# Práctica UD2

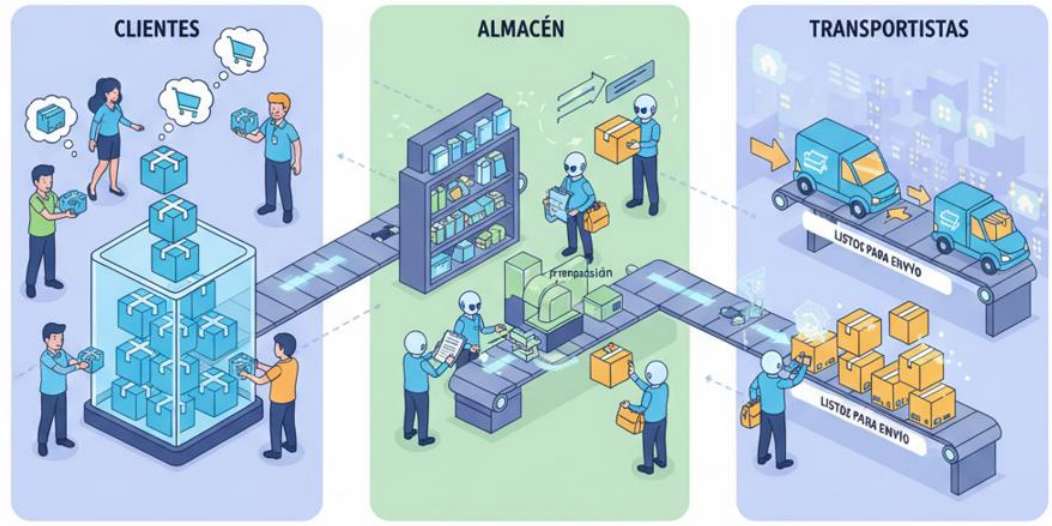
La empresa **Amadzon** desea desarrollar una aplicación que simule el funcionamiento interno de su tienda online.

En este sistema, los pedidos realizados por los clientes pasan por tres fases: **recepción**, **procesamiento** y **envío**.

La tienda recibe múltiples pedidos de forma simultánea y dispone de una **capacidad limitada de almacenamiento**, por lo que es necesario diseñar un sistema concurrente que gestione correctamente la entrada y salida de pedidos, evitando bloqueos y garantizando que no se supere el límite del almacén.

El sistema debe simular una cadena de trabajo concurrente compuesta por tres tipos de hilos:

1. **Clientes:** generan pedidos de manera continua y los depositan en la cola del almacén.
2. **Gestores de almacén:** recogen los pedidos de la cola, los procesan (simulando una preparación) y los marcan como listos para envío.
3. **Transportistas:** recogen los pedidos procesados y simulan su envío al cliente final.




Cada pedido tendrá un **identificador**, un **nombre de producto** y un **estado** (PENDIENTE, PROCESANDO, ENVIADO).

El sistema deberá garantizar que el número de pedidos almacenados no supere un **límite predefinido** (25 pedidos simultáneos).

Existen tres versiones del sistema (V1, V2 y V3) debe implementar los tres tipos de hilos —clientes, gestores y transportistas— utilizando **métodos de creación distintos**, según se detalla en el **Anexo I**.

## Gestión de productos exclusivos

Además del flujo normal de pedidos, la tienda dispone de una gama de **productos exclusivos**:

**Producto A, Producto B, Producto C, Producto D, Producto E y Producto F.**


Cada uno de estos productos tiene **una única unidad disponible**.

Distintos clientes pueden intentar comprarlos de manera simultánea, lo que genera situaciones de **competencia por recursos escasos** y, por tanto, problemas típicos de concurrencia.

El comportamiento funcional del módulo de productos exclusivos debe ser el mismo en todas las versiones; lo que cambia es **la forma en la que se implementa la sincronización y el control de acceso**.

**Reglas de negocio**

**Características funcionales (iguales en V1, V2 y V3):**

1. **Stock:** cada producto exclusivo dispone de **una única unidad**.
2. **Compra doble por cliente:** cada cliente puede comprar **exactamente dos productos exclusivos que se venden de forma conjunta** (paquete de dos).
3. **Regla de dependencia ("reserva encadenada"):**
    - Para completar la compra del **paquete de dos**, el cliente debe **asegurarse** el **primer producto** (A/B/C/D/E/F) y, **acto seguido**, asegurarse el **segundo producto**.
    - Solo si el cliente **consigue ambos**, la compra se **confirma**; si **falla el segundo**, debe **liberar el primero** (no vale quedarse con uno solo).
4. **Tiempos aleatorios:** cada intento de asegurar o confirmar un producto incluirá **retardos aleatorios**, simulando operaciones reales (conexión a servidor, pasarela de pago, etc.).
5. **Contención:** si otro cliente está asegurando o comprando un producto, el hilo actual deberá **esperar** o **desistir** (según la versión).
6. **Notificación:** cuando un producto quede libre (porque no se completó el paquete o se canceló), se deberá **notificar** a los compradores en espera.
7. **Invariante: ningún producto** puede acabar **vendido a dos clientes**.

**En la implementación se deben:**

- **Observar los efectos de la falta de sincronización**, como resultados incoherentes o condiciones de carrera.
- **Analizar interbloqueos y esperas indefinidas**, y aplicar estrategias para evitarlos.
- **Comparar distintos modelos de sincronización** en Java y valorar sus ventajas y limitaciones.

**Salida esperada:**

```
Cliente-1 produce pedido → Pedido{id=1, producto='Ratón'}
Gestor procesa pedido ← Pedido{id=1, producto='Ratón'}
Transportista envía Pedido{id=1, producto='Ratón'}

Cliente-2 intenta asegurar Producto A
Cliente-3 intenta asegurar Producto A
Cliente-2 asegura Producto A
Cliente-2 intenta asegurar Producto B
Cliente-3 espera por Producto A
Cliente-2 confirma paquete [A, B]
Cliente-3 reintenta Producto A → libre
```

**Requisitos de implementación**

## Versión 1: Sincronización clásica synchronized, wait() y notifyAll().

### 1. Clase Pedido

Define una clase con los atributos id, producto y estado. Cada pedido representa una operación de compra o envío dentro del sistema.

### 2. Clase ColaPedidos (monitor compartido)

Esta clase debe funcionar como un **almacén sincronizado** de pedidos, implementado sobre una estructura Queue<Pedido>.

- Los métodos añadir() y retirar() deben declararse como synchronized.
- Si la cola está llena, el hilo productor debe esperar (wait()).
- Si la cola está vacía, el hilo consumidor debe esperar (wait()).
- Cuando se añade o retira un pedido, se deben **despertar los hilos bloqueados** mediante notifyAll().

### 3. Tres tipos de hilos

- **Cliente:** genera pedidos de forma aleatoria y los deposita en la cola. Además, intentará comprar productos exclusivos.
- **GestorAlmacen:** retira pedidos de la cola, los marca como "procesando" y simula la preparación mediante Thread.sleep().
- **Transportista:** recoge los pedidos procesados y los marca como "enviados".

## 4. Gestión de productos exclusivos (sincronización clásica)

Los productos exclusivos (A–F) deben gestionarse utilizando **monitores clásicos** para garantizar que no puedan ser adquiridos simultáneamente por varios clientes.

- Usa la palabra clave **synchronized** para proteger la estructura compartida que almacena el estado de los productos.
- La operación de compra de dos productos exclusivos debe realizarse como una **secuencia sincronizada**, siguiendo estos pasos:
    1. **Asegurar el primero:**
        - Si el producto está ocupado por otro cliente, el hilo debe **esperar** (wait()) hasta que quede libre.
        - Una vez disponible, el cliente lo marca como **asegurado** (en proceso de compra).
    2. **Intentar asegurar el segundo:**
        - Si el segundo producto también está libre, se asegura y se continúa con la compra.
        - Si el segundo producto está ocupado y no se libera en un **tiempo razonable**, el cliente debe **liberar el primero** y **notificar** (notifyAll()) al resto de hilos en espera para evitar bloqueos.
    3. **Confirmación de compra:**
        - Si el cliente consigue ambos productos, realiza la **confirmación atómica** del pedido conjunto (por ejemplo, actualizando su estado a *COMPRADO*).
        - Si ocurre algún fallo durante el proceso, debe **liberar ambos productos** y **notificar** a los demás hilos.
- Añade **retardos aleatorios** de entre **2 y 5 segundos** para simular tiempos reales de procesamiento (pago, validación o envío), de modo que se puedan observar condiciones de carrera y bloqueos.

## Versión 2: Concurrencia java.util.concurrent ReentrantLock, Semaphore y estructuras concurrentes

Esta versión reescribe el sistema utilizando las herramientas del paquete java.util.concurrent, introduciendo mecanismos más robustos y menos propensos a errores.

El objetivo es mostrar cómo las estructuras modernas permiten **sincronización automática y mayor control de recursos**.

**Estructuras y gestión general**

- Sustituye ColaPedidos por una BlockingQueue<Pedido> (por ejemplo, LinkedBlockingQueue).
- Usa un Semaphore para limitar el número de pedidos que pueden estar en procesamiento.
- Emplea un AtomicInteger para llevar la cuenta de pedidos generados y enviados.
- Gestiona los hilos mediante un ExecutorService, usando pools fijos o dinámicos según el tipo de tarea.

**Gestión de productos exclusivos (sincronización avanzada)**

La gestión de los productos exclusivos se implementará mediante **mecanismos explícitos de control de concurrencia**, sustituyendo los monitores clásicos por estructuras más flexibles del paquete java.util.concurrent.

- Cada producto exclusivo debe asociarse a un **ReentrantLock** o un **Semaphore(1)** que controle su acceso.
- Los clientes deberán **adquirir los bloqueos de los dos productos** antes de confirmar la compra, manteniendo el orden global (por ejemplo, A antes que B) para evitar interbloqueos.
    1. **Asegurar el primero:**
        - Llamar a lock() sobre el primer producto.
        - Si está ocupado, el hilo esperará automáticamente hasta que se libere.
    2. **Intentar asegurar el segundo:**

- Llamar a tryLock(timeout, TimeUnit.SECONDS) para el segundo producto.
- Si no se obtiene dentro del tiempo límite, **liberar el primero** (unlock()) y volver a intentarlo más tarde.
- Esto evita esperas indefinidas y garantiza que el sistema no se bloquee.

3. **Confirmar o liberar:**
   - Si ambos bloqueos se adquieren correctamente, se realiza la **compra atómica** del conjunto.
   - Si ocurre un error, ambos bloqueos se liberan.

- Se puede configurar **equidad (fairness)** en los ReentrantLock (new ReentrantLock(true)) para garantizar que los clientes sean atendidos por orden de llegada, reduciendo la **inanición (starvation)**.
- Se recomienda usar **ExecutorService** para ejecutar los clientes y **semáforos**

```
Cliente-1 intenta comprar Producto B
Cliente-2 intenta comprar Producto B
Cliente-1 compra Producto B
Cliente-2 no puede comprar Producto B (agotado)
Cliente-3 intenta comprar Producto C
Cliente-3 compra Producto C
```

## Versión 3: Concurrencia avanzada y modelo Loom. CompletableFuture, tryLock(timeout) y hilos virtuales.

Esta versión se trabaja con **hilos virtuales** y **asincronía declarativa** manteniendo estilo bloqueante pero con **alta escalabilidad**, incluyendo **hilos virtuales (Project Loom)** y las **APIs asíncronas de alto nivel** como CompletableFuture y StructuredTaskScope.

**Estructuras y gestión general**

- **Clientes:** se ejecutarán como hilos virtuales creados con Thread.ofVirtual() o Thread.startVirtualThread(). Esto permite simular miles de clientes concurrentes sin saturar el sistema operativo.
- **Gestores:** implementados con CompletableFuture.runAsync() o supplyAsync(), encadenando operaciones (preparar → procesar → enviar).
- **Transportistas:** gestionados mediante flujos paralelos (Parallel Streams) o un ForkJoinPool.

**Gestión de productos exclusivos (modelo no bloqueante)**

Los productos exclusivos se controlarán mediante **bloqueos no bloqueantes** (tryLock()) y **tareas asincrónicas** (CompletableFuture), integradas con **hilos virtuales**.

1. **Ejecución concurrente:**
    - Cada cliente se ejecuta como **hilo virtual** (Thread.ofVirtual() o Thread.startVirtualThread()), permitiendo simular **miles de compradores concurrentes** sin saturar el sistema operativo.
2. **Control de acceso a productos:**
    - Cada producto exclusivo se asocia a un **ReentrantLock** o estructura similar.
    - Los intentos de compra se realizan con **tryLock(timeout, TimeUnit.SECONDS)**, evitando bloqueos prolongados.
    - Si el cliente no obtiene el bloqueo a tiempo, puede **cancelar la operación** o **volver a intentarlo más tarde**.

3. **Flujo asincrónico de compra:**
    - La secuencia (asegurar → pagar → confirmar) se implementa con **CompletableFuture** para encadenar tareas:

```
CompletableFuture.runAsync(() -> intentarCompra(productoA))
   .thenRun(() -> intentarCompra(productoB))
   .thenRun(this::confirmarCompra)
   .exceptionally(e -> { liberarProductos(); return null; });
```

    - Este enfoque elimina bloqueos directos y facilita la **recuperación automática ante fallos**.
4. **Confirmación y coherencia:**
    - Al completar la compra del primer producto, el hilo puede intentar el segundo.
    - Si cualquiera de los intentos falla, ambos productos deben **liberarse** y **notificar** a otros hilos.
5. **Simulación realista:**
    - Se pueden introducir **retrasos aleatorios** (Thread.sleep() o CompletableFuture.delayedExecutor()) para simular latencias reales de red o pago.
    - Gracias a los **hilos virtuales**, se pueden ejecutar **miles de clientes concurrentes**, cada uno con comportamiento bloqueante lógico pero bajo consumo real de recursos.

```
Cliente-1 (hilo virtual) intenta comprar Producto C
Cliente-2 (hilo virtual) intenta comprar Producto C
Cliente-1 compra Producto C
Cliente-2 timeout: Producto C ocupado
Cliente-3 compra Producto D
```

## Extensiones opcionales

### 1. Monitorización y visibilidad de memoria

Implementar un hilo de supervisión que muestre periódicamente el estado del sistema (pedidos en cola, pedidos procesados y productos exclusivos vendidos). El hilo deberá poder detenerse limpiamente mediante una variable compartida marcada como volatile, para garantizar la visibilidad del cambio entre hilos.

**Conceptos trabajados:** visibilidad de memoria, sincronización indirecta y apagado controlado (*graceful shutdown*).

### 2. Salida sincronizada de transportistas

Simular que los camiones de reparto solo pueden salir del almacén cuando todos han terminado de cargarse.

- En la versión clásica, puede resolverse mediante wait() y notifyAll().
- En la versión moderna, mediante CyclicBarrier o CountDownLatch.

**Conceptos trabajados:** sincronización de fases, coordinación sin exclusión y barreras cíclicas.

### 3. Fairness (evitar inanición)

Asegurar que todos los clientes tienen la misma oportunidad de acceder a los recursos compartidos, incluso en situaciones de alta contención.

- Comparar el comportamiento entre bloqueos justos (ReentrantLock(true)) y bloqueos normales.
- Implementar un pequeño **backoff aleatorio** (retardo) antes de reintentar una operación fallida.

**Conceptos trabajados:** planificación justa (*fairness*), inanición (*starvation*) y backoff exponencial.

### 4. Control de recursos con semáforos

Limitar el número de **gestores de almacén** activos simultáneamente para simular la disponibilidad real del personal o maquinaria.

- Implementar con Semaphore (en la versión moderna) o con un contador sincronizado (en la versión clásica).

**Conceptos trabajados:** control de acceso concurrente, gestión de recursos escasos y sincronización proporcional.

### 5. Hilos virtuales para I/O bloqueante masivo

Simular la llegada de entre **2 000 y 10 000 solicitudes concurrentes** de compra, cada una realizando varias operaciones bloqueantes (consultar stock, procesar pago, registrar pedido).

- Implementar con un ExecutorService de hilos virtuales (Executors.newVirtualThreadPerTaskExecutor()).
- Medir el tiempo de ejecución total y la eficiencia respecto a hilos tradicionales.

**Conceptos trabajados:** escalabilidad masiva, eficiencia en tareas I/O bloqueantes y ventajas del nuevo modelo de concurrencia en Java.

### 6. Scoped Values y Concurrencia Estructurada

**Contexto:**
En esta práctica, los hilos comparten información a través de objetos sincronizados (la ColaPedidos, semáforos o colas bloqueantes).

Sin embargo, en muchos sistemas reales, cada hilo necesita mantener **información contextual propia** —por ejemplo, el identificador del cliente, el número de pedido o la sesión activa— sin compartirla con los demás.

Hasta Java 24, esto se resolvía mediante la clase ThreadLocal, que permite almacenar datos ligados a un hilo concreto.

El problema es que ThreadLocal requiere limpieza manual y puede provocar **fugas de memoria o inconsistencias** si los hilos se reciclan o reaprovechan, como ocurre en los pools de ejecución.

**Novedad en Java 25:**
La nueva API de **Scoped Values**, junto con la **Concurrencia Estructurada**, ofrece una forma moderna y segura de compartir datos entre tareas concurrentes.
Los valores se definen en un ámbito temporal y se propagan de forma controlada a las subtareas que se ejecutan dentro de ese contexto.

Se deberán implementar dos versiones del mismo escenario —una con ThreadLocal (modelo clásico) y otra con ScopedValue (Java 25)— para comparar cómo cambia la gestión del contexto entre hilos. Ambas implementaciones deben ejecutarse dentro del flujo de compra de *Amadzon*, mostrando cómo cada cliente mantiene su propio identificador durante las operaciones concurrentes.

**Parte A – Versión con ThreadLocal (enfoque clásico)**

1. Define un ThreadLocal<String> que almacene el identificador del cliente actual.

   ```
   private static final ThreadLocal<String> contextoCliente = new
   ThreadLocal<>();
   ```
2. Antes de procesar un pedido, asigna el valor correspondiente al hilo:

   ```
   contextoCliente.set("Cliente-5");
   ```

3. Las subtareas del pedido (verificar stock, procesar pago, registrar pedido) acceden a ese valor sin recibirlo por parámetro:

   ```
   System.out.println("Procesando pedido de " + contextoCliente.get());
   ```

4. Al finalizar, limpia el contexto:

   ```
   contextoCliente.remove();
   ```

Este enfoque funciona correctamente, pero si un hilo se reutiliza en un ExecutorService sin limpiar el valor, **otro cliente podría heredar información ajena**, provocando errores sutiles.

**Parte B – Versión con Scoped Value (Java 25)**

1. Declara un valor de ámbito:

   ```
   static final ScopedValue<String> CLIENTE = ScopedValue.newInstance();
   ```
2. Al iniciar una compra, crea un nuevo ámbito:

```
ScopedValue.where(CLIENTE, "Cliente-5").run(() -> {
    verificarStock();
    procesarPago();
    registrarPedido();
});
```

3. Las subtareas pueden acceder al valor con:

```
System.out.println("Procesando pedido de " + CLIENTE.get());
```

El valor solo existe dentro del bloque run(). Al salir del ámbito, se elimina automáticamente, evitando fugas o contaminaciones entre hilos.

4. Si se desea ejecutar tareas en paralelo, se puede combinar con **concurrencia estructurada**:

```
try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
    scope.fork(() -> verificarStock());
    scope.fork(() -> procesarPago());
    scope.fork(() -> registrarPedido());
    scope.join();
}
```

**Ejemplo de salida**

```
Cliente-5 inicia compra estructurada
 → Verificando stock
 → Procesando pago
 → Registrando pedido
Compra completada correctamente
```

# Anexo I. Creación y gestión de hilos en cada versión

### Versión 1: API clásica de hilos

- **Clientes:** extienden la clase Thread.
- **Gestores:** implementan la interfaz Runnable y los ejecutan en instancias de Thread.
- **Transportistas:** crean hilos mediante clases anónimas o expresiones lambda.

### Versión 2: API java.util.concurrent

- **Clientes:** lanzan tareas con un ExecutorService o un ScheduledExecutorService.
- **Gestores:** ejecutan tareas Runnable o Callable en un FixedThreadPool.
- **Transportistas:** usan un CachedThreadPool o un ejecutor de hilos virtuales (Executors.newVirtualThreadPerTaskExecutor()).

### Versión 3: Concurrencia avanzada y modelo Loom

- **Clientes:** se ejecutan como hilos virtuales (Thread.ofVirtual() o Thread.startVirtualThread()).
- **Gestores:** utilizan CompletableFuture.runAsync() o supplyAsync().
- **Transportistas:** trabajan con flujos paralelos (Parallel Streams) o con un ForkJoinPool.

# Tabla resumen: creación y gestión de hilos en Java

| Nº | Creación | API / Clase usada | Gestión de hilos |
|---|---|---|---|
| 1 | extends Thread | Thread | Manual |
| 2 | implements Runnable | Thread(Runnable) | Manual |
| 3 | Lambda / clase anónima | Thread | Manual |
| 4 | Callable + FutureTask | FutureTask, Thread | Manual |
| 5 | Virtual Thread | Thread.ofVirtual() o Thread.startVirtualThread() | Gestionado por la JVM |
| 6 | ExecutorService | Executors.newFixedThreadPool(...) | Pool gestionado |
| 7 | ScheduledExecutorService | Executors.newScheduledThreadPool(...) | Pool planificado |
| 8 | ExecutorService (virtual) | Executors.newVirtualThreadPerTaskExecutor() | Pool virtual gestionado |
| 9 | CompletableFuture | CompletableFuture.runAsync() o supplyAsync() | Automático |
| 10 | ForkJoinPool /Parallel Streams | ForkJoinPool, Stream.parallel() | Automático |

| Versión | Tipo de hilo | Forma de creación | API / Clase usada |
|---|---|---|---|
| V1 | Cliente | extends Thread | java.lang.Thread |
| V1 | Gestor | implements Runnable | java.lang.Thread |
| V1 | Transportista | Clase anónima / lambda | Runnable |
| V2 | Cliente | ExecutorService o ScheduledExecutorService | Executors.newFixedThreadPool() |
| V2 | Gestor | Runnable o Callable | Executors.newFixedThreadPool() |
| V2 | Transportista | Pool dinámico | Executors.newCachedThreadPool() |
| V3 | Cliente | Hilos virtuales | Thread.ofVirtual() / Thread.startVirtualThread() |
| V3 | Gestor | Asincronía encadenada | CompletableFuture.runAsync() o supplyAsync() |
| V3 | Transportista | Ejecución paralela | Parallel Streams o ForkJoinPool |