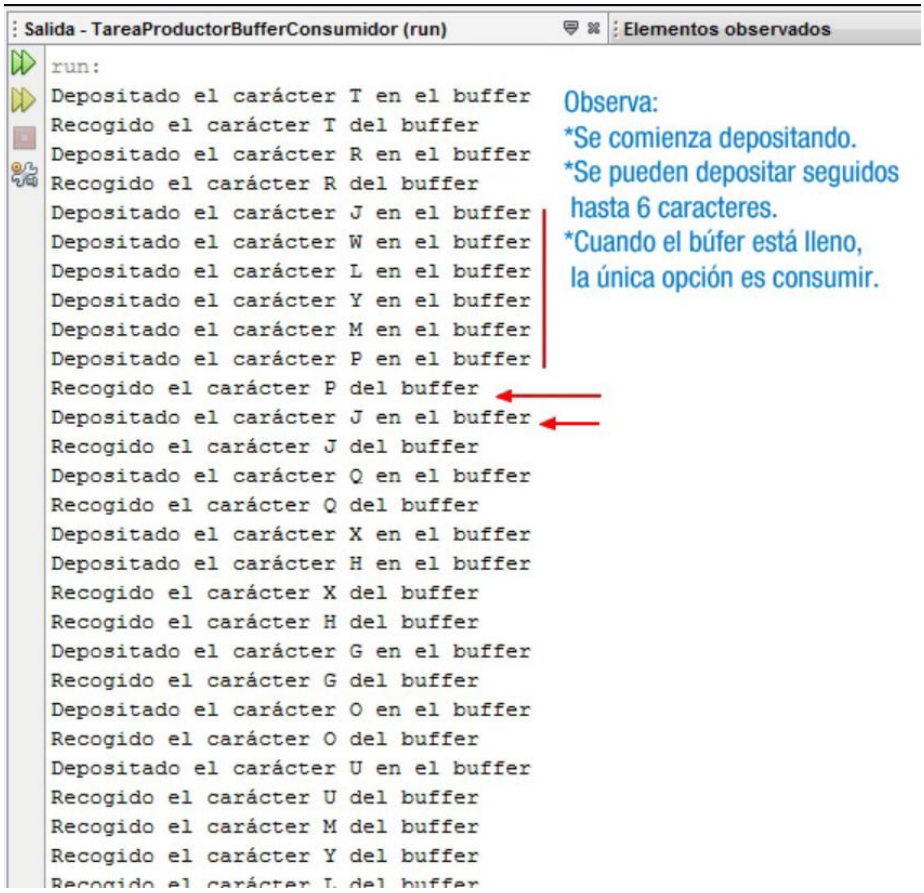
Tarea 1

De igual manera a lo visto en el tema, ahora te proponemos un ejercicio del tipo productor-consumidor que mediante un hilo productor almacene datos (15 caracteres) en un búfer compartido, de donde los debe recoger un hilo consumidor (consume 15 caracteres). La capacidad del búfer ahora es de 6 caracteres, de manera que el consumidor podrá estar cogiendo caracteres del búfer siempre que éste no esté vacío. El productor sólo podrá poner caracteres en el búfer, cuando esté vacío o haya espacio.

Te mostramos una posible salida del programa que debes realizar

Ten en cuenta que ahora el problema del productor-consumidor utiliza un búfer de tamaño 6, que lo puedes implementar mediante un array. La forma de ir introduciendo caracteres será de izquierda a derecha y se consumirán de derecha a izquierda (el último que se produzca será el primero en consumirse).



Clase Buffer

```java
package com.mycompany.psp02_tarea1;

import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Clase Buffer que implementa un buffer circular de caracteres con un tamaño fijo.
 * Proporciona métodos sincronizados para producir y consumir caracteres,
 * permitiendo la comunicación entre hilos productores y consumidores.
 */
public class Buffer {

    /**
     * Array de caracteres que actúa como el buffer.
     */
    private char[] buffer;

    /**
     * Índice actual para la posición de inserción/consumo en el buffer.
     */
    private int posicion;

    /**
     * Indica si el buffer está lleno.
     */
    private boolean estaLleno;

    /**
     * Indica si el buffer está vacío.
     */
    private boolean estaVacio;

    /**
     * Constructor que inicializa el buffer con un tamaño dado.
     *
     * @param tamanio Tamaño del buffer.
     */
    public Buffer(int tamanio) {
        this.buffer = new char[tamanio];
        this.posicion = 0;
        this.estaLleno = false;
        this.estaVacio = true;
    }
```

```java
/**
 * Método sincronizado que permite a un hilo productor agregar un carácter al buffer.
 * Si el buffer está lleno, el hilo se bloquea hasta que haya espacio disponible.
 *
 * @param c Carácter a añadir al buffer.
 */
public synchronized void producir(char c) {

    // Esperar si el buffer está lleno
    while (this.estaLleno) {
        try {
            wait();
        } catch (InterruptedException ex) {
            Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Añadir el carácter al buffer y actualizar la posición
    this.buffer[this.posicion] = c;
    this.posicion++;
    this.estaVacio = false;

    // Marcar el buffer como lleno si hemos alcanzado su capacidad
    if (this.buffer.length == this.posicion) {
        this.estaLleno = true;
    }

    // Notificar a todos los hilos que están esperando
    notifyAll();
}
```

Clase Productor

```java
package com.mycompany.psp02_tarea1;

import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * La clase Productor representa un hilo que produce caracteres aleatorios y los
 * deposita en un buffer compartido. Hereda de la clase Thread para ejecutarse
 * de manera concurrente.
 */
public class Productor extends Thread {

    /**
     * Buffer compartido donde el productor depositará los caracteres.
     */
    private Buffer buffer;

    /**
     * Conjunto de letras que el productor puede generar aleatoriamente.
     */
    private final String letras = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    /**
     * Contador de caracteres producidos por este hilo.
     */
    private int producido;

    /**
     * Número máximo de caracteres que el productor generará.
     */
    private int maximo = 15;

    /**
     * Constructor que inicializa el productor con un buffer compartido.
     *
     * @param buffer El buffer donde se depositarán los caracteres producidos.
     */
    public Productor(Buffer buffer) {
        this.buffer = buffer;
        this.producido = 0;
    }
```

```java
/**
 * Método que ejecuta el hilo. Genera caracteres aleatorios y los deposita en
 * el buffer hasta alcanzar el máximo permitido.
 */
@Override
public void run() {
    while (producido < maximo) {
        try {
            // Generar un carácter aleatorio de la cadena 'letras'
            char letra = letras.charAt((int) (Math.random() * letras.length()));

            // Depositar el carácter en el buffer
            buffer.producir(letra);
            producido++;

            // Imprimir el carácter depositado
            System.out.println("Depositado el caracter " + letra + " en el buffer.");

            // Pausa aleatoria antes de producir el próximo carácter
            sleep((long) (Math.random() * 4000));

        } catch (InterruptedException ex) {
            Logger.getLogger(Productor.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Clase Consumidor

```java
package com.mycompany.psp02_tarea1;

import static java.lang.Thread.sleep;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * La clase Consumidor representa un hilo que consume caracteres de un buffer
 * compartido. Hereda de la clase Thread para ejecutarse de manera concurrente.
 */
public class Consumidor extends Thread {

    /**
     * Buffer compartido desde donde el consumidor retirará los caracteres.
     */
    private Buffer buffer;

    /**
     * Contador de caracteres consumidos por este hilo.
     */
    private int consumido;

    /**
     * Número máximo de caracteres que el consumidor extraerá del buffer.
     */
    private int maximo = 15;

    /**
     * Constructor que inicializa el consumidor con un buffer compartido.
     *
     * @param buffer El buffer desde donde se consumirán los caracteres.
     */
    public Consumidor(Buffer buffer) {
        this.buffer = buffer;
        this.consumido = 0;
    }

    /**
     * Método que ejecuta el hilo. Extrae caracteres del buffer hasta alcanzar el
     * número máximo permitido.
     */
    @Override
    public void run() {
        while (consumido < maximo) {
            try {
                // Extraer un carácter del buffer
                char letra = buffer.consumir();

                // Imprimir el carácter consumido
                System.out.println("Consumido el caracter " + letra + " del buffer.");
                consumido++;

                // Pausa aleatoria antes de consumir el próximo carácter
                sleep((long) (Math.random() * 4000));

            } catch (InterruptedException ex) {
                Logger.getLogger(Consumidor.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

Clase Buffer

```java
package com.mycompany.psp02_tarea1;

import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * Clase Buffer que implementa un buffer circular de caracteres con un tamaño fijo.
 * Proporciona métodos sincronizados para producir y consumir caracteres,
 * permitiendo la comunicación entre hilos productores y consumidores.
 */
public class Buffer {

    /**
     * Array de caracteres que actúa como el buffer.
     */
    private char[] buffer;

    /**
     * Índice actual para la posición de inserción/consumo en el buffer.
     */
    private int posicion;

    /**
     * Indica si el buffer está lleno.
     */
    private boolean estaLleno;

    /**
     * Indica si el buffer está vacío.
     */
    private boolean estaVacio;

    /**
     * Constructor que inicializa el buffer con un tamaño dado.
     *
     * @param tamanio Tamaño del buffer.
     */
    public Buffer(int tamanio) {
        this.buffer = new char[tamanio];
        this.posicion = 0;
        this.estaLleno = false;
        this.estaVacio = true;
    }
```

```java
/**
 * Método sincronizado que permite a un hilo productor agregar un carácter al buffer.
 * Si el buffer está lleno, el hilo se bloquea hasta que haya espacio disponible.
 *
 * @param c Carácter a añadir al buffer.
 */
public synchronized void producir(char c) {

    // Esperar si el buffer está lleno
    while (this.estaLleno) {
        try {
            wait();
        } catch (InterruptedException ex) {
            Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Añadir el carácter al buffer y actualizar la posición
    this.buffer[this.posicion] = c;
    this.posicion++;
    this.estaVacio = false;

    // Marcar el buffer como lleno si hemos alcanzado su capacidad
    if (this.buffer.length == this.posicion) {
        this.estaLleno = true;
    }

    // Notificar a todos los hilos que están esperando
    notifyAll();
}
```

```java
/**
 * Método sincronizado que permite a un hilo consumidor retirar un carácter del buffer.
 * Si el buffer está vacío, el hilo se bloquea hasta que haya elementos disponibles.
 *
 * @return El carácter retirado del buffer.
 */
public synchronized char consumir() {

    // Esperar si el buffer está vacío
    while (this.estaVacio) {
        try {
            wait();
        } catch (InterruptedException ex) {
            Logger.getLogger(Buffer.class.getName()).log(Level.SEVERE, null, ex);
        }
    }

    // Reducir la posición y obtener el carácter correspondiente
    this.posicion--;
    this.estaLleno = false;

    // Marcar el buffer como vacío si no quedan elementos
    if (this.posicion == 0) {
        this.estaVacio = true;
    }

    // Notificar a todos los hilos que están esperando
    notifyAll();

    // Devolver el carácter consumido
    return this.buffer[posicion];
}
```

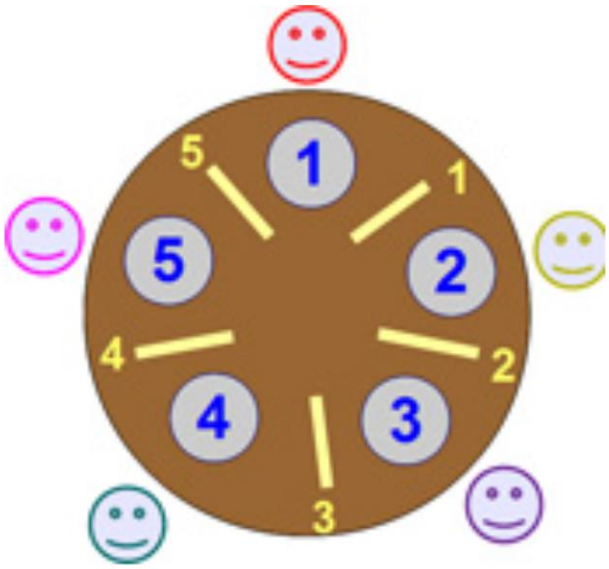
Clase Main

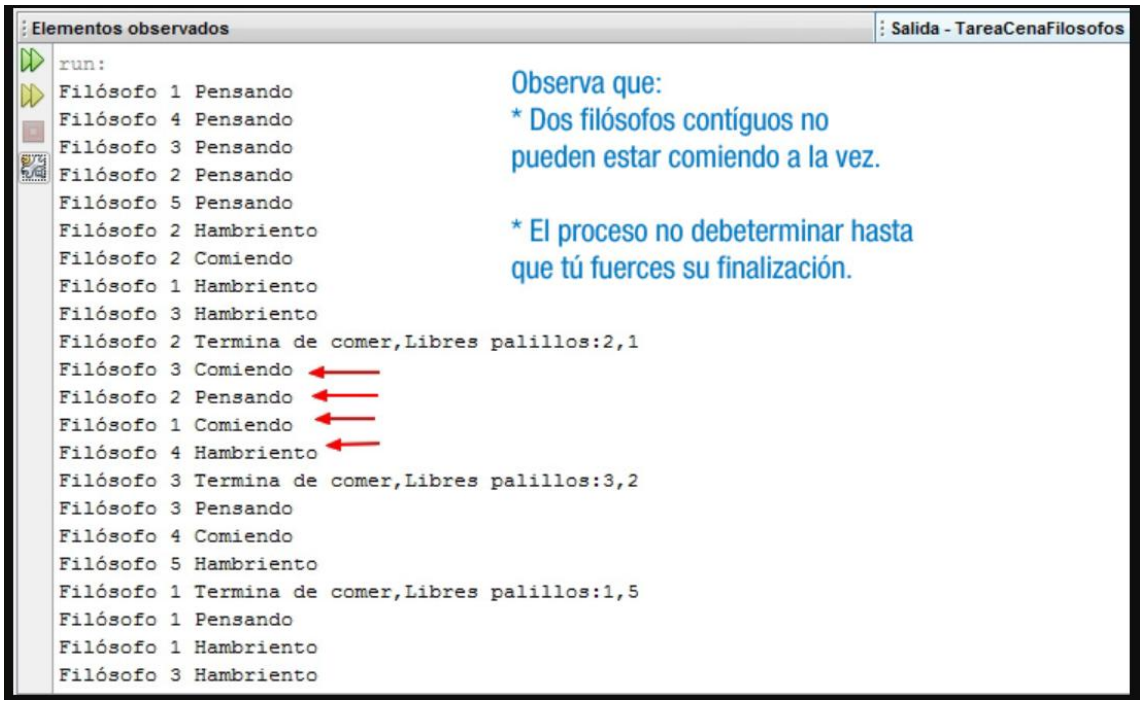
```java
package com.mycompany.psp02_tarea1;

import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * La clase principal inicia el programa productor-consumidor.
 * Crea un buffer compartido y lanza un hilo productor y un hilo consumidor.
 */

public class PSP02_Tarea1 {

    public static void main(String[] args) {

        try {

            // Crear un buffer con una capacidad de 6 elementos
            Buffer b = new Buffer(6);
            // Crear instancias de productor y consumidor que comparten el mismo buffer
            Productor p = new Productor(b);
            Consumidor c = new Consumidor(b);
            // Iniciar el hilo productor
            p.start();
            // Esperar 2 segundos antes de iniciar el hilo consumidor
            Thread.sleep(2000);
            // Iniciar el hilo productor
            c.start();
            // Esperar a que ambos hilos terminen su ejecución
            p.join();
            c.join();
            System.out.println("Fin del proceso");

        } catch (InterruptedException ex) {
            Logger.getLogger(PSP02_Tarea1.class.getName()).log(Level.SEVERE, null, ex);
        }

    }

}
```

Ejercicio 2

De igual manera a lo visto en el tema, ahora te proponemos que resuelvas el clásico problema denominado "La cena de los filósofos" utilizando la clase Semaphore del paquete java.util.concurrent.

El problema es el siguiente: Cinco filósofos se sientan alrededor de una mesa y pasan su vida comiendo y pensando. Cada filósofo tiene un plato de arroz chino y un palillo a la izquierda de su plato. Cuando un filósofo quiere comer arroz, cogerá los dos palillos de cada lado del plato y comerá. El problema es el siguiente: establecer un ritual (algoritmo) que permita comer a los filósofos. El algoritmo debe satisfacer la exclusión mutua (dos filósofos no pueden emplear el mismo palillo a la vez), además de evitar el interbloqueo y la inanición.





Clase Filósofo

```java
package com.mycompany.psp02_tarea2;

import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * La clase Filosofo representa a un filósofo en el problema de los filósofos cenando.
 * Cada filósofo alterna entre los estados de pensando y comiendo.
 * Para comer, un filósofo debe tomar dos palillos (el de su izquierda y el de su derecha).
 * Esta clase extiende la clase Thread para ejecutarse de forma concurrente.
 */
public class Filosofo extends Thread {

    /**
     * La mesa compartida con los palillos que los filósofos comparten.
     */
    private Mesa mesa;

    /**
     * El número que identifica al filósofo (comienza en 1).
     */
    private int comensal;

    /**
     * La posición del filósofo en la mesa (índice del array de palillos).
     */
    private int posicionMesa;

    /**
     * Constructor que inicializa un nuevo filósofo.
     *
     * @param m        La instancia de Mesa que contiene los palillos.
     * @param comensal El número que identifica al filósofo (comienza en 1).
     */
    public Filosofo(Mesa m, int comensal) {
        this.mesa = m;
        this.comensal = comensal;
        this.posicionMesa = comensal - 1; // Convertir a índice basado en 0
    }
```

```java
/**
 * Simula el estado de pensamiento del filósofo.
 * El filósofo estará pensando durante 2 segundos.
 */
public void pensando() {
    try {
        Thread.sleep(2000); // Simular el tiempo de pensar durante 2000 milisegundos.
    } catch (InterruptedException ex) {
        Logger.getLogger(Filosofo.class.getName()).log(Level.SEVERE, null, ex);
    }
    System.out.println("Filosofo " + this.comensal + " pensando...");
}

/**
 * Simula el estado de comer del filósofo.
 * El filósofo estará comiendo durante 2 segundos.
 */
public void comiendo() {
    try {
        Thread.sleep(2000); // Simular el tiempo de comida durnate 2000 milisegundos.
    } catch (InterruptedException ex) {
        Logger.getLogger(Filosofo.class.getName()).log(Level.SEVERE, null, ex);
    }
    System.out.println("Filosofo " + this.comensal + " comiendo...");
}
```

```java
/**
 * Método principal que ejecuta el ciclo de vida del filósofo.
 * El filósofo alterna entre pensar y comer indefinidamente.
 */
@Override
public void run() {
    while (true) {
        try {
            // El filósofo piensa durante un tiempo aleatorio
            this.pensando();

            System.out.println("Filosofo " + this.comensal + " Hambriento");

            // El filósofo intenta coger los palillos para comer
            mesa.cogerPalillo(this.posicionMesa);

            // El filósofo está comiendo
            this.comiendo();

            System.out.println("Filosofo " + this.comensal + " Termina de comer, Libres palillos: " +
                    (this.mesa.palilloIzquierdo(this.posicionMesa) + 1) + ", " +
                    (this.mesa.palilloDerecho(this.posicionMesa) + 1));

            // El filósofo deja los palillos después de comer
            mesa.dejarPalillo(this.posicionMesa);

        } catch (InterruptedException ex) {
            Logger.getLogger(Filosofo.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```

Clase Mesa

```java
/**
 * La clase Mesa representa una mesa con un conjunto de palillos utilizados en el
 * problema de los filósofos cenando. Cada filósofo necesita dos palillos (uno a su
 * izquierda y otro a su derecha) para comer. Los palillos son gestionados mediante
 * semáforos para asegurar la sincronización entre los filósofos.
 */
public class Mesa {

    /**
     * Array de semáforos que representa los palillos en la mesa.
     * Cada semáforo asegura que un palillo solo sea utilizado por un filósofo a la vez.
     */
    private Semaphore[] palillos;

    /**
     * Constructor que inicializa la mesa con un número específico de palillos.
     * Cada palillo es representado por un semáforo con un permiso.
     *
     * @param numPalillos El número de palillos disponibles en la mesa (debería ser igual al número de filósofos).
     */
    public Mesa(int numPalillos) {
        this.palillos = new Semaphore[numPalillos];

        // Inicializar cada palillo (semáforo) con un permiso.
        for (int i = 0; i < numPalillos; i++) {
            this.palillos[i] = new Semaphore(1);
        }
    }
```

```java
    /**
     * Devuelve el índice del palillo izquierdo en función del índice del filósofo.
     *
     * @param palillo El índice del filósofo.
     * @return El índice del palillo a la izquierda del filósofo.
     */
    public int palilloIzquierdo(int palillo) {
        return palillo;
    }

    /**
     * Devuelve el índice del palillo derecho en función del índice del filósofo.
     * Si el filósofo es el primero (índice 0), el palillo derecho será el último.
     *
     * @param palillo El índice del filósofo.
     * @return El índice del palillo a la derecha del filósofo.
     */
    public int palilloDerecho(int palillo) {
        if (palillo == 0) {
            return this.palillos.length - 1;
        }
        return palillo - 1;
    }
```

```java
/**
 * Permite a un filósofo coger los dos palillos necesarios para comer.
 * Utiliza los métodos acquire() de los semáforos para bloquear los palillos.
 *
 * @param filosofo El índice del filósofo que quiere coger los palillos.
 * @throws InterruptedException Si el hilo es interrumpido mientras espera por un palillo.
 */
public void cogerPalillo(int filosofo) throws InterruptedException {
    // El filósofo coge el palillo izquierdo y luego el derecho
    this.palillos[this.palilloIzquierdo(filosofo)].acquire();
    this.palillos[this.palilloDerecho(filosofo)].acquire();
}

/**
 * Permite a un filósofo dejar los dos palillos después de comer.
 * Utiliza los métodos release() de los semáforos para liberar los palillos.
 *
 * @param filosofo El índice del filósofo que quiere dejar los palillos.
 */
public void dejarPalillo(int filosofo) {
    // El filósofo suelta el palillo izquierdo y luego el derecho
    this.palillos[this.palilloIzquierdo(filosofo)].release();
    this.palillos[this.palilloDerecho(filosofo)].release();
}
```

Clase Main

```java
package com.mycompany.psp02_tarea2;

public class PSP02_Tarea2 {

    public static void main(String[] args) {

        // Crear una mesa con 5 palillos
        Mesa m = new Mesa(5); // En la mesa se colocan 5 palillos disponibles para los filósofos

        // Crear y comenzar 5 filósofos
        for (int i = 1; i <= 5; i++) {  // Iteramos 5 veces para crear 5 filósofos

            // Creamos un filósofo con el número correspondiente (comenzando desde 1)
            // 'i' es el índice del bucle.
            Filosofo filosofo = new Filosofo(m, i);

            // Iniciar el hilo del filósofo (cada uno comenzará su ejecución)
            filosofo.start(); // Comienza el hilo para que el filósofo comience a pensar, comer, etc.
        }
    }
}
```