TEMA 1: Programas, Ejecutables, Procesos y Servicios

unir FORMACIÓN PROFESIONAL PROEDUCA

A veces los distintos hilos de un proceso necesitan trabajar cooperativamente para alcanzar un objetivo común.

Varios hilos podrían compartir información accediendo a la misma variable, objeto, chero, etc. Esto podría crear **zonas críticas**, es decir, áreas de código que podrían crear problemas de concurrencia.

Vamos a crear una aplicación que utiliza dos clases, llamadas *Productor* y *Consumidor*.

Ambas clases implementan hilos que acceden a un objeto común, un objeto de la clase *BufferRadios*.

La clase *BufferRadios* contiene una *cola* (objeto *LinkedList*), que se llenará de números enteros que representan valores de los radios de varias circunferencias.

Los objetos de tipo *Productor* se encargarán de colocar elementos en la cola y los de tipo *Consumidor*, de sacar elementos de la cola y utilizarlos para calcular la longitud y el área de la circunferencia.

LinkedList: Estructura y Funcionamiento

Una **LinkedList** es una estructura de datos lineal que consiste en una secuencia de nodos.
Cada nodo contiene dos partes:

- **Dato**: La información que almacena el nodo.
- **Enlace**: Una referencia al siguiente nodo de la lista.

Tipos de LinkedList:

1. **Simplemente enlazada**: Cada nodo se conecta solo al siguiente nodo.
2. **Doblemente enlazada**: Cada nodo se conecta tanto al siguiente como al nodo anterior.
3. **Circular**: El último nodo se conecta al primer nodo, formando un ciclo.

Operaciones comunes:

- **Inserción**: Añadir un nodo (al principio, al final o en una posición específica).
- **Eliminación**: Remover un nodo de la lista.
- **Búsqueda**: Encontrar un nodo específico.
- **Recorrido**: Visitar cada nodo desde el primero hasta el último.

Ventajas:

1. **Memoria dinámica**: No requiere una cantidad fija de espacio en memoria.
2. **Inserción/Eliminación eficiente**: Las operaciones son rápidas comparadas con los arrays, ya que no necesitan mover todos los elementos.

Desventajas:

1. **Acceso secuencial**: No se puede acceder a un elemento directamente, es necesario recorrer la lista.

```
import java.util.LinkedList;
...
cola = new LinkedList<Integer>();
```

1. **add(E element)**
   - **Descripción**: Añade un elemento al final de la lista.
   - **Ejemplo**: list.add("A");

2.add(int index, E element)
   - **Descripción**: Inserta un elemento en una posición específica.
   - **Ejemplo**: list.add(1, "B");

3.get(int index)
   - **Descripción**: Retorna el elemento en la posición especificada.
   - **Ejemplo**: list.get(0);

4.remove(int index)
   - **Descripción**: Elimina el elemento en la posición indicada.
   - **Ejemplo**: list.remove(1);

5.remove(Object o)
   - **Descripción**: Elimina la primera aparición de un elemento específico.
   - **Ejemplo**: list.remove("A");

6.size()
   - **Descripción**: Retorna el número de elementos en la lista.
   - **Ejemplo**: list.size();

7.isEmpty()
   - **Descripción**: Verifica si la lista está vacía.
   - **Ejemplo**: list.isEmpty();

8.clear()
   - **Descripción**: Elimina todos los elementos de la lista.
   - **Ejemplo**: list.clear();

9.contains(Object o)
   - **Descripción**: Verifica si la lista contiene el elemento especificado.
   - **Ejemplo**: list.contains("A");

10.indexOf(Object o)
   - **Descripción**: Retorna el índice de la primera aparición del elemento.
   - **Ejemplo**: list.indexOf("B");

Queue es una **interfaz** en Java que representa una **estructura de datos en cola** (FIFO: First In, First Out). Los elementos que se agregan primero son los primeros en salir.

```
import java.util.Queue;
.....
private Queue<Integer> cola;
```

```
import java.util.LinkedList;
import java.util.Queue;

public class BufferRadios {
	private Queue<Integer> cola;

	public BufferRadios() {
		cola = new LinkedList<Integer>();
	}

	public void poner(Integer r) {
		cola.add(r);
	}

	public Integer sacar() {
		if (cola.isEmpty()) {
			return null;
		}
		else {
			Integer radio = cola.remove();
			return radio;
		}
	}
}
```

La clase *Productor*

Implementa un hilo que actúa como productor de radios de circunferencias, generando números al azar y colocándolos en un objeto de tipo *BfferRadios*, llamado *buffer*, que será compartido por un consumidor.

```
public class Productor implements Runnable
{
   private BufferRadios buffer;
   private static int contador = 0;
   private Thread hilo;

   public Productor(BufferRadios buffer)
   {
      contador ++;
      this.buffer = buffer;
      hilo = new Thread(this, "Productor"+contador);
      hilo.start();
   }

   @Override
   public void run()
   {
      int radio;
      radio = (int) (Math.random()*30+1);
      System.out.println(hilo.getName() + " va a generar un radio = " +
radio);
      buffer.poner(radio);
   }
}
```

private static int contador = 0; pertenece a la clase no a las instancias

contador++; en el constructor, se incrementa en 1

hilo = new Thread(this, "Productor" + contador); diferenciamos diferentes instancias de Productor

```
public class Consumidor implements Runnable {
     private Thread hilo;
     private BufferRadios buffer;
     private static int contador = 0;

     public Consumidor(BufferRadios buffer) {
          contador++;
          hilo = new Thread(this, "Consumidor" + contador);
          this.buffer = buffer;
          hilo.start();
     }

     @Override
     public void run() {
          consumirUnRadio();
     }

     public void consumirUnRadio() {
          Integer radio = buffer.sacar();
          if (radio != null) {
               double l = 2 * Math.PI * radio;
               double a = Math.PI * radio * radio;
               System.out.println("Radio = " + radio + ", Longitud = " + l
+ ", Area = " + a);
          }
          else {
               System.out.println("Cola vacía");
          }
     }
}
```

La clase *Principal* es la encargada de lanzar los hilos de ejecución, creando objetos de tipo *Productor* o *Consumidor*. Aquí se ve bien claro cómo los hilos productores y consumidores comparten el mismo objeto *buffer*.

```
public class Principal {
        public static void main(String[] args) {
                BufferRadios buffer = new BufferRadios();
                new Productor(buffer);
                new Consumidor(buffer);
                new Productor(buffer);
                new Consumidor(buffer);
                new Productor(buffer);
                new Consumidor(buffer);
        }
}
```

Se supone que deseamos producir tres radios de circunferencia y consumir los tres. Sin embargo, el resultado de la ejecución es imprevisible. Ejecuta varias veces y comprueba los resultados.

```
Cola vacía
Productor2 va a generar un radio = 22
Productor1 va a generar un radio = 4
Cola vacía
Productor3 va a generar un radio = 26
Consumidor3 - Radio = 4, Longitud = 25.132741228718345, Area = 50.26548245743669
```

Se han producido tres radios, pero solo se ha podido consumir uno de ellos, el motivo es que los consumidores se han adelantado a los productores, intentando consumir antes de que los radios hayan sido colocados en el *buffer*.

```
Productor1 va a generar un radio = 12
Productor2 va a generar un radio = 29
Productor3 va a generar un radio = 22
Consumidor1 - Radio = 12, Longitud = 75.39822368615503, Area = 452.3893421169302
Consumidor2 - Radio = 29, Longitud = 182.212373908208, Area = 2642.079421669016
Consumidor3 - Radio = 22, Longitud = 138.23007675795088, Area = 1520.5308443374597
```

se han consumido todos los radios producidos

```
Productor1 va a generar un radio = 11
Cola vacía
Productor2 va a generar un radio = 14
Productor3 va a generar un radio = 23
Consumidor3 - Radio = 14, Longitud = 87.96459430051421, Area = 615.7521601035994
Consumidor2 - Radio = 11, Longitud = 69.11503837897544, Area = 380.1327110843649
```

Se ha perdido uno de los radios producidos.

Incluso si lo ejecutas muchas veces, terminará produciéndose una excepción

```
Productor1 va a generar un radio = 24
Cola vacía
Productor2 va a generar un radio = 28
Productor3 va a generar un radio = 13
Exception in thread "Consumidor3" java.util.NoSuchElementException
        at java.util.LinkedList.removeFirst(LinkedList.java:270)
        at java.util.LinkedList.remove(LinkedList.java:685)
        at BufferRadios.sacar(BufferRadios.java:20)
        at Consumidor.consumirUnRadio(Consumidor.java:19)
        at Consumidor.run(Consumidor.java:15)
        at java.lang.Thread.run(Thread.java:745)
Consumidor2 - Radio = 28, Longitud = 175.92918860102841, Area = 2463.0086404143976
```

La excepción la ha provocado el método *sacar()* de la clase *BufferRadios*, estamos intentando ejecutar un método *remove()* sobre una cola vacía.

No lo teníamos controlado con la expresión condicional *cola.isEmpty()*?

```
public Integer sacar() {
     if (cola.isEmpty()) {
          return null;
     }
     else {
          Integer radio = cola.remove();
          return radio;
     }
}
```

Estamos trabajando con varios hilos de ejecución simultáneos. Seguro que la cola no estaba vacía en el momento de evaluarse el *if*, pero luego, antes de sacar el elemento con el método *remove()*, otro hilo se adelantó, colocándose en medio y sacando el último elemento que quedaba.

zona crítica!!

Cómo sincronizar hilos?

Métodos *wait()* y *notify()*

Los métodos *wait()* y *notify()* pertenecen a la superclase *Object*, por consiguiente, todos los objetos cuentan con ellos.

El método *wait()* deja bloqueado el hilo que lo llama, hasta que es liberado por otro hilo por medio de la ejecución del método *notify()* o cuando han transcurrido el número de milisegundos especificados. En nuestro ejemplo, mientras la cola esté vacía, bloquearemos repetitivamente la ejecución del método *sacar()* hasta que sea liberado por la ejecución del método *notify()*, llamado desde *poner()*, o hasta que transcurra un segundo.

La utilización de los métodos *wait()* y *notify()* requiere de métodos marcados con el modificador *synchronized*.

El modificador *synchronized*

Cuando un método o bloque de código está marcado como **synchronized**, se asegura de que solo un hilo pueda **obtener el monitor** o **bloqueo** (lock) asociado al objeto y ejecutar ese código a la vez. Otros hilos que intenten acceder a la misma sección de código **esperarán** hasta que el hilo actual libere el bloqueo.

Cuando un método se marca con synchronized, el bloqueo se aplica al objeto al que pertenece el método. Si otro hilo intenta acceder a cualquier otro método sincronizado del mismo objeto, tendrá que esperar a que el primer hilo termine.

```
public class Contador
{
    private int count = 0;

    public synchronized void incrementar()
    {
        count++;
    }

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

Synchronized()

- En lugar de sincronizar un método completo, también puedes sincronizar un **bloque de código** específico dentro de un método. Esto permite que otras partes del método sean ejecutadas por otros hilos, pero la parte crítica (el bloque sincronizado) esté protegida.

```
public class Contador
{
    private int count = 0;

    public void incrementar()
    {
         synchronized(this)
         {
             count++;
         }
     }

    public int obtenerValor()
    {
        synchronized(this)
        {
            return count;
        }
    }
}
```

Synchronized() método vs bloque de codigo

Cuando un método se declara como synchronized, **toda la ejecución del método** está sincronizada sobre el objeto que lo invoca.

Esto significa que **todo el cuerpo del método** estará protegido por un monitor (bloqueo), y solo un hilo podrá acceder a ese método a la vez para cualquier instancia particular del objeto.

Cuando sincronizas un bloque de código dentro de un método (en lugar de todo el método), puedes controlar más finamente **qué parte del código** está sincronizada.

Esto significa que solo la sección del código que realmente necesita protección estará bajo el bloqueo.

¿Qué es el monitor?

La clave de la sincronización está en la palabra *monitor*. Como hemos comentado anteriormente, sólo un hilo de ejecución puede acceder a un método sincronizado al mismo tiempo;

Se dice que ese hilo es el que **tiene el monitor y tendrá bloqueado el proceso hasta que finalice su ejecución y lo libere.**

Cuando marcas un método o bloque de código como synchronized, estás trabajando con el **monitor** (o **bloqueo**) de un objeto.
Solo un hilo puede tener el control del monitor de un objeto a la vez.
Otros hilos que intenten ejecutar código sincronizado en el mismo objeto **esperarán** hasta que el hilo actual libere el bloqueo.

- **synchronized(this)**: El bloqueo se asocia con la instancia actual del objeto (es decir, el objeto en el que se está ejecutando el método).

- **synchronized(otroObjeto)**: El bloqueo se asocia con el objeto pasado, y el control del bloqueo es sobre ese objeto específico.

wait()

El método wait() se usa para **poner en espera un hilo** hasta que otro hilo lo notifique de que puede continuar. Este método debe ser llamado dentro de un bloque sincronizado para que funcione correctamente, ya que está estrechamente ligado al sistema de monitores de Java.

Funcionalidad:

•Un hilo que llama a wait() **libera el bloqueo (monitor)** que tenía sobre el objeto y **entra en un estado de espera**. Esto significa que el hilo está suspendido y no continuará su ejecución hasta que sea notificado.

•Solo puede ser llamado desde un bloque **sincronizado** que está sincronizado sobre el mismo objeto cuyo monitor se ha adquirido.

```
synchronized (objetoCompartido) { while (condiciónNoSatisfecha)
{
        objetoCompartido.wait(); // El hilo actual se suspende y espera hasta que sea notificado.
}
// Código que se ejecuta después de que el hilo haya sido notificado. }
```

notify()

El método notify() se usa para **despertar un hilo que está en espera** en el mismo monitor (objeto compartido) al que se llamó wait(). Si hay múltiples hilos esperando en ese monitor, uno de ellos será despertado de forma aleatoria (no necesariamente el que fue suspendido primero).

Funcionalidad:

•Un hilo que llama a notify() no libera el bloqueo inmediatamente. El hilo que fue notificado tendrá que esperar hasta que el hilo que llamó a notify() **libere el bloqueo** (al salir del bloque sincronizado).

•Despierta solo **un hilo** que esté esperando en el monitor asociado al objeto. Si hay varios hilos en espera, no se garantiza cuál será notificado.

```
synchronized (objetoCompartido)
{ // Realiza alguna operación y notifica a un hilo en espera.
        objetoCompartido.notify();
}
```

notifyAll()

Además de notify(), existe también el método notifyAll(), que **despierta todos los hilos** que están en espera en el monitor asociado. Aunque todos los hilos son notificados, solo uno podrá continuar inmediatamente, ya que el bloqueo del objeto todavía estará retenido por el hilo que llamó a notifyAll(). Los demás hilos esperarán a que se libere el monitor.

```
synchronized (objetoCompartido)
{ // Notifica a todos los hilos que están esperando en este objeto.
        objetoCompartido.notifyAll();
}
```

Actuaremos sobre la clase *BufferRadios*, que es la que contiene el método *sacar()* que constituye la zona crítica.

```
import java.util.LinkedList;
import java.util.Queue;
public class BufferRadios
{
   private Queue<Integer> cola;
   public BufferRadios()
   {
      cola = new LinkedList<Integer>();
   }
   public synchronized void poner(Integer r)
   {
      cola.add(r);
      notify();
   }
```

```
   public synchronized Integer sacar()
   {
      while (cola.isEmpty())
      {
         try
         {
            wait();
         }
         catch (InterruptedException e)
         {
            System.out.println(e.getMessage());
         }
      }
      if (!cola.isEmpty())
      {
         Integer radio = cola.remove();
         return radio;
      }
      else
      {
         return null;
      }
   }
}
```

Observa cómo el método *sacar()* ejecuta repetitivamente un *wait()* mientras la cola de radios está vacía.

Cada espera será de un segundo (1000 milisegundos) o menos.

Ahora mismo nuestro programa funciona perfectamente, pero
.....

si se construyeran solo dos objetos *Productor* y 3 objetos *Consumidor*, el último *Consumidor* entraría en un bucle infinito dentro de la estructura mientras está esperando a que el método *poner()* coloque un radio.

Esto podría solucionarse colocando un contador para limitar el número de repeticiones del *while*.

Vamos a cambiar el código para que solo se admita un máximo de veinte repeticiones.

```
import java.util.LinkedList;
import java.util.Queue;

public class BufferRadios {
	private Queue<Integer> cola;

	public BufferRadios() {
		cola = new LinkedList<Integer>();
	}

	public synchronized void poner(Integer r) {
		cola.add(r);
		notify();
	}

	public synchronized Integer sacar() {
		int esperas = 0;
		while (cola.isEmpty() && esperas<20) {
			esperas ++;
			try {
				wait(10);
			} catch (InterruptedException e) {
				System.out.println(e.getMessage());
			}
		}
		if (esperas<20) {
			Integer radio = cola.remove();
			return radio;
		} else {
			return null;
		}
	}
}
```

```
public class Consumidor implements Runnable {
	private Thread hilo;
	private BufferRadios buffer;
	private static int contador = 0;

	public Consumidor(BufferRadios buffer) {
		contador++;
		hilo = new Thread(this, "Consumidor" + contador);
		this.buffer = buffer;
		hilo.start();
	}

	@Override
	public void run() {
		consumirUnRadio();
	}

	public void consumirUnRadio() {
		Integer radio = buffer.sacar();
		if (radio != null) {
			double l = 2 * Math.PI * radio;
			double a = Math.PI * radio * radio;
			System.out.println(hilo.getName() + " - Radio = " + radio + ", Longitud
= " + l + ", Area = " + a);
		}
		else {
			System.out.println("Agotado tiempo de espera y no hay más elementos que
consumir");
		}
	}
}
```

Si transcurridas las veinte repeticiones todavía sigue vacía la lista, se dejará de esperar a que se produzcan más radios y nalizará el método *sacar()*, devolviendo el valor *null*.

https://create.kahoot.it/details/3167681c-b8a7-4218-8fcd-84a4c40c83b2

ACTIVIDAD OPTATIVA UF 1.2

- **Enunciado**:
- Esta práctica consiste en el desarrollo de un programa que simulará la gestión de un centro de exámenes, donde podrán examinarse varios alumnos. El programa lanzará varios hilos de ejecución, cada uno de los cuales representa un alumno que se está examinando.

```
public class Principal
{
    public static void main(String[] args) throws InterruptedException
    {
            BufferExamenes generador = new BufferExamenes();
            new ProductorExamenes(generador);
            new Examinador("Rosa", generador);
            new ProductorExamenes(generador);
            new Examinador("Miguel", generador);
            new ProductorExamenes(generador);
            new Examinador("Carlos", generador);
     }
}
```

Escribe la clase *BufferExamenes en base al contenido inicial facilitado:*

Para cada alumno que va a examinarse se debe imprimir un examen, que tendrá un código diferenciado. La clase *BufferExamenes* mantiene una cola (objeto *LinkedList*) de códigos de examen. Para cada alumno se extrae un examen de la cola.

```
import java.util.LinkedList;
import java.util.Queue;

public class BufferExamenes {
private Queue<String> colaExamenes;

public BufferExamenes() {
colaExamenes = new LinkedList<String>();
}

public synchronized void fabricarNuevoExamen(String
codigo) {
// Aquí se fabrica un nuevo examen.
// Un hilo de la clase ProductorExamenes
// se encargará de fabricarlo
// y pasarlo como argumento a este método.
}
```

```
// Añade el código pasado como argumento a la
//cola y libera el hilo que está intentando
consumir un nuevo examen.

public synchronized String consumirExamen() {
// Este método se encargará de suministrar un
//examen a cada hilo de tipo Examinador que lo
//solicite.

// Para suministrar el examen habrá antes que
//esperar hasta que haya algún examen para
//consumir en la cola.

// Haz aquí una pausa hasta que se haya fabricado
//algún examen.

// Si la cola sigue sin estar vacía, saca un examen y
// entrégalo como retorno de esta función.
}
}
```

La clase *ProductorExamenes* se encargará de generar exámenes, asignándole a cada uno un código que estará formado por la letra E seguida del número de examen, un guión y el año, por ejemplo: "E2-2024 (segundo examen del año 2024)." El número de examen se establece a partir de la variable estática *numeroExamen*.

```
import java.time.LocalDateTime;
public class ProductorExamenes implements
Runnable {
private BufferExamenes buffer;
private static int numeroExamen = 0;
private Thread hilo;

public ProductorExamenes(BufferExamenes
buffer) {
// Incrementa el contador de exámenes
(variable numeroExamen).

// Construye el hilo. El nombre será la letra E
seguida
// del valor de la variable numeroExamen.

// Establece el valor de la propiedad buffer

// Inicia el hilo.
}
```

```
@Override
public void run() {
int aa = LocalDateTime.now().getYear();
// Generación del código de examen.
String codigo = this.hilo.getName() + "-" +aa;

// Añade el nuevo código al buffer de exámenes.

// Muestra un mensaje en consola informando sobre el
// código del examen que se acaba de producir.
}
}
```

La clase *Examinado* se lanza cada vez que un alumno va a examinarse, simulando la realización del examen por parte del alumno. Un ejemplo de salida de examen podría ser:

E2-2024;Miguel; Pregunta 1;C
E2-2024;Miguel; Pregunta 2;-
E2-2024;Miguel; Pregunta 3;A
E2-2024;Miguel; Pregunta 4;C
E2-2024;Miguel; Pregunta 5;D
E2-2024;Miguel; Pregunta 6;-
E2-2024;Miguel; Pregunta 7;-
E2-2024;Miguel; Pregunta 8;B
E2-2024;Miguel; Pregunta 9;D
E2-2024;Miguel; Pregunta 10;-

```
public class Examinado implements Runnable {
private Thread hilo;
BufferExamenes buffer;

public Thread getHilo() {
return hilo;
}
public Examinado(String alumno, BufferExamenes
generador) {
// Construye el hilo. El nombre será el nombre del
alumno.

// Establece el valor de la propiedad buffer

// Inicia el hilo.

}
```

```
@Override
public synchronized void run() {
String codigoExamen = this.buffer.consumirExamen();
if (codigoExamen != null) {
// Simula aquí un examen de 10 preguntas
// cuyas respuestas se seleccionarán al azar
// entre A, B, C, D o – (sin contestar).
}
else {
System.out.println("Agotado tiempo de espera y " +
"no hay más exámenes");
}
}
}
```

La ejecución completa del programa debe ofrecer una salida similar a esta:

```
Producido examen E2-2024
E1-2024;Carlos; Pregunta 1;B
E1-2024;Carlos; Pregunta 2;B
E2-2024;Rosa; Pregunta 1;A
E2-2024;Rosa; Pregunta 2;A
Producido examen E3-2024
E2-2024;Rosa; Pregunta 3;C
E2-2024;Rosa; Pregunta 4;-
E3-2024;Miguel; Pregunta 1;C
Producido examen E1-2024
E1-2024;Carlos; Pregunta 3;-
E1-2024;Carlos; Pregunta 4;D
E3-2024;Miguel; Pregunta 2;-
E2-2024;Rosa; Pregunta 5;D
E3-2024;Miguel; Pregunta 3;D
E1-2024;Carlos; Pregunta 5;C
E3-2024;Miguel; Pregunta 4; B
E2-2024;Rosa; Pregunta 6;-
E3-2024;Miguel; Pregunta 5;B
E1-2024;Carlos; Pregunta 6;D
E3-2024;Miguel; Pregunta 6;C
E2-2024;Rosa; Pregunta 7;-
E3-2024;Miguel; Pregunta 7;-
E1-2024;Carlos; Pregunta 7;C
E3-2024;Miguel; Pregunta 8;B
E2-2024;Rosa; Pregunta 8;A
E3-2024;Miguel; Pregunta 9;C
E1-2024;Carlos; Pregunta 8;D
E3-2024;Miguel; Pregunta 10;A
E2-2024;Rosa; Pregunta 9;C
E2-2024;Rosa; Pregunta 10;A
E1-2024;Carlos; Pregunta 9;B
E1-2024;Carlos; Pregunta 10;-
```

MUCHAS GRACIAS POR VUESTRA ATENCIÓN!