecute. En otras palabras, el cambio de contexto ocurre cuando el procesador pasa de ejecutar un proceso a ejecutar otro.



- Cada cambio de contexto requiere que el procesador guarde y restaure el estado del proceso interrumpido.
- La eficiencia en la concurrencia depende de la capacidad del sistema para administrar los cambios de contexto de manera eficiente.
- Importante considerar los cambios de contexto al diseñar sistemas concurrentes.

Procesos e hilos

Un proceso es una instancia de un programa en ejecución, mientras que un hilo es una secuencia de instrucciones dentro de un proceso.



Subprogramas

Un subprograma es una sección de código que puede ser ejecutada concurrentemente en un programa principal o en otro subprograma. Un subprograma también se conoce como función, procedimiento o método.

```
public class EjemploSubprograma {  
  
    public static void main(String[] args) {  
        int num1 = 5;  
        int num2 = 3;  
  
        int suma = sumar(num1, num2);  
  
        System.out.println("La suma de " + num1 + " y " + num2 + " es " + suma);  
    }  
  
    public static int sumar(int a, int b) {  
        return a + b;  
    }  
}
```

Thread

Constructores



```
Thread( )
Thread(Runnable target)
Thread(String name)
Thread(Runnable target, String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
```

Thread

Metodos



```
start(): // Inicia la ejecución del hilo.  
  
run(): // Define el punto de entrada para el hilo y se ejecuta cuando se llama al método start().  
  
join(): // Espera a que el hilo en cuestión termine su ejecución antes de que el programa continúe.  
  
interrupt(): // Interrumpe la ejecución del hilo en cuestión.  
  
isAlive(): // Verifica si el hilo está en ejecución o no.  
  
setName(String name): // Establece el nombre del hilo.  
  
getName(): // Obtiene el nombre del hilo.  
  
setPriority(int priority): // Establece la prioridad del hilo.  
  
getPriority(): // Obtiene la prioridad del hilo.  
  
setDaemon(boolean on): // Establece el hilo como un hilo demonio.  
  
isDaemon(): // Verifica si el hilo es un hilo demonio.  
  
currentThread(): // Obtiene una referencia al hilo actual en el que se está ejecutando el código.  
  
sleep(long millis): // Hace que el hilo actual suspenda su ejecución durante un período de tiempo determinado en milisegundos.  
  
yield(): // Hace que el hilo actual ceda su turno al siguiente hilo disponible en el mismo grupo de hilos.
```

Thread.start()

- Es utilizado para iniciar la ejecución de un hilo de forma concurrente.
- Debe ser invocado en un objeto de la clase Thread.
- Al invocar start(), se crea un nuevo hilo de ejecución y se comienza a ejecutar el método run() asociado a dicho hilo.
- Si el objeto Thread no tiene un método run() definido, entonces no ocurrirá nada cuando se invoque start().
- El momento exacto en que se inicie el hilo depende del sistema operativo y del planificador de procesos.
- No garantiza que el hilo se inicie inmediatamente.
- El método run() es el que define el comportamiento que se ejecutará en el hilo, mientras que el método start() es el que inicia la ejecución del hilo en sí.

Thread.join()

En Java, el método join() es un método de la clase Thread que permite que un hilo espere hasta que otro hilo termine su ejecución

```
/*
 * Wait forever for the Thread in question to die.
 *
 * @throws InterruptedException if the Thread is interrupted; it's
 *          <i>interrupted status</i> will be cleared
 */
public final void join() throws InterruptedException
{
    join(0, 0);
}
```

Thread.sleep()

El método Thread.sleep() en Java es utilizado para hacer que un hilo se suspenda temporalmente durante un cierto período de tiempo.

```
public static void sleep(long ms) throws InterruptedException
{
    sleep(ms, 0);
}

public class EjemploSleep {
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            try {
                // Imprime el número del ciclo actual
                System.out.println("Ciclo " + i);

                // Suspende el hilo actual durante 1 segundo
                Thread.sleep(1000);
            } catch (InterruptedException ex) {
                // Maneja la excepción si se produce un error al suspender el hilo
                ex.printStackTrace();
            }
        }
    }
}
```

Ciclo 0
Ciclo 1
Ciclo 2
Ciclo 3
Ciclo 4

synchronized

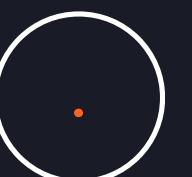
En Java, el método synchronized se utiliza para sincronizar el acceso a bloques de código o métodos entre múltiples hilos. Cuando un método o bloque de código se declara como synchronized, Java asegura que solo un hilo puede acceder a ese método o bloque de código en un momento dado. Esto evita que varios hilos accedan simultáneamente a la misma sección crítica del código y potencialmente modifiquen datos compartidos de forma incorrecta.

```
public synchronized void miMetodo() {  
    // código a sincronizar  
}  
  
synchronized(miObjeto) {  
    // código a sincronizar  
}
```

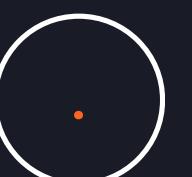


EJEMPLO:

```
public class Main {  
  
    public static void Suma(int a, int b){  
        System.out.println("La suma de los dos numeros es: "+(a+b));  
    }  
  
    Run | Debug  
    public static void main(String[] args) throws InterruptedException {  
  
        Thread hilo = new Thread(()->{  
            Suma(a:2,b:3);  
        });  
  
        hilo.start();  
        hilo.join();  
  
        System.out.println("Fin del programa.");  
    }  
}
```



El hilo que llama al método `join()` se bloqueará hasta que el hilo en el que se llama termine su ejecución.



El método `join()` tiene varias formas de uso, pero la más común es la que espera a que un solo hilo termine su tarea.



El método `join()` es útil en situaciones en las que necesitamos asegurarnos de que un hilo termine su tarea antes de continuar con otra parte del programa.

EJEMPLO:

```
public class Main {  
    static class SumaHilo implements Runnable {  
        private int num1;  
        private int num2;  
  
        public SumaHilo(int num1, int num2) {  
            this.num1 = num1;  
            this.num2 = num2;  
        }  
  
        public void run() {  
            synchronized(Main.class) {  
                System.out.println("La suma es: "+(num1+num2));  
            }  
        }  
    }  
  
    Run | Debug  
    public static void main(String[] args) throws InterruptedException {  
  
        Thread[] sumas = new Thread[5];  
  
        for (int i = 0; i < sumas.length; i++) {  
            sumas[i] = new Thread(new SumaHilo(i+10,i+11));  
        }  
  
        for (int i = 0; i < sumas.length; i++) {  
            sumas[i].start();  
        }  
  
        for (int i = 0; i < sumas.length; i++) {  
            sumas[i].join();  
        }  
  
        System.out.println("Fin del programa.");  
    }  
}
```

EJERCICIO:

Descripción del ejercicio

Se desea crear un programa en Java que simule una carrera de estudiantes. El programa deberá crear un array de threads que contenga los threads correspondientes a cada estudiante y deberá iniciar la carrera. Cada estudiante avanzará una cantidad aleatoria de metros (entre 1 y 6) en cada iteración, y el avance será simulado mediante un tiempo de espera aleatorio (entre 10 y 100 milisegundos) entre iteraciones. El estudiante que llegue primero a la meta será el ganador de la carrera.

EJERCICIO:

Pautas del ejercicio

1. Crear una clase `Estudiante` que implemente la interfaz `Runnable`. Esta clase deberá tener los siguientes atributos:
 - `id`: un identificador numérico único para cada estudiante.
 - `posicion`: la posición actual del estudiante en la carrera (initialmente 0).
 - `distanciaCarrera`: la distancia total de la carrera (constante).
 - `ganador`: una variable compartida por todos los estudiantes que indica quién es el ganador de la carrera (initialmente -1).
2. En el método `run` de la clase `Estudiante`, implementar la lógica de la carrera. En cada iteración, el estudiante deberá avanzar una cantidad aleatoria de metros (entre 1 y 6) y luego esperar un tiempo de espera aleatorio (entre 10 y 100 milisegundos) antes de continuar con la siguiente iteración. La carrera termina cuando un estudiante llega a la posición `distanciaCarrera`. Cuando esto sucede, el estudiante se convierte en el ganador y se actualiza la variable `ganador`.
3. En el método `main` del programa, crear un array de threads `estudiantes` que contenga los threads correspondientes a cada estudiante. Iniciar la carrera llamando al método `start` de cada thread.
4. Esperar a que todos los threads terminen de ejecutarse llamando al método `join` de cada thread.
5. Imprimir un mensaje indicando quién es el ganador de la carrera.
6. Agregar sincronización en el método `avanzar` para evitar problemas de concurrencia.
7. Si se desea, se puede agregar un tiempo de espera antes de iniciar la carrera para dar la oportunidad a todos los estudiantes de llegar a la línea de salida.

Thread.interrupt()

El método `Thread.interrupt()` es un método de la clase `Thread` en Java que se utiliza para interrumpir un hilo de ejecución. Cuando se llama a este método en un hilo, se establece una bandera interna en el hilo que indica que se ha solicitado la interrupción.

Es importante tener en cuenta que la interrupción no detiene directamente el hilo objetivo.

```
/**  
 * Interrupt this Thread. First, there is a security check,  
 * <code>checkAccess</code>. Then, depending on the current state of the  
 * thread, various actions take place:  
 *  
 * <p>If the thread is waiting because of {@link #wait()},  
 * {@link #sleep(long)}, or {@link #join()}, its <i>interrupt status</i>  
 * will be cleared, and an InterruptedException will be thrown. Notice that  
 * this case is only possible if an external thread called interrupt().  
 *  
 * <p>If the thread is blocked in an interruptible I/O operation, in  
 * {@link java.nio.channels.InterruptibleChannel}, the <i>interrupt  
 * status</i> will be set, and ClosedByInterruptException will be thrown.  
 *  
 * <p>If the thread is blocked on a {@link java.nio.channels.Selector}, the  
 * <i>interrupt status</i> will be set, and the selection will return, with  
 * a possible non-zero value, as though by the wakeup() method.  
 *  
 * <p>Otherwise, the interrupt status will be set.  
 *  
 * @throws SecurityException if you cannot modify this Thread  
 */  
public synchronized void interrupt()  
{  
    checkAccess();  
    VMThread t = vmThread;  
    if (t != null)  
        t.interrupt();  
}
```

Thread.interrupt()

```
public class MyThread extends Thread {  
    public void run() {  
        try {  
            while (!Thread.interrupted()) {  
                // Código del hilo aquí  
            }  
        } catch (InterruptedException e) {  
            // Manejar la excepción  
        }  
    }  
}
```

Operaciones Atómicas

Una operación atómica es una operación que se realiza de manera indivisible e irreducible, es decir, que se realiza en su totalidad o no se realiza en absoluto. En otras palabras, una operación atómica es una operación que se ejecuta de manera completa y sin interrupciones, sin que se pueda acceder o modificar su estado intermedio.

```
import java.util.concurrent.atomic.AtomicInteger;

public class Example {
    private static AtomicInteger counter = new AtomicInteger(0);

    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.incrementAndGet();
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 1000; i++) {
                counter.incrementAndGet();
            }
        });

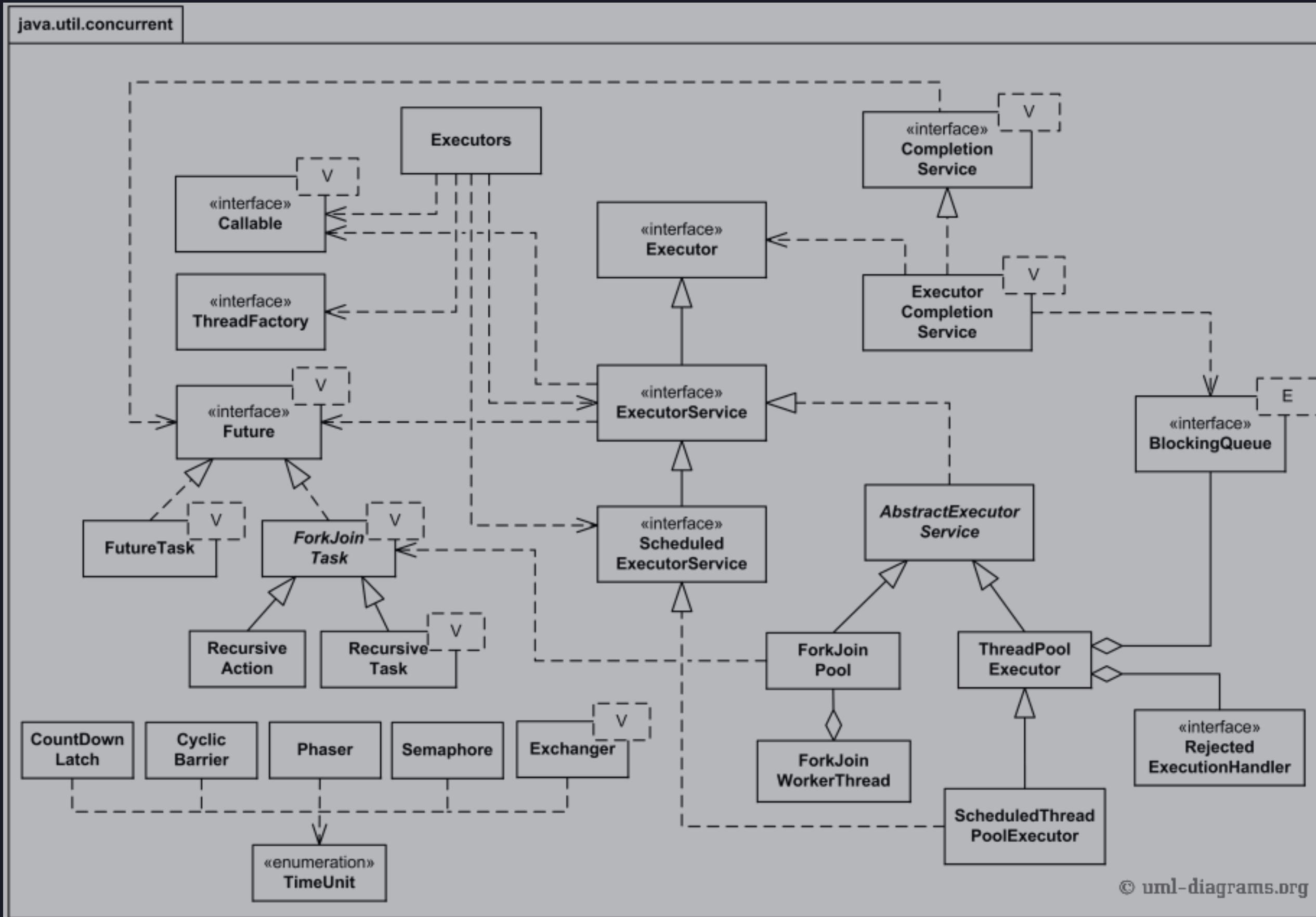
        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted!");
        }

        System.out.println("Counter value: " + counter.get());
    }
}
```

java.util.concurrent

Este paquete es muy útil para la programación concurrente y paralela en Java, y proporciona una variedad de herramientas y mecanismos para gestionar eficazmente los hilos y las tareas en un entorno de programación multihilo.



java.util.concurrent.atomic

La clase `java.util.concurrent.atomic` es un conjunto de clases que proporciona operaciones atómicas para variables en entornos multi-hilo. Estas clases se utilizan para garantizar que las operaciones de lectura y escritura de una variable sean atómicas, lo que significa que una operación de un hilo no será interrumpida por otra operación de otro hilo.

Algunas de las clases más comunes en el paquete `java.util.concurrent.atomic` son:

- **AtomicBoolean**: Proporciona operaciones atómicas para una variable booleana.
- **AtomicInteger**: Proporciona operaciones atómicas para una variable entera.
- **AtomicLong**: Proporciona operaciones atómicas para una variable long.
- **AtomicReference**: Proporciona operaciones atómicas para una variable de referencia.
- **AtomicStampedReference**: Proporciona operaciones atómicas para una variable de referencia junto con un sello de tiempo.

Ejercicio

Descripción del ejercicio

Crear un programa que cree 10 hilos que incrementen una variable compartida de tipo `'AtomicInteger'` en una unidad cada uno 1000 veces. Al finalizar la ejecución del programa, se debe imprimir el valor final de la variable compartida. (10000)

Ejercicio

Requisitos del ejercicio

1. Crear una clase `Incrementor` que implemente la interfaz `Runnable`. teniendo de unico atributo un counter : AtomicInteger que sera constante. En el constructor debera recibir counter : AtomicInteger y asignarle dicho valor al counter del objeto.
2. Implementar el método `run()` de la clase `Incrementor` para incrementar la variable compartida en una unidad en cada iteración.
3. En el método `main()` del programa, inicializar el counter : AtomicInteger a 0, crear 10 hilos utilizando la clase `Incrementor` pasandole el counter y arrancarlos. Esperar a que todos los hilos terminen y luego imprimir el valor final de la variable compartida.

Locking

El locking en concurrencia es un mecanismo utilizado para garantizar la exclusión mutua entre hilos que acceden a un recurso compartido, lo que significa que sólo un hilo a la vez puede tener acceso al recurso en un momento dado.

Java ofrece una variedad de mecanismos de locking para ayudar a controlar el acceso concurrente a los recursos compartidos.

Algunos de estos mecanismos incluyen el uso de la palabra clave `synchronized`, los métodos `wait()` y `notify()` de los objetos, la clase `Lock` del paquete `java.util.concurrent.locks`, entre otros.

Deadlock

Un deadlock, también conocido como interbloqueo, es una situación en la que dos o más hilos o procesos se bloquean mutuamente al esperar indefinidamente por un recurso que el otro tiene y no puede liberar.

Deadlock

```
public class Example {  
    Run | Debug  
    public static void main(String[] args) {  
        Object lock1 = new Object();  
        Object lock2 = new Object();  
  
        Thread thread1 = new Thread(() -> {  
            synchronized (lock1) {  
                System.out.println("Thread 1: locked lock1");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {}  
  
                synchronized (lock2) {  
                    System.out.println("Thread 1: locked lock2");  
                }  
            }  
        });  
  
        Thread thread2 = new Thread(() -> {  
            synchronized (lock2) {  
                System.out.println("Thread 2: locked lock2");  
                try {  
                    Thread.sleep(1000);  
                } catch (InterruptedException e) {}  
  
                synchronized (lock1) {  
                    System.out.println("Thread 2: locked lock1");  
                }  
            }  
        });  
  
        thread1.start();  
        thread2.start();  
    }  
}
```

Ejercicio

Realizar el deadlock anterior pero hacer que
funcione correctamente

Volatile

Si una variable es declarada como "volatile", se garantiza que cualquier cambio realizado por un hilo se verá reflejado inmediatamente en la variable compartida por todos los demás hilos.

El uso de "volatile" es importante en aplicaciones donde múltiples hilos acceden y modifican una variable compartida, ya que ayuda a garantizar que los hilos no estén trabajando con valores obsoletos o inconsistentes.

Colecciones concurrentes

Las colecciones concurrentes en Java son un conjunto de clases que proporcionan estructuras de datos que pueden ser accedidas y modificadas por múltiples hilos de ejecución simultáneamente. Estas clases se encuentran en el paquete `java.util.concurrent` y son altamente eficientes y seguras para su uso en aplicaciones concurrentes.

Algunas de las colecciones concurrentes más comunes son:

1. `ConcurrentHashMap`: una versión segura para subprocessos de la clase `HashMap`, que permite la lectura y escritura simultánea de múltiples hilos.
2. `CopyOnWriteArrayList`: una versión segura para subprocessos de la clase `ArrayList`, que crea una copia de la lista original cada vez que se realiza una operación de escritura.
3. **ConcurrentLinkedQueue**: una cola concurrente que permite la inserción y eliminación de elementos desde múltiples hilos.
4. `LinkedBlockingQueue`: una cola de bloqueo que se utiliza para la comunicación entre subprocessos y que admite la inserción y eliminación de elementos desde múltiples hilos.
5. `ConcurrentSkipListMap`: una versión segura para subprocessos de la clase `TreeMap` que utiliza una estructura de datos de lista enlazada para mantener los elementos ordenados.
6. `ArrayBlockingQueue`: una cola de bloqueo que tiene una capacidad fija y que admite la inserción y eliminación de elementos desde múltiples hilos.
7. `SynchronousQueue`: una cola de intercambio que se utiliza para la comunicación entre subprocessos y que permite la inserción de un elemento solo cuando otro subprocesso está esperando para recibirla.

Estas colecciones concurrentes son muy útiles en aplicaciones que requieren acceso concurrente a estructuras de datos compartidas entre múltiples hilos, y proporcionan un alto rendimiento y seguridad en su uso.

Concurrent Linked Queue

```
import java.util.concurrent.ConcurrentLinkedQueue;

public class ConcurrentLinkedQueueExample {

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

        // Agregar elementos a la cola
        queue.offer("elemento 1");
        queue.offer("elemento 2");
        queue.offer("elemento 3");

        // Mostrar los elementos de la cola
        System.out.println("Elementos en la cola: " + queue);

        // Eliminar un elemento de la cola
        String elementoEliminado = queue.poll();
        System.out.println("Elemento eliminado de la cola: " + elementoEliminado);

        // Mostrar los elementos de la cola actualizados
        System.out.println("Elementos en la cola después de eliminar un elemento:");
    }
}
```

Ejercicio

Crea una ConcurrentLinkedQueue llamada queue que almacene strings

Crea dos hilos uno para el productor y otro para el consumidor

En el primero el productor agrega 10 elementos, cada uno con la siguiente sintaxis:

"Elemento " + numero de elemento

Mostraremos en consola lo siguiente: "Productor agregó: " + elemento

En el segundo hilo el consumidor quita elementos en un while infinito (while(true))

mostrando por cada elemento lo siguiente:

"Consumidor eliminó: " + elemento

Si 'elemento' es nulo, es decir que ya no hay mas elementos por quitar, no mostrará nada y saldrá