EJERCICIOS HILOS

Se desea implantar un huerto en un centro público y se han instalado varios sensores ESP32 para monitorizar la temperatura, la humedad del suelo y el estado de las plantas en distintas áreas del huerto. Se requiere un sistema en Java que simule la lectura de estos sensores de manera concurrente para obtener los datos en tiempo real.

Requisitos del sistema:

1. Crear tres hilos para simular los sensores: uno para la temperatura, otro para la humedad del suelo y otro para el estado de las plantas.

2. Cada sensor (hilo) debe:

Generar un valor aleatorio para la medición correspondiente.
Dormir un tiempo aleatorio entre 1 y 3 segundos para simular la espera entre lecturas.
Imprimir el valor generado con un mensaje que indique qué sensor hizo la lectura y en qué momento (puede usarse `System.currentTimeMillis()`).

3. El programa debe ejecutar 10 ciclos de medición por cada sensor.
Funcionalidades requeridas (FR):
FR1: Crear un hilo para cada sensor (temperatura, humedad y estado de las plantas).
FR2: Generar valores aleatorios para cada medición de sensor.
FR3:Simular la espera de tiempo entre lecturas con `Thread.sleep()`.
FR4:Mostrar las lecturas de cada sensor con su correspondiente marca de tiempo.
FR5: Ejecutar correctamente los 10 ciclos de medición por cada sensor.

SOLUCIÓN:

```
import java.util.Random;

// Clase principal

public class HuertoSensores {

  public static void main(String[] args) {

    // Creamos hilos para cada sensor usando clases internas

    Thread sensorTemperatura = new Thread(new Sensor("Temperatura", 15, 35));

    Thread sensorHumedad = new Thread(new Sensor("Humedad", 20, 80));

    Thread sensorEstadoPlantas = new Thread(new Sensor("Estado de plantas", 0, 1)); //
0=mal, 1=bien

    // Iniciamos los hilos: se ejecutarán de manera concurrente
```

```
        sensorTemperatura.start();
        sensorHumedad.start();
        sensorEstadoPlantas.start();
    }
}
// Clase que representa un sensor
class Sensor implements Runnable {
    private String nombre;      // Nombre del sensor
    private int minValor;       // Valor mínimo posible
    private int maxValor;       // Valor máximo posible
    private Random random;
    public Sensor(String nombre, int minValor, int maxValor) {
        this.nombre = nombre;
        this.minValor = minValor;
        this.maxValor = maxValor;
        this.random = new Random();
    }
    @Override
    public void run() {
        // Ejecutamos 10 ciclos de medición
        for (int i = 1; i <= 10; i++) {
            // Generamos valor aleatorio para el sensor
```

```
        int valor = random.nextInt(maxValor - minValor + 1) + minValor;

            // Mostramos lectura con marca de tiempo

            System.out.println(nombre + " - Lectura " + i + ": " + valor + " (Timestamp: " +
System.currentTimeMillis() + ")");

        // Simulamos tiempo de espera entre 1 y 3 segundos

        try {

            Thread.sleep((random.nextInt(3) + 1) * 1000);

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

  }

}
```

Enunciado: Cajeros Automáticos Simulados

Se quiere simular un banco con tres cajeros automáticos que atienden clientes de manera concurrente.

Requisitos del sistema:

1. Crear tres hilos, cada uno representando un cajero automático:
    - Cajero 1
    - Cajero 2
    - Cajero 3

2. Cada cajero debe:
    - Atender a 10 clientes consecutivos.
    - Para cada cliente, generar un tiempo de servicio aleatorio entre 1 y 5 segundos.
    - Imprimir un mensaje indicando:
        - Qué cajero está atendiendo.
        - Número del cliente.
        - Tiempo estimado de atención.
3. Todos los cajeros deben trabajar de manera concurrente, es decir, los mensajes pueden aparecer mezclados en consola.

Funcionalidades Requeridas:

- FR1: Crear un hilo para cada cajero.
- FR2: Generar valores aleatorios para el tiempo de atención.
  FR3: Simular la espera entre clientes con Thread.sleep().
- FR4: Mostrar los mensajes de atención de cada cajero.
- FR5: Cada cajero debe atender correctamente a 10 clientes.

SOLUCIÓN:

```
import java.util.Random;

public class Main {
  public static void main(String[] args) {

     Thread cajero1 = new Thread(new Cajero("Cajero 1"));
     Thread cajero2 = new Thread(new Cajero("Cajero 2"));
     Thread cajero3 = new Thread(new Cajero("Cajero 3"));

  cajero1.start();
  cajero2.start();
  cajero3.start();

  }
}

class Cajero implements Runnable {
  private String nombre;
  private Random random;

  public Cajero (String nombre){
     this.nombre = nombre;
     this.random = new Random();
  }

  @Override
  public void run() {

     for (int i = 1; i <= 10; i++) {
        int numeroale = random.nextInt((5-1+1)+1);
        System.out.println(nombre + " | Cliente: " + i + " | Tiempo: " + numeroale);

        try {
           Thread.sleep(numeroale * 1000);

        } catch (InterruptedException e) {
           e.printStackTrace();
        }
     }
  }
}
```

MÉTODO JOSE DAVID Y SEPARADO POR CLASES (ESTO ES PARA UNA CUENTA DE BANCO COMPARTIDA):

```java
// Clase principal: crea la cuenta y los hilos de los clientes

public class BancoSinSincronizar {
  public static void main(String[] args) {
     Cuenta cuenta = new Cuenta(); // crea una cuenta compartida

     // Crea dos hilos (clientes) que usan la misma cuenta
     Thread t1 = new Thread(new Cliente(cuenta, "Cliente 1"));
     Thread t2 = new Thread(new Cliente(cuenta, "Cliente 2"));

     // Inicia los hilos
     t1.start();
     t2.start();
  }
}
```

```java
// Clase que implementa Runnable: cada cliente será un hilo

public class Cliente implements Runnable {
  private Cuenta cuenta; // referencia compartida a la cuenta
  private String nombre; // nombre del cliente

  // Constructor: recibe la cuenta y el nombre del cliente
  public Cliente(Cuenta cuenta, String nombre) {
     this.cuenta = cuenta;
     this.nombre = nombre;
  }

  // Método que ejecuta el hilo

  @Override
  public void run() {
     for (int i = 0; i < 3; i++) { // intenta retirar dinero 3 veces
        cuenta.retirarDinero(nombre, 50); // realiza la operación
        try {
           Thread.sleep(100); // pausa para simular tiempo real
        } catch (InterruptedException e) {
           e.printStackTrace();
        }
     }
  }
}
```

```java
// Clase que representa una cuenta bancaria compartida

public class Cuenta {
  private int saldo = 100; // saldo inicial

  // Método para retirar dinero de la cuenta
  public synchronized void retirarDinero(String nombre, int cantidad) {
    if (saldo >= cantidad) { // comprueba si hay saldo suficiente
      System.out.println(nombre + " va a retirar " + cantidad);
      saldo -= cantidad; // actualiza el saldo
      System.out.println(nombre + " ha retirado. Saldo restante: " + saldo);
    } else {
      System.out.println(nombre + " no puede retirar, saldo insuficiente.");
    }
  }
}
```

UNA CUENTA COMPARTIDA Y LAS OTRAS NO (EJERCICIO COMPLICADO):

```java
// Clase principal que crea las cuentas y los clientes
public class Banco {
  public static void main(String[] args) {
    // Cuentas: una compartida y tres individuales
    Cuenta cuentaCompartida = new Cuenta(200); // usada por Cliente 1 y Cliente 2
    Cuenta cuenta3 = new Cuenta(150);
    Cuenta cuenta4 = new Cuenta(100);
    Cuenta cuenta5 = new Cuenta(300);

    // Clientes (5 en total)
    Thread c1 = new Thread(new Cliente(cuentaCompartida, "Cliente 1", true));
    Thread c2 = new Thread(new Cliente(cuentaCompartida, "Cliente 2", true));
    Thread c3 = new Thread(new Cliente(cuenta3, "Cliente 3", false));
    Thread c4 = new Thread(new Cliente(cuenta4, "Cliente 4", false));
    Thread c5 = new Thread(new Cliente(cuenta5, "Cliente 5", false));

    // Iniciar todos los hilos
    c1.start();
    c2.start();
    c3.start();
    c4.start();
    c5.start();
  }
}
```

```java
import java.util.Random;

// Clase que representa a un cliente que actúa como hilo
public class Cliente implements Runnable {
  private Cuenta cuenta;
  private String nombre;
```

```
    private boolean cuentaCompartida; // indica si debe esperar más tiempo
    private Random random = new Random();

    // Constructor
    public Cliente(Cuenta cuenta, String nombre, boolean cuentaCompartida) {
        this.cuenta = cuenta;
        this.nombre = nombre;
        this.cuentaCompartida = cuentaCompartida;
    }

    @Override
    public void run() {
        for (int i = 0; i < 3; i++) { // cada cliente intenta retirar 3 veces
            cuenta.retirarDinero(nombre, 50);

            try {
                // si comparten cuenta, esperan entre 3 y 6 segundos
                if (cuentaCompartida) {
                    int espera = 3000 + random.nextInt(3000); // entre 3000 y 6000 ms
                    Thread.sleep(espera);
                } else {
                    Thread.sleep(1000); // si es cuenta individual, solo 1 segundo
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
// Clase que representa una cuenta bancaria compartida
public class Cuenta {
    private int saldo;

    // Constructor: define el saldo inicial
    public Cuenta(int saldoInicial) {
        this.saldo = saldoInicial;
    }

    // Método sincronizado para evitar accesos simultáneos peligrosos
    public synchronized void retirarDinero(String nombre, int cantidad) {
        if (saldo >= cantidad) { // hay suficiente dinero
            System.out.println(nombre + " va a retirar " + cantidad);
            saldo -= cantidad; // resta del saldo
            System.out.println(nombre + " ha retirado. Saldo restante: " + saldo);
        } else {
            System.out.println(nombre + " no puede retirar, saldo insuficiente. (Saldo actual: " +
saldo + ")");
        }
    }
```

```
    // Getter opcional si se quiere consultar saldo desde fuera
    public int getSaldo() {
        return saldo;
    }
}
```

🗒️ **Tarea: Simulación concurrente de sensores en un huerto inteligente (EJERCICIO MÁS COMPLICADO)**

Se ha decidido implantar un huerto en un centro público, y se han instalado varios sensores ESP32 para monitorizar la temperatura, la humedad del suelo y el estado de las plantas en distintas áreas del huerto.
Se requiere un sistema en Java que simule la lectura de estos sensores de manera concurrente para obtener los datos en tiempo real.

🧩 **Tu tarea:**
Diseñar una solución que simule el monitoreo de estos sensores usando hilos en Java.
El sistema debe realizar lo siguiente:

Crear tres hilos para simular los sensores:
Uno para la temperatura.
Uno para la humedad del suelo.
Uno para el estado de las plantas.

Cada sensor (hilo) debe:
Generar un valor aleatorio para su medición correspondiente.
Dormir durante un tiempo aleatorio entre 1 y 3 segundos para simular la espera entre lecturas.
Imprimir el valor generado con un mensaje que indique qué sensor hizo la lectura y en qué momento (System.currentTimeMillis()).
Realizar 10 ciclos de medición (10 iteraciones por sensor).
Añadir un recurso compartido, por ejemplo una clase RegistroCentral, que almacene los valores leídos por todos los sensores.
Todos los sensores deben usar el mismo objeto compartido para guardar sus mediciones, empleando synchronized para evitar conflictos de concurrencia.
Además, simular que existen dos zonas distintas del huerto, cada una con su propio conjunto de sensores.
La Zona 1 y la Zona 2 tendrán cada una sus propios sensores de temperatura, humedad y plantas (por tanto, sensores independientes).
Los sensores de ambas zonas usarán el mismo registro central (compartido) para enviar sus datos, pero cada uno imprimirá la zona de la que procede.
Implementar constructores que permitan configurar el nombre del sensor, la zona y el registro compartido.

⚙️ **Funcionalidades Requeridas (FR):**

FR1 (2 puntos): Crear un hilo para cada sensor (temperatura, humedad y estado de las plantas).
FR2 (2 puntos): Generar valores aleatorios para cada medición de sensor.
FR3 (2 puntos): Simular la espera de tiempo entre lecturas con Thread.sleep().
FR4 (2 puntos): Mostrar las lecturas de cada sensor con su correspondiente marca de tiempo y zona.
FR5 (2 puntos): Ejecutar correctamente los 10 ciclos de medición por cada sensor.

FR6 (2 puntos): Implementar un recurso compartido sincronizado (por ejemplo, una clase RegistroCentral) donde todos los sensores almacenen los datos concurrentemente.
FR7 (2 puntos): Crear sensores pertenecientes a distintas zonas del huerto que trabajen de forma independiente pero registren sus lecturas en el mismo objeto compartido.
FR8 (2 puntos): Utilizar constructores con varios parámetros (nombre, zona, registro) para la inicialización de los hilos.

SOLUCIÓN:

```
public class Huerto {
  public static void main(String[] args) {

    RegistroCentral registro = new RegistroCentral(); // Compartido

    //Zona 1
    Thread sensorTemperatura1 = new Thread(new Sensor("Temperatura", "Zona 1", registro));
    Thread sensorHumedad1 = new Thread(new Sensor("Humedad", "Zona 1", registro));
    Thread sensorEstado1 = new Thread(new Sensor("Estado", "Zona 1", registro));

    //Zona 2
    Thread sensorTemperatura2 = new Thread(new Sensor("Temperatura", "Zona 2", registro));
    Thread sensorHumedad2 = new Thread(new Sensor("Humedad", "Zona 2", registro));
    Thread sensorEstado2 = new Thread(new Sensor("Estado", "Zona 2", registro));

    //Iniciar Hilos
    sensorTemperatura1.start();
    sensorHumedad1.start();
    sensorEstado1.start();
    sensorTemperatura2.start();
    sensorHumedad2.start();
    sensorEstado2.start();


  }
}
```

```
import java.sql.Time;
import java.util.Random;

public class Sensor implements Runnable {
  private String nombre;
  private String zona;
  private RegistroCentral registro;
  private Random random = new Random();


  public Sensor(String nombre, String zona, RegistroCentral registro) {
    this.nombre = nombre;
    this.zona = zona;
    this.registro = registro;
```

```
  }

  @Override
  public void run() { //Ejecución

      for (int i = 1; i <= 10; i++) {
          long timestamp = System.currentTimeMillis();
          if (nombre.equalsIgnoreCase("Estado")) {
              int estado = random.nextInt(2);
              String linea = "Estado: " + estado + " | " + zona + " | Lectura: " + i + " | MS: " +
timestamp;
              System.out.println(linea);
              registro.registrar(linea);

          } else if (nombre.equalsIgnoreCase("Temperatura")) {
              double valor = 15 + random.nextDouble() * 20; // 15–35
              String linea = nombre + ":" + valor +  " | " + zona + " | Lectura: " + i  + " | MS: " +
timestamp;
              System.out.println(linea);
              registro.registrar(linea);

          } else if (nombre.equalsIgnoreCase("Humedad")) {
              double valor = random.nextDouble() * 70; // 0–100
              String linea = nombre + ":" + valor +  "% | " + zona + " | Lectura: " + i  + " | MS: " +
timestamp;
              System.out.println(linea);
              registro.registrar(linea);
          }

          try {
              Thread.sleep(1000 + random.nextInt(2000));
          } catch (InterruptedException e) {
              e.printStackTrace();
          }
      }
  }
}
```

```
import java.util.ArrayList;
import java.util.List;

public class RegistroCentral {
private final List<String> registros = new ArrayList<>();

public synchronized void registrar(String Lectura) {
  registros.add(Lectura);
  System.out.println("Registro correcto.");
}
}
```

EJERCICIOS PROCESOS

ENUNCIADO:
Crea un programa Java que implemente las siguientes Funcionalidades Requeridas (FRs):
FR1: lea una cadena de caracteres desde la entrada estándar hasta recibir un carácter de terminación, en concreto, un asterisco *
FR2: una vez recibido el caracter de terminación, muestre por pantalla toda la información leída
FR3: Crea después otro programa que ejecute el anterior
Implementa el control de errores -
Documenta el código

SOLUCIÓN:

```
import java.io.BufferedReader;      // Importa para leer texto de manera eficiente
import java.io.InputStreamReader;   // Permite leer desde System.in (entrada estándar)
import java.io.IOException;         // Para capturar errores de entrada/salida

*/public class App {
  public static void main(String[] args) {

    // StringBuilder para almacenar todo lo que escribe el usuario
    StringBuilder input = new StringBuilder();

    // Mensaje inicial para indicar cómo terminar la entrada
    System.out.println("Introduce letras (acaba con '*'):");

    // try-with-resources: crea un lector de entrada y lo cierra automáticamente al terminar
    try (BufferedReader reader = new BufferedReader(new InputStreamReader(System.in))) {

      int letrita; // variable donde se guarda temporalmente cada carácter leído (como número)

      // Bucle principal: lee carácter por carácter hasta que se detecte el final (-1) o el '*'
      while ((letrita = reader.read()) != -1) { // lee un carácter del teclado
        char letrota = (char) letrita;        // convierte el número leído en carácter

        if (letrota == '*') {                 // FR1: condición de parada cuando se introduce '*'
          break;                              // sale del bucle
        }

        input.append(letrota);                // añade el carácter leído al texto acumulado
      }

      // FR2: cuando se termina la lectura, muestra todo el texto introducido
      System.out.println("\n--- Texto introducido ---");
      System.out.println(input.toString());     // imprime el contenido completo

    } catch (IOException e) {                   // FR4: captura errores de lectura
      System.err.println("Error al leer la entrada: " + e.getMessage());
    }
  }
}
```

```java
import java.io.IOException;  // Necesario para manejar errores de entrada/salida

public class Launcher {
  public static void main(String[] args) {

    try {
      // FR3: Crea el proceso que ejecutará el programa App
      // "java -cp bin App" => ejecuta la clase App que está dentro de la carpeta 'bin'
      ProcessBuilder pb = new ProcessBuilder("java", "-cp", "bin", "App");

      // Redirige la entrada/salida del proceso hijo para usar la misma consola que el padre
      pb.inheritIO();

      // Inicia la ejecución del proceso (App comienza a ejecutarse)
      Process process = pb.start();

      // Espera a que el proceso hijo (App) termine antes de continuar
      int exitCode = process.waitFor();

      // Muestra el código de salida del proceso hijo (0 = correcto)
      System.out.println("\nEl proceso App terminó con código: " + exitCode);

    } catch (IOException | InterruptedException e) { // FR4: controla errores al crear o ejecutar el proceso
      System.err.println("Error al ejecutar el proceso: " + e.getMessage());
    }
  }
}
```

ENUNCIADO:

Crea un programa en Java que admita argumentos desde main() y devuelva con System.exit() los siguientes valores:
Si el número de argumentos es < 1 debe devolver 1
Si el argumento es una cadena debe devolver 2
Si el argumento es un número entero menor que 0 debe devolver 3
En cualquier otro caso debe devolver 0

Realiza un segundo programa Java que ejecute al anterior. Este programa deberá mostrar por pantalla lo que sucede según el valor devuelto al ejecutar el programa principal.

Crea un programa Java que implemente las siguientes Funcionalidades Requeridas (FRs):
FR1: admita argumentos desde main() - 1 punto
FR2: devuelva con System.exit() los siguientes valores:
Si el número de argumentos es < 1 debe devolver 1 - 1 punto
Si el argumento es una cadena debe devolver - 1 punto
Si el argumento es un número entero menor que 0 debe devolver 3 - 1 punto
En cualquier otro caso debe devolver 0 - 1 punto
FR3: Crea después otro programa que ejecute el anterior

SOLUCIÓN:

```
public class Principal {
  public static void main(String[] args) {
          // FR1: Comprobar si se proporcionó al menos un argumento
     if (args.length < 1) {
        System.exit(1);  // FR2: devuelve 1 si no hay argumentos
     }

     try {

        int num = Integer.parseInt(args[0]); // Intentar convertir el primer argumento a entero

        if (num < 0) {
           System.exit(3);
        } else {
           System.exit(0);
        }

     } catch (NumberFormatException e) {
        System.exit(2);
     }
  }
}
```

```
import java.io.*;

public class Lanzador {
  public static void main(String[] args) {
          // FR1: comprobar que se proporcionó un argumento al lanzador
     if (args.length < 1) {
        System.out.println("Uso: java Lanzador <argumento>");
        return;
     }

     try {
          // Crear un proceso que ejecute el programa Principal con el argumento recibido
        ProcessBuilder pb = new ProcessBuilder("java", "Principal", args[0]);
        pb.inheritIO();     // hereda la entrada/salida del proceso padre
        Process proceso = pb.start();

        int exitCode = proceso.waitFor(); // Espera a que el proceso termine y captura el código
de salida

        switch (exitCode) {
           case 0:
              System.out.println("Todo chachi");
              break;
           case 1:
              System.out.println("Error: no se proporcionaron argumentos.");
              break;
           case 2:
```

```
                System.out.println("no es un número entero.");
                break;
            case 3:
                System.out.println("el número es negativo.");
                break;
            default:
                System.out.println("error por defecto " + exitCode);
        }

    } catch (IOException e) {
        System.err.println("Error al ejecutar el proceso: " + e.getMessage());
    } catch (InterruptedException e) {
        System.err.println("La ejecución fue interrumpida: " + e.getMessage());
    }
  }
}
```

EJERCICIO GENERAL:

```
import java.util.Scanner;
import java.io.*;

// Programa que ejecuta varios comandos del sistema usando ProcessBuilder y
muestra/guarda la salida
public class EjecutaComando {
  public static void main(String[] args) throws IOException, InterruptedException {

    // 1) Ejecuta "ping www.google.com" y muestra la salida en consola
    ProcessBuilder pb1 = new ProcessBuilder("ping", "www.google.com");
    pb1.redirectErrorStream(true);              // redirige errores a la salida estándar
    Process p1 = pb1.start();                   // inicia el proceso

    // Lee la salida del proceso línea por línea
    try (BufferedReader br = new BufferedReader(new
InputStreamReader(p1.getInputStream()))) {
        String linea;
        while ((linea = br.readLine()) != null) {
            System.out.println(linea);          // imprime en consola
        }
    }

    // 2) Ejecuta "ping www.direccionInexistente.test" y muestra la salida/error
    ProcessBuilder pb2 = new ProcessBuilder("ping", "www.direccionInexistente.test");
    pb2.redirectErrorStream(true);
    Process p2 = pb2.start();

    try (BufferedReader br = new BufferedReader(new
InputStreamReader(p2.getInputStream()))) {
        String linea;
        while ((linea = br.readLine()) != null) {
            System.out.println(linea);          // muestra fallo de conexión
        }
```

```
    }

    // 3) Ejecuta "ping www.google.com" y guarda la salida en "salida.txt"
    String fichero = "salida.txt";
    ProcessBuilder pb3 = new ProcessBuilder("ping", "www.google.com");
    pb3.redirectErrorStream(true);
    Process p3 = pb3.start();

    // Lee la salida y la escribe también en un archivo
    try (
            BufferedReader br = new BufferedReader(new
InputStreamReader(p3.getInputStream()));
            BufferedWriter bw = new BufferedWriter(new FileWriter(fichero))
    ) {
        String linea;
        while ((linea = br.readLine()) != null) {
            System.out.println(linea);            // imprime
            bw.write(linea);                      // escribe en el fichero
            bw.newLine();
        }
    }

    // 4) Ejecuta "dir" y guarda la salida en "lista.txt"
    String ficheroLista = "lista.txt";
    ProcessBuilder pb4 = new ProcessBuilder("cmd", "/c", "dir");
    pb4.redirectOutput(new File(ficheroLista));     // redirige la salida al archivo
    Process p4 = pb4.start();
    int exitCode4 = p4.waitFor();                 // espera a que termine

    // Si todo salió bien, muestra el contenido del archivo en consola
    if (exitCode4 == 0) {
        ProcessBuilder pb4b = new ProcessBuilder("cmd", "/c", "type", ficheroLista);
        pb4b.inheritIO();            // hereda IO del proceso padre
        Process p4b = pb4b.start();
        p4b.waitFor();
    } else {
        System.out.println("Fallo: " + exitCode4);
    }

    // 5) Permite al usuario introducir un comando manualmente
    Scanner sc = new Scanner(System.in);
    System.out.print("Introduce un comando: ");
    String comando = sc.nextLine();

    // Ejecuta el comando introducido y muestra su salida
    ProcessBuilder pb5 = new ProcessBuilder("cmd", "/c", comando);
    pb5.redirectErrorStream(true);
    Process p5 = pb5.start();

    try (BufferedReader br = new BufferedReader(new
InputStreamReader(p5.getInputStream()))) {
        String linea;
```

```
        while ((linea = br.readLine()) != null) {
            System.out.println(linea);          // imprime línea a línea
        }
    }
    sc.close();                             // cierra el Scanner
  }
}
```

EJERCICIO:

FR1: Haz un programa en C que genere una estructura de procesos con un PADRE y 3 HIJOS, del mismo padre se entiende - 2 puntos
FR2: Visualiza por cada hijo su identificador (si es el hijo 1, 2 ó 3), su PID y el del padre, utilizando para ello una función definida por ti a la que llamen los procesos hijos - 2 puntos
FR3: Justo antes de finalizar el programa PADRE, se debe imprimir por pantalla el PID del padre de todos una única vez. Debe hacerlo el programa PADRE - 2 puntos
FR4: Implementa el control de errores - 2 puntos
FR5: Documenta y estructura el código - 2 puntos

SOLUCIÓN:

```
import java.io.IOException;

public class EstructuraProcesos {

   public static void main(String[] args) {
       // FR1: Crear 3 procesos hijos
       for (int i = 1; i <= 3; i++) {
           try {
               ProcessBuilder pb = new ProcessBuilder("java", "Hijo",
String.valueOf(i));
               pb.inheritIO(); // Redirige salida del hijo a la consola del
padre
               pb.start();
           } catch (IOException e) {
               System.err.println("Error al crear el proceso hijo " + i +
": " + e.getMessage());
           }
       }

       // FR3: El padre imprime su PID al final
       System.out.println("\n[PADRE] Finalizando. Mi PID es: " +
ProcessHandle.current().pid());
   }
}
```

```java
public class Hijo {

    public static void main(String[] args) {
        if (args.length < 1) {
            System.err.println("Error: falta número de hijo.");
            System.exit(1);
        }

        try {
            int numeroHijo = Integer.parseInt(args[0]);
            mostrarInfoHijo(numeroHijo);
        } catch (NumberFormatException e) {
            System.err.println("Error: argumento inválido.");
        }
    }

    // FR2: Función definida por el alumno que muestra PID hijo y PID padre
    public static void mostrarInfoHijo(int num) {
        long pidHijo = ProcessHandle.current().pid();
        long pidPadre =
ProcessHandle.current().parent().map(ProcessHandle::pid).orElse(-1L);
        System.out.println("[HIJO " + num + "] PID: " + pidHijo + " | PID
PADRE: " + pidPadre);
    }
}
```

TEORÍA:

Proceso: Programa en ejecución con su propio espacio de memoria y recursos del sistema. Es la unidad básica de ejecución en un sistema operativo.

Hilo (thread): Subproceso que comparte memoria y recursos con otros hilos del mismo proceso. Permite realizar varias tareas dentro de un mismo programa simultáneamente.

Programación secuencial: las instrucciones se ejecutan una tras otra, de forma lineal, sin solapamiento ni ejecución simultánea.
Ejemplo: una app de escanear documentos que procesa página a página de forma lineal.

Programación concurrente: varios procesos o hilos pueden estar activos al mismo tiempo, aunque no necesariamente se ejecuten simultáneamente (pueden alternarse en el tiempo).
Ejemplo: una app de descargas que inicia varios ficheros al mismo tiempo, alternando hilos en un solo CPU.

Programación paralela: varias tareas se ejecutan al mismo tiempo en procesadores o núcleos distintos, aumentando el rendimiento.
Ejemplo: un render 3D que calcula diferentes partes de la escena simultáneamente en varios núcleos.

Programación distribuida: varias tareas se ejecutan en distintos equipos conectados en red, colaborando para resolver un mismo problema.
Ejemplo: un sistema de cálculo de estadísticas que reparte datos entre varios servidores conectados en red.

Runnable vs Thread: Usar implements Runnable es mejor porque permite que la clase herede de otra si hace falta y separa la lógica del hilo.

Synchronized: Se usa solo cuando varios hilos acceden o modifican el mismo recurso a la vez; si cada hilo trabaja con datos propios, no hace falta.

ProcessBuilder: Clase de Java que permite crear y gestionar procesos externos desde un programa Java, configurando su entorno, directorio de trabajo y flujos de entrada/salida.

Diferencia start() y run():
● **start()** crea un nuevo hilo y ejecuta run() en él, permitiendo concurrencia real.
● **run()** se llama como un método normal, ejecutando el código en el hilo actual sin generar concurrencia.