UT1 - Programacion multihilo

Introducción

Qué vamos a trabajar...

RA 1. Desarrolla aplicaciones compuestas por varios procesos reconociendo y aplicando principios de programación paralela

🧠 **Parte teórica (conceptos y fundamentos)**

a) Programación concurrente: características y ámbitos.
b) Programación paralela vs distribuida: diferencias, ventajas e inconvenientes.
c) Procesos: características y ejecución por el sistema operativo.
d) Hilos de ejecución: relación con los procesos.

🛠️ **Parte práctica (aplicación y desarrollo)**

e) Clases para crear subprocesos.
f) Mecanismos para compartir información con subprocesos.
g) Mecanismos para sincronizar y obtener valores de subprocesos.
h) Aplicaciones que gestionan procesos en paralelo.
i) Depuración y documentación de aplicaciones.

RA 2. Desarrolla aplicaciones compuestas por varios hilos de ejecución analizando y aplicando librerías específicas del lenguaje de programación

🧠 **Parte teórica (conceptos y fundamentos)**

a) Situaciones útiles para varios hilos
d) Estados de ejecución de un hilo y su gestión
g) Prioridad de hilos de ejecución
i) Contexto de ejecución de los hilos
k) Problemas al compartir información entre hilos

🛠️ **Parte práctica (aplicación y desarrollo)**

b) Mecanismos para crear, iniciar y finalizar hilos
c) Programación de aplicaciones con varios hilos
e) Mecanismos para compartir información entre hilos
f) Programas con varios hilos sincronizados
h) Depuración y documentación de programas
j) Librerías específicas para programación multihilo

🧠 RA1a) Programación concurrente: características y ámbitos

- **Definición:** La programación concurrente permite que **varias tareas se ejecuten de forma solapada en el tiempo**, compartiendo recursos del mismo sistema.
- **Objetivo:** Mejorar la **eficiencia y capacidad de respuesta** de los programas.
- **Características:**
  - Varias tareas activas de forma lógica al mismo tiempo.
  - Comparten recursos del mismo proceso (memoria, CPU).
  - Requiere mecanismos de sincronización para evitar interferencias.
- **Ámbitos de aplicación:**
  - Aplicaciones con interfaces gráficas (mantener la UI reactiva).
  - Servidores que gestionan múltiples peticiones de usuarios.
  - Simulaciones, videojuegos, sistemas de tiempo real.

⚡ RA1b) Programación paralela vs distribuida

Programación paralela

- **Qué es:** Ejecutar varias tareas **simultáneamente en varios núcleos de una misma máquina**.
- **Memoria:** Compartida entre hilos/procesos.
- **Comunicación:** Rápida (memoria compartida).
- **Ventajas:**
  - Alta velocidad de comunicación.
  - Ideal para cálculos intensivos.
- **Inconvenientes:**
  - Escalabilidad limitada al número de núcleos disponibles.
  - Necesita sincronización cuidadosa para evitar condiciones de carrera.

- **Ejemplos:** Hilos (`Thread`), `ExecutorService` , `ForkJoinPool` .

Programación distribuida

- **Qué es:** Ejecutar varias tareas en **distintas máquinas conectadas por red**.
- **Memoria:** Independiente en cada nodo.
- **Comunicación:** Más lenta, mediante mensajes/red.
- **Ventajas:**
  - Muy escalable (puedes añadir más nodos).
  - Tolerancia a fallos (si un nodo falla, los demás siguen).
- **Inconvenientes:**
  - Mayor complejidad de coordinación.
  - Comunicación más lenta que en memoria compartida.
- **Ejemplos:** Sistemas con `RMI` (Remote Method Invocation), `RPC` (Remote Procedure Call), `sockets` , `servicios web` , `microservicios` .

Aspecto	Paralela	Distribuida
Ubicación de tareas	Una máquina	Varias máquinas
Tipo de memoria	Compartida	Independiente
Comunicación	Memoria compartida	Red (mensajes)
Velocidad de comunicación	Muy rápida	Más lenta
Escalabilidad	Vertical y limitada (aumentando los recursos de la maquina)	Horizontal, muy alta (añadiendo maquinas a un cluster)

❓ Pregunta

¿Qué tipo de programación tiene mayor complejidad de implementación: la programación paralela o la programación distribuida?

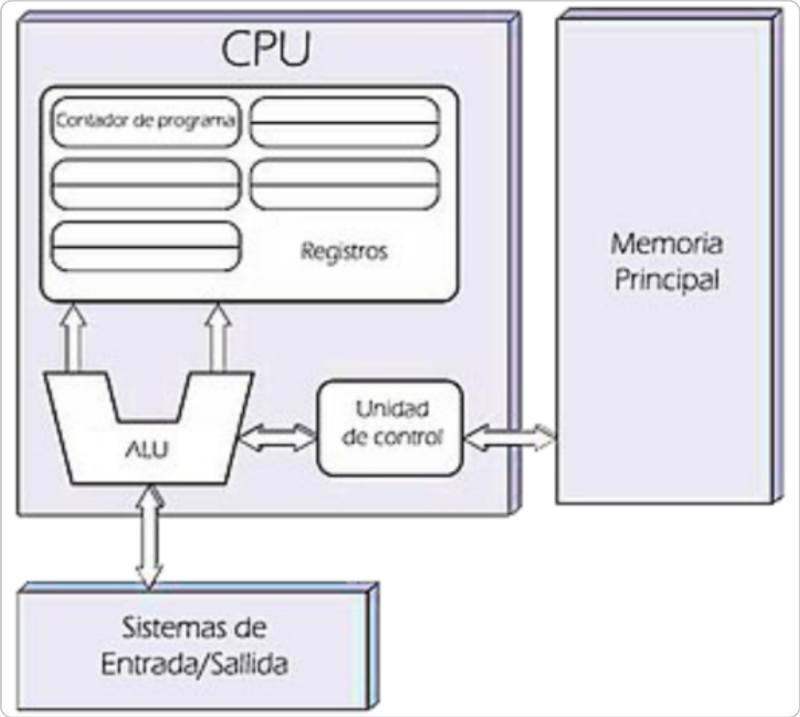
> ✅ La **programación distribuida** es más compleja porque requiere coordinar múltiples máquinas, comunicación por red, tolerancia a fallos y sincronización entre nodos, mientras que la paralela ocurre en una sola máquina con memoria compartida.

⚙️ RA1c) Procesos: características y ejecución por el sistema operativo

Concepto de programa

- Conjunto ordenado de instrucciones escritas en un lenguaje de programación para realizar una tarea concreta.

- Sus instrucciones se cargan en memoria principal para ser ejecutadas por la CPU.



🧮 Contador de programa (PC — Program Counter)

- Registro especial de la CPU.
- **Guarda la dirección de la siguiente instrucción** que debe ejecutarse.
- Cada vez que se ejecuta una instrucción, el PC se **incrementa automáticamente** para apuntar a la siguiente.
- Permite que la CPU sepa en qué punto del programa está.

💾 Registros

- **Pequeñas memorias muy rápidas** dentro de la CPU.
- Guardan **datos temporales o direcciones** durante la ejecución de instrucciones.
- Tipos habituales:
  - **Registro de instrucciones (IR):** almacena la instrucción actual.
  - **Acumulador:** guarda resultados de operaciones aritméticas y lógicas.
  - **Registros de propósito general:** almacenan datos intermedios.
  - **Registros de direcciones o punteros:** indican posiciones de memoria.

🔹 Son mucho más rápidos que la RAM y esenciales para que la CPU funcione eficientemente.

⚡ ALU (Unidad Aritmético-Lógica)

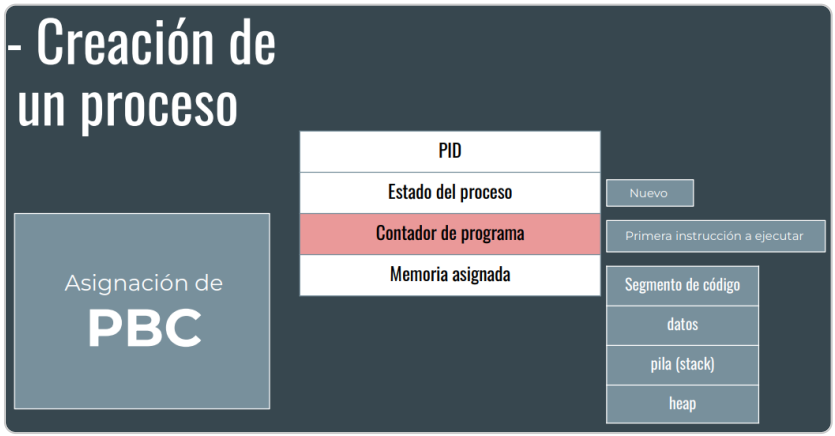
- Parte de la CPU que **realiza operaciones matemáticas y lógicas**:
- Sumas, restas, multiplicaciones, divisiones.
  - Comparaciones (>, <, ==, etc.).
  - Operaciones lógicas (AND, OR, NOT…).
- Recibe datos de los registros, realiza la operación y devuelve el resultado a otro registro

Concepto de proceso

- Un proceso es un **programa en ejecución**, gestionado por el sistema operativo (SO).
- Instancia de un programa en ejecución.
- Usa recursos del sistema: CPU, memoria principal, dispositivos de E/S.

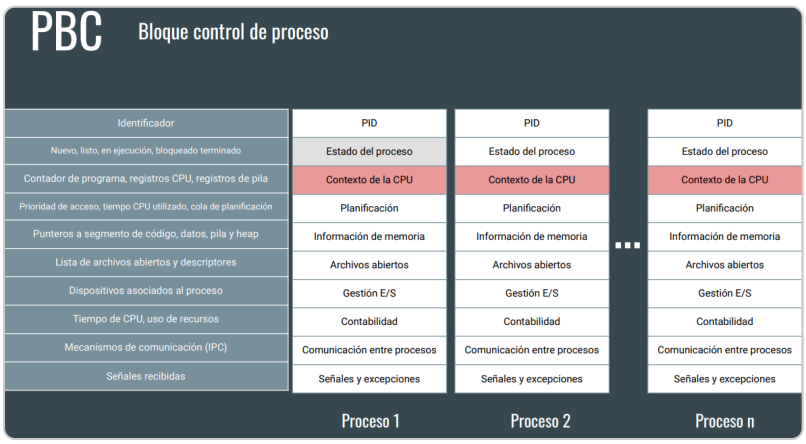
Características principales de un proceso

- El **SO recibe una solicitud para ejecutar un programa**
- Recoge toda la informacion del proceso en **PCB (Process Control Block)**
- Se ejecuta de forma secuencial en la CPU.
- Tiene recursos asignados (memoria, archivos, dispositivos).
- Está **aislado** de otros procesos por seguridad y estabilidad.
- Puede **cambiar de contexto** cuando el SO decide ejecutar otro proceso.
- Tiene su **propio espacio de memoria** independiente.



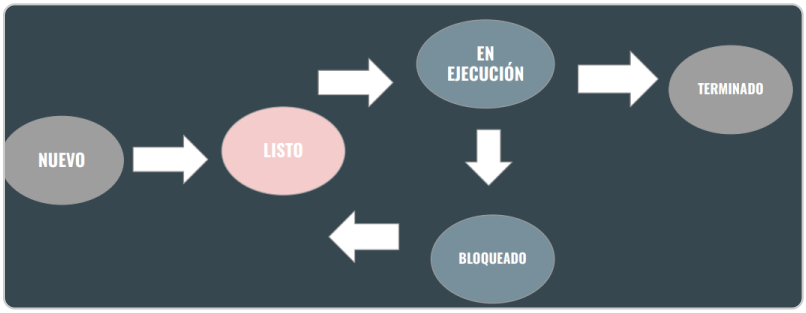
Bloque control de proceso (PBC)

- **PCB (Process Control Block)**: el SO guarda en memoria principal toda la información del proceso (registros, contador de programa...).



Componentes de un proceso

- **Espacio de direcciones**: código, datos, pila, heap.
- **Programa ejecutable**: instrucciones que ejecuta la CPU.
- **Estado del proceso** (nuevo, listo, en ejecución...).

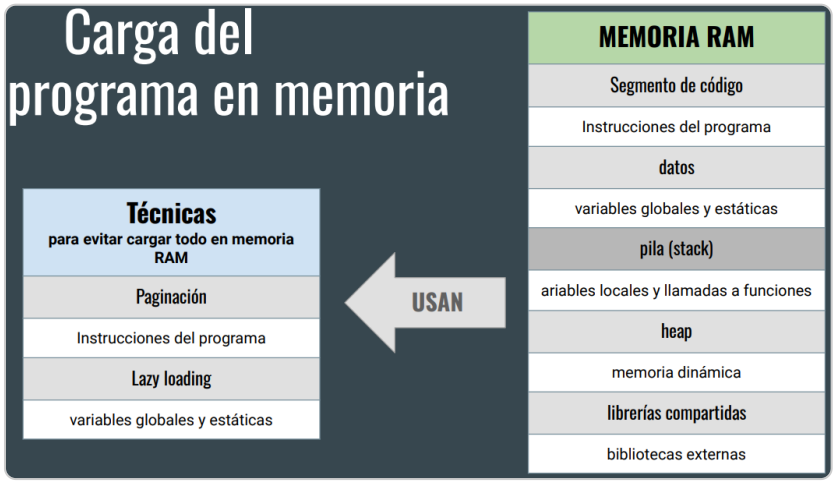


- **PCB (Process Control Block)**: el SO guarda en memoria principal toda la información del proceso (registros, contador de programa...).
- **Recursos asignados**: memoria, archivos, dispositivos.
- **Identificador (PID)**: lo distingue de otros procesos.

Ejemplo: la clase `ProcessBuilder` en Java sirve para:

- **Crear y ejecutar procesos** externos del sistema operativo desde un programa Java. Lanza otros programas o comandos (como si los escribieras en la terminal).
- Permite leer su salida y enviarles datos de entrada.
- Cada proceso creado se ejecuta de forma independiente del programa Java.

🧠 Zonas de memoria de un proceso



Zona	Contenido	Tamaño	Acceso	Ejemplos típicos
Código (code/text)	Instrucciones del programa	Fijo (solo lectura)	Solo lectura	Funciones, métodos, bucles, sentencias `if`
Datos (data)	Variables globales y estáticas	Fijo	Lectura/escritura	`static int contador;` `int x = 5;`
Pila de llamadas (stack)	Variables locales y direcciones de retorno	Dinámico (crece/disminuye)	L/E	Variables de funciones, parámetros, llamadas anidadas
Heap	Objetos y memoria dinámica	Dinámico (asignado por el programador)	L/E	`new Persona()`, `malloc()`, estructuras dinámicas

⚙️ Ejecución por el Sistema Operativo

- El **sistema operativo crea, planifica y finaliza los procesos** que se ejecutan en el sistema.
- Usa **planificadores de CPU** para **asignar el tiempo de procesador entre los distintos procesos secuencialmente**, decidiendo cuál se ejecuta en cada momento. **FCFS** (orden de llegada), **SJF** (más corto primero), **SRTF** (más corto restante), **RoundRobin** (turnos fijos), **Prioridades** (mayor prioridad primero), **MLFQ** (colas con prioridad dinámica).
- Controla las **transiciones entre estados** de los procesos:
  🆕 **Nuevo → Listo → En ejecución → Bloqueado → Terminado**

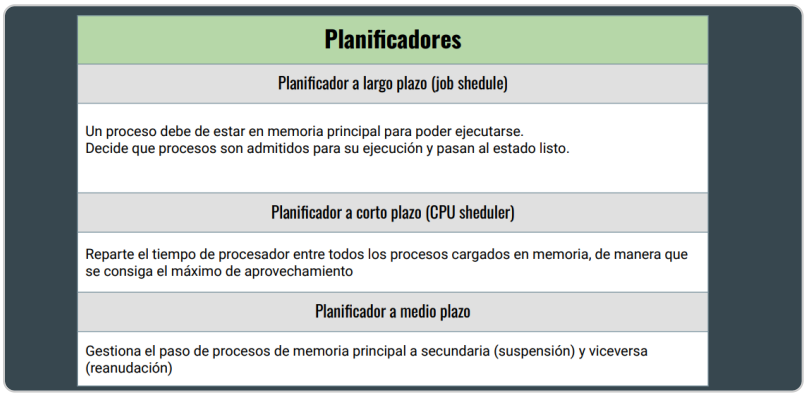
Paralelismo

- En un sistema con **una sola CPU o un solo núcleo**, **no existe paralelismo real**: solo **se simula mediante multitarea**, alternando muy rápido entre procesos mediante **cambios de contexto**.
- **Pseudoparalelismo (paralelismo virtual)** → un solo núcleo, los procesos se ejecutan alternadamente dando una sensacion de ejecucion paralela o simultánea de aplicaciones·
- **Paralelismo real** → varios núcleos o CPUs, procesos se ejecutan al mismo tiempo de verdad, de manera simultánea en paralelo.

Planificadores del SO

Los tres (job scheduler, medium-term scheduler y CPU scheduler) pertenecen al sistema operativo.

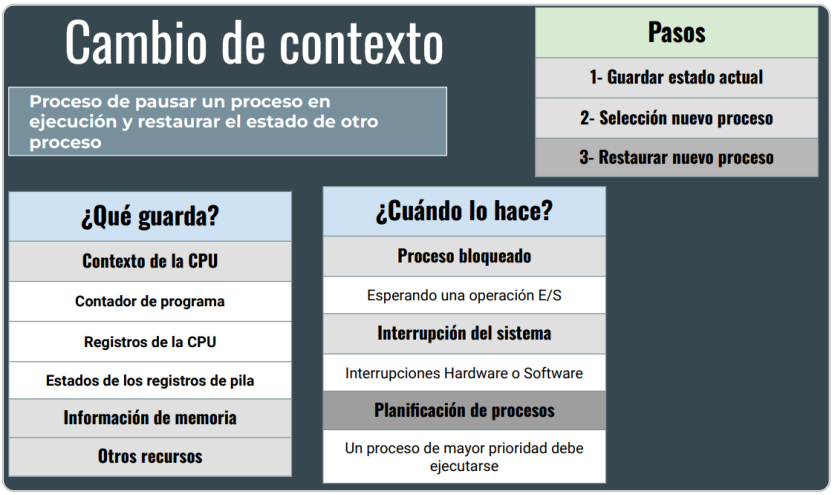
- **Job scheduler**: decide qué procesos entran en memoria desde el almacenamiento (pocas veces, largo plazo).
- **CPU scheduler**: decide qué proceso en memoria usa la CPU ahora (muy frecuente, corto plazo).
- **Medium-term scheduler**: decide qué procesos en memoria se suspenden o reanudan (libera o recupera RAM).



¿Qué implica el cambio de contexto?

El planificador del SO pausa la ejecucion de un proceso en ejecución para empezar a ejecutar otro proceso.

1. **Guardar el estado del proceso actual** (el que se pausa):
   - Contador de programa (la siguiente instrucción a ejecutar)
   - Registros de la CPU
   - Punteros de memoria, etc.
   - Toda esta información se guarda en el PCB (Process Control Block) del proceso.
2. **Cargar el estado del siguiente proceso** (el que se va a ejecutar
   - Restaura sus registros, contador de programa, etc., desde su PCB.
3. **La CPU continúa ejecutando ese nuevo proceso** desde donde se había quedado.



Comunicación entre procesos

- Se hace mediante **pipes, sockets, ficheros compartidos, etc**, ya que **no comparten memoria**.
- Estos mecanismos se denominan **(IPC - Inter-Process Communication)**

Mecanismo	Nivel	Qué hace	Uso típico	Ejemplo breve
Memoria compartida	Bajo (IPC clásico)	Varios procesos acceden a una misma zona de memoria	Procesos locales que comparten datos grandes	`MappedByteBuffer` en Java
Pipes (tuberías)	Bajo (IPC clásico)	Canal unidireccional: uno escribe y otro lee	Comunicación padre-hijo (`ls	grep`)
Sockets	Bajo (IPC clásico)	Canal bidireccional entre procesos (locales o remotos)	Comunicación cliente-servidor	`new Socket("localhost", 5000)`
Paso de mensajes	Medio (IPC clásico)	Los procesos se envían mensajes estructurados a través del SO	Sistemas distribuidos que necesitan coordinar tareas	`cola.enviar()` / `cola.recibir()` *(pseudocódigo)*
Base de datos compartida	Medio (IPC indirecto)	Los procesos comparten datos a través de registros persistentes	Aplicaciones que leen/escriben datos comunes	`INSERT / SELECT` sobre una tabla
RPC (Remote	Alto (sobre	Un proceso llama funciones	Aplicaciones distribuidas	`servidor.sumar(2,3)` devuelve `5`

Mecanismo	Nivel	Qué hace	Uso típico	Ejemplo breve
Procedure Call)	IPC)	que se ejecutan en otro proceso	cliente-servidor	
Servicios web (REST, SOAP)	Alto (sobre IPC/RPC)	Permiten exponer y consumir servicios mediante mensajes estructurados (HTTP/JSON…)	Sistemas distribuidos a través de Internet	Petición `HTTP GET` a `https://api.datos.com`

⚙️ Clase Runtime en Java

La clase `Runtime` forma parte de la librería estándar de Java y permite **interactuar con el entorno de ejecución de la JVM**.
Ofrece acceso a **funciones del sistema operativo subyacente**, como ejecutar comandos, obtener información del sistema o finalizar la JVM.

📌 Características clave

1. **Obtener la instancia de Runtime**
   - Sigue el patrón *Singleton* → solo hay una instancia disponible.
   - Se obtiene con `Runtime.getRuntime()`.
2. **Ejecutar comandos del sistema**
   - Método `exec()` → ejecuta comandos del sistema operativo desde un programa Java.
   - Puede pasarse como `String` o como array de argumentos.
3. **Capturar la salida del proceso**
   - Se lee lo que el proceso genera por consola mediante `InputStream` y `BufferedReader`.
4. **Controlar el consumo de memoria**
   - Permite consultar memoria disponible o usada por la JVM (`freeMemory()`).
5. **Finalizar la JVM**
   - `exit(int)` → finaliza la JVM (0 = finalización correcta; otro valor = error).
   - `halt(int)` → finaliza de forma abrupta (sin ejecutar bloques `finally`).

📋 Principales métodos de Runtime

Método	Funcionalidad
`static Runtime getRuntime()`	Devuelve el objeto `Runtime` asociado a la aplicación actual.
`Process exec(String command)`	Ejecuta un comando del sistema operativo.

Método	Funcionalidad
`Process exec(String[] commandArgs)`	Ejecuta un comando con argumentos.
`Process exec(String command, String[] envp)`	Ejecuta un comando indicando variables de entorno.
`Process exec(String[] cmdarray, String[] envp, File dir)`	Ejecuta un comando con entorno y directorio de trabajo.
`void exit(int status)`	Termina la ejecución de la JVM (0 = normal, ≠0 = anormal).
`void halt(int status)`	Termina la JVM de forma inmediata y abrupta.
`long freeMemory()`	Devuelve la cantidad de memoria libre en la JVM.
`int availableProcessors()`	Devuelve el número de procesadores disponibles para la JVM.

⚙️ Clase Process en Java

Cuando se ejecuta un comando con `exec()`, se obtiene un objeto `Process`.
La clase `Process` representa un **proceso en ejecución iniciado por** `Runtime` **o** `ProcessBuilder` y permite **interactuar con él**.

📌 Funcionalidades clave

- Obtener la **salida estándar del comando** (`getInputStream()`).
- Obtener la **salida de errores** (`getErrorStream()`).
- **Enviar datos de entrada** al proceso si los necesita (`getOutputStream()`).
- **Esperar a que termine el proceso** (`waitFor()`).
- **Comprobar si sigue activo** (`isAlive()`).
- **Finalizarlo manualmente** (`destroy()`).

📋 Principales métodos de Process

Método	Funcionalidad
`int exitValue()`	Devuelve el código de salida del proceso (error si aún no ha terminado).
`void destroy()`	Finaliza el proceso de forma forzosa.
`InputStream getInputStream()`	Devuelve un flujo para leer la salida estándar del proceso.

Método	Funcionalidad
`InputStream getErrorStream()`	Devuelve un flujo para leer la salida de error estándar del proceso.
`OutputStream getOutputStream()`	Devuelve un flujo para enviar datos a la entrada estándar del proceso.
`boolean isAlive()`	Indica si el proceso aún está en ejecución.
`int waitFor()`	Espera a que el proceso finalice y devuelve su código de salida.

⚠️ **Consideraciones de seguridad**

- Al ejecutar comandos del sistema, hay que validar las entradas para evitar **ejecución de comandos maliciosos** que puedan comprometer la aplicación.

📌 **Resumen clave**

- `Runtime` → ejecuta comandos del SO y gestiona recursos de la JVM.
- `Process` → representa y controla el proceso externo lanzado con esos comandos.

📝 Ejercicio 1: Java Runtime y Process

🎯 Objetivo del ejercicio

Crear un programa en Java que:

- Pida por pantalla la **ruta de una carpeta**.
- Ejecute el comando del sistema (`dir` en Windows o `ls` en Linux/Unix) sobre esa carpeta usando **las clases** `Runtime` y `Process`
- Muestre por pantalla la **información devuelta por el comando**.

✅ Requisitos del ejercicio:

- Desarrollo en IntelliJ
- Usa las clases **Runtime** y **Process** que se describen a continuación.

ProcessBuilder en Java

La clase **ProcessBuilder** proporciona una forma más **flexible y potente** de crear y gestionar procesos del sistema operativo desde una aplicación Java, en comparación con `Runtime.exec()`.

Permite controlar aspectos como:

- Entrada y salida del proceso

- Directorio de trabajo
- Variables de entorno

Es útil cuando se necesitan ejecutar **programas externos o comandos del sistema operativo** y controlar cómo se ejecutan e interactúan con el sistema.

Principios de funcionamiento

- Ejecuta comandos del sistema como si fuese una terminal.
- Los comandos dependen del sistema operativo (ej.: `ls`, `cat` en Unix/Linux o `dir`, `ipconfig` en Windows).

Características principales

- **Construcción del proceso**
  Configurar el comando o los comandos a ejecutar antes de iniciarlo.
- **Personalización del entorno de ejecución**
  Configurar variables de entorno y el directorio de trabajo del proceso.
- **Redirección de entrada/salida**
  Redirigir la salida y la entrada del proceso a archivos u otros flujos.
- **Gestión avanzada de errores**
  Redirigir la salida estándar y la de error a diferentes flujos o combinarlas para facilitar la depuración.

ProcessBuilder: Creación y ejecución de un proceso

1. Construcción del proceso

Un proceso se define por una lista de comandos y sus argumentos.
El primer elemento de la lista es el comando a ejecutar (por ejemplo, `ls`, `dir`), y los siguientes elementos son los argumentos de ese comando (por ejemplo, `-l` para listar los archivos en formato detallado).

```
ProcessBuilder builder = new ProcessBuilder("ls", "-l");
ProcessBuilder builder = new ProcessBuilder("cmd.exe", "/c", "dir");
```

2. Ejecutar el proceso

Una vez que se ha configurado el proceso, puedes **iniciarlo** utilizando el método `start()`.
Este método devuelve una instancia de la clase `Process`, que representa el proceso en ejecución.

```
Process process = builder.start();
```

3. Capturar la salida del proceso

Puedes capturar la **salida estándar** del proceso, que es el equivalente a lo que verías en la terminal cuando ejecutas un comando manualmente.

Para ello, puedes leer desde el **InputStream** del proceso.

```
BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
String line;
while ((line = reader.readLine()) != null) {
    System.out.println(line);
}
```

4. Esperar a que el proceso termine

El método `waitFor()` detiene la ejecución del programa hasta que el proceso finaliza y devuelve el **código de salida** del proceso.

```
int exitCode = process.waitFor();
System.out.println("Proceso terminado con código: " + exitCode);
```

Redirección de la entrada, salida y error estándar

`ProcessBuilder` permite redirigir la entrada estándar (stdin), la salida estándar (stdout) y la salida de error (stderr) del proceso a diferentes destinos, como archivos o flujos.

```
ProcessBuilder builder = new ProcessBuilder("ls", "-l");
builder.redirectOutput(new File("salida.txt"));
builder.start();
```

Combinar salida estándar y salida de error

Puedes combinar la salida estándar y la salida de error para manejar ambos en un solo flujo.
Esto es útil para capturar tanto los resultados como los errores generados por el proceso.

```
builder.redirectErrorStream(true);
```

Especificar el directorio de trabajo del proceso

Si el proceso necesita ejecutarse desde un directorio específico, puedes configurar el directorio de trabajo usando el método `directory()`.

```
builder.directory(new File("/ruta/a/mi/directorio"));
```

Modificar el entorno del proceso

La clase `ProcessBuilder` te permite modificar o añadir **variables de entorno** para el proceso que se va a ejecutar.
Esto es útil si necesitas que el proceso tenga acceso a ciertas variables del sistema.

```
Map<String, String> environment = builder.environment();
environment.put("MI_VARIABLE", "valor");
```

Ventajas de ProcessBuilder sobre Runtime.exec()

1. **Mayor flexibilidad**
   Permite un mayor control sobre la ejecución del proceso, incluyendo la redirección de entrada y salida, el directorio de trabajo y las variables de entorno.
2. **Manejo de múltiples argumentos**
   Facilita la ejecución de comandos con múltiples argumentos y reduce los problemas de manejo de espacios o caracteres especiales.
3. **Mejor manejo de errores**
   Puedes redirigir y capturar fácilmente los errores generados por el proceso.

Métodos principales de ProcessBuilder

Método	Funcionalidad
`ProcessBuilder(List<String> command)`	Constructor que toma una lista de comandos y argumentos para el proceso
`start()`	Inicia el proceso configurado y devuelve una instancia de la clase `Process`
`redirectOutput(File file)`	Redirige la salida estándar del proceso a un archivo
`redirectError(File file)`	Redirige la salida de error del proceso a un archivo
`redirectInput(File file)`	Redirige la entrada estándar del proceso desde un archivo
`directory(File dir)`	Establece el directorio de trabajo desde donde se ejecutará el proceso
`environment()`	Devuelve un mapa que representa el entorno del proceso y permite modificar variables de entorno
`redirectErrorStream(boolean)`	Si se establece en `true`, combina la salida estándar y la salida de error en un solo flujo

📝 Ejercicio 2: ProccessBuilder

Escribe un programa en **Java** que solicite por pantalla la **ruta de una carpeta** de la que se desea obtener información y, haciendo uso del comando `dir` o `ls` (según el sistema operativo), **muestre la información** tras su ejecución.

Consideraciones adicionales

- Se **almacenará la información** en un archivo **/data/out/ejercicio2.log** y se indicará por pantalla unicamente **la ubicación del archivo** que contiene la información.
- **En la última línea del archivo con el resultado se indicará el PID** del proceso que generó la información.

- Es **obligatorio** utilizar la clase `ProcessBuilder` y el metodo **directory()** para cambiar la ruta del proceso.

✅ Requisitos:

- Usa las clases ProcessBuilder y Process
- Usa PrintWriter para imprimir la salida en el fichero.

Invocar un programa Java de manera externa

¿Qué es un JAR?

- **JAR (Java ARchive)** es un archivo comprimido que contiene clases compiladas (`.class`), recursos (imágenes, ficheros de texto, etc.) y un **MANIFEST.MF** que indica la configuración del programa.
- Permite **empaquetar y distribuir aplicaciones Java** en un solo archivo ejecutable.
- Si incluye una **clase**

`Main` **definida en el manifiesto**, se puede ejecutar con:

```
java -jar programa.jar arg0 arg1 argn
```

- Desde el main se recogen los argumentos en String [] args

```java
public class Main {
    public static void main(String[] args) {
        // Comprobar que se pasa al menos un argumento
        if (args.length == 0) {
            System.out.println("No se han pasado argumentos.");
            return;
        }

        // Recoger el primer argumento
        String arg0 = args[0];
        System.out.println("Primer argumento: " + arg0);
```

Crear un JAR en IntelliJ IDEA

1. Ir a **File → Project Structure... → Artifacts**
2. Pulsar **+ → JAR → From modules with dependencies...**
3. Seleccionar la clase principal (

`Main`) como **Main Class**
   - IntelliJ generará automáticamente el archivo **MANIFEST.MF**, donde quedará registrada la clase principal.
   - Este archivo se guarda dentro del JAR y es el que indica qué clase debe ejecutarse.

4. Aceptar y aplicar los cambios
5. Ir a **Build → Build Artifacts → Build**
6. El archivo `.jar` aparecerá en:
   `out/artifacts/nombre/`

📝 Ejercicio 2B: Ejecuta un Jar desde ProccessBuilder

Objetivo: Comprender el uso de `ProcessBuilder` para invocar un programa Java desde otro.

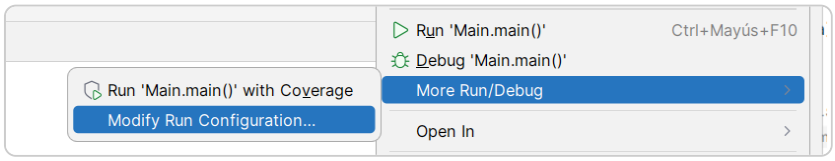
Crea un nuevo proyecto en **IntelliJ IDEA** denominado `listaruta` con las siguientes características:

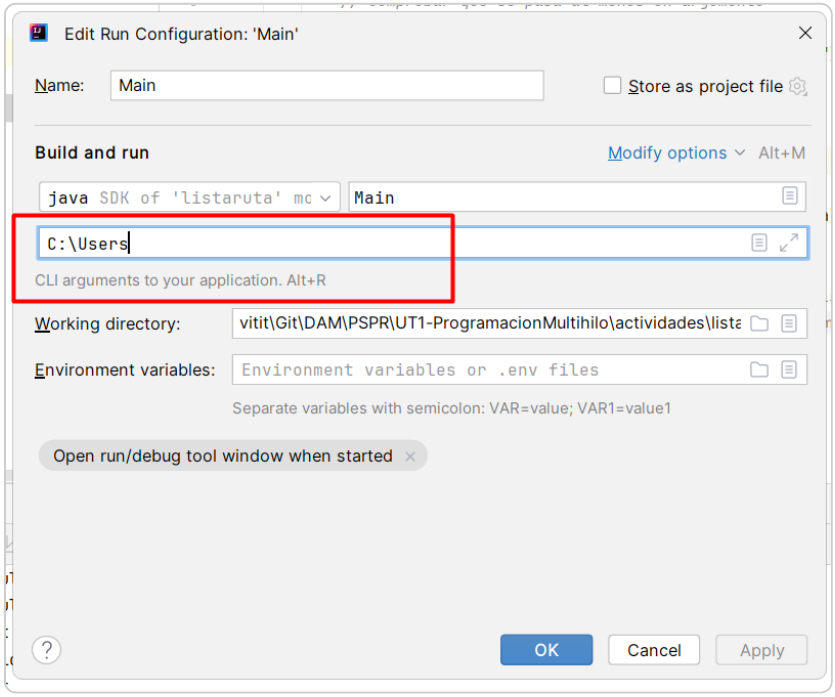
1. Define una clase `Main` que contenga el método `public static void main(String[] args)`.
2. El programa debe recibir **por argumento** la **ruta a un directorio**.
3. Si no se pasa ningún argumento, el programa debe mostrar un mensaje de error indicando que falta la ruta.
4. Si la ruta existe y es un directorio, el programa debe listar su contenido (archivos y subdirectorios), de forma similar al ejercicio anterior donde utilizaste `dir`.
5. Si la ruta no existe o no es un directorio válido, el programa debe mostrar un mensaje de error.

📌 Desde terminal se ejecutaría así:

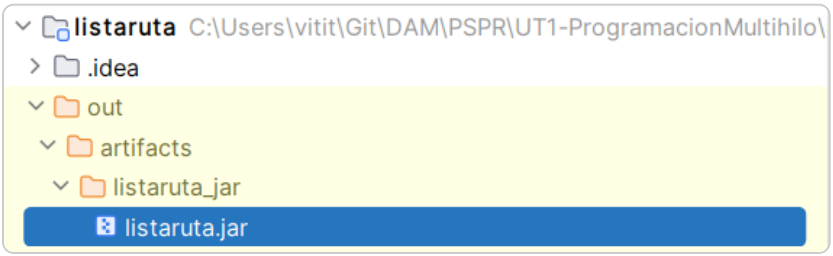
```
java -jar listaruta.jar "C:/Users/Alumno/Documents"
```

📌 Para ejecutarlo en intellij podemos indicar los argumentos desde `Run Configuration...`





6. **Genera el archivo/artifact** listaruta.jar



Desde el proyecto original realiza la llamada al jar utilizando la clase ProcessBuilder.

1. **Copia el archivo** `listaruta.jar` generado en el proyecto anterior dentro de la carpeta del nuevo proyecto (por ejemplo en `/libs`).
2. Crea una clase Main con un método `main`.
3. En este método, utiliza la clase `ProcessBuilder` para invocar el programa `listaruta.jar`.

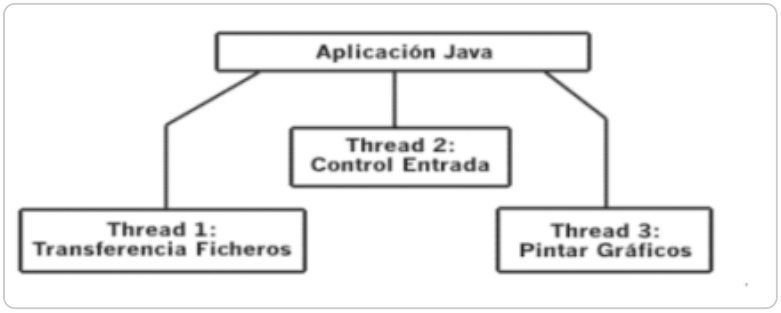
```
// Ruta a listar
  String ruta = "C:/Users/Alumno/Documents";

  // Crear el proceso para ejecutar el jar externo
  ProcessBuilder pb = new ProcessBuilder(
      "java", "-jar", "listaruta.jar", ruta
  );

  // Directorio donde está el JAR
  pb.directory(new File("libs"));

  // Redirigir salida/errores a la consola actual
  pb.inheritIO();

  // Iniciar el proceso
  Process p = pb.start();

  // Esperar a que termine
  int exitCode = p.waitFor();
```

📝 RA1d) Hilos de ejecución: relación con los procesos

- Los **Hilos** tambien llamados **procesos ligeros o hebras comparten el contexto del programa/proceso**.
- El **hilo es iniciado por el propio programa**, por lo que **es éste quien se encarga de su gestión y sincronización**.
- Cuando se inicia un programa en Java, **se crea un proceso con un hilo principal que invoca al método** `main()`.
- Este **hilo principal terminará al finalizar el** `main()`.
- Si el **hilo principal crea otros hilos, se ejecutarán de manera concurrente**.
- **Qué es un hilo:** Es una **unidad de ejecución dentro de un proceso**.
- **Relación con los procesos:**
  - Un proceso puede tener **uno o varios hilos** que comparten su misma memoria.
  - Todos los hilos de un proceso pueden acceder a sus variables y recursos.

- Los hilos permiten realizar **tareas concurrentes dentro del mismo proceso**.

- **Características:**
  - Comparten el mismo espacio de direcciones y recursos del proceso.
  - Son más ligeros que los procesos.
  - Necesitan sincronización para evitar conflictos al acceder a datos compartidos.



Programación concurrente (gestionando hilos)

La **programación concurrente** es una técnica que permite **generar distintos procesos o hilos simultáneos** que **colaboran mediante mecanismos de coordinación y sincronización para resolver un problema común**.

Permite la **ejecución de varias actividades a la vez (en paralelo lógico)**, **optimizando el uso de los recursos del procesador**.
📝 *Ejemplo: mientras proceso solicita realizar una opertacion de E/S, otros hilos pueden aprovechar el procesador para realizar otras operaciones.*

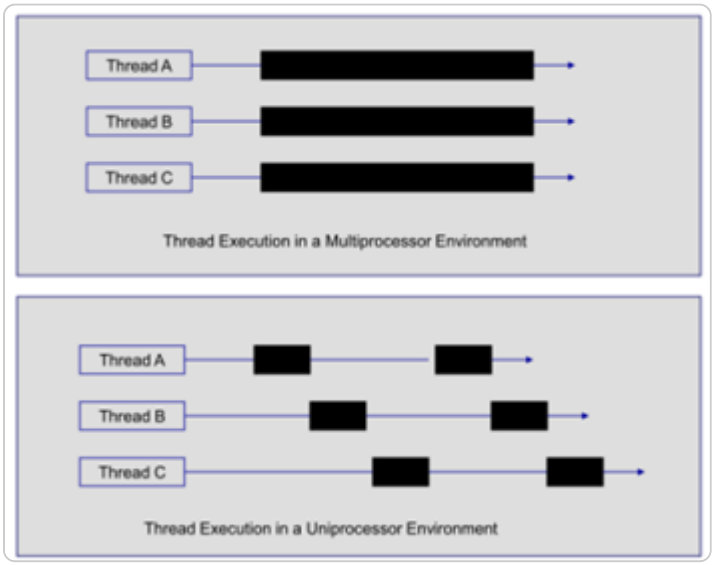
Operaciones básicas de la programación concurrente:

- 🧩 **Definición de procesos/hilos**
- ⏳ **Sincronización de procesos/hilos**
- 💬 **Comunicación/comparticion de información entre procesos/hilos**

Ventajas de la implementación de hilos

1. **Mayor capacidad de respuesta**
   - Permiten atender varias tareas a la vez (E/S y cálculos), mejorando la fluidez y rapidez del sistema.
2. **Uso eficiente de los recursos**
   - Comparten la memoria y recursos del proceso, evitando el coste de crear múltiples procesos independientes.
   - Paralelizacion de tareas, lo que mejora tiempos de respuesta (baja latencia).
3. **Comunicacion entre ellos**
   - Todos los hilos de un mismo proceso comparten el mismo espacio de memoria, así que pueden leer y escribir en las mismas variables y estructuras de datos directamente, sin mecanismos complejos.
   - Esto hace que la comunicación entre hilos sea rápida y sencilla.
4. **Mayor modularidad y facilidad de programación**
   - Facilitan el desarrollo de aplicaciones concurrentes al compartir datos y coordinar tareas de forma sencilla.

- **Ejemplo en Java:** Clases `Thread`, `Runnable`, `ExecutorService`.



Ejecucion de hilos en entorno paralelo real y pseudoparalelo

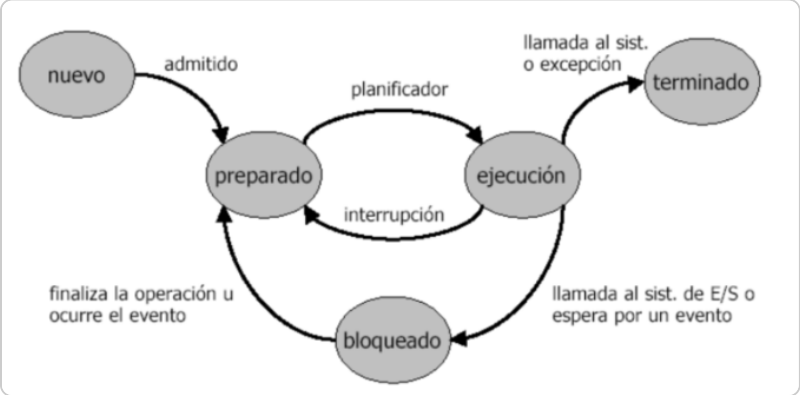
🧠 RA2a) Situaciones útiles de uso de hilos

- Cuando se desea **ejecutar varias tareas de forma simultánea** para mejorar el rendimiento o la capacidad de respuesta.
- Situaciones comunes:
  - **Interfaces gráficas**: mantener la UI activa mientras se hacen tareas en segundo plano.
  - **Servidores**: atender múltiples peticiones de clientes a la vez.
  - **Juegos o simulaciones**: gestionar varios elementos (IA, físicas, renderizado...) de forma concurrente.
  - **Procesamiento de grandes volúmenes de datos** en paralelo.
- Ventajas:
  - Mayor aprovechamiento de los núcleos de CPU.
  - Reducción de tiempos de espera (bloqueos de red, E/S...).

🔁 RA2d) Estados de ejecución de un hilo y su gestión

Los hilos pasan por varios **estados gestionados por la JVM y el sistema operativo**: En Java se utilizan la interfaz `Runnable` y clase `Thread` para la gestion de hilos.

- **NEW (Nuevo):** creado con `new Thread()`, pero no iniciado aún.
- **RUNNABLE (Preparado/listo):** iniciado con `start()`, esperando turno para ejecutarse. Si ocurre una interrupcion (por ejemplo, por la finalizacion de una operacion de E/S) vuelve a este estado.
- **RUNNING (En ejecución):** actualmente ejecutando su código en la CPU.
- **WAITING / TIMED_WAITING (Esperando):** suspendido temporalmente (por `sleep()`, `join()`, `wait()`, etc.).
- **BLOCKED (Bloqueado):** esperando a acceder a un recurso o a un evento (E/S...). Por ejemplo, el hilo solicita una operación de entrada/salida (E/S).
- **TERMINATED (Finalizado):** ha terminado su ejecución.



Ejemplo de gestión de estados en Java (clase Thread):

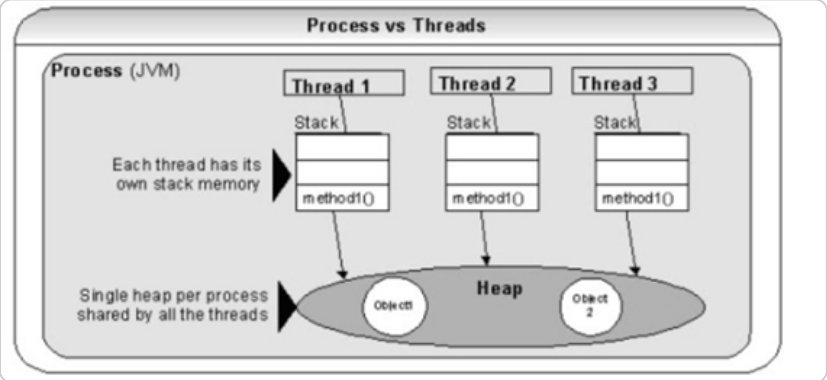
- `start()` → pasa de NEW a RUNNABLE.
- `sleep(ms)` → pausa un hilo en ejecución (pasa a TIMED_WAITING).
- `join()` → hace que un hilo espere a que termine otro.
- `interrupt()` → interrumpe un hilo en espera o en ejecución.
- No se puede reiniciar un hilo que ya ha terminado (TERMINATED).

⚖️ RA2g) Prioridad de hilos de ejecución

- La prioridad de los hilos en Java es solo una sugerencia para la JVM y puede influir en el orden de ejecución **entre hilos de la misma aplicación**, pero **el sistema operativo puede ignorarlas**, por lo que **no garantizan ningún orden real de ejecución.**
- Cada hilo tiene un **nivel de prioridad numérico** entre `Thread.MIN_PRIORITY (1)` y `Thread.MAX_PRIORITY (10)`.
- Por defecto es `Thread.NORM_PRIORITY (5)`.
- El **planificador de hilos** del sistema operativo **intenta dar más tiempo de CPU a los de mayor prioridad**, aunque no lo garantiza.
- Se establece con `thread.setPriority(nivel)`.
- Se usa solo como **sugerencia**, no asegura el orden exacto de ejecución.

⚙️ RA2i) Contexto de ejecución de los hilos

- Es el **entorno en el que se ejecuta un hilo**:
  - Variables y memoria compartida del proceso.
  - Registros, contador de programa y pila propios de cada hilo.
- Cada hilo mantiene su propio **stack (pila de ejecución)**, pero **comparte el heap (memoria dinámica)** con otros hilos del mismo proceso.
- Cambiar entre hilos implica un **cambio de contexto**, que guarda y restaura el estado del hilo.



⚠️ RA2k) Problemas al compartir información entre hilos

- Como los hilos comparten memoria, pueden producirse **condiciones de carrera** si varios acceden a los mismos datos sin control.
- Problemas comunes:
  - **Race conditions:** resultados impredecibles por accesos simultáneos.
  - **Deadlocks:** dos hilos se bloquean mutuamente esperando recursos.
  - **Starvation:** un hilo nunca obtiene CPU o recursos porque otros acaparan.
  - **Inconsistencia de datos:** lectura de datos modificados a medias.
- Para evitarlo:
  - Uso de `synchronized`, `Lock`, `Semaphore`, estructuras de datos o colecciones que soporten concurrencia, por ejemplo del paquete `java.util.concurrent`., etc.
  - Diseñar secciones críticas correctamente.

Programación Multihilo en Java

Creación de Hilos en Java

Java proporciona dos maneras principales de crear hilos: **extendiendo la clase** `Thread` o **implementando la interfaz** `Runnable`.

🔹 Extender `Thread`:

- Esta técnica implica crear una subclase de `Thread` y sobrescribir el método `run` para definir la tarea que el hilo debe ejecutar.
- Es simple y adecuada para tareas específicas y aisladas, pero limita la herencia de la clase, ya que en Java no es posible heredar de más de una clase.
- Es la menos usada.

```
public class MiHilo extends Thread {
    @Override
    public void run() {
        System.out.println("Hilo ejecutado extendiendo Thread");
```

```
    }
}

public class Main {
    public static void main(String[] args) {
        MiHilo hilo = new MiHilo();
        hilo.start(); // Inicia el hilo
    }
}
```

🔹 Implementar `Runnable`:

- Se crea una clase que implementa la interfaz `Runnable` y define la tarea en el método `run`.
- La ventaja de `Runnable` es que permite que la clase implemente otras interfaces o extienda de otra clase, brindando mayor flexibilidad y modularidad.

```
public class MiRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Hilo ejecutado implementando Runnable");
    }
}

public class Main {
    public static void main(String[] args) {
        Thread hilo = new Thread(new MiRunnable());
        hilo.start(); // Inicia el hilo
    }
}
```

🔹 Función o expresión Lambda

La **lambda define el método** `run()`. Es útil para crear hilos de forma más simple y concisa.

```
public static void main(String[] args) {
    Thread hilo = new Thread(() -> {
        System.out.println("¡Hola desde un hilo con lambda!");
    });

    hilo.start();
}
```

Metodos clave de la clase Thread

Método	Funcionalidad
`start()`	Inicia el hilo, creando un nuevo hilo de ejecución y llamando internamente al método `run()` en paralelo.
`run()`	Define el código que se ejecutará en el hilo. **No debe llamarse directamente**, ya que en ese caso no se creará un nuevo hilo (se ejecuta en

Método	Funcionalidad
	el hilo actual).
`sleep(long millis)`	Hace que el hilo actual se "duerma" durante el tiempo especificado en milisegundos, **liberando la CPU para que otros hilos puedan ejecutarse**.
`join()`	Hace que el hilo actual espere a que **otro hilo termine su ejecución** antes de continuar.
`interrupt()`	Marca el hilo como interrumpido. Si el hilo está en espera o dormido, **lanza** `InterruptedException` **y lo despierta**; si está activo, **solo cambia su estado a "interrumpido"** (no lo detiene automáticamente).

📝 Ejercicio 3: Trabajando con Hilos

Realiza un programa que demuestre el uso de **hilos concurrentes en Java** para simular varias tareas de impresión que se ejecutan al mismo tiempo.

📌 Clases

`TareaImpresion` **(implementa** `Runnable` **)**

- Representa una tarea de impresión.
- Tiene un atributo `nombre` para identificar el hilo.
- En su método `run()`:
  - Imprime 5 veces un mensaje `"nombre + Impresión i"`.
  - Entre cada impresión **duerme 1 segundo (** `Thread.sleep(1000)` **)** simulando el tiempo de impresión.
  - Si se interrumpe, captura la excepción `InterruptedException`.
  - Imprime por pantalla que ha finalizado el hilo `"nombre + Finalizado"`.

> 📍 Esto define lo que hará cada hilo cuando se ejecute.

```
@Override
   public void run() {
       System.out.println(nombre + " Iniciado ");

       //TODO: Realiza las 5 impresiones //

       System.out.println(nombre + " Finalizado");

   }
```

`Main`

- Crea 3 tareas `TareaImpresion` (`tarea1`, `tarea2`, `tarea3`) con nombres distintos.
- Crea 3 objetos `Thread` a partir de esas tareas.
- Llama a `start()` en cada hilo para ejecutarlos **en paralelo**.
- Llama a `join()` sobre cada hilo para que el **hilo principal espere a que los tres terminen antes de finalizar el programa.**
- Imprime por pantalla que ha finalizado el hilo principal.

```
public static void main(String[] args) {
	System.out.println("Hilo principal Iniciado ");

	// Crear instancias de tareaimpresion
	TareaImpresion tarea1 = new TareaImpresion("Hilo 1");
	TareaImpresion tarea2 = new TareaImpresion("Hilo 2");
	TareaImpresion tarea3 = new TareaImpresion("Hilo 3");

	// Crear hilos para ejecutar las tareas
	Thread hilo1 = new Thread(tarea1);
	Thread hilo2 = new Thread(tarea2);
	Thread hilo3 = new Thread(tarea3);

	//TODO: Iniciar los hilos
	//TODO: El hilo principal debe esperar a que terminen

	System.out.println("Hilo principal Finalizado ");
}
```

Resultado de la Ejecución

- Los tres hilos imprimirán sus mensajes **de forma concurrente (intercalada)**.
- Cada hilo hace 5 impresiones con pausas de 1 segundo entre ellas.
- El hilo principal no avanza hasta que los tres hilos terminen gracias a `join()`.

✅ Requisitos:

- Usa la interfaz Runnable y la clase Thread

Sincronización de hilos

La **sincronización es crucial cuando varios hilos acceden a recursos compartidos**.
Sin ella, pueden producirse **condiciones de carrera**, donde el resultado depende del orden en que los hilos acceden a los recursos.

Problemas en la sincronizacion

Cuando varios hilos acceden a **recursos compartidos** (listas, objetos, buffers, variables), pueden aparecer:

- **Condiciones de carrera (Race conditions)** → resultados aleatorios.
- **Bloqueos (DeadLock)** → un hilo espera indefinidamente.

Condiciones de carrera

- Se producen cuando **varios hilos acceden y modifican un recurso compartido sin coordinación**, "compiten" por un mismo recurso.
- Esto puede provocar **resultados inesperados o erróneos**.
- Por ejemplo, varios hilos acceden a datos de variables o ficheros compartidos desde distintos métodos, si lo hacen al mismo tiempo pueden existir inconsistencias debido al distinto orden en que se produzcan lecturas y modificaciones.
- **Datos inconsistentes** → lecturas/escrituras simultáneas.

Mecanismos de sincronizacion en Java

Java proporciona varios mecanismos para sincronizar hilos

- Exclusion mutua `synchronized`
- Esperas y notificaciones `metodos wait()/notify()`
- Semaforos `clase Semaphore`
- Uso de datos atomicos y colecciones synchronized/concurrent `Ej: AtomicInteger, Collections.synchronizedList`
- Uso de la libreria `java.util.concurrent`.

Exclusion mutua o Bloqueo de sincronización (`synchronized`)

- Java ofrece la palabra clave `synchronized` para **sincronizar el acceso a métodos o bloques de código**.
- Se utiliza en secciones criticas donde varios hilos acceden a datos/objetos compartidos
- Esto garantiza que **solo un hilo pueda acceder a la sección sincronizada a la vez**, protegiendo los recursos compartidos.
- Es decir, fuerza a que el metodo se ejecute de manera sincronizada cuando 2 o mas hilos pretenden ejecutar la seccion crítica.

📌 Ejemplo:

```
public synchronized void incrementar() {
    contador++; // variable compartida
}
```

📌 Recuerda...

- Cada **hilo tiene su propio stack (pila)**:
  - Aquí se almacenan **variables locales y llamadas de métodos**.
  - **Nadie más puede acceder a este stack**, es privado del hilo.
- Las **variables compartidas entre hilos están en el heap (montón)**:
  - Aquí se guardan los **objetos y sus atributos**.
  - **Todos los hilos pueden acceder al mismo objeto en el heap.**

📝 Ejercicio 4: Exclusión mutua. Sincronización de hilos con `synchronized`

Partiendo del ejercicio anterior, en el que cada hilo imprimía varios mensajes de forma concurrente, vas a **añadir un recurso compartido entre todos los hilos** y **proteger su acceso usando** `synchronized`.

🧩 Instrucciones

1. Añade en la clase `TareaImpresion` una **variable estática** `int totalImpresiones` que represente el **número total de impresiones realizadas por todos los hilos.**
2. Cada vez que un hilo imprima un mensaje, **incrementará** `totalImpresiones` **en 1.**
3. Protege el acceso a `totalImpresiones` usando un **bloque** `synchronized` para evitar que varios hilos la modifiquen a la vez.
4. Haz que cada hilo imprima también el valor actualizado de `totalImpresiones` tras incrementarlo.
5. Ejecuta el programa varias veces y comprueba que **los valores de** `totalImpresiones` **son siempre consistentes y no se pierden incrementos.**
6. Muestra el **total de impresiones al finalizar el hilo principal**.

📌 Nota

- Sin `synchronized`, podrían producirse **condiciones de carrera** que harían que el contador se actualice incorrectamente.

📝 Ejercicio 5: Compartiendo objetos entre hilos

Modifica el ejercicio anterior para que los hilos compartan un **objeto contador** (en vez de la variable `static` int contador).

- Crea una clase `Contador` con un atributo `int`
- Crea **2 metodos**:
  - `incrementar()` : método `synchronized` para incrementarlo.
  - `getTotal()` : método `synchronized` para devolver el valor.
- Pasa la **misma instancia de** `Contador` **a todos los hilos** desde Main en cada constructor.
- Cada hilo, tras imprimir su mensaje, debe **usar el contador compartido para incrementar y mostrar el total de impresiones realizadas.**

EXTRA: Depurar

Practica la depuración del ejercicio del **objeto compartido entre hilos**.

1. **Configura breakpoints** en `TareaImpresion.run()` :
   - En la línea donde se imprime `"<nombre> - Impresión i"` .
   - Justo antes de llamar a `contadorCompartido.incrementar()` .
2. **Depura el programa** (modo Debug en IntelliJ):
   - El **valor del contador compartido en la tercera impresión del** `Hilo 2` (cuando `i == 3` para `Hilo 2`) .
3. **Puedes añadir observadores (Watches) y condiciones** en el depurador para pararlo en el punto concreto:
   - `i` , `nombre` , `this.contadorCompartido` , `contadorCompartido.totalImpresiones` *(si es* `private` *, usa el getter o inspección de objeto en el depurador).*

Coordinacion de hilos con esperas y notificaciones

Java proporciona un mecanismo adicional de **espera y notificación** para coordinar hilos.

- `wait()`
  - El hilo **se bloquea**.
  - Solo puede usarse dentro de un bloque `synchronized`.
  - El hilo se queda dormido hasta que otro lo despierte.
- `notify()`
  - Despierta a **un hilo** en espera.
  - Si hay varios hilos esperando, no se garantiza cuál.
- `notifyAll()`
  - Despierta a **todos los hilos** que están esperando.

Son **metodos de la clase padre Object** por lo que están disponibles desde cualquier clase.

Ventajas :

- **Mejora la comunicación entre hilos**: notify() y notifyAll() permiten que un hilo avise a otros cuando hay cambios (ej: "nuevo pedido disponible").
- Permite tener varios hilos productores y consumidores **trabajando en paralelo sin pisarse**. (ej: los productores añaden a una lista de pedidos y los consumidores procesan los pedidos)
- **Uso eficiente de los recursos**: Los hilos esperan dormidos con wait() cuando no pueden trabajar y se despiertan en el momento preciso con notifyAll().

📝 Ejercicio6: Productor–Consumidor con varios hilos

El **problema productor-consumidor** es un ejemplo clásico de **programación concurrente**.
Se trata de **coordinar** a varios hilos que comparten un recurso común: un **buffer** (normalmente una cola).

- **Hilos Productores** → generan datos y los introducen en el buffer.
- **Hilos Consumidores** → extraen datos del buffer y los procesan.
- **Buffer** → tiene **capacidad máxima**, por lo que no puede crecer indefinidamente.

🔹 Retos principales

1. **Exclusión mutua** → evitar que dos hilos accedan al buffer a la vez.
2. **Buffer lleno** → los productores deben esperar.
3. **Buffer vacío** → los consumidores deben esperar.
4. **Varios hilos** → pueden existir múltiples productores y múltiples consumidores.
5. **Finalización ordenada** → el `main` debe esperar a que todos los hilos terminen usando `join()`.

🔹 Conceptos clave

- `synchronized` → asegura acceso exclusivo al buffer.
- `wait()` → suspende el hilo hasta que la condición cambie.
- `notifyAll()` → despierta a todos los hilos en espera (productores o consumidores).
- `join()` → hace que el **hilo principal** espere a que terminen los hilos secundarios.

```
// Clase Buffer (Recurso compartido)
class Buffer {
    private Queue<Integer> cola = new LinkedList<>(); // Cola compartida
    private int capacidadMax = 5; // Capacidad máxima del buffer

    // Añade un elemento la cola, por tanto debe ser syncronized //
    public synchronized void producir(int valor) throws InterruptedException {
        //TODO Mientras la cola este llena, no se produce//
        {
           //TODO Espera
        }
        //TODO Añadir a la cola el valor
        System.out.println("Producido: " + valor);

        //TODO Avisamos a los hilos dormidos esperando (los consumidores)
    }

    // Lee un elemento de la cola, por tanto debe ser syncronized //
    public synchronized int consumir() throws InterruptedException {
        //TODO Mientras la cola esta vacia, no se consume //
        {
            //TODO Espera
        }
        //TODO recuperar el valor de la cabeza de la cola poll()
        System.out.println("Consumido valor: " + valor);
        //TODO Avisamos a los hilos dormidos esperando (los productores)
        return valor;
    }

  public Queue<Integer> getCola() {
        return cola;
    }

}

// Clase Productor
class Productor extends Thread {
    private Buffer buffer; // buffer compartido
    private Contador contador; // Contador de valores compartido

     public Productor(Buffer buffer, Contador contador) {
        this.buffer = buffer;
        this.contador = contador;
    }

    public void run() {
        try {
                System.out.println(getName() + " Iniciado ");

                //TODO producir(valor), pasamos el valor incrementado del contador
                //Simula tiempo de producir
                Thread.sleep(1000);

                System.out.println(getName() + " Finalizado ");
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}
```

```
// Clase Consumidor
class Consumidor extends Thread {
    private Buffer buffer;// buffer compartido
    public Consumidor(Buffer buffer) { this.buffer = buffer; }

    public void run() {
        try {
            System.out.println(getName() + " Iniciado ");

                //TODO consumir
                //Simula tiempo de consumir
                Thread.sleep(1000);

            System.out.println(getName() + " Finalizado ");

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

// Clase Principal
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Buffer buffer = new Buffer();

        // Crear varios productores
        Productor[] productores = new Productor[2];
        //TODO configurar e iniciar hilos productores

        // Crear varios consumidores
        Consumidor[] consumidores = new Consumidor[2];
        //TODO Configurar e iniciar hilos consumidores

        //TODO Esperar a que terminen todos los hilos

        System.out.println("Todos los productores y consumidores han terminado.");
        System.out.println("¿El Buffer está vacio? " + buffer.getCola().isEmpty());
    }
}
```

⚠️ Deadlock (Bloqueo de la muerte)

Un **deadlock** es una situación en la que **todos los hilos están bloqueados esperando**, y ninguno puede avanzar.
El programa queda **congelado** aunque no haya terminado.

🔹 ¿Cuándo puede ocurrir en Productor–Consumidor?

1. **Uso incorrecto de** `notify()`
   - Si hay **varios hilos** (ej. 3 productores y 3 consumidores) y usamos solo `notify()`, puede despertarse un hilo del **mismo tipo** que el que ya estaba activo.
   - Ejemplo:

        - El buffer está lleno → todos los productores esperan.
        - Un consumidor consume 1 elemento y hace `notify()`.
        - Se despierta otro **productor** (pero sigue lleno) → vuelve a esperar.
        - Los consumidores nunca se despiertan → **deadlock**.
    - ✅ Solución: usar `notifyAll()`.
2. **Error en las condiciones de espera**
    - Si en lugar de `while` usamos `if`:

```
if (cola.isEmpty()) wait();
```

    - Al despertar el hilo es importante que vuelva a realizarse la comprobacion del buffer.
    - Puede ocurrir que varios consumidores se despierten a la vez, y al entrar uno de ellos el buffer quede vacío otra vez.
    - Los demás intentarían consumir de un buffer vacío → se bloquean sin salida.
    - ✅ Solución: usar siempre `while` en lugar de `if`.
3. **Productores y consumidores no balanceados**
    - Si los productores producen muchos menos elementos de los que esperan los consumidores, llegará un momento en que habrá consumidores que queden **bloqueados para siempre**.
    - ✅ Solución: controlar la **cantidad total de producción**.

Resumiendo...

El deadlock aparece cuando:

- ❌ Se usa `notify()` en lugar de `notifyAll()`.
- ❌ Se emplea `if` en vez de `while` en las condiciones de espera.
- ❌ Productores y consumidores no están balanceados.

Colecciones Syncronized

Proximamente...

Semaforos

Proximamente... Clase Semaphore de java

Libreria Concurrent

java.util.concurrent