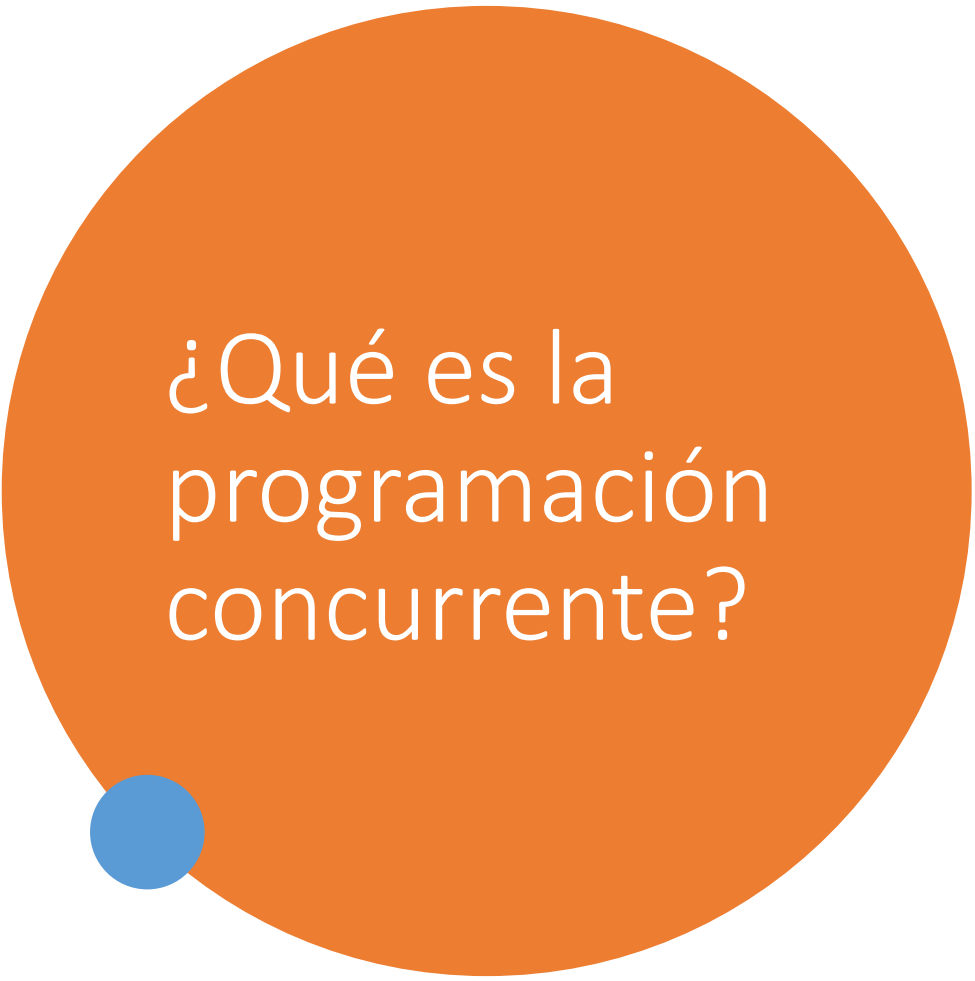




PROGRAMACIÓN CONCURRENTE

Hecho por Alfredo Maldonado,
Juan Antonio, Rubén



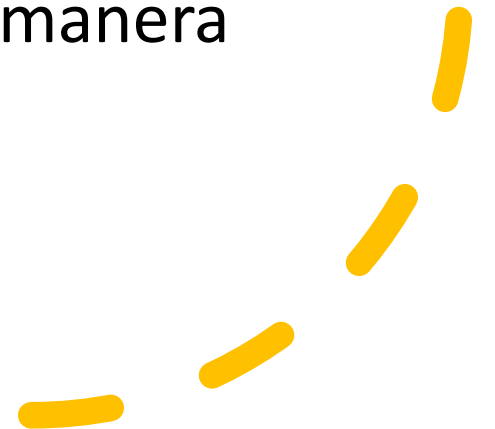


¿Qué es la programación concurrente?

- Capacidad de realizar múltiples tareas al mismo tiempo, aunque no necesariamente en paralelo.
- Ejemplo: Imagina a una persona que trabaja en varias tareas simultáneamente y cambia rápidamente de una a otra.
- Programación: varios flujos de control (como hilos o procesos) colaboran para resolver un problema.

Colas Concurrentes: ¿Qué son y para qué sirven?*

- Estructuras de datos diseñadas para manejar la concurrencia de manera eficiente.
- Java proporciona implementaciones de colas concurrentes como `BlockingQueue`, `ConcurrentLinkedQueue` y `DelayQueue`.
- Estas colas son especialmente útiles cuando varios hilos o procesos necesitan acceder y modificar datos compartidos de manera segura.



Casos de uso

- Productores y consumidores*: Las colas concurrentes son ideales para escenarios donde varios productores generan datos y varios consumidores los procesan. Por ejemplo, en sistemas de mensajería o procesamiento de eventos.
- Tareas programadas*: `DelayQueue` se utiliza para programar tareas o eventos futuros. Puedes agregar elementos con un tiempo de espera y recuperarlos cuando llega el momento.
- *Control de recursos*: Las colas concurrentes ayudan a coordinar el acceso a recursos compartidos, como conexiones de red o bases de datos



Ventajas

- Seguridad: Proporcionan sincronización segura entre hilos o procesos.
- Eficiencia: Minimizan bloqueos y esperas innecesarias.
- Escalabilidad: Funcionan bien en sistemas con alta concurrencia.

Inconvenientes

- ***Complejidad***: Requieren un buen entendimiento de la concurrencia y sincronización.
- ***Posible sobrecarga***: Algunas operaciones pueden ser más costosas que en estructuras no concurrentes.

BlockingQueue

- Es una cola que admite operaciones de espera bloqueante. Es útil cuando necesitas sincronizar hilos de manera eficiente. Puedes usar `put()` para agregar elementos y `take()` para obtenerlos.

```

class Producer implements Runnable {
    private BlockingQueue<String> blockingQueue;

    public Producer(BlockingQueue<String> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }

    @Override
    public void run() {
        try {
            // Produciendo elementos y poniéndolos en la cola
            for (int i = 1; i <= 5; i++) {
                String element = "Elemento " + i;
                System.out.println("Productor produce: " + element);
                blockingQueue.put(element);
                Thread.sleep(1000); // Simula la producción de un elemento cada segundo
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

```

// Clase Consumidor
class Consumer implements Runnable {
    private BlockingQueue<String> blockingQueue;

    public Consumer(BlockingQueue<String> blockingQueue) {
        this.blockingQueue = blockingQueue;
    }

    @Override
    public void run() {
        try {
            // Consumiendo elementos de la cola
            for (int i = 1; i <= 5; i++) {
                String element = blockingQueue.take();
                System.out.println("Consumidor consume: " + element);
                Thread.sleep(2000); // Simula el consumo de un elemento cada dos segundos
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

```

```

import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class BlockingQueueExample {

    public static void main(String[] args) {
        // Crear una cola bloqueante con una capacidad de 3
        BlockingQueue<String> blockingQueue = new ArrayBlockingQueue<>(3);

        // Crear un productor y un consumidor
        Thread producerThread = new Thread(new Producer(blockingQueue));
        Thread consumerThread = new Thread(new Consumer(blockingQueue));

        // Iniciar los hilos
        producerThread.start();
        consumerThread.start();
    }
}

```

```

Productor produce: Elemento 1 y el tamaño de la cola actualmente es: 0
Productor produce: Elemento 2 y el tamaño de la cola actualmente es: 1
Productor produce: Elemento 3 y el tamaño de la cola actualmente es: 2
Productor produce: Elemento 4 y el tamaño de la cola actualmente es: 3
Consumidor consume: Elemento 1 y el tamaño de la cola actualmente es: 2
Productor produce: Elemento 5 y el tamaño de la cola actualmente es: 3
Consumidor consume: Elemento 2 y el tamaño de la cola actualmente es: 2
Consumidor consume: Elemento 3 y el tamaño de la cola actualmente es: 2
Consumidor consume: Elemento 4 y el tamaño de la cola actualmente es: 1
Consumidor consume: Elemento 5 y el tamaño de la cola actualmente es: 0

Process finished with exit code 0

```


ConcurrentLinkedQueue

- Una cola no bloqueante basada en enlaces donde los elementos se agregan al final y se retiran desde el principio, siguiendo el principio FIFO (primero en entrar, primero en salir). Es eficiente para aplicaciones con alta concurrencia.
- Puedes usar `add()` para agregar elementos y `poll()` para obtenerlos.

```
Tamaño de la cola: 3
Primer elemento de la cola: 10
Elementos retirados de la cola:
10
20
30
Tamaño final de la cola: 0

Process finished with exit code 0
```

- Se agregan tres elementos ('10', '20' y '30') a la cola usando el método `offer()`. Con `queue.size()` mostramos el tamaño de la cola.
- Accedemos al primer elemento de la cola sin retirarlo usando `queue.peek()` y se muestra en la consola. Después se retiran y se muestran todos los elementos de la cola usando un bucle `while` y el método `queue.poll()` retira y devuelve el primer elemento de la cola. El bucle se ejecutará mientras la cola no esté vacía ('!queue.isEmpty()').
- Por último, se muestra el tamaño final de la cola después de retirar todos los elementos.

```
java Copy code

import java.util.concurrent.ConcurrentLinkedQueue;

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

        // Agregando elementos a la cola
        queue.offer(10);
        queue.offer(20);
        queue.offer(30);

        // Mostrando el tamaño de la cola
        System.out.println("Tamaño de la cola: " + queue.size());

        // Accediendo y mostrando el primer elemento de la cola sin eliminarlo
        System.out.println("Primer elemento de la cola: " + queue.peek());

        // Retirando elementos de la cola y mostrándolos
        System.out.println("Elementos retirados de la cola:");
        while (!queue.isEmpty()) {
            System.out.println(queue.poll());
        }

        // Mostrando el tamaño de la cola después de retirar elementos
        System.out.println("Tamaño final de la cola: " + queue.size());
    }
}
```

DelayQueue

- Una cola de prioridad en la que los elementos se ordenan por tiempo de espera. Útil para programar tareas o eventos futuros.
- Puedes usar `put()` para agregar elementos y `take()` para obtenerlos.

```
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

4 usages
class DelayedElement implements Delayed {
    public String data;

    4 usages
    private long expiryTime;

    2 usages
    public DelayedElement(String data, long delay) {
        this.data = data;
        this.expiryTime = System.currentTimeMillis() + delay;
    }

    @Override
    public long getDelay(TimeUnit unit) {
        return unit.convert( sourceDuration: expiryTime - System.currentTimeMillis(), TimeUnit.MILLISECONDS);
    }

    @Override
    public int compareTo(Delayed other) { return Long.compare(this.expiryTime, ((DelayedElement) other).expiryTime); }
}
```

```
import java.util.concurrent.DelayQueue;

no usages
public class DelayQueueExample {

    no usages
    public static void main(String[] args) throws InterruptedException {
        DelayQueue<DelayedElement> queue = new DelayQueue<>();

        // Agregamos dos elementos con diferentes tiempos de espera
        queue.put(new DelayedElement( data: "Tarea 1", delay: 5000)); // Espera 2 segundos
        queue.put(new DelayedElement( data: "Tarea 2", delay: 2000)); // Espera 5 segundos

        // Retiramos los elementos cuando expira su tiempo
        while (!queue.isEmpty()) {
            System.out.println(queue.take().data);
        }
    }
}
```

Tarea 2

Tarea 1

Process finished with exit code 0