

PROGRAMACION DE SERVICIOS

TAREA I UDI



ALUMNO CESUR 25/26

Alejandro Muñoz de la Sierra

PROFESOR

Santiago Martin-palomo Garcia

INTRODUCCION

En nuestra incursión inicial en el mundo de la Programación de Servicios y Procesos, nos enfrentamos a un desafío común en el desarrollo de software: la concurrencia. Imaginemos múltiples hilos de ejecución intentando acceder, al mismo tiempo, a un recurso compartido. Este primer caso práctico nos sumergió en esta problemática, particularmente relevante en el contexto de aplicaciones empresariales.

Para ilustrarlo, consideremos el escenario de una empresa que gestiona un listado de precios de un producto, provenientes de diferentes proveedores. Este listado, almacenado en un array, está sujeto a lecturas y modificaciones simultáneas desde distintos puntos de la aplicación. Aquí radica el problema: sin una gestión adecuada, el acceso concurrente puede derivar en inconsistencias en los datos, o incluso en errores derivados de la sobrescritura de valores.

El objetivo central, por lo tanto, fue la implementación de un mecanismo de exclusión mutua. Esta técnica asegura que, en un instante dado, solo un hilo pueda modificar el array, previniendo conflictos y preservando la integridad de los datos.

EXPLORACIÓN DEL PROBLEMA Y CONCEPTOS CLAVE

Inicialmente, carecíamos de experiencia en programación concurrente y gestión de hilos en Java. Por ello, nuestra primera tarea fue sumergirnos en la documentación proporcionada en la Unidad 1, complementándola con investigación en línea.

Nos enfocamos en términos como “exclusión mutua en Java”, “sección crítica” y “problemas de concurrencia”. ¿Qué descubrimos? Que una "sección crítica" es la porción de código donde se accede al recurso compartido (en nuestro caso, el array de precios). La "exclusión mutua", por su parte, garantiza que solo un hilo a la vez pueda ejecutar dicha sección. Java ofrece diversas herramientas para lograr esto, como **synchronized**, **Semaphore** o **Lock**.

Tras evaluar las opciones, nos inclinamos por **synchronized**. Si bien existían alternativas, **synchronized** nos pareció la más directa, segura y comprensible para esta primera aproximación a la programación concurrente.

DISEÑO DE LA SOLUCIÓN: UN ENFOQUE MODULAR

El caso práctico requería la implementación de dos métodos esenciales:

```
public void entrada_seccion_critica() { /* Código */ }  
public void salida_seccion_critica() { /* Código */ }
```

Estos métodos debían controlar el acceso al array, asegurando que cualquier interacción se realizara dentro de una zona protegida por exclusión mutua.

Para facilitar la comprensión y el mantenimiento del código, optamos por una estructura modular, dividiendo la solución en tres clases principales:

PrecioProducto: Representa el precio de un producto y su proveedor.

AlmacenPreciosConSeccionCritica: Gestiona el array de precios y controla el acceso mediante un "**candado**" (**synchronized**), garantizando el acceso exclusivo para escritura.

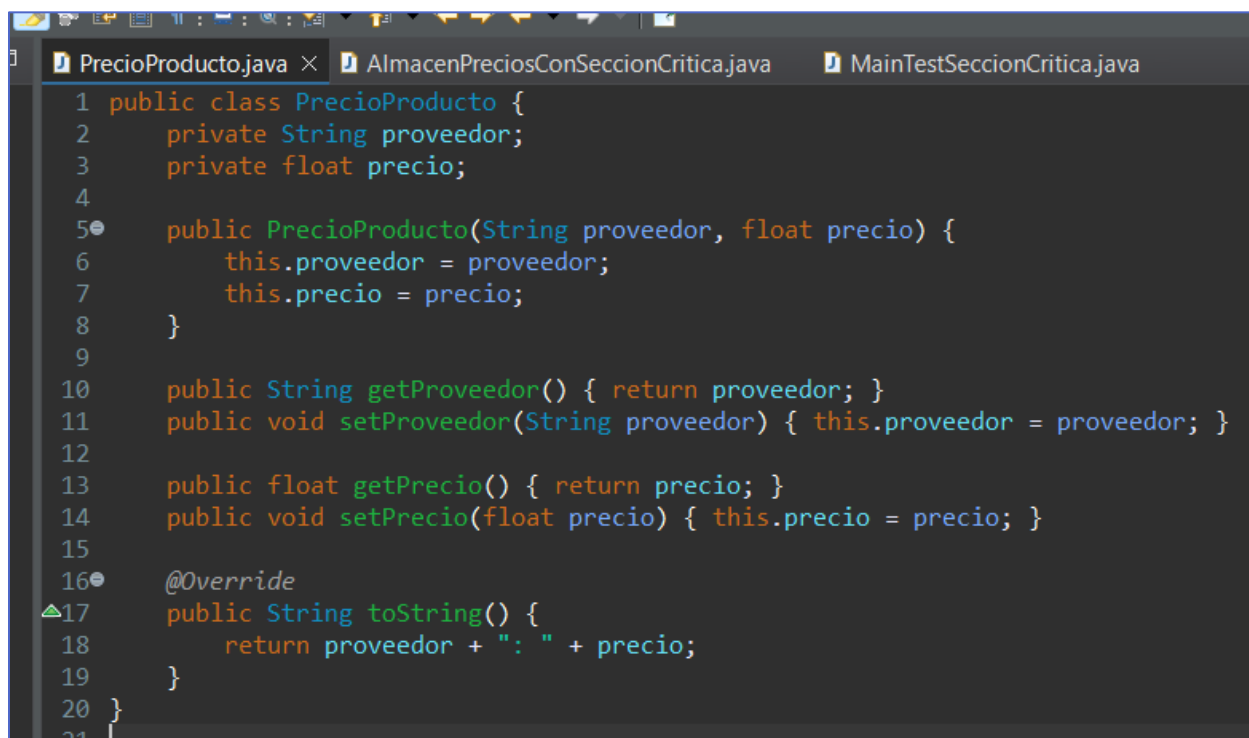
MainTestSeccionCritica: Genera múltiples **hilos** para simular procesos concurrentes intentando acceder y modificar el array.

Esta estructura no solo mantuvo el código ordenado y modular, sino que también facilitó la comprensión de la lógica y la identificación de posibles problemas de concurrencia.

DESARROLLO DE LAS CLASES

3.1. Creación de la clase PrecioProducto

Empezamos definiendo la clase más sencilla del proyecto, una clase que tiene la simple función de representar los datos básicos del problema: el nombre del proveedor y el precio que ofrece.



```
1 public class PrecioProducto {
2     private String proveedor;
3     private float precio;
4
5     public PrecioProducto(String proveedor, float precio) {
6         this.proveedor = proveedor;
7         this.precio = precio;
8     }
9
10    public String getProveedor() { return proveedor; }
11    public void setProveedor(String proveedor) { this.proveedor = proveedor; }
12
13    public float getPrecio() { return precio; }
14    public void setPrecio(float precio) { this.precio = precio; }
15
16    @Override
17    public String toString() {
18        return proveedor + ": " + precio;
19    }
20 }
21 }
```

Esta clase, aunque no participa directamente en el baile de la concurrencia, es fundamental, pues será el tipo de objeto que los hilos intentarán modificar a la vez en el array compartido.

Creo que trabajar con esta clase nos ayudó a comprender mejor la diferencia entre una clase, que es como un modelo general, y un objeto, que es una instancia concreta con valores específicos.

3.2. Creación de la clase AlmacenPreciosConSeccionCritica

Después de la clase **PrecioProducto**, lo que tocaba era darle forma a **AlmacenPreciosConSeccionCritica**. Esta clase, a fin de cuentas, se encarga de guardar los precios y de controlar que los hilos accedan al array de precios de forma organizada, sin que se pisen entre ellos.

Yo diría que esta clase es la pieza central de todo el ejercicio. ¿Por qué? Porque aquí es donde realmente se pone en marcha el sistema para que solo un hilo a la vez pueda tocar el array compartido. Digamos que es como el portero de una discoteca, pero en versión código.

```
PrecioProducto.java  AlmacenPreciosConSeccionCritica.java  MainTestSeccionCritica.java
1 public class AlmacenPreciosConSeccionCritica {
2     private PrecioProducto[] producto;
3
4     // Candado para la sección crítica
5     private final Object Candado = new Object();
6
7     public AlmacenPreciosConSeccionCritica(int tamaño) {
8         this.producto = new PrecioProducto[tamaño];
9     }
10
11     // Entrada a sección crítica
12     public void entrada_seccion_critica(String hilo) {
13         System.out.println(hilo + " está intentando entrar en la sección crítica...");
14     }
15
16     // Salida de sección crítica
17     public void salida_seccion_critica(String hilo) {
18         System.out.println(hilo + " ha salido de la sección crítica");
19     }
20
21     // Método para escribir precio en sección crítica
22     public void escribirPrecio(int indice, PrecioProducto p, String hilo) {
23         entrada_seccion_critica(hilo);
24
25         synchronized (Candado) { // bloque crítico
26             System.out.println(hilo + " está escribiendo en la posición " + indice + ": " + p);
27
28             if (indice < 0 || indice >= producto.length) {
29                 System.out.println(hilo + " intentó un índice fuera de rango");
30                 return;
31             }
32             producto[indice] = p;
33
34             System.out.println(hilo + " terminó de escribir en la posición " + indice);
35         }
36
37         salida_seccion_critica(hilo);
38     }
39
40     // Mostrar estado del array
41     public void mostrarPrecios() {
42         System.out.println("Estado actual del array:");
43         for (int i = 0; i < producto.length; i++) {
44             System.out.println "[" + i + "] " + (producto[i] == null ? "null" : producto[i]);
45         }
46     }
47 }
48
```

Declaración de la clase

```
PrecioProducto.java  AlmacenPreciosConSeccionCritica.java ×  Ma
1 public class AlmacenPreciosConSeccionCritica {
2     private PrecioProducto[] producto;
3
4     // Candado para la sección crítica
5     private final Object Candado = new Object();
6
```

Ahora, si le echamos un vistazo más de cerca, dentro de la clase tenemos dos atributos clave:

`private PrecioProducto[] producto`: Aquí es donde vamos a guardar los precios que nos dan los proveedores. Por ahora, solo declaramos el array, pero no te preocupes, que lo crearemos más adelante, en el constructor.

`private final Object Candado = new Object()`: Este objeto es el que va a actuar como "candado", controlando quién entra y quién sale de la sección crítica. Al ser "final", nos aseguramos de que siempre sea el mismo objeto, para que el `synchronized` funcione correctamente.

Constructor

```
6
7 public AlmacenPreciosConSeccionCritica(int tamaño) {
8     this.producto = new PrecioProducto[tamaño];
9 }
10
```

El constructor recibe el tamaño del array, que viene a ser la cantidad de proveedores que vamos a manejar.

Básicamente, con `new PrecioProducto[tamaño]` se crea un array vacío, y cada hueco empieza estando "vacío" (con `null`). Por ejemplo, si creamos `new AlmacenPreciosConSeccionCritica(5)`, tendremos cinco lugares listos para guardar precios.

Métodos de control visual: entrada seccion critica() y salida seccion critica()

```
11 // Entrada a sección crítica
12● public void entrada_seccion_critica(String hilo) {
13     System.out.println(hilo + " está intentando entrar en la sección crítica...");
14 }
15
16 // Salida de sección crítica
17● public void salida_seccion_critica(String hilo) {
18     System.out.println(hilo + " ha salido de la sección crítica");
19 }
20
```

Estos métodos, la verdad, no bloquean nada. Pero nos ayudan a ver, de manera más clara, cuándo un hilo está intentando entrar o salir de la sección crítica.

En la vida real, en un programa serio, esto se usaría para logs o estadísticas, pero aquí nos sirve para entender qué está pasando en cada momento.

Método principal: escribirPrecio()

```
21 // Método para escribir precio en sección crítica
22● public void escribirPrecio(int indice, PrecioProducto p, String hilo) {
23     entrada_seccion_critica(hilo);
24
25     synchronized (Candado) { // bloque crítico
26         System.out.println(hilo + " está escribiendo en la posición " + indice + ": " + p);
27
28         if (indice < 0 || indice >= producto.length) {
29             System.out.println(hilo + " intentó un índice fuera de rango");
30             return;
31         }
32         producto[indice] = p;
33
34         System.out.println(hilo + " terminó de escribir en la posición " + indice);
35     }
36
37     salida_seccion_critica(hilo);
38 }
39
```

Entrada a la sección crítica: Primero, llamamos a **entrada_seccion_critica()** para avisar de que un hilo quiere entrar.

Bloque **synchronized**: Aquí es donde se pone seria la cosa, con la exclusión mutua. Solo un hilo puede estar dentro de este bloque a la vez, mientras que los demás esperan pacientemente.

Mensajes de escritura: Dentro del bloque, mostramos información de qué hilo está escribiendo y en qué posición.

Comprobación de rango: Nos aseguramos de que la posición en la que queremos escribir sea válida, para no liarla parda con errores.

Asignación al array: El hilo modifica el array, poniendo el precio donde corresponde, sin que otros hilos interfieran.

Salida de la sección crítica: Llamamos a **salida_seccion_critica()**, y en ese momento el "**candado**" se libera automáticamente, dando paso a otro hilo.

Piensa en el Candado como una llave: mientras un hilo la tiene, nadie más puede entrar; y al terminar, la devuelve automáticamente.

Método mostrarPrecios()

```
40 // Mostrar estado del array
41 public void mostrarPrecios() {
42     System.out.println("Estado actual del array:");
43     for (int i = 0; i < producto.length; i++) {
44         System.out.println "[" + i + "] " + (producto[i] == null ? "null" : producto[i]);
45     }
46 }
```

Básicamente, este método itera a través del array producto, mostrando en la consola el contenido de cada índice. Si un precio no ha sido asignado aún a un producto, verás "null"; de lo contrario, se desplegará el proveedor y su precio correspondiente.

Si bien no se emplea synchronized directamente en este método, podríamos, en teoría, protegerlo con el mismo "candado" utilizado en otras partes de la clase. Esto sería crucial si existiera la posibilidad de que otro hilo intentara modificar el array mientras este se está mostrando. Sin embargo, en mi opinión, como este método se llama al finalizar la ejecución del programa (al menos en la forma en que lo he implementado), esta protección adicional no es estrictamente necesaria.

En resumen...

Con la clase **AlmacenPreciosConSeccionCritica**, hemos logrado, creo, cubrir los puntos principales del ejercicio:

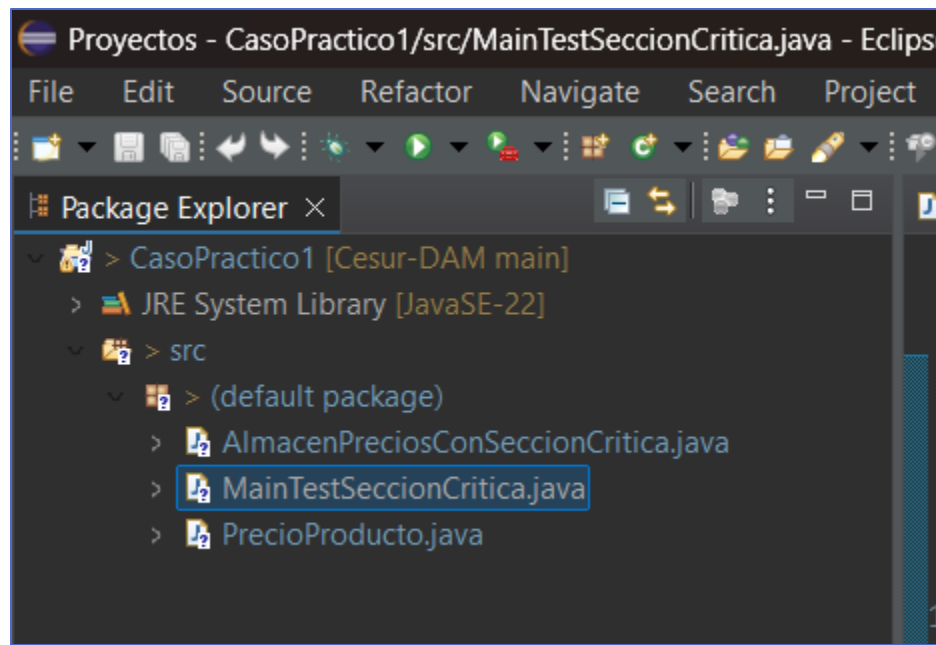
Hemos implementado un mecanismo de exclusión mutua utilizando **synchronized**.

Garantizamos que solo un hilo pueda acceder al array compartido a la vez, lo que evita confusiones y datos incoherentes.

A través de los métodos de "entrada" y "salida" a la sección crítica, podemos observar de forma bastante clara cómo se comportan los hilos en este contexto.

Esta experiencia me ha ayudado a entender, de manera práctica, cómo Java maneja la concurrencia y por qué los bloqueos son tan importantes para mantener la integridad de los datos cuando varios hilos están involucrados.

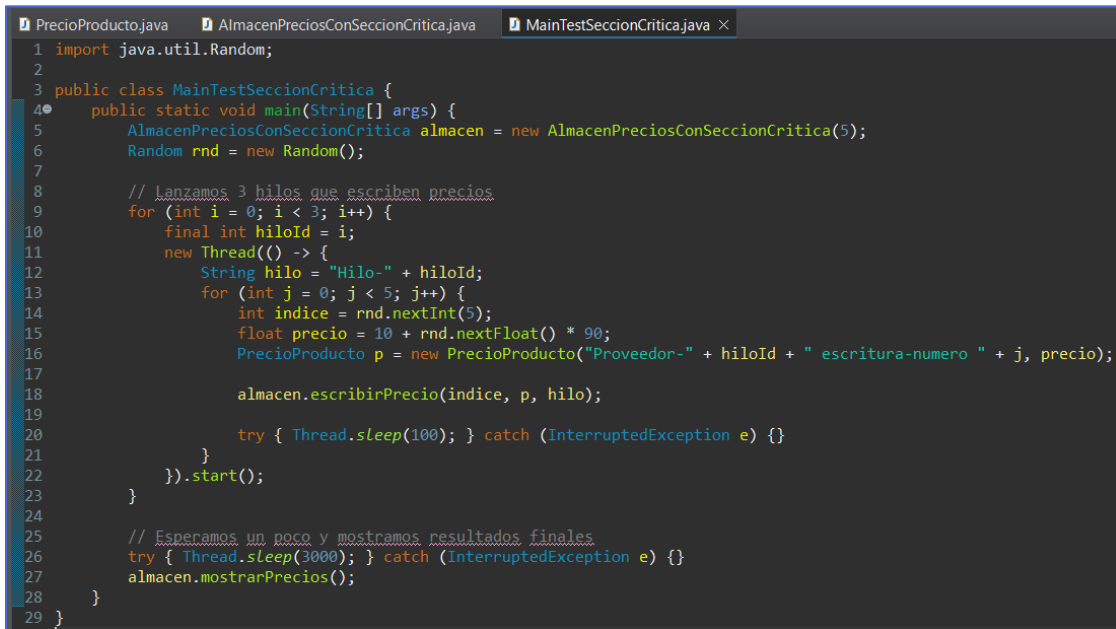
Estructura del entorno de desarrollo y del proyecto para ejecutar tests con la ultima clase:



3.3. Creación de la clase MainTestSeccionCritica

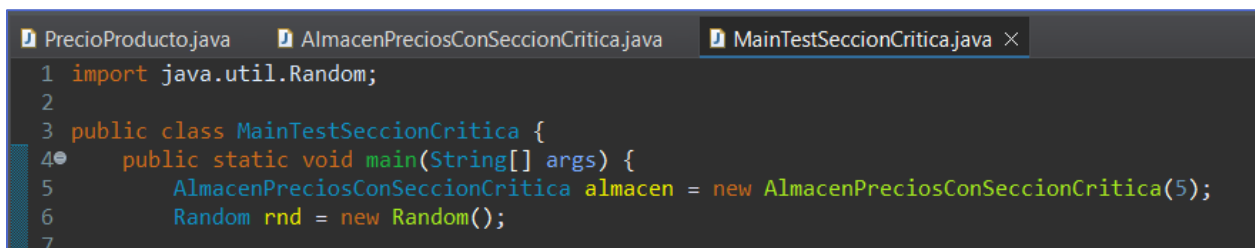
Una vez que teníamos listas las clases **PrecioProducto** y **AlmacenPreciosConSeccionCritica**, necesitábamos una clase que actuara como el motor de arranque de todo el sistema. Así nació **MainTestSeccionCritica**, la cual alberga el famoso método **main()**, que es, como sabemos, el punto de inicio de cualquier programa Java.

El objetivo primordial de esta clase es replicar, lo más fielmente posible, la concurrencia que se da en la vida real cuando varios procesos intentan acceder al mismo recurso, ese espacio compartido que puede generar problemas si no se maneja con cuidado.



```
1 import java.util.Random;
2
3 public class MainTestSeccionCritica {
4     public static void main(String[] args) {
5         AlmacenPreciosConSeccionCritica almacen = new AlmacenPreciosConSeccionCritica(5);
6         Random rnd = new Random();
7
8         // lanzamos 3 hilos que escriben precios
9         for (int i = 0; i < 3; i++) {
10             final int hiloId = i;
11             new Thread(() -> {
12                 String hilo = "Hilo-" + hiloId;
13                 for (int j = 0; j < 5; j++) {
14                     int indice = rnd.nextInt(5);
15                     float precio = 10 + rnd.nextFloat() * 90;
16                     PrecioProducto p = new PrecioProducto("Proveedor-" + hiloId + " escritura-numero " + j, precio);
17
18                     almacen.escribirPrecio(indice, p, hilo);
19
20                     try { Thread.sleep(100); } catch (InterruptedException e) {}
21                 }
22             }).start();
23         }
24
25         // Esperamos un poco y mostramos resultados finales
26         try { Thread.sleep(3000); } catch (InterruptedException e) {}
27         almacen.mostrarPrecios();
28     }
29 }
```

Inicialización: Preparando el Escenario



```
1 import java.util.Random;
2
3 public class MainTestSeccionCritica {
4     public static void main(String[] args) {
5         AlmacenPreciosConSeccionCritica almacen = new AlmacenPreciosConSeccionCritica(5);
6         Random rnd = new Random();
7     }
```

En primer lugar, creamos una instancia de nuestro almacén de precios, con una capacidad para almacenar cinco precios diferentes. Esto significa que internamente, el array que guarda los precios tendrá índices que van desde 0 hasta 4. Además, instanciamos un objeto **Random**. Este objeto es esencial, ya que nos permitirá generar tanto índices aleatorios como precios que también varían aleatoriamente. La idea es simular cómo diferentes proveedores podrían ofrecer valores distintos en un momento dado.

Creación y Ejecución de Hilos

```
8      // lanzamos 3 hilos que escriben precios
9      for (int i = 0; i < 3; i++) {
10         final int hiloId = i;
11         new Thread() -> {
12             String hilo = "Hilo-" + hiloId;
13             for (int j = 0; j < 5; j++) {
14                 int indice = rnd.nextInt(5);
15                 float precio = 10 + rnd.nextFloat() * 90;
16                 PrecioProducto p = new PrecioProducto("Proveedor-" + hiloId + " escritura-numero " + j, precio);
17
18                 almacen.escribirPrecio(indice, p, hilo);
19
20                 try { Thread.sleep(100); } catch (InterruptedException e) {}
21             }
22         }).start();
23     }
```

Un bucle que se repite tres veces: Aquí creamos tres hilos, a los que llamaremos "Hilo-0", "Hilo-1" y "Hilo-2". Cada uno de estos hilos representa un proceso completamente independiente que intentará escribir datos en el mismo almacén de precios.

`final int hiloId = i;` Esta línea es importante. Asegura que cada hilo tenga un identificador que no cambie, un valor inmutable que es necesario para poder usarlo sin problemas dentro de la expresión lambda que definirá el comportamiento del hilo.

`new Thread((() -> { ... })).start();`: Esta sintaxis es una forma concisa de crear y lanzar hilos. Nos permite definir el comportamiento de cada hilo de manera directa, sin tener que crear una clase aparte que implemente la interfaz Runnable.

Nombrando a los hilos: Dentro de la **expresión lambda**, la línea String **hilo** = "Hilo-" + hiloid; asigna un nombre a cada hilo. Esto facilita mucho la tarea de identificar qué hilo está realizando cada acción cuando vemos los mensajes impresos por la consola.

Un **bucle** interno: Cada hilo realizará cinco intentos de escritura aleatoria en el array que representa el almacén. En cada iteración:

1. Generamos un índice aleatorio que estará entre 0 y 4 (correspondiente a las posiciones válidas del array).
2. Creamos un precio que estará entre 10 y 100 (un valor flotante aleatorio dentro de ese rango).
3. Instanciamos un objeto **PrecioProducto**, asignándole un nombre único para cada "proveedor" (que en realidad es el hilo que está intentando escribir el precio).
4. **almacen.escribirPrecio(indice, p, hilo)**: Aquí es donde ocurre la magia de la sincronización. Esta llamada garantiza la exclusión mutua mediante la utilización de `synchronized` sobre el objeto Candado. Esto significa que solo un hilo puede estar modificando el array en un momento dado. Los demás hilos tendrán que esperar su turno.
5. **Thread.sleep(100)**: Esta línea introduce una pequeña pausa de 100 milisegundos. Esto simula que cada hilo está "trabajando" o haciendo algo antes de intentar la siguiente escritura. Esto permite que las ejecuciones de los hilos se entrelacen y podamos observar la concurrencia en acción.

Finalmente, **start()** lanza cada hilo. El sistema operativo se encarga de gestionar estos hilos de forma paralela, asignándoles tiempo de CPU según su propio planificador.

La resolución: Espera y Visualización de Resultados

```
25         // Esperamos un poco y mostramos resultados finales
26         try { Thread.sleep(3000); } catch (InterruptedException e) {}
27         almacen.mostrarPrecios();
28     }
29 }
30
```

Después de lanzar los hilos al ruedo, el hilo principal (el que ejecuta el método **main()**) se toma un respiro y espera durante 3 segundos. Esto le da tiempo a los hilos secundarios para que terminen sus operaciones de escritura. Luego, la llamada a **mostrarPrecios()** imprime en la consola el estado final del array, mostrando qué proveedores (hilos) y qué precios quedaron almacenados en cada posición del almacén.

Es importante recordar que en cada ejecución del programa, los hilos podrían intentar acceder a la sección crítica en un orden diferente. Sin embargo, el "candado" que implementamos con `synchronized` garantiza que nunca habrá dos hilos escribiendo al mismo tiempo, cumpliendo con el principio fundamental de la exclusión mutua. Al menos, eso es lo que esperamos...

En Resumen: MainTestSeccionCritica

En esencia, la clase **MainTestSeccionCritica** sirve como un banco de pruebas para demostrar cómo la sincronización mediante secciones críticas (en este caso, el objeto `Candado` y la palabra clave **synchronized**) puede ayudar a gestionar el acceso concurrente a un recurso compartido y prevenir condiciones de carrera. La verdad, esta práctica fue como vivir un simulacro de la vida real, donde varios programas chocan intentando cambiar la misma cosa al mismo tiempo.

Pero, gracias al **Candado**, ese objeto tan útil, y al bloque `synchronized` dentro de la clase `AlmacenPreciosConSeccionCritica`, las cosas se pusieron en orden. Todo se escribía de forma segura, sin peleas ni errores raros. Es como si tuviéramos un semáforo para los datos.

Y no solo eso, también me ayudó a entender mejor cosas importantes de Java, como:

Hacer hilos y ponerlos a correr.

Compartir datos usando instancias que todos ven.

Y lo más importante, controlar todo para que la información no se vuelva un desastre. Si me preguntas, este tipo de ejercicios son cruciales para entender la programación concurrente.

3.4 Ejecución del código:

```
Problems Javadoc Declaration Console X
<terminated> Practica1ProgramacionMultiHilo [Java Application] C:\Program Files\Java\jdk-23\bin\javaw.exe (20 c
Hilo-1 está intentando entrar en la sección crítica...
Hilo-0 está intentando entrar en la sección crítica...
Hilo-2 está intentando entrar en la sección crítica...
Hilo-1 está escribiendo en la posición 1: Proveedor-1 escritura-numero 0: 55.43013
Hilo-1 terminó de escribir en la posición 1
Hilo-2 está escribiendo en la posición 2: Proveedor-2 escritura-numero 0: 31.260078
Hilo-2 terminó de escribir en la posición 2
Hilo-2 ha salido de la sección crítica
Hilo-1 ha salido de la sección crítica
Hilo-0 está escribiendo en la posición 4: Proveedor-0 escritura-numero 0: 48.90001
Hilo-0 terminó de escribir en la posición 4
Hilo-0 ha salido de la sección crítica
Hilo-0 está intentando entrar en la sección crítica...
Hilo-1 está intentando entrar en la sección crítica...
Hilo-2 está intentando entrar en la sección crítica...
Hilo-0 está escribiendo en la posición 1: Proveedor-0 escritura-numero 1: 53.265545
Hilo-0 terminó de escribir en la posición 1
Hilo-0 ha salido de la sección crítica
```

```
Hilo-1 está escribiendo en la posición 4: Proveedor-1 escritura-numero 4: 61.69733
Hilo-1 terminó de escribir en la posición 4
Hilo-1 ha salido de la sección crítica
Estado actual del array:
[0] Proveedor-1 escritura-numero 3: 90.314964
[1] Proveedor-0 escritura-numero 4: 90.75083
[2] Proveedor-2 escritura-numero 3: 20.726717
[3] Proveedor-2 escritura-numero 4: 76.51081
[4] Proveedor-1 escritura-numero 4: 61.69733
```

En esencia, el programa se asemeja a un pequeño universo de clases, colaborando entre sí para emular la concurrencia de una forma que diría que es bastante segura. Para empezar, la clase **MainTestSeccionCritica** genera una instancia de **AlmacenPreciosConSeccionCritica**. Este último actúa como el depósito compartido donde los precios, aportados por los proveedores, se mantienen guardados.

Después, se ponen en marcha varios hilos (**Thread**). Cada uno de ellos ejecuta el método **escribirPrecio()** sobre esa misma instancia del almacén. Este método es clave, porque se encarga de que el acceso al array sea seguro, implementando un candado. Así, se asegura que solo un hilo pueda modificar el array en un momento dado. Digamos que es una forma de mantener el orden.

Por otro lado, **PrecioProducto** juega el rol de modelo de datos. Cada hilo trabaja con distintas instancias de esta clase, representando la información concreta que se va a grabar en el array.

Así, las tres clases se coordinan de una manera bastante clara:

PrecioProducto: Se ocupa de mantener los datos de cada proveedor junto con su precio.

AlmacenPreciosConSeccionCritica: Controla el acceso concurrente al array y aplica la exclusión mutua.

MainTestSeccionCritica: Crea y administra los hilos, simulando la ejecución paralela y mostrando cómo las operaciones se van intercalando.

Al ejecutar el programa, se hace evidente cómo los hilos intentan acceder a la sección crítica al mismo tiempo, pero, gracias al candado, solo uno lo consigue a la vez. Los demás esperan, por decirlo así, en la cola. Esto previene problemas y mantiene la consistencia de los datos, que es, en mi opinión, lo fundamental.

Curiosamente, el orden en que los hilos se ejecutan puede variar en cada ocasión, dado que lo determina el planificador del sistema operativo. Sin embargo, la regla de oro siempre se cumple: nunca, bajo ninguna circunstancia, dos hilos están escribiendo simultáneamente en la misma posición del array. Esto, desde mi punto de vista, demuestra que la exclusión mutua funciona según lo previsto y que el diseño es, en definitiva, bastante seguro y confiable.

CONCLUSIONES

Esta práctica nos echó una mano para visualizar y comprender claramente qué es una sección crítica y por qué la exclusión mutua es imprescindible en la programación concurrente.

Al principio, los conceptos de hilos, sincronización y bloqueo nos parecían bastante complejos, pero al implementarlos poco a poco con un ejemplo concreto y visual, entendimos cómo funcionan en la práctica.

Un aspecto que me llamó la atención fue observar que el orden de ejecución de los hilos cambia en cada ejecución, lo que demuestra que el sistema operativo decide la planificación de los procesos. Sin embargo, gracias a `synchronized`, el programa se mantiene seguro y consistente, independientemente del orden en que se ejecuten los hilos, generalmente hablando.

También comprendimos que el objeto `Candado` no es más que una referencia que se usa para controlar el acceso a la sección crítica. Java, por suerte, se encarga automáticamente de liberar el bloqueo cuando termina el bloque sincronizado, lo que simplifica mucho la gestión de la concurrencia.

En resumen, esta práctica nos permitió pasar de no tener conocimientos sobre concurrencia a entender y aplicar los principios de exclusión mutua en Java, un concepto fundamental para desarrollar aplicaciones robustas y seguras en entornos donde múltiples procesos comparten datos.

0 5

REFERENCIAS

<https://docs.oracle.com/javase/tutorial/essential/concurrency/>

<https://docs.oracle.com/javase/tutorial/essential/concurrency/>

<https://jenkov.com/tutorials/java-concurrency/synchronized.html>

<https://www.baeldung.com/java-synchronized>

<https://jenkov.com/tutorials/java-concurrency/race-conditions-and-critical-sections.html>

<https://oregoom.com/java/sincronizacion-de-hilos/>

<https://www.makigas.es/series/concurrencia-en-java>