Programación de Servicios y Procesos. 2020. Recupera Primera Evaluación.
Simulación Bancaria

Queremos simular el funcionamiento de los clientes que operan en un banco, para ello nuestra aplicación cuenta con las clases:

1. **Banco**, que es la aplicación principal (tiene el main[]) que inicia toda la simulación.
2. **Persona**, que es la clase que simula al cliente y se ejecuta como hilo (tiene el run())
3. **Cuenta**. Controla las operaciones que podemos hacer sobra la cuenta, retirar o ingresar dinero.

Seguidamente se muestran las clases

```
public class Banco {

	public static void main(String[] args) {

		Cuenta cuenta = new Cuenta(2000, 10000);

		Vector<Persona> gente = new Vector<Persona>();

		gente.addElement(new Persona("Juan", 0, cuenta));
		gente.addElement(new Persona("Pepe", 1, cuenta));
		gente.addElement(new Persona("Ana" , 2, cuenta));

		for (int i = 0; i < gente.size(); i++){
			gente.elementAt(i).start();
		}//end-for
	}//end-main

}

public class Persona extends Thread{
	// Propiedades
	private String nombre;
	private int numCliente;
	private Cuenta cuenta;
	private boolean terminar;

	// Constructor
	public Persona(String nombre, int numCliente, Cuenta cuenta){
		this.nombre = nombre;
		this.numCliente = numCliente;
		this.cuenta = cuenta;
	}

	// Métodos
	public void run(){
		while(true){
			cuenta.ingresar(generarCifra(), numCliente, this);
			try{
				sleep(300);
			}
			catch(InterruptedException e){}
			if (terminar == true)
				break;
			cuenta.retirar(generarCifra(), numCliente, this);
			try{
```

```
                        sleep(300);
                    }
                    catch(InterruptedException e){}
                    if (terminar == true)
                        break;
                } //end-while
        } //end-run

        public int generarCifra(){
                return (int) (Math.random()*1000+1);
        }

        public String getNombre(){
                return nombre;
        }

        public void terminar() {
                this.terminar = true;

        }
}

public class Cuenta {
        // Propiedades
        private int saldoActual;
        private int saldoMaximo;

        // Constructor
        public Cuenta(int saldoActual, int saldoMaximo){

                this.saldoActual = saldoActual;
                this.saldoMaximo = saldoMaximo;
        }

        // Metodos
        public void ingresar(int cantidad, int numCliente, Persona ultimoCliente){

                // SECCION CRITICA
                seccionCriticaIngreso(cantidad, ultimoCliente);
                //espera
                try {
                        Thread.sleep(300);
                } catch (InterruptedException e) {}

        } //end-ingresar

        public void retirar(int cantidad, int numCliente, Persona ultimoCliente){

                // SECCION CRITICA
                seccionCriticaRetirada(cantidad, ultimoCliente);
                //espera
                try {
                        Thread.sleep(300);
                } catch (InterruptedException e) {}
        } //end-retirar


        private void seccionCriticaIngreso(int cantidad, Persona ultimoCliente){
                if (saldoActual + cantidad <= saldoMaximo){
                        saldoActual = saldoActual + cantidad;
```

```
                  System.out.println(ultimoCliente.getNombre() + " ha
      ingresado " + cantidad + " euros");
                  System.out.println(ultimoCliente.getNombre() + " Saldo: "+
saldoActual);
            }
            else{
                  System.out.println(ultimoCliente.getNombre() + " ha   superado
el ingreso máximo permitido");
                  System.out.println(ultimoCliente.getNombre() + " Saldo: "+
saldoActual);
                  ultimoCliente.terminar();
            }//end-if
      }

      private void seccionCriticaRetirada(int cantidad, Persona ultimoCliente){
            if (saldoActual - cantidad >= 0){
                  saldoActual = saldoActual + cantidad;
                  System.out.println(ultimoCliente.getNombre() + " ha   retirado
" + cantidad + " euros");
                  System.out.println(ultimoCliente.getNombre() + " Saldo: "+
saldoActual);
            }
            else{
                  System.out.println(ultimoCliente.getNombre() + " ha   superado
la retirada máxima permitido");
                  System.out.println(ultimoCliente.getNombre() + " Saldo: "+
saldoActual);
                  ultimoCliente.terminar();
            } //end-if
      }
}
```

1. Modificar el código para que el padre inicie todos los hilos a la vez (**SYNC** mediante **CyclicBarrier**). Esta barrera llevará **asociada la ejecución de un método** encargado de imprimir que todos los hilos se han creado. Además, **si transcurridos 2 segundos, no se ha levantado la barrera**, los hilos que la hayan alcanzado **se iniciarán automáticamente.**

2. Dado que las operaciones sobre una cuenta son críticas modificar el código añadiendo un mecanismo de sincronización (**SYNC**) para que únicamente se pueda realizar concurrentemente una sola operación sobre la cuente, es decir. **Solo un cliente puede ejecutar una operación de retirada o ingreso concurrentemente**.

3. Modificar el código para que cada cliente, en vez de realizar un ingreso y una retirada en la secuencia prefijada, **decidan si realizan un ingreso o una retirada cada vez de forma aleatoria**.

4. Para **simular clientes muy activos**, que van muchas veces al banco, y **clientes tranquilos**, **modifica la clase Cliente** con el tiempo que transcurrirá entre dos movimientos consecutivos (TIEMPOMIN y TIEMPOMAX).

```
      tiempo = ((int) (Math.random()*(TIEMPOMAX - TIEMPOMIN) + TIEMPOMIN)); //ms
```

5. Añade una **nueva clase Contador** que se ejecuta como hilo, esta clase va a **recibir mensajes de los clientes utilizando un único pipe**. Cada vez que un cliente realiza una operación envía un mensaje por el pipe, así esta clase contará en número total de operaciones realizadas por los clientes.

6. Cuando todos los clientes terminen, la clase Contador deberá **imprimir el número total de operaciones realizadas**, antes de terminar el main.

Funcionamiento de la clase **main**

1. Crear todos los pipes e hilos
2. Despertar a todos los clientes (**SYNC**) para iniciar la simulación
3. Esperar a que todos los clientes terminen (**SYNC**)
5. Notificar al hilo contador que todos los clientes han terminado

El funcionamiento del cliente (**Persona**) es el siguiente:

1. Inicializar variables, pipes o lo que sea necesario
2. Esperar que todos los hilos Persona sean creados (**SYNC**)
3. Simular el comportamiento hasta que el hilo termine. Cada ver determina aleatoriamente un tipo de operación a realizar (ingreso/reintegro) [ojo con la concurrencia de las operaciones], la realiza y la comunica por el pipe al Contador
4. Comunicar al padre que ha terminado

La clase **Contador** realiza:

1. Iniciar variables
2. Esperar información por el pipe, e incrementar el contador de operaciones
3. Cuando el padre se lo comunique (**SYNC**) imprime el número de operaciones y termina

Criterios de evaluación:

1. Las clases se ha implementado adecuadamente, realizan la función sugerida y los parámetros de las clases son coherentes con su funcionalidad. (2 puntos)
2. La sincronización de la simulación del main, Persona y Contador se ha realizado adecuadamente, utilizando las clases de concurrencia que java proporciona. (3 puntos)
3. Los problemas de la sección crítica se han realizado correctamente. (1 punto)
4. La comunicación Personas-Contador se ha realizado atendiendo a las especificaciones propuestas. (2 puntos)

Entrega del ejercicio

El ejercicio deberá ser entregado en un archivo comprimido con el nombre y apellidos y todos los archivos fuente utilizados (dejad el package por defecto)

Seguidamente y a modo de ejemplo se muestran los posibles parámetros de la clase Persona

```
public Persona(String nombre, int numCliente, Cuenta cuenta, CyclicBarrier 
esperaCrearHilos, int tiempoMin, int tiempoMax, PipedWriter emisor, 
CountDownLatch esperaFinClientes)
```