Técnicas de programación avanzada

Curso 2023-2024

Tema 5.1 Concurrencia

Tema 1: Objetos y memoria.

1.1 Características básicas del lenguaje. Primer programa. Compilación y Ejecución. IDE.

1.2 Sentencias de control. Secuencia, selección e iteración.

1.3 Abstracción. Clases, objetos, métodos y atributos.

1.4 Sobrecarga de métodos y encapsulamiento.

Tema 2. Otros conceptos fundamentales de la Programación Orientada a Objetos

2.1 Herencia. Interfaces y clases abstractas. Agregación.

2.2 Polimorfismo.

2.3 Gestión de Excepciones.

2.4 Genericidad y plantillas.

2.5 Utilidades. Entrada y Salida.

2.6 Anotaciones.

Tema 3. Patrones de Diseño.

3.1 Concepto de Patrones de Diseño.

3.2 Patrones de creación.

3.3 Patrones estructurales.

3.4 Patrones de comportamiento.

Tema 4. Programación de Interfaces.

4.1 Interfaces Gráficas de Usuario.

4.2 Gestión de eventos.

Tema 5. Temas Avanzados.

5.1 Concurrencia.

5.2 Inversión de Control. Definición y ejemplos. Inyección de dependencias.

5.3 Expresiones avanzadas del lenguaje.

5.1 Concurrencia

Intro

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

Definición Básica: La concurrencia en programación se refiere a la capacidad de un sistema para ejecutar dos o más tareas simultáneamente. Esto puede ser a nivel de múltiples procesos o dentro de un mismo proceso a través de múltiples hilos de ejecución.

Importancia de la Concurrencia en Java

- *Mejora del Rendimiento*: En aplicaciones donde se requiere realizar varias tareas que no dependen entre sí (por ejemplo, manejo de solicitudes de usuarios en un servidor web), la concurrencia puede mejorar significativamente el rendimiento y la eficiencia.

- *Uso Eficiente de Recursos*: La concurrencia permite que una aplicación haga un uso más eficiente de los recursos del sistema, ejecutando tareas en el fondo mientras se continúa con otras operaciones.

- *Aplicaciones Modernas*: En el contexto actual, donde las aplicaciones a menudo requieren manejar múltiples operaciones al mismo tiempo (como interfaces de usuario, procesamiento de datos, operaciones de red), la concurrencia se ha vuelto esencial.

5.1 Concurrencia

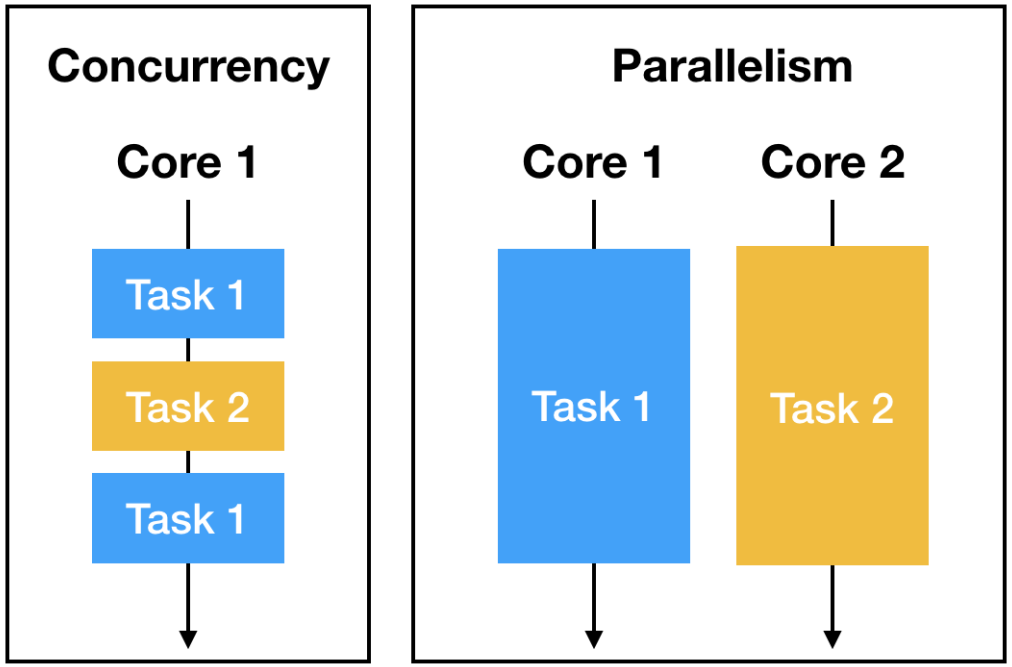
Concurrency vs Parallelism

- Más de un procesador corriendo simultáneamente y accediendo a los mismos recursos (“un ratito tu, un ratito yo,…”) Context switch

- Puede haber problemas si se quiere modificar la misma información a la vez

- Para evitar conflictos, deben coordinarse / sincronizarse las tareas (synchronized) → solo un hilo tiene acceso en cada momento, el resto está “bloqueado”

- A veces cuando queremos usar paralelismo, hacemos uso de concurrencia (falso paralelismo) – 1 CPU

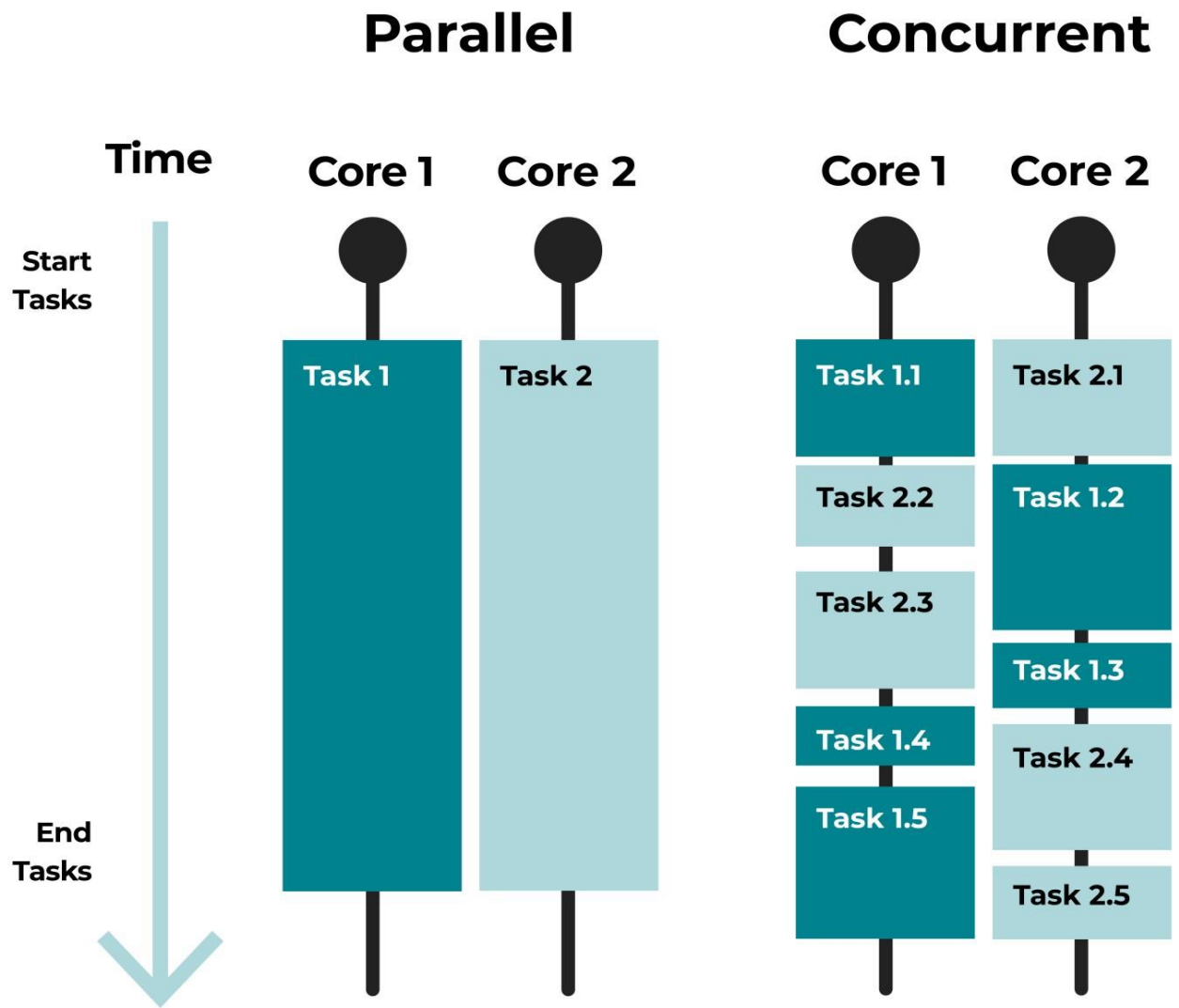


- Se ejecutan realmente diferentes tareas a la vez

- Requiere que el equipo disponga de diferentes CPUs (unidades de procesamiento)

5.1 Concurrencia

Concurrency vs Parallelism



- Tener multiples procesadores no implica que la ejecución de nuestro Código vaya a ser en paralelo

5.1 Concurrencia

Hilos (Thread)

Java utiliza hilos (threads) como su principal unidad de concurrencia. Un hilo en Java es un camino de ejecución dentro de un programa. Cada programa Java tiene al menos un hilo, conocido como el hilo principal, que es ejecutado al iniciar el programa. Los hilos adicionales se pueden crear para realizar tareas específicas.

Cómo implementar un hilo:

1. Extendiendo la clase *Thread* → https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
2. Implementando la interfaz *Runnable* → https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html

En ambos casos, tendremos que instanciar después un Thread e invocar al método “start”.

5.1 Concurrencia

Hilos (Thread)

Extendiendo la Clase Thread:

- Concepto: La clase Thread en Java proporciona un esqueleto para crear y ejecutar un hilo. Al extender Thread, se heredan sus capacidades de manejo de hilos.

- Implementación: Creas una subclase de Thread y sobrescribes su método run(), que define la tarea que el hilo ejecutará.

Ejemplo:

```
class MyThread extends Thread {
    public void run() {
        // código a ejecutar en el hilo
    }
}

MyThread t = new MyThread();
t.start(); // Inicia la ejecución del hilo
```

5.1 Concurrencia

Hilos (Thread)

Implementando la Interfaz Runnable:

- Concepto: Runnable es una interfaz funcional en Java que define un solo método abstracto run(). Al implementar Runnable, defines la tarea que el hilo realizará, pero la gestión del hilo se maneja a través de una instancia de Thread.

- Implementación: Creas una clase que implementa Runnable y defines el método run(). Luego, pasas una instancia de esta clase al constructor de Thread y llamas a start().

Ejemplo:

```
class MyRunnable implements Runnable {
    public void run() {
        // código a ejecutar en el hilo
    }
}

Thread t = new Thread(new MyRunnable());
t.start(); // Inicia la ejecución del hilo
```

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

UNIVERSIDAD NEBRIJA

Control del Ciclo de Vida: Java permite controlar el ciclo de vida de un hilo mediante métodos como start(), join(), interrupt(), sleep(), entre otros.

- **start()**: Inicia la ejecución del hilo.
- **join()**: Espera a que un hilo termine su ejecución.
- **interrupt()**: Solicita la interrupción de un hilo.
- **sleep(long millis)**: Hace que el hilo actual se detenga durante un periodo específico.

Prioridades de Hilos: Puedes establecer u obtener la prioridad de un hilo usando **setPriority(int newPriority)** y **getPriority()**. Las prioridades ayudan al planificador de hilos a decidir qué hilo ejecutar primero, aunque la implementación exacta de cómo se manejan las prioridades depende de la JVM y del sistema operativo.

Daemon Threads: Un hilo daemon en Java es un hilo de baja prioridad que se ejecuta en segundo plano para realizar tareas de servicio, como la recolección de basura. Se puede configurar un hilo como daemon mediante **setDaemon(true)** antes de que el hilo sea iniciado.

Manejo de Excepciones: Las excepciones en los hilos deben manejarse cuidadosamente, ya que una excepción no capturada en un hilo puede afectar a otros hilos y al estado de la aplicación. Java proporciona **UncaughtExceptionHandler** para manejar tales excepciones.

5.1 Concurrencia

Ejemplo de implementación

https://github.com/aalonsopuig/Java_Ejemplos_5/tree/main/src/a_hilos

UNIVERSIDAD NEBRIJA

```
import java.lang.Math ;

class EjemploThread extends Thread {
   int numero;
   EjemploThread (int n) { numero = n; }

   @Override
   public void run() {
      try {
         while (true) {
               System.out.println (numero);
               sleep((long)(1000*Math.random()));
         }
      } catch (InterruptedException e) { return; } // acaba este thread
   }

   public static void main (String args[]) {
      for (int i=0; i<10; i++)
      new EjemploThread(i).start();
   }
}
```

Importante: se lanza con START para crear un nuevo hilo.

Necesidad de Sincronización

Cuando múltiples hilos acceden y modifican los mismos recursos (como variables o estructuras de datos), pueden surgir condiciones de carrera y datos inconsistentes. La sincronización es el proceso de controlar el acceso a estos recursos compartidos para garantizar la consistencia y evitar problemas concurrentes.

Bloqueo

La sincronización a menudo implica bloquear el acceso a ciertas partes del código para que solo un hilo pueda ejecutarlas a la vez.

Sincronización de Métodos:

- Utilizas la palabra clave **synchronized** en la definición de un método para hacer que el método sea sincronizado.
- Cuando un hilo invoca un método sincronizado, adquiere un bloqueo (o monitor) para el objeto que contiene el método. Otros hilos que intentan invocar cualquier método sincronizado en el mismo objeto serán bloqueados hasta que el hilo actual libere el bloqueo.

Sincronización de Bloques

- Dentro de un método, puedes sincronizar solo una parte del código utilizando bloques sincronizados.
- Esto se hace encerrando el código en **{}** y precediéndolo con la palabra clave **synchronized** y el objeto que sirve como bloqueo.

```
synchronized(objectidentifier) {
   // Access shared variables and other shared resources
}
```

UNIVERSIDAD NEBRIJA

Consideraciones Importantes

- *Deadlocks*: Un problema común en la programación concurrente es el deadlock, que ocurre cuando dos o más hilos están bloqueados indefinidamente, esperando que los otros liberen los recursos.
- *Rendimiento*: La sincronización puede afectar el rendimiento, ya que restringe la ejecución paralela. Por lo tanto, es importante usarla de manera eficiente y solo cuando sea necesario.
- *Buenas Prácticas*: Se recomienda minimizar el alcance de la sincronización, evitar la sincronización anidada cuando sea posible y ser consciente de los deadlocks.

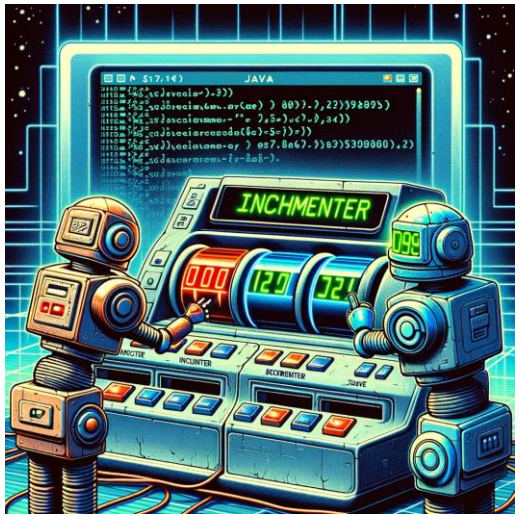
```java
public class SharedResource {
    private int counter = 0;

    // Método sincronizado
    public synchronized void increment() {
        counter++;
    }

    // Bloque sincronizado
    public void decrement() {
        synchronized (this) {
            counter--;
        }
    }
}
```

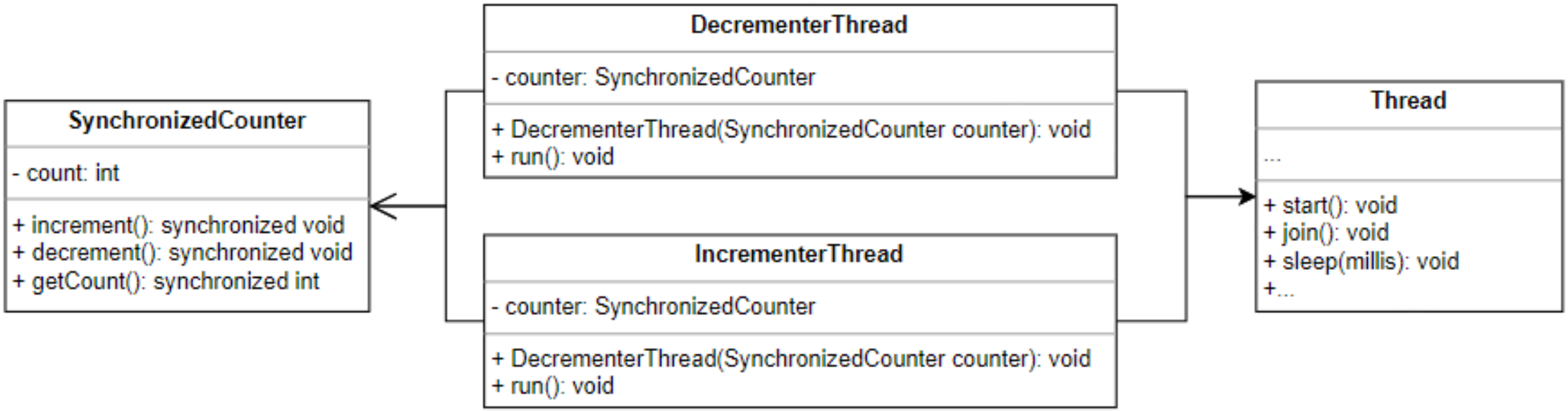
5.1 Concurrencia

Ejemplo

Objetivo: Desarrollar un programa en Java que demuestre el uso efectivo de la sincronización en un entorno de programación multihilo.

Descripción: Implementar un programa en Java que mantendrá un contador compartido. Este contador será modificado por múltiples hilos para demostrar la sincronización de acceso a un recurso compartido. El programa debe evitar condiciones de carrera y garantizar la consistencia de los datos del contador.

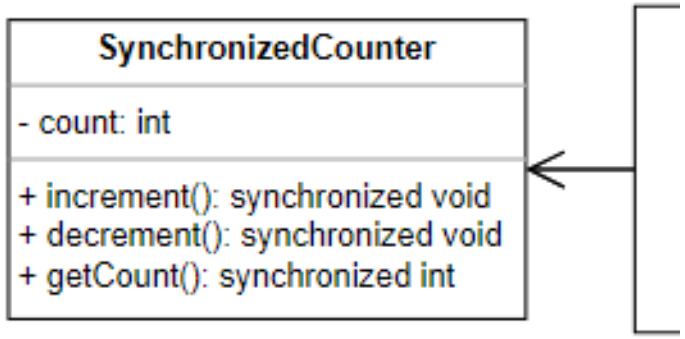




CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_5/tree/main/src/b_hilos_synch_sumador_restador

5.1 Concurrencia

Ejemplo




```java
public class SynchronizedCounter {

    // Variable que mantiene el valor actual del contador.
    private int count = 0;

    public synchronized void increment() {
        count++;
        System.out.println("Incrementado: " + count);
    }

    public synchronized void decrement() {
        count--;
        System.out.println("Decrementado: " + count);
    }

    public synchronized int getCount() {
        return count;
    }
}
```

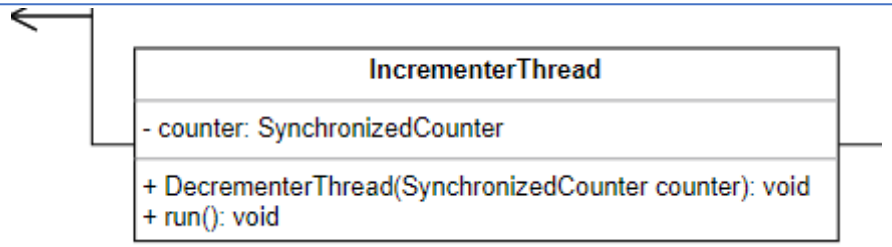
CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_5/tree/main/src/b_hilos_synch_sumador_restador

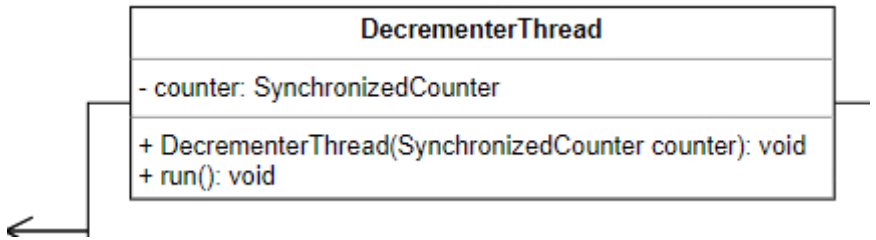
5.1 Concurrencia

Ejemplo

```java
public class IncrementerThread extends Thread {
    private SynchronizedCounter counter;

    IncrementerThread(SynchronizedCounter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            counter.increment();
            try {
                Thread.sleep(100); // Simula trabajo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```java
public class DecrementerThread extends Thread {
    private SynchronizedCounter counter;

    DecrementerThread(SynchronizedCounter counter) {
        this.counter = counter;
    }

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            counter.decrement();
            try {
                Thread.sleep(100); // Simula trabajo
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```





CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_5/tree/main/src/b_hilos_synch_sumador_restador

5.1 Concurrencia

Ejemplo

```java
public class MainClass {
    public static void main(String[] args) {
        SynchronizedCounter counter = new SynchronizedCounter();

        // Creación de hilos
        Thread incrementerThread = new IncrementerThread(counter);
        Thread decrementerThread = new DecrementerThread(counter);

        // Iniciando los hilos
        incrementerThread.start();
        decrementerThread.start();

        try {
            incrementerThread.join();
            decrementerThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("Valor final del contador: " + counter.getCount());
    }
}
```

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_5/tree/main/src/b_hilos_synch_sumador_restador

5.1 Concurrencia

Ejemplo. Explicación

El programa Java que hemos mostrado es un ejemplo clásico de cómo implementar y manejar la sincronización en un ambiente multihilo. A continuación, te explico cada parte del programa:

Clase SynchronizedCounter
Esta clase sirve como un recurso compartido que será accedido y modificado por múltiples hilos. Contiene un contador (int count) que representa el recurso compartido.
Métodos Sincronizados:

- increment(): Aumenta el valor del contador en uno. Está sincronizado para evitar que múltiples hilos modifiquen el contador al mismo tiempo, lo que podría llevar a resultados inconsistentes o una condición de carrera.
- decrement(): Disminuye el valor del contador en uno, también sincronizado por las mismas razones.
- getCount(): Devuelve el valor actual del contador. Este método también es sincronizado para asegurar una lectura consistente del valor del contador mientras otros hilos pueden estar intentando modificarlo.

Clases de Hilos: IncrementerThread y DecrementerThread
Estas clases son hilos que modifican el contador. Cada una de estas clases extiende Thread y realiza operaciones específicas sobre el contador.

- IncrementerThread: Incrementa el contador. En su método run(), este hilo llama repetidamente al método increment() de SynchronizedCounter.
- DecrementerThread: Decrementa el contador. Similar al IncrementerThread, pero llama al método decrement() del contador en su método run().

Ejecución y Prueba
En el método main, se inician los hilos y se espera a que terminen usando join(). Esto asegura que el programa principal espere a que todos los hilos hayan completado sus operaciones antes de imprimir el valor final del contador.

CÓDIGO: https://github.com/aalonsopuig/Java_Ejemplos_5/tree/main/src/b_hilos_synch_sumador_restador

Ejercicios

5.1 Concurrencia

Ejercicios

Carrera de Camellos Virtual

Desarrollar un programa en Java que simule una carrera de camellos. Cada camello será representado por un hilo que avanza a lo largo de una pista. Deberás utilizar mecanismos de sincronización para controlar el acceso a la línea de meta y asegurar que solo un camello sea declarado ganador.

Clase CarreraCamellos:

- Deberá tener una longitud definida para la pista de carrera.
- Implementar un método sincronizado **cruzarMeta** que determine y anuncie el camello ganador.

Clase Camello (Extiende Thread):

- Representa a un competidor en la carrera.
- Cada camello debe avanzar a lo largo de la pista y, al llegar al final, intentar cruzar la meta llamando al método **cruzarMeta**.

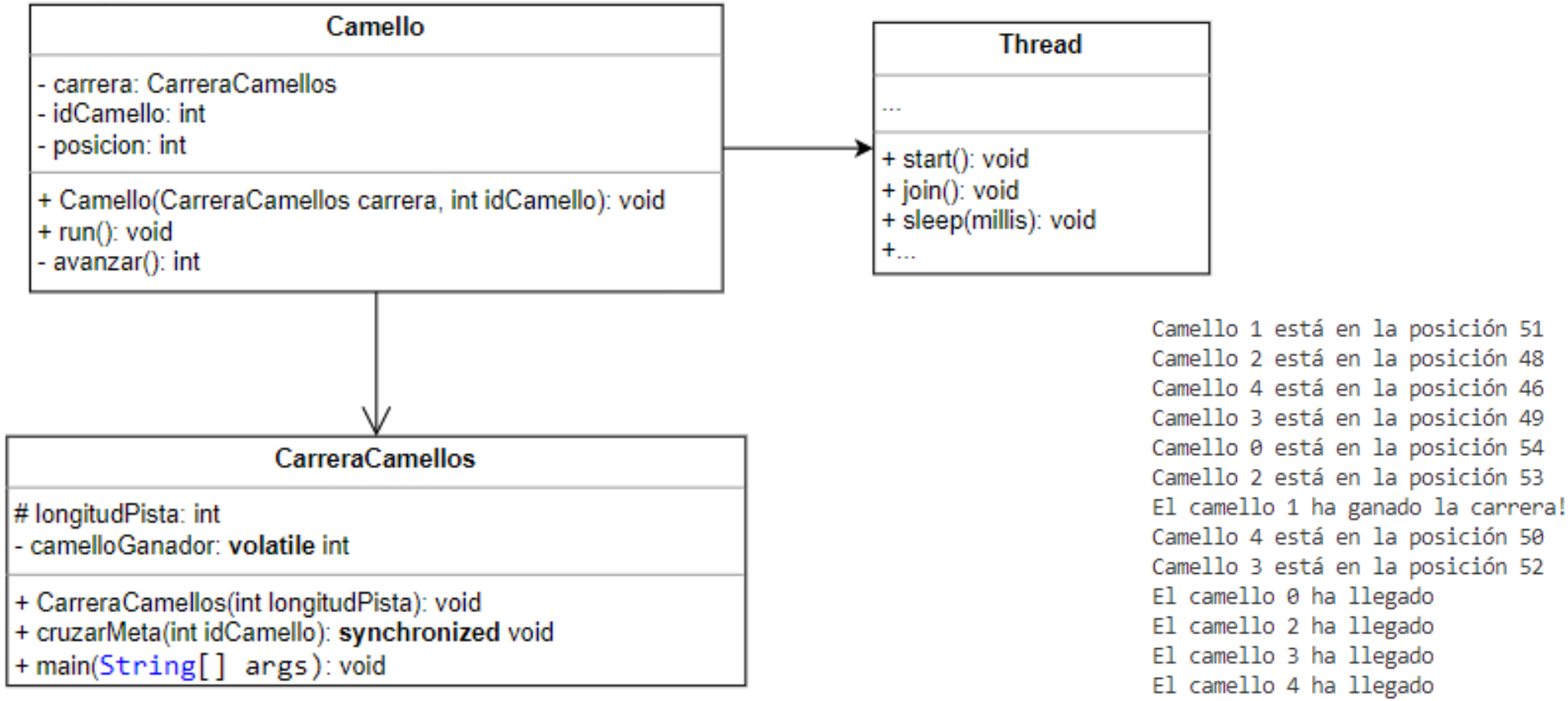
Sincronización:

- Asegurar que el acceso al método **cruzarMeta** sea sincronizado para evitar conflictos entre los hilos.

Visualización y Resultado:

- El programa debe mostrar el avance de cada camello y anunciar al ganador de la carrera.

Pistas




```
Camello 1 está en la posición 51
Camello 2 está en la posición 48
Camello 4 está en la posición 46
Camello 3 está en la posición 49
Camello 0 está en la posición 54
Camello 2 está en la posición 53
El camello 1 ha ganado la carrera!
Camello 4 está en la posición 50
Camello 3 está en la posición 52
El camello 0 ha llegado
El camello 2 ha llegado
El camello 3 ha llegado
El camello 4 ha llegado
```

5.1 Concurrencia

Ejercicios

Gestión de Reservas en un Restaurante

Desarrollar un programa en Java que simule un sistema de reservas en un restaurante. Los clientes intentarán hacer reservas en el restaurante. Deberás emplear mecanismos de sincronización para gestionar las reservas y evitar conflictos, como la sobreocupación o reservas duplicadas, incluyendo la identificación de cada cliente en el proceso.

Clase Restaurante:

- Contiene un número limitado de mesas disponibles.
- Implementa métodos sincronizados para hacer y cancelar reservas, teniendo en cuenta la identificación de los clientes.

Clase Cliente (Extiende Thread):

- Representa a un cliente que intenta hacer una reserva en el restaurante.
- Cada cliente debe tener un identificador único y debe intentar hacer una reserva llamando al método correspondiente en **Restaurante**.

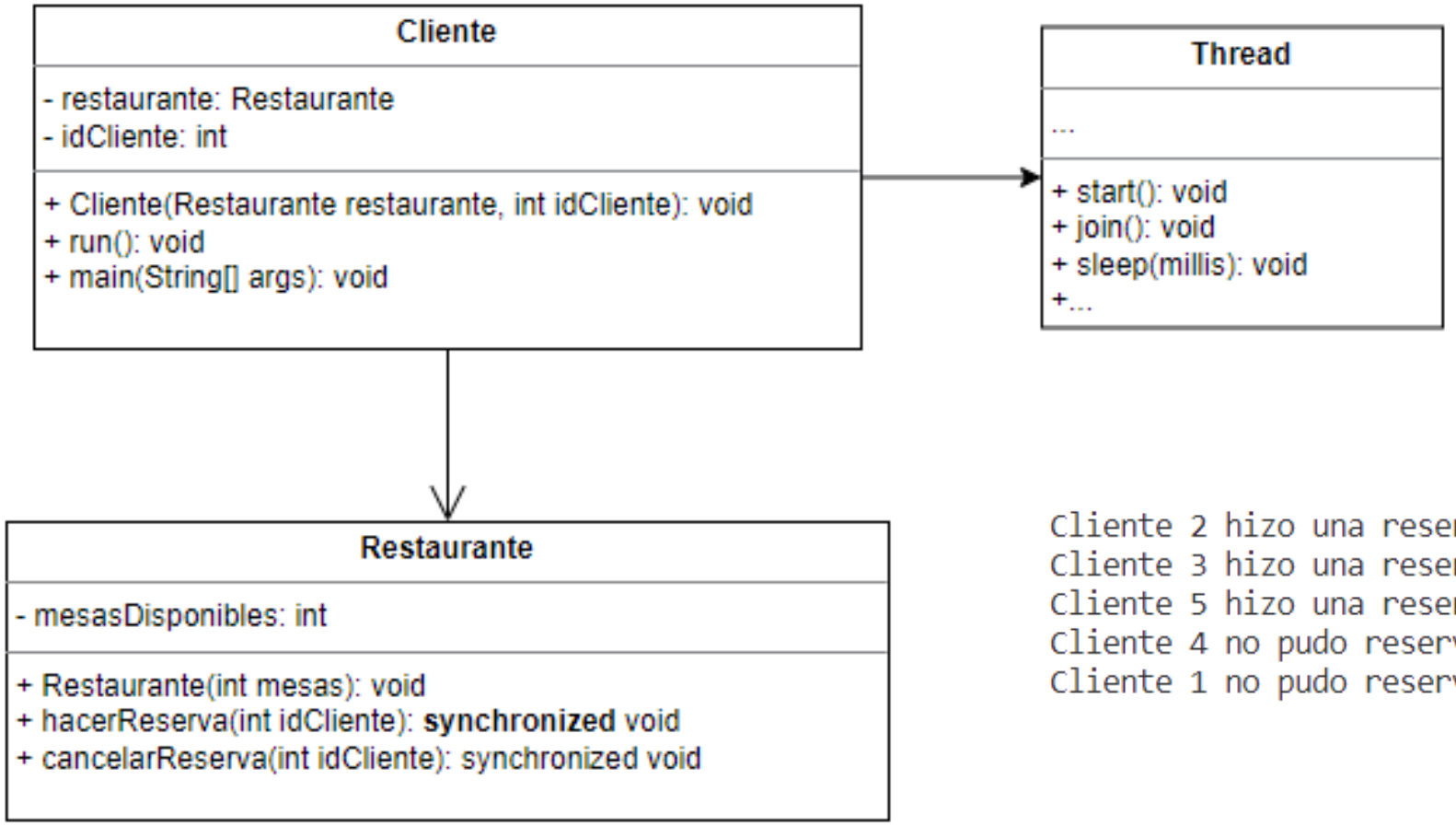
Sincronización:

- Garantiza que el acceso a los métodos de reserva y cancelación sea sincronizado para prevenir conflictos entre múltiples clientes.

Visualización y Resultado:

- El programa debe mostrar información relevante, como cuándo un cliente hace o cancela una reserva y la disponibilidad actual de mesas en el restaurante, incluyendo la identificación del cliente.

Pistas




```
Cliente 2 hizo una reserva. Mesas disponibles: 2
Cliente 3 hizo una reserva. Mesas disponibles: 1
Cliente 5 hizo una reserva. Mesas disponibles: 0
Cliente 4 no pudo reservar. No hay mesas disponibles.
Cliente 1 no pudo reservar. No hay mesas disponibles.
```

Bibliografía

- Deitel, H. M. & Deitel, P. J.(2008). *Java: como programar*. Pearson education. Séptima edición.
- The Java tutorials. https://docs.oracle.com/javase/tutorial/
- Páginas de apoyo para la sintaxis, bibliotecas, etc.:
  - https://www.sololearn.com
  - https://www.w3schools.com/java/default.asp
  - https://docstore.mik.ua/orelly/java-ent/jnut/index.htm

Técnicas de Programación Avanzada

```
exit(); //Gracias!
```