1. Visión General de la Clase

- **Nombre**: `public class Order implements Directions`

- **Responsabilidad**: Orquestar la creación y gestión de trenes, reservar y liberar celdas en el mapa, y controlar señales de operador («4:20» y «11:00») mediante semáforos y variables `volatile`.

---

2. Campos Principales

```
private final Semaphore intersectionSemaphore;        // Controla intersección crítica
private final Semaphore cisnerosToSanAntonioSemaphore;
private final Semaphore lineBExitSemaphore;
private final Semaphore lineATallerSemaphore;
private final Semaphore depotEntranceSemaphore;
private final Object depotLock;                       // Lock adicional para entrada al depósito
private volatile boolean waitingFor420 = true;        // Flag para señal de inicio del servicio
public volatile boolean isReturningToDepot = false;  // Flag para señal de retorno al taller
public int[][] map = new int[36][21];                 // Mapa de ocupación de celdas
private final int[] NIQUIA_COORDS = {35, 19};         // Coordenadas fijas de estaciones
private final int[] ESTRELLA_COORDS = {16, 1};
private final int[] SANJAVIER_COORDS = {1, 11};
```

- **Semáforos**: Cada intersección o tramo crítico tiene su propio `Semaphore` para garantizar exclusividad.

- **Mapa**: Reserva (`1`) y libera (`0`) casillas para prevenir colisiones.

- **Locks y flags**: Variables `volatile` definen puntos de sincronización no bloqueantes.

---

3. Constructor e Inicialización

```
public Order() {
    intersectionSemaphore = new Semaphore(1);
    cisnerosToSanAntonioSemaphore = new Semaphore(1);
    lineBExitSemaphore = new Semaphore(1);
    lineATallerSemaphore = new Semaphore(1);
    depotEntranceSemaphore = new Semaphore(1);
    depotLock = new Object();
```

```
    initializeTrains();
}
```

- Asigna cada semáforo con un único permiso (`1`).
- Llama a `initializeTrains()` para construir los hilos `Train`.

---

4. `initializeTrains()`: Creando y configurando trenes

```
private void initializeTrains() {
    List<Train> creationOrder = new ArrayList<>();
    for (int i = 0; i < 32; i++) {
        String route = (i % 3 == 0 ? "AN" : i % 3 == 1 ? "AE" : "ASJ");
        boolean isInitial = (i < 3);
        Train t = new Train(startRow, startCol, Directions.NORTH, 1, trainColor, route, this);
        t.isInitialTrain = isInitial;
        creationOrder.add(t);
    }
    // Hilos listos: el orden de arranque usa creationOrder
}
```

- Alterna rutas entre los 32 trenes.
- Marca los primeros tres como `initialTrain`.
- Prepara la lista para `startSystem()`.

---

5. `startSystem()`: Control del ciclo de servicio

```
public void startSystem() {
    // 1. Arranca los 3 trenes iniciales:
    for (Train t : creationOrder.subList(0, 3)) t.start();
    // 2. Espera a que lleguen a sus estaciones terminales (map[...]=1)
    waitForInitialPosition();
    // 3. Espera input "4:20" para desactivar waitingFor420
    readOperatorSignal("4:20");
    waitingFor420 = false;
    // 4. Arranca los trenes restantes
    for (Train t : creationOrder.subList(3, creationOrder.size())) t.start();
    // 5. Lee "11:00" y activa isReturningToDepot
```

```
    readOperatorSignal("11:00");
    isReturningToDepot = true;
}
```

- **Fases**: arranque inicial, señal de inicio, arranque completo, señal de retorno.
- **Lectura de señales**: Utiliza bucles de I/O ligeros sin bloquear semáforos.

---

6. Métodos de Adquisición y Liberación

```
public void acquireIntersection() throws InterruptedException {
    intersectionSemaphore.acquire();
}
public void releaseIntersection() {
    intersectionSemaphore.release();
}
// Métodos similares para cada semáforo...
public void acquireDepotEntrance() throws InterruptedException {
    depotEntranceSemaphore.acquire();
    synchronized(depotLock) {
        Thread.sleep(50); // Retraso entre entradas
    }
}
public void releaseDepotEntrance() {
    depotEntranceSemaphore.release();
}
```

- **Control granular**: Cada semáforo maneja una zona distinta.
- `depotLock`: Asegura retrasos y orden en la entrada al taller.

---

7. Gestión del Mapa de Ocupación

```
public synchronized void updateMap(int oldRow, int oldCol,
                                   int newRow, int newCol) {
    map[oldRow][oldCol] = 0;
    map[newRow][newCol] = 1;
}
```

- **Sincronizado**: Para evitar accesos concurrentes al array.
- **Reservas**: Marca salida y llegada en cada movimiento.

---

8. Señales y Variables `volatile`

- `waitingFor420`: Controla que los trenes iniciales pausen antes de arrancar el servicio completo.
- `isReturningToDepot`: Indica a cada hilo que debe abandonar el bucle comercial y volver al depósito.
- Mantenerlas `volatile` garantiza visibilidad inmediata entre hilos.

---

9. Manejo de Posicionamiento Inicial y Señales del Operador

```
private void waitForInitialPosition() throws InterruptedException {
    // Espera hasta que las 3 estaciones terminales estén ocupadas en el mapa
    while (map[NIQUIA_COORDS[0]][NIQUIA_COORDS[1]] == 0
            || map[ESTRELLA_COORDS[0]][ESTRELLA_COORDS[1]] == 0
            || map[SANJAVIER_COORDS[0]][SANJAVIER_COORDS[1]] == 0) {
        Thread.sleep(100);
    }
}

private void readOperatorSignal(String expected) {
    // Lectura en bucle de consola sin bloquear semáforos
    Scanner sc = new Scanner(System.in);
    String input;
    do {
        System.out.print("Ingrese señal ('" + expected + "'): ");
        input = sc.nextLine().trim();
    } while (!input.equals(expected));
}
```

- **`waitForInitialPosition()`**: Evita el arranque completo hasta que los trenes iniciales alcancen sus terminales.
- **`readOperatorSignal()`**: Ilustra el patrón de lectura ligera de I/O para señales, sin retener locks ni semáforos.

---

10. Manejo de Excepciones y Robustez

- Cada método de adquisición lanza `InterruptedException`, que se propaga para permitir la finalización limpia de hilos.
- Uso de `synchronized` en `updateMap` evita corrupciones de datos en escenarios de alta contención.

---

11. Conclusión

La clase `Order` demuestra cómo centralizar la orquestación de múltiples hilos en un sistema educativo con necesidades reales de concurrencia. Sus mecanismos de sincronización, reserva de recursos y manejo de señales ofrecen un marco robusto y adaptable a cambios futuros en rutas, reglas o tiempos de operación.