RA2. Desarrolla aplicaciones compuestas por varios hilos de ejecución analizando y aplicando librerías específicas del lenguaje de programación. a) Se han identificado situaciones en las que resulte útil la utilización de varios hilos en un programa.

b) Se han reconocido los mecanismos para crear, iniciar y finalizar hilos.

c) Se han programado aplicaciones que implementen varios hilos.

d) Se han identificado los posibles estados de ejecución de un hilo y programado aplicaciones que los gestionen.

e) Se han utilizado mecanismos para compartir información entre varios hilos de un mismo proceso.

f) Se han desarrollado programas formados por varios hilos sincronizados mediante técnicas específicas.

g) Se ha establecido y controlado la prioridad de cada uno de los hilos de ejecución.

h) Se han depurado y documentado los programas desarrollados.

i) Se ha realizado control de versiones en el código generado.

j) Se ha analizado el contexto de ejecución de los hilos.

k) Se han analizado librerías específicas del lenguaje de programación que permiten la programación multihilo.

l) Se han reconocido los problemas derivados de la compartición de información entre los hilos de un mismo proceso.

m) Se ha utilizado una metodología para la planificación, asignación y desarrollo de las funcionalidades generadas.

1. Crea un repositorio en github, llámalo práctica2.2-psp, en este repositorio debes subir los siguientes ejercicios y enviar el enlace del repositorio.

2. En el siguiente enlace puedes encontrar un programa que usa el concepto syncronized de la programación multihilo. Analiza el programa, ejecútalo y explica lo que hace.
https://drive.google.com/file/d/1pIWv63lyiliSqQhes8Z1Y_jM6RWBuX64/view?usp=sharing

1. Clases y roles

El programa tiene dos clases:

1. **`Turnos`**:
    a. Controla el turno de ejecución de los hilos.
    b. Tiene un atributo `turno` que indica qué hilo debe ejecutar:
        i. `0` → hilo A
        ii. `1` → hilo B
        iii. `2` → hilo C
    c. Métodos sincronizados (`synchronized`):
        i. `esperarTurno(int miTurno)`: hace que el hilo espere hasta que sea su turno. Usa `wait()` dentro de un `while` para esperar.
        ii. `siguiente()`: cambia al siguiente turno y despierta todos los hilos con `notifyAll()`.
2. **`HiloImprimePorTurno`**:
    a. Hereda de `Thread`.
    b. Tiene atributos:
        i. `Turnos turno`: referencia al objeto compartido para controlar los turnos.
        ii. `miTurno`: indica el turno asignado a ese hilo (0, 1 o 2).
        iii. `inicio`: primer número que imprimirá.
        iv. `paso`: incremento (aunque no se usa directamente en el bucle, ya que `i += 3` está hardcodeado).
    c. En `run()`, cada hilo:
        i. Espera su turno usando `turno.esperarTurno(miTurno)`.
        ii. Imprime su número.
        iii. Llama a `turno.siguiente()` para pasar el turno al siguiente hilo.

2. Ejecución de los hilos

En `main()`:

```
HiloImprimePorTurno a = new HiloImprimePorTurno(turno, 0, 1, 3);
HiloImprimePorTurno b = new HiloImprimePorTurno(turno, 1, 2, 3);
HiloImprimePorTurno c = new HiloImprimePorTurno(turno, 2, 3, 3);
```

- Hilo A imprime 1, 4, 7, 10
- Hilo B imprime 2, 5, 8
- Hilo C imprime 3, 6, 9

El bucle `for (int i = inicio; i <= 10; i += 3)` asegura que cada hilo imprime su secuencia.

3. Uso de synchronized

- `esperarTurno()` y `siguiente()` están **sincronizados**:
  - Esto garantiza que solo un hilo acceda a estas secciones críticas al mismo tiempo.
  - Evita que varios hilos lean/escriban el turno simultáneamente, lo que podría desordenar la impresión.
- `wait()` hace que un hilo se bloquee hasta que `notifyAll()` despierte a todos los hilos que están esperando.
- `notifyAll()` permite que todos los hilos despierten y vuelvan a comprobar si es su turno.

4. Salida esperada

El programa imprime los números del 1 al 10 en **orden alternado por hilos**:

```
A imprime: 1
B imprime: 2
C imprime: 3
A imprime: 4
B imprime: 5
C imprime: 6
A imprime: 7
B imprime: 8
C imprime: 9
A imprime: 10
Fin impresión ordenada.
```

- La impresión está **ordenada por turnos**, gracias a la sincronización.

5. Resumen de funcionamiento

1. Se crean tres hilos A, B y C.
2. Cada hilo imprime números en una secuencia específica.
3. El objeto `Turnos` asegura que solo el hilo cuyo turno coincide con `turno` pueda imprimir.
4. La sincronización (`synchronized`, `wait`, `notifyAll`) garantiza que los hilos no interfieran entre sí y que la impresión sea **ordenada y predecible**.

3. En los apuntes, en la página 38 tienes el ejemplo de un programa productor-consumidor. Analízalo, desarróllalo y ejecútalo, una vez realizados estos pasos explica su funcionamiento.
   https://aulasciclos2526.castillalamancha.es/pluginfile.php/806684/mod_resource/content/1/PSPUT2%20-%2002%20Programacio%CC%81n%20Sincronizada%20Multihilo.pdf

Cómo funciona

1. El productor agrega elementos al buffer hasta que llega a la capacidad máxima.
2. Si el buffer está lleno, el productor se bloquea (`wait`) hasta que el consumidor retire un elemento.
3. El consumidor quita elementos del buffer. Si está vacío, se bloquea hasta que el productor agregue algo.
4. `notifyAll()` asegura que los hilos que esperan se despierten cuando se produce un cambio en el buffer.
5. `sleep()` simula el tiempo que tarda en producir o consumir, mostrando el efecto en tiempo real.

4. Simula un sistema de control de acceso a una sala de conferencias en un edificio de oficinas.
   En este caso, habrá varios empleados que desean usar una sala de conferencias compartida.
   Los empleados deben esperar si la sala está ocupada o si hay demasiados empleados dentro de la sala, ya que la sala tiene un límite de ocupación (por ejemplo, 5 personas como máximo)

   Escenario:

Sala de conferencias compartida: Tiene una capacidad máxima de 5 personas. Si ya hay 5 personas en la sala, el siguiente empleado debe esperar fuera.
Empleados: Son hilos que intentan entrar en la sala. Si la sala está llena, deben esperar hasta que alguien salga. Además, solo puede haber un número máximo de empleados dentro al mismo tiempo.
Salida: Cuando un empleado sale de la sala, debe notificar a los demás empleados esperando fuera para que puedan entrar si hay espacio.

Reglas:
La sala tiene una capacidad máxima de 5 personas.
Hay 30 empleados. Si llegan y ven que la sala está llena deben esperar fuera, tienen 3 intentos, deja 12 segundos entre intentos.
Un empleado solo puede entrar cuando hay espacio disponible (cuando otro haya salido).
Los empleados están 10 segundos en la sala y tardan 1,5 en salir.
Cuando un empleado sale de la sala, debe notificar a los demás empleados para que puedan entrar.

1. **SalaConferencias**
    a. Mantiene la ocupación actual.
    b. Usa un `Lock` y una `Condition` para controlar el acceso.
    c. Método `entrar`:
        i. Si la sala está llena, el empleado espera hasta 12 segundos.
        ii. Cada empleado tiene 3 intentos para entrar.
    d. Método `salir`:
        i. Reduce la ocupación y notifica a todos los empleados esperando.
2. **Empleado**
    a. Cada empleado es un hilo.
    b. Intenta entrar a la sala usando el método `entrar`.
    c. Si entra, permanece 10 segundos en la sala.
    d. Tarda 1,5 segundos en salir y luego llama a `salir` para notificar a otros.
3. **SimulacionSala**
    a. Crea 30 empleados que intentan acceder a la sala de conferencias de 5 plazas.

5. Simulación Cuenta Bancaria
Crea una simulación de una cuenta bancaria compartida por 5 clientes. Cada cliente debe poder ingresar dinero, sacar dinero y ver su saldo. Sin embargo,

el banco tiene una limitación, sólo 2 clientes pueden acceder a la cuenta simultáneamente.

Reglas:
Define las clases necesarias con sus métodos.
Debes implementar la clase Thread para el manejo de los hilos.
Cada hilo representará un cliente.
Asegúrate de evitar condiciones de carrera.
El cliente aleatoriamente realizará una operación de ingresar o de sacar dinero.
El programa debe terminar cuando todos los clientes hayan realizado 3 operaciones del tipo ingresar o sacar dinero.
Cada operación debe tardar un tiempo aleatorio entre 1 y 3 segundos.
En las operaciones de ingresar y sacar dinero, la cantidad debe ser aleatoria entre 10 y 1000 euros.
Al finalizar cada operación, el cliente debe consultar el saldo de la cuenta.
Comprueba que, en las operaciones de retirada de dinero, haya suficiente saldo. En caso contrario, muestra un mensaje.
Imprime por consola todos los movimientos para poder controlar que se está haciendo de manera correcta.

1. **CuentaBancaria**
    a. Mantiene el saldo.
    b. Usa un `Semaphore` con 2 permisos para limitar el acceso simultáneo.
    c. Métodos `ingresar`, `retirar` y `consultarSaldo` **adquieren el semáforo** antes de operar y lo liberan después, evitando condiciones de carrera.
2. **Cliente**
    a. Cada cliente es un hilo que realiza **3 operaciones aleatorias** (ingresar o retirar) con cantidades entre 10 y 1000 €.
    b. Cada operación dura entre 1 y 3 segundos.
3. **SimulacionBanco**
    a. Crea 5 clientes que acceden a la misma cuenta compartida.
    b. Se asegura que solo **2 clientes pueden operar a la vez** gracias al semáforo.