1. ¿Qué es un hilo?

Un **hilo** es una *unidad de ejecución* dentro de un proceso. Un mismo programa puede ejecutar varios hilos simultáneamente para realizar tareas en paralelo. En Java pueden crearse de dos formas:

- **Extender `Thread`**
- **Implementar `Runnable`** (la forma recomendada, porque desacopla la lógica del hilo)

Cada hilo define su comportamiento en el método **`run()`**.

---

2. Creación y gestión de hilos

✔ Crear un hilo

Extendiendo `Thread`:

```
class Hilo extends Thread {
    public void run() { ... }
}
new Hilo().start();
```

Implementando `Runnable`:

```
class Tarea implements Runnable {
    public void run() { ... }
}
new Thread(new Tarea()).start();
```

✔ Suspender o dormir un hilo

- **`sleep(ms)`**: bloquea temporalmente un hilo.
- Métodos antiguos como `suspend` y `resume` están obsoletos.
- Lo correcto es gestionar una variable de parada o invocar **`interrupt()`**.

✔ Parar un hilo

- `stop()` está **depreciado** porque puede dejar recursos inconsistentes.
- Lo recomendable es:
    - lanzar una interrupción con `interrupt()`
    - o esperar la finalización con `join()`

---

3. Estados de un hilo

Los estados básicos de un hilo en Java son:

- **New**: creado pero no iniciado.
- **Runnable**: listo para ejecutarse (o ejecutándose).
- **Blocked**: esperando recursos (E/S, `sleep`, `wait`, bloqueo de monitor…).
- **Dead**: ya ha finalizado su `run()` o ha ocurrido una excepción.

Estos estados evolucionan mediante llamadas como `start()`, `sleep()`, `wait()`, `notify()`, etc.

---

4. Gestión de prioridades

- Cada hilo tiene una prioridad entre **1 y 10**.
- Se modifica con **`setPriority()`.**
- El planificador *puede* dar más CPU a los hilos con mayor prioridad, pero no es garantizado (depende del sistema operativo).
- Hoy en día **casi nunca se recomienda cambiar prioridades**.

---

5. Comunicación y sincronización entre hilos

La concurrencia sin control genera **condiciones de carrera**, es decir, accesos desordenados a recursos compartidos.

✔ Herramientas fundamentales:

🟦 `synchronized`

Protege secciones críticas para que solo un hilo pueda entrar a la vez.

Puede aplicarse a:

- **bloques sincronizados**
- **métodos enteros**

🟦 `wait()`, `notify()`, `notifyAll()`

- `wait()` → el hilo se suspende y libera el monitor.

- `notify()` → despierta *uno* de los hilos en espera.
- `notifyAll()` → despierta *todos* (más seguro y recomendado).

Estos métodos **solo se usan dentro de un bloque/método sincronizado**.

---

6. Métodos importantes de la clase `Thread`

Del README complementario:

🔹 **`isAlive()`**

Indica si el hilo se ha iniciado y aún no ha terminado.

🔹 **`toString()`**

Devuelve información sobre el hilo (nombre, prioridad, grupo).

🔹 **`getId()`**

Devuelve el identificador único del hilo.

🔹 **`yield()`**

Sugiere al planificador que el hilo actual ceda la CPU.

🔹 **`setPriority(int)`**

Cambia la prioridad del hilo (entre 1 y 10).

🔹 **`interrupt()`**

Interrumpe un hilo bloqueado o dormido, provocando `InterruptedException`.

🔹 **`join()`**

El hilo actual espera a que otro hilo termine.

---

7. El patrón Productor–Consumidor

Es un problema clásico de concurrencia:

- **Productor** → genera elementos.

- **Consumidor** → los procesa.
- **Búfer compartido (monitor)** → controla el acceso.

Problemas resueltos:

- Evitar que el productor produzca cuando el búfer está lleno
- Evitar que el consumidor consuma cuando está vacío
- Coordinar ambos hilos sin condiciones de carrera

Herramientas usadas:

- `synchronized`
- `wait()`
- `notifyAll()`
- Variables que indican si el búfer está lleno o vacío

Funcionamiento:

- Si el consumidor intenta consumir y **no hay datos**, hace `wait()`.
- Si el productor intenta producir y **el búfer está lleno**, hace `wait()`.
- Cada vez que se produce o consume un dato se ejecuta **`notifyAll()`**.

Se utiliza en colas de tareas, servidores web, apps móviles, mensajería y prácticamente cualquier solución concurrente moderna.

Conclusión general

El multihilo en Java permite que un programa realice varias tareas de forma concurrente. Para usarlo correctamente es esencial:

- Conocer la clase `Thread` y la interfaz `Runnable`
- Entender los estados de un hilo y cómo arrancarlo, suspenderlo o detenerlo
- Utilizar de forma segura `synchronized`, `wait()`, `notifyAll()`
- Dominar métodos clave como `join()`, `interrupt()`, `isAlive()` y `yield()`
- Comprender patrones fundamentales como **Productor–Consumidor**, base de muchas arquitecturas concurrentes reales