PROGRAMACIÓN DE SERVICIOS Y PROCESOS

CASO PRÁCTICO 1 - JOSE CRUCES APARICIO 2 º DAM

SECCIÓN CRITICA

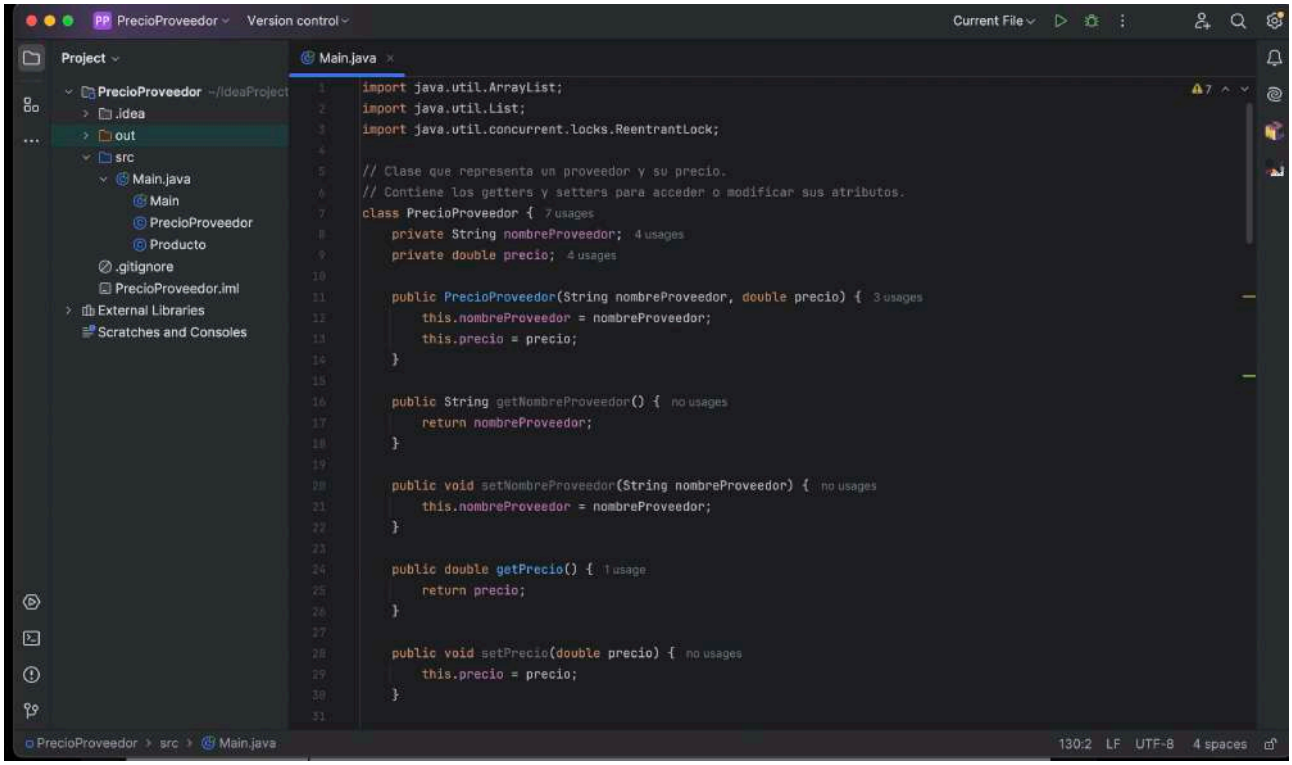
Aquí muestro el codigo del main.java de un proyecto que gestiona una sección crítica en la que a la vez se le manda 3 precios y tiene que gestionar ese momento e incluir el mejor precio de los 3, esta muy comentado para explicar todo lo mas claro posible :

Main.java :

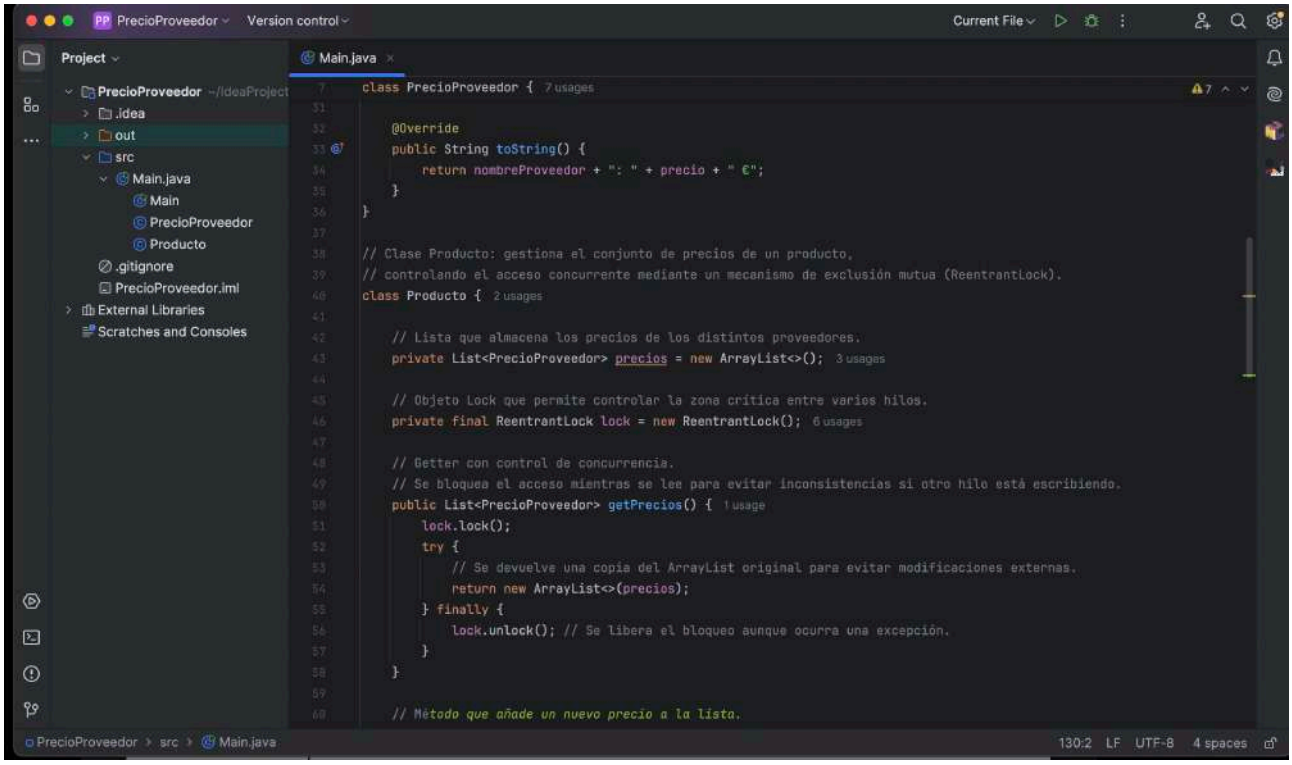
```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.locks.ReentrantLock;

// Clase que representa un proveedor y su precio.
// Contiene los getters y setters para acceder o modificar sus atributos.
class PrecioProveedor {
    private String nombreProveedor;
    private double precio;

    public PrecioProveedor(String nombreProveedor, double precio) {
        this.nombreProveedor = nombreProveedor;
        this.precio = precio;
    }

    public String getNombreProveedor() {
        return nombreProveedor;
    }

    public void setNombreProveedor(String nombreProveedor) {
        this.nombreProveedor = nombreProveedor;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }

    @Override
    public String toString() {
        return nombreProveedor + ": " + precio + " €";
    }
}

// Clase Producto: gestiona el conjunto de precios de un producto,
// controlando el acceso concurrente mediante un mecanismo de exclusión mutua (ReentrantLock).
class Producto {

    // Lista que almacena los precios de los distintos proveedores.
    private List<PrecioProveedor> precios = new ArrayList<>();

    // Objeto Lock que permite controlar la zona crítica entre varios hilos.
    private final ReentrantLock lock = new ReentrantLock();

    // Getter con control de concurrencia.
    // Se bloquea el acceso mientras se lee para evitar inconsistencias si otro hilo está escribiendo.
    public List<PrecioProveedor> getPrecios() {
        lock.lock();
        try {
            // Se devuelve una copia del ArrayList original para evitar modificaciones externas.
            return new ArrayList<>(precios);
        } finally {
            lock.unlock(); // Se libera el bloqueo aunque ocurra una excepción.
        }
    }

    // Método que añade un nuevo precio a la lista.
    // Representa la zona crítica, ya que varios hilos pueden intentar escribir simultáneamente.
    public void agregarPrecio(PrecioProveedor precioProveedor) {
        lock.lock(); //  Inicio de la sección crítica.
```

```
        try {
            System.out.println(Thread.currentThread().getName() + " agregando " + precioProveedor);
            precios.add(precioProveedor);

            // Simulamos el tiempo de escritura para mostrar el efecto del bloqueo.
            Thread.sleep(500);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock(); //  Fin de la sección crítica (libera el lock).
        }
    }

    // Calcula y devuelve el mejor precio (el menor) entre todos los proveedores.
    // También se protege con el bloqueo para asegurar una lectura consistente.
    public double obtenerMejorPrecio() {
        lock.lock();
        try {
            return precios.stream()
                    .mapToDouble(PrecioProveedor::getPrecio)
                    .min()
                    .orElse(Double.NaN);
        } finally {
            lock.unlock();
        }
    }
}

// Clase principal: simula la ejecución concurrente de varios proveedores que actualizan precios.
public class Main {
    public static void main(String[] args) {
        Producto producto = new Producto();

        // Creamos varios hilos que intentarán escribir en la lista de precios simultáneamente.
        Thread proveedor1 = new Thread(() -> {
            producto.agregarPrecio(new PrecioProveedor("Proveedor A", 10.5));
        }, "Hilo-Proveedor-A");

        Thread proveedor2 = new Thread(() -> {
            producto.agregarPrecio(new PrecioProveedor("Proveedor B", 9.8));
        }, "Hilo-Proveedor-B");

        Thread proveedor3 = new Thread(() -> {
            producto.agregarPrecio(new PrecioProveedor("Proveedor C", 11.2));
        }, "Hilo-Proveedor-C");

        // Iniciamos los tres hilos.
        proveedor1.start();
        proveedor2.start();
        proveedor3.start();

        // Con join() esperamos a que todos los hilos terminen antes de continuar.
        try {
            proveedor1.join();
            proveedor2.join();
            proveedor3.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // Mostramos los resultados finales.
        System.out.println("\nLista de precios final:");
        producto.getPrecios().forEach(System.out::println);

        System.out.println("\n Mejor precio: " + producto.obtenerMejorPrecio() + " €");
    }
}
```

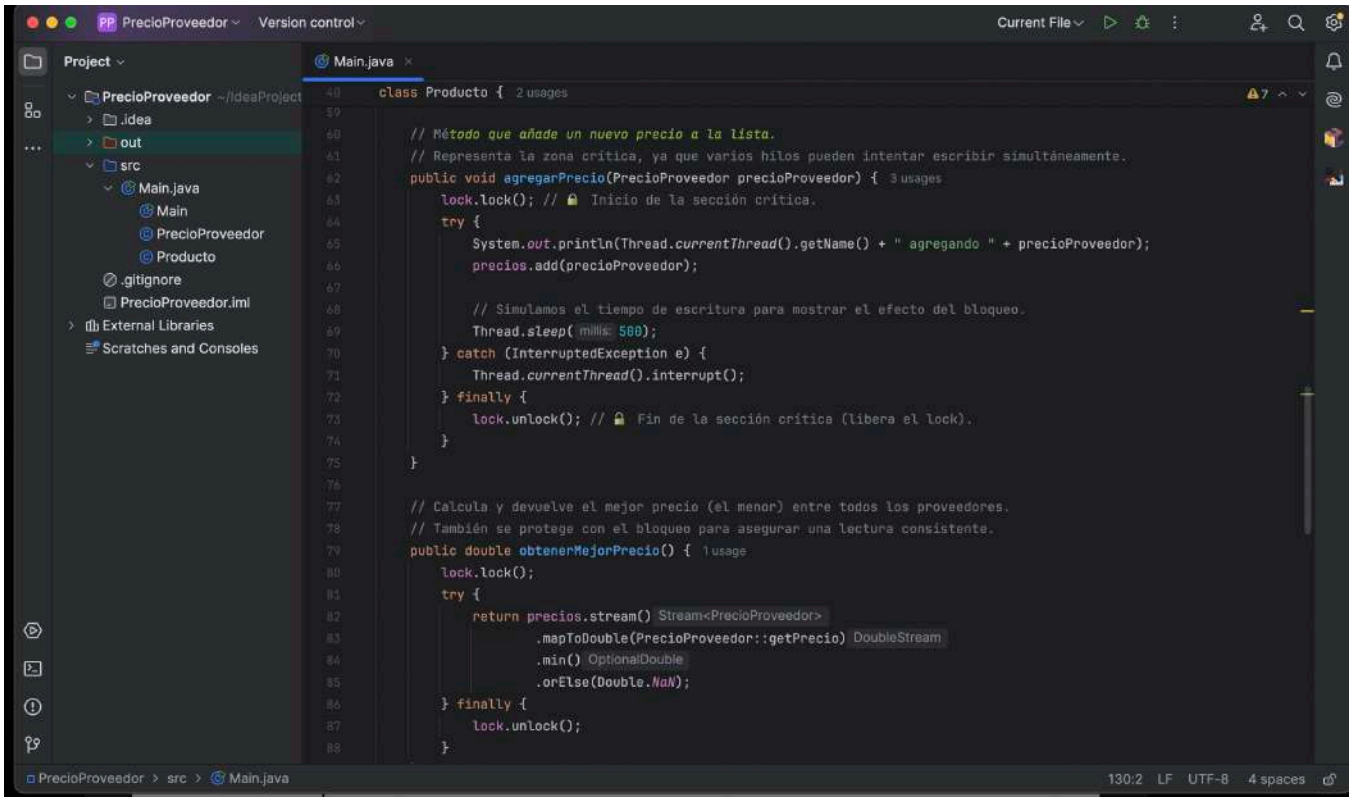
Adjunto capturas de pantalla para mostrar el código y despues una captura de pantalla que como se tienen que mandar comprimidos los pdf no se ve del todo bien por eso esta el codigo entero encima y entrego el archivo java tambien ,pero ademas muestro el resultado del ejericicio viendo la gestión.



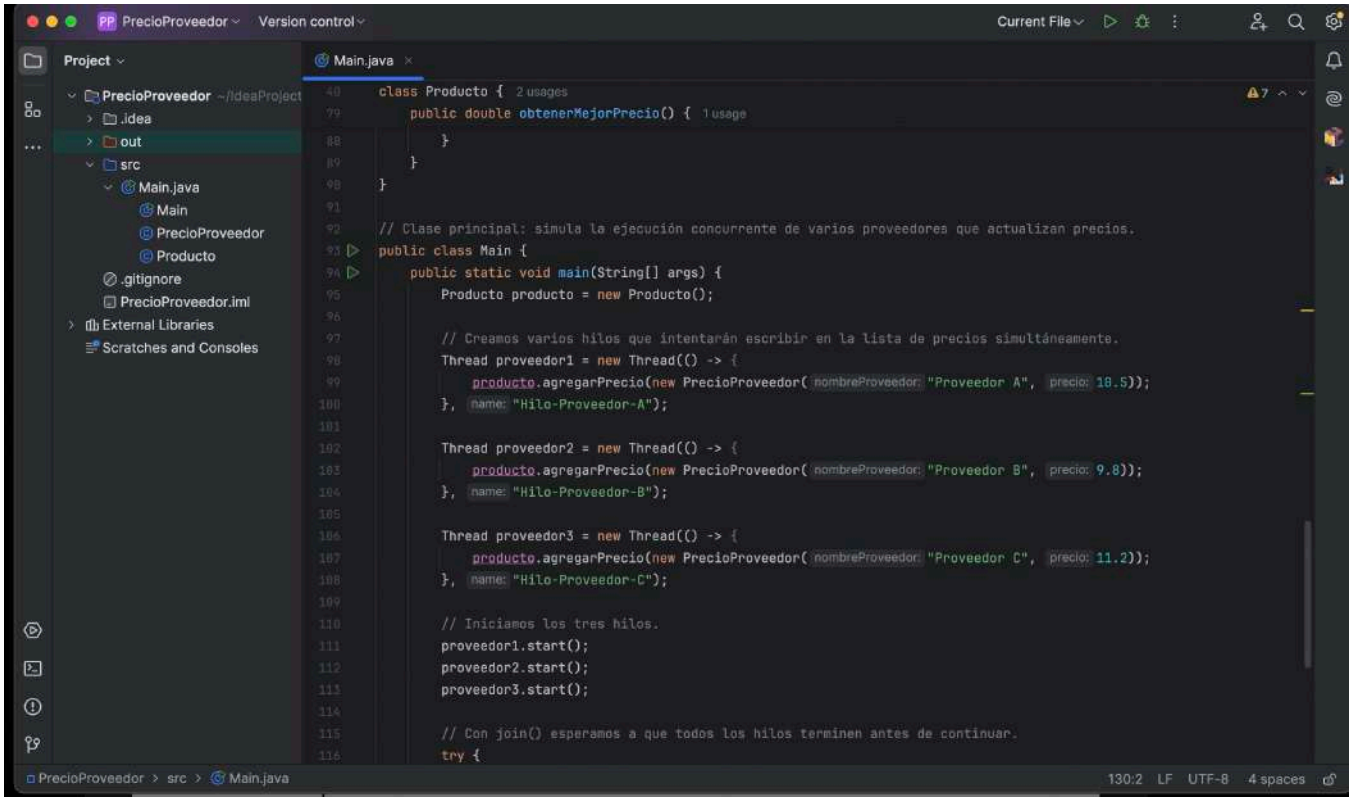
```java
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.locks.ReentrantLock;

// Clase que representa un proveedor y su precio.
// Contiene los getters y setters para acceder o modificar sus atributos.
class PrecioProveedor {
    private String nombreProveedor;
    private double precio;

    public PrecioProveedor(String nombreProveedor, double precio) {
        this.nombreProveedor = nombreProveedor;
        this.precio = precio;
    }

    public String getNombreProveedor() {
        return nombreProveedor;
    }

    public void setNombreProveedor(String nombreProveedor) {
        this.nombreProveedor = nombreProveedor;
    }

    public double getPrecio() {
        return precio;
    }

    public void setPrecio(double precio) {
        this.precio = precio;
    }
```



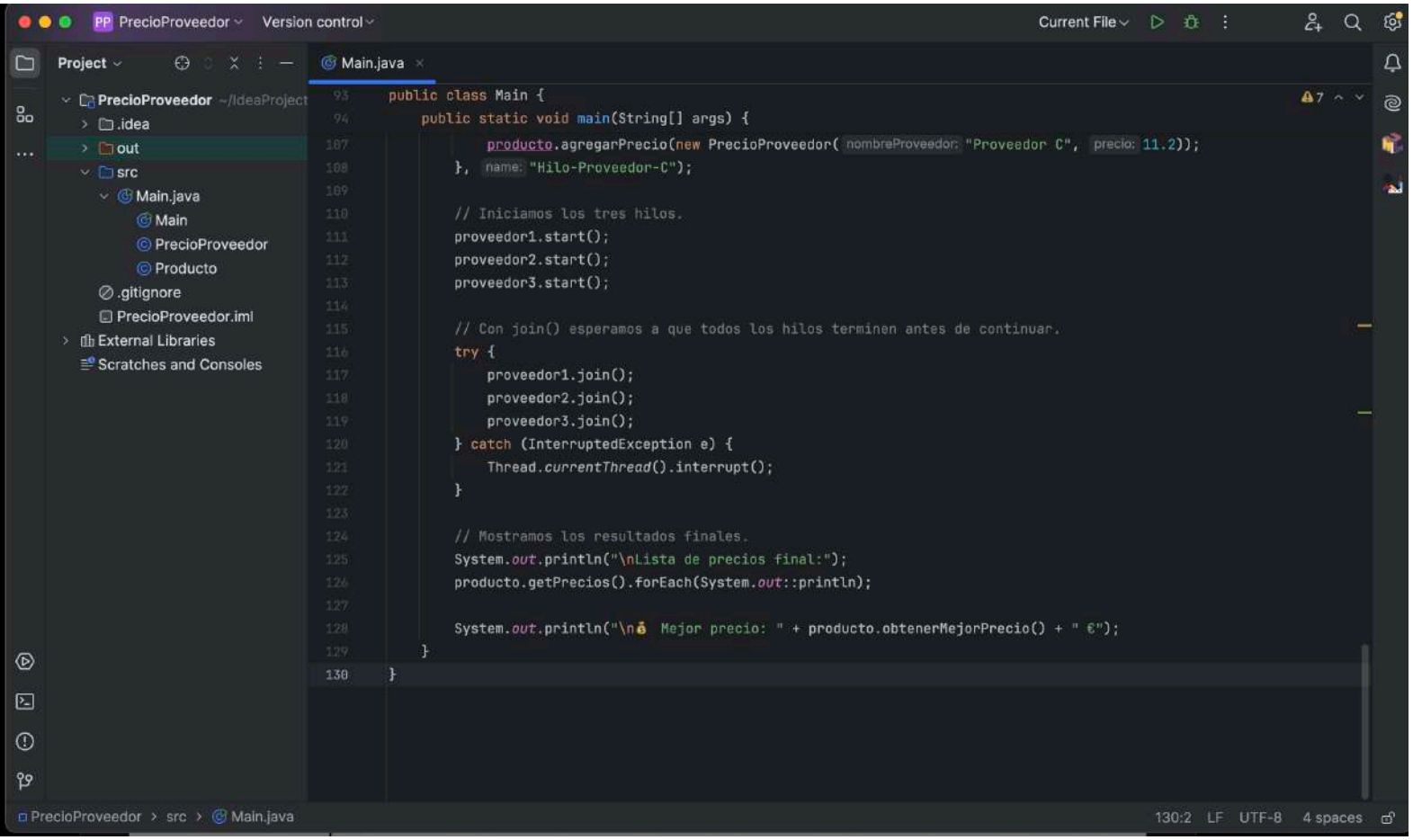
```java
class PrecioProveedor {

    @Override
    public String toString() {
        return nombreProveedor + ": " + precio + " €";
    }
}

// Clase Producto: gestiona el conjunto de precios de un producto,
// controlando el acceso concurrente mediante un mecanismo de exclusión mutua (ReentrantLock).
class Producto {

    // Lista que almacena los precios de los distintos proveedores.
    private List<PrecioProveedor> precios = new ArrayList<>();

    // Objeto Lock que permite controlar la zona crítica entre varios hilos.
    private final ReentrantLock lock = new ReentrantLock();

    // Getter con control de concurrencia.
    // Se bloquea el acceso mientras se lee para evitar inconsistencias si otro hilo está escribiendo.
    public List<PrecioProveedor> getPrecios() {
        lock.lock();
        try {
            // Se devuelve una copia del ArrayList original para evitar modificaciones externas.
            return new ArrayList<>(precios);
        } finally {
            lock.unlock(); // Se libera el bloqueo aunque ocurra una excepción.
        }
    }

    // Método que añade un nuevo precio a la lista.
```



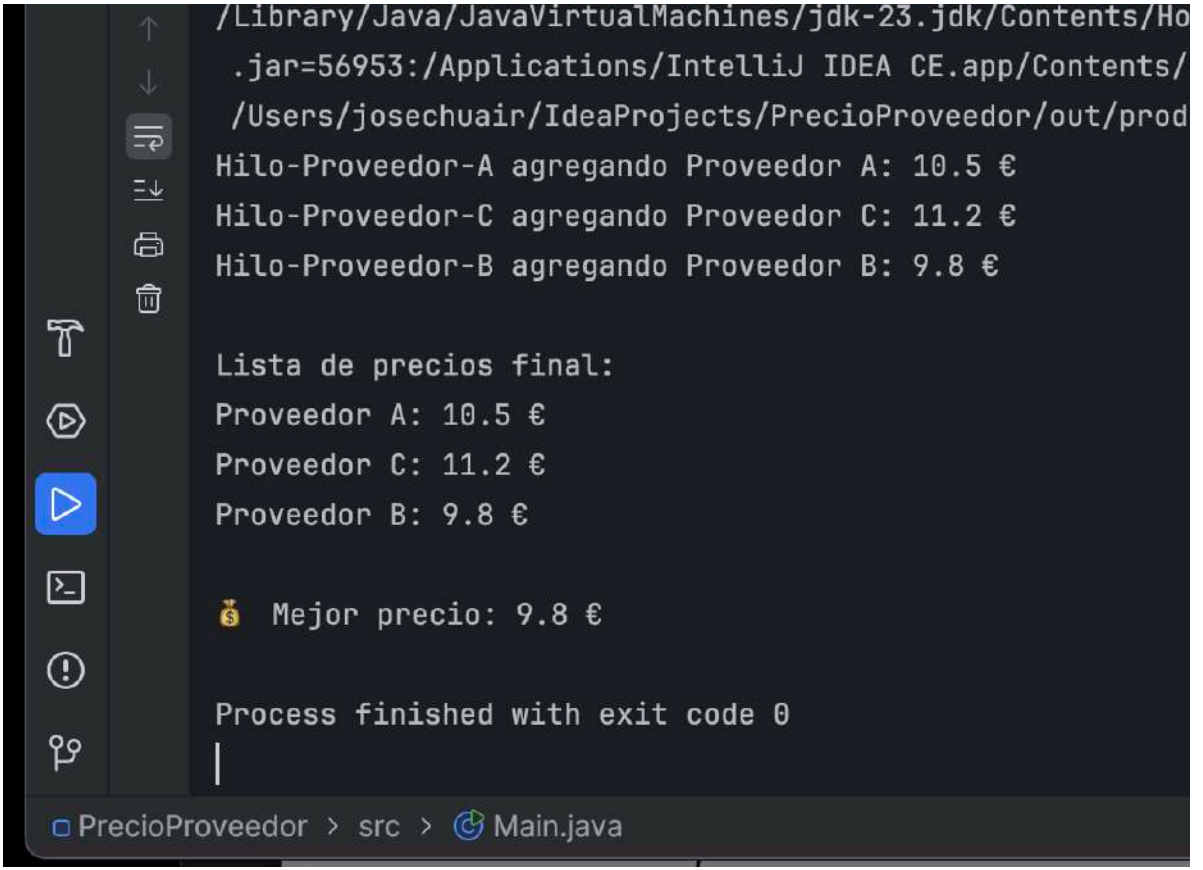
```
class Producto {  2 usages
    // Método que añade un nuevo precio a la lista.
    // Representa la zona crítica, ya que varios hilos pueden intentar escribir simultáneamente.
    public void agregarPrecio(PrecioProveedor precioProveedor) {  3 usages
        lock.lock(); // 🔒 Inicio de la sección crítica.
        try {
            System.out.println(Thread.currentThread().getName() + " agregando " + precioProveedor);
            precios.add(precioProveedor);

            // Simulamos el tiempo de escritura para mostrar el efecto del bloqueo.
            Thread.sleep( millis: 500);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock(); // 🔓 Fin de la sección crítica (libera el lock).
        }
    }

    // Calcula y devuelve el mejor precio (el menor) entre todos los proveedores.
    // También se protege con el bloqueo para asegurar una lectura consistente.
    public double obtenerMejorPrecio() {  1 usage
        lock.lock();
        try {
            return precios.stream()  Stream<PrecioProveedor>
                    .mapToDouble(PrecioProveedor::getPrecio)  DoubleStream
                    .min()  OptionalDouble
                    .orElse(Double.NaN);
        } finally {
            lock.unlock();
        }
```



```
class Producto {  2 usages
    public double obtenerMejorPrecio() {  1 usage
        }
    }
}

// Clase principal: simula la ejecución concurrente de varios proveedores que actualizan precios.
public class Main {
    public static void main(String[] args) {
        Producto producto = new Producto();

        // Creamos varios hilos que intentarán escribir en la lista de precios simultáneamente.
        Thread proveedor1 = new Thread(() -> {
            producto.agregarPrecio(new PrecioProveedor( nombreProveedor: "Proveedor A",  precio: 10.5));
        },  name: "Hilo-Proveedor-A");

        Thread proveedor2 = new Thread(() -> {
            producto.agregarPrecio(new PrecioProveedor( nombreProveedor: "Proveedor B",  precio: 9.8));
        },  name: "Hilo-Proveedor-B");

        Thread proveedor3 = new Thread(() -> {
            producto.agregarPrecio(new PrecioProveedor( nombreProveedor: "Proveedor C",  precio: 11.2));
        },  name: "Hilo-Proveedor-C");

        // Iniciamos los tres hilos.
        proveedor1.start();
        proveedor2.start();
        proveedor3.start();

        // Con join() esperamos a que todos los hilos terminen antes de continuar.
        try {
```



Aqui muestro la ejecución del programa y el resultado en la terminal :



Notas

La implementación de exclusión mutua mediante `ReentrantLock` garantiza la integridad de los datos en entornos multihilo.

Gracias a este mecanismo, se evita que varios procesos escriban simultáneamente en el mismo recurso, eliminando las condiciones de carrera.

Este tipo de control es esencial en sistemas concurrentes, especialmente en contextos empresariales donde se manejan precios, inventarios o información crítica compartida.

Documentacion :

Material de la Unidad 1

https://webprogramacion.com/exclusion-mutua/

Youtube

https://es.wikipedia.org/wiki/Exclusi%C3%B3n_mutua_(inform%C3%A1tica)

https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/concurrent/locks/ReentrantLock.html

https://es.scribd.com/document/832852201/Exclusion-Mutua