
RA2. Programació multafil en Java

Índex

1. Context d'execució dels fils	1
1.1 Procés vs. fil	1
1.2 Quan té sentit utilitzar fils?	1
1.3 Recursos compartits	1
2. Llibreries i classes per crear fils	2
2.1 Estendre la classe Thread	2
2.2 Implementar la interfície Runnable	3
2.3 Thread vs. Runnable: quina triem?	3
2.4 Diverses instàncies del mateix fil	4
3. Estats d'un fil	4
3.1 Exemple: observar els estats	5
3.2 Finalització correcta d'un fil	5
4. Gestió de fils: mètodes fonamentals	7
4.1 <code>join()</code> : esperar que un fil acabi	7
4.2 Fils daemon	8
5. Compartició d'informació entre fils i condicions de carrera	9
5.1 El problema	9
5.2 La solució: <code>synchronized</code>	10
5.3 Un exemple més realista: places lliures d'un pàrquing	11
6. Sincronització avançada: <code>wait()</code>, <code>notify()</code> i <code>notifyAll()</code>	12
6.1 El problema productor-consumidor	12
7. Gestió de prioritats	14
8. Més enllà de Thread: una mirada a <code>java.util.concurrent</code>	15
9. Depuració d'aplicacions multifil	16
Índex de figures	17

1. Context d'execució dels fils

1.1 Procés vs. fil

Un **procés** és un programa en execució amb el seu propi espai de memòria, els seus propis descriptors de fitxers i els seus propis recursos del sistema operatiu. Un **fil** (en anglès *thread*) és una línia d'execució que viu **dins** d'un procés. Un mateix procés pot tenir un o més fils, i tots els fils d'un procés **comparteixen** el mateix espai de memòria (variables estàtiques, heap, fitxers oberts...).

Per aquest motiu als fils se'ls anomena sovint **processos lleugers**: crear-los i canviar de context entre ells és molt més barat que fer-ho amb processos, perquè el sistema operatiu no ha de duplicar tot l'espai de memòria.

```
Procés
├── Fil principal (main)
├── Fil 1
└── Fil 2
```

1.2 Quan té sentit utilitzar fils?

Els fils són útils sempre que una part de la feina d'un programa pugui (o hagi de) executar-se sense bloquejar la resta. Alguns exemples típics:

- Un **servidor** que ha d'atendre diverses connexions de clients alhora (un fil per client).
- Una **interfície gràfica** que ha de continuar responent mentre es fa una operació lenta en segon pla (descàrrega, càlcul, accés a disc...).
- Un programa que **monitora diversos sensors** de forma independent i simultània.
- Un editor de text que **corregeix l'ortografia** mentre l'usuari continua escrivint.
- Una aplicació que **desa periòdicament** l'estat sense interrompre l'usuari.

1.3 Recursos compartits

Com que tots els fils d'un mateix procés comparteixen memòria, poden accedir als mateixos objectes. Això és un avantatge (permet comunicar-se fàcilment) però també la font del problema principal de la programació multifil: si dos fils **llegeixen i escriuen** el mateix recurs sense cap control, el resultat final pot dependre de l'ordre exacte en què el planificador del sistema operatiu decideixi executar-los, i aquest ordre no és determinista. Aquest fenomen es coneix com a **condició de carrera** (*race condition*) i el tractarem a Compartició d'informació entre fils i condicions de carrera.

NOTA

El planificador de fils del sistema operatiu decideix quin fil s'executa en cada moment i durant quant de temps. Un programa multifil ben dissenyat no ha de fer cap suposició sobre aquest ordre.

2. Llibreries i classes per crear fils

En Java, tot el suport per a fils viu al paquet `java.lang`, per la qual cosa no cal cap import addicional per a les classes bàsiques (`Thread`, `Runnable`). Hi ha **dues formes** de dotar una classe de comportament de fil:

Tècnica	Quan usar-la
Estendre <code>Thread</code>	Quan la classe no necessita heretar de cap altra
Implementar <code>Runnable</code>	Quan la classe ja hereta d'una altra classe, o quan volem separar la tasca del fil que l'executa

2.1 Estendre la classe `Thread`

```
public class ComptadorFil extends Thread {
    private final String etiqueta;
    private final int repeticions;

    public ComptadorFil(String etiqueta, int repeticions) {
        this.etiqueta = etiqueta;
        this.repeticions = repeticions;
    }

    @Override
    public void run() {
        for (int i = 1; i <= repeticions; i++) {
            System.out.println(etiqueta + " -> iteració " + i);
        }
    }
}
```

Per crear-lo i arrencar-lo:

```
ComptadorFil f = new ComptadorFil("Sensor-A", 5);
f.start();      // NO cridis mai run() directament!
```

IMPORTANT

`start()` és el mètode que demana a la JVM que reservi un fil natiu del sistema operatiu i, quan estigui llest, cridi el nostre `run()`. Si, en canvi, crides `f.run()` directament, el codi s'executarà de manera **síncrona** dins del fil que l'ha cridat, sense crear cap fil nou.

2.2 Implementar la interfície Runnable

Runnable és una interfície funcional amb un únic mètode abstracte: `void run()`. És la tècnica recomanada quan la nostra classe ja estén d'una altra (Java no permet herència múltiple) o quan volem que la tasca es pugui reutilitzar amb diferents mecanismes d'execució (fils clàssics, `ExecutorService`, etc.):

```
public class TascaSensor implements Runnable {
    private final String nom;

    public TascaSensor(String nom) {
        this.nom = nom;
    }

    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println(nom + " llegint mostra " + i);
        }
    }
}
```

Per executar-lo cal embolicar-lo dins d'un Thread:

```
Thread t = new Thread(new TascaSensor("Sensor-B"));
t.start();
```

Des de Java 8 també es pot fer servir una expressió lambda, ja que Runnable és una interfície funcional:

```
Thread t = new Thread(() -> {
    for (int i = 1; i <= 5; i++) {
        System.out.println("Fil lambda, iteració " + i);
    }
});
t.start();
```

2.3 Thread vs. Runnable: quina triem?

En la pràctica professional, **Runnable és la millor opció per defecte**:

- Separa la lògica de la tasca de la maquinària que l'executa.
- Permet reaprofitar la mateixa tasca amb `ExecutorService` (vegeu punt 8) sense tocar-ne el codi.
- No malgasta l'única herència disponible en Java.

Thread només té sentit quan volem **sobreescriure** algun altre comportament de la classe Thread a més del `run()`.

2.4 Diverses instàncies del mateix fil

Un error freqüent és crear un objecte `Thread/Runnable` i intentar arrencar-lo dues vegades:

```
Thread t = new Thread(() -> System.out.println("Hola"));
t.start();
t.start();    // llança IllegalStateException!
```

Un objecte `Thread` només es pot arrencar **una vegada**. Si necessitem repetir la tasca, cal crear una nova instància.

3. Estats d'un fil

Java modela l'estat d'un fil amb l'enumeració `Thread.State`, que es pot consultar amb `getState()`. Els estats possibles són:

Estat	Significat
NEW	El fil s'ha creat (<code>new Thread(...)</code>) però encara no s'ha cridat <code>start()</code>
RUNNABLE	El fil és candidat a executar-se; pot estar realment corrent a la CPU o esperant que el planificador li assigni temps
BLOCKED	El fil espera a poder entrar en una secció <code>synchronized</code> que un altre fil té bloquejada
WAITING	El fil espera indefinidament que un altre fil el notifiqui (<code>wait()</code> sense timeout, <code>join()</code> sense timeout)
TIMED_WAITING	Igual que <code>WAITING</code> però amb un temps màxim (<code>sleep(ms)</code> , <code>wait(ms)</code> , <code>join(ms)</code>)
TERMINATED	El mètode <code>run()</code> ha acabat, sigui amb normalitat o per una excepció no capturada

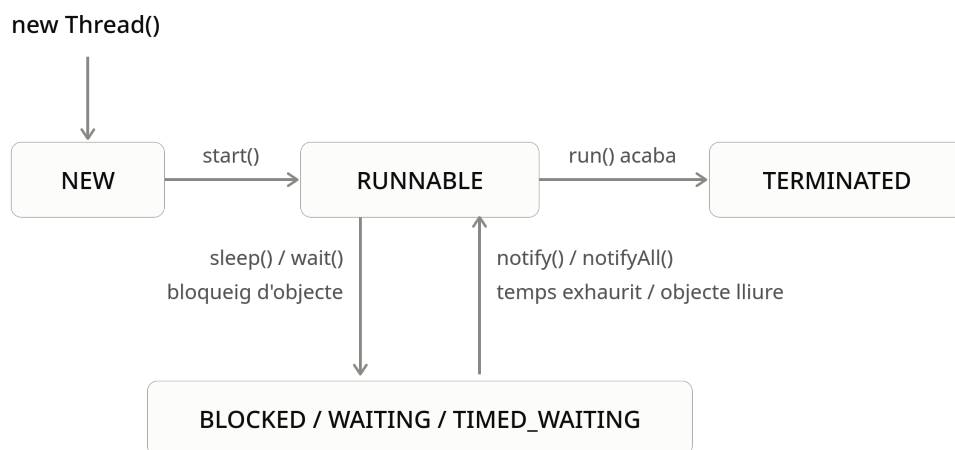


Figura 1: Estats d'un fil

3.1 Exemple: observar els estats

```
public class DemoEstats {
    public static void main(String[] args) throws InterruptedException
    {
        ↪ {
            Thread t = new Thread(() -> {
                try {
                    Thread.sleep(2000);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });

            System.out.println("Abans de start(): " + t.getState()); //
            ↪ NEW
            t.start();
            Thread.sleep(200);
            System.out.println("Mentre dorm: " + t.getState()); //
            ↪ TIMED_WAITING
            t.join();
            System.out.println("Després de join(): " + t.getState()); //
            ↪ TERMINATED
        }
    }
}
```

NOTA

El llibre de text clàssic sol descriure només quatre estats (*new*, *runnable*, *blocked*, *dead*), un model simplificat previ a Java 5. Des de Java 5, `Thread.State` distingeix `BLOCKED`, `WAITING` i `TIMED_WAITING`, cosa que permet diagnosticar amb més precisió per què un fil no avança (per exemple, amb una eina com `jstack` o el *thread dump* de l'IDE).

3.2 Finalització correcta d'un fil

Els mètodes `stop()`, `suspend()` i `resume()` de la classe `Thread` estan **obsolets (deprecated)** des de fa moltíssimes versions de Java i **no s'han d'utilitzar mai**: `stop()` pot deixar objectes compartits a mig actualitzar perquè no allibera els blocatges que el fil tenia adquirits, i `suspend()`/`resume()` poden provocar interbloquejos.

La forma correcta i segura d'aturar un fil és mitjançant una **variable de control** o la **interrupció cooperativa**:

Opció A --- variable booleana pròpia:

```
public class FilAturable extends Thread {
    private volatile boolean actiu = true;

    public void aturar() {
        actiu = false;
    }
}
```

```

@Override
public void run() {
    while (actiu) {
        // feina del fil
    }
    System.out.println("Fil aturat de forma neta");
}
}

```

Cal marcar la variable com a `volatile` perquè els canvis fets des d'un altre fil siguin visibles immediatament (sense `volatile`, el compilador o la CPU podrien mantenir una còpia en cache del valor i el bucle no es tallaria mai).

Opció B --- interrupció estàndard de Java:

```

public class FilInterrompible extends Thread {
    @Override
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                // feina del fil
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Interromput mentre dormia");
        }
        System.out.println("Fil finalitzat");
    }
}

```

```

FilInterrompible f = new FilInterrompible();
f.start();
Thread.sleep(500);
f.interrupt(); // marca el flag d'interrupció (i llança
↳ InterruptedException si estava en sleep/wait/join)

```

`interrupt()` **no atura el fil per la força**: només activa un indicador intern (i, si el fil estava bloquejat en `sleep()`, `wait()` o `join()`, provoca que aquestes crides llencin `InterruptedException`). És responsabilitat del mateix codi del `run()` comprovar aquest indicador i decidir acabar de manera ordenada.

4. Gestió de fils: mètodes fonamentals

Mètode	Què fa
<code>start()</code>	Arrenca el fil; la JVM crida internament <code>run()</code>
<code>run()</code>	Conté el codi que executarà el fil
<code>sleep(long ms)</code> (estàtic)	Posa a dormir el fil actual el temps indicat
<code>join()</code>	El fil que crida <code>join()</code> sobre un altre s'espera fins que aquest acabi
<code>join(long ms)</code>	Igual, però amb un temps màxim d'espera
<code>interrupt()</code>	Envia un senyal d'interrupció al fil
<code>isInterrupted()</code>	Retorna si el fil ha estat interromput
<code>isAlive()</code>	Retorna si el fil ha arrencat i encara no ha acabat
<code>getState()</code>	Retorna l'estat actual (<code>Thread.State</code>)
<code>setName(String) / getName()</code>	Assigna o consulta el nom del fil
<code>setDaemon(boolean)</code>	Marca el fil com a <i>daemon</i> (la JVM no espera els fils daemon per acabar el programa)
<code>currentThread()</code> (estàtic)	Retorna una referència al fil que executa el codi actual

4.1 `join()`: esperar que un fil acabi

`join()` és imprescindible quan el fil principal (main) necessita **esperar** que un o més fils secundaris hagin acabat abans de continuar (per exemple, per combinar-ne els resultats):

```
public class TascaSuma extends Thread {
    private final int desde, fins;
    private long resultat;

    public TascaSuma(int desde, int fins) {
        this.desde = desde;
        this.fins = fins;
    }

    @Override
    public void run() {
        long suma = 0;
        for (int i = desde; i <= fins; i++) suma += i;
        resultat = suma;
    }

    public long getResultat() { return resultat; }
}

public class SumaParalela {
    public static void main(String[] args) throws InterruptedException
    ↪ {
        TascaSuma part1 = new TascaSuma(1, 500_000);
        TascaSuma part2 = new TascaSuma(500_001, 1_000_000);

        part1.start();
        part2.start();
    }
}
```

```

        part1.join();    // esperem que acabin tots dos
        part2.join();

        System.out.println("Suma total: " + (part1.getResultat() +
        ↪ part2.getResultat()));
    }
}

```

Sense els `join()`, el `main()` podria arribar a la línia del `System.out.println` **abans** que els fils haguessin acabat de calcular, i el resultat imprès seria incorrecte (probablement 0).

4.2 Fils daemon

Un fil **daemon** és un fil de “servei” que la JVM no té en compte per decidir si el programa ha d'acabar: quan tots els fils normals (no daemon) acaben, la JVM finalitza encara que hi hagi fils daemon en marxa. S'utilitzen típicament per a tasques de fons com un recol·lector d'estadístiques o un *keep-alive*:

```

Thread rellotgeFons = new Thread(() -> {
    while (true) {
        System.out.println("Tic...");
        try { Thread.sleep(1000); } catch (InterruptedException e) {
            ↪ return; }
    }
});
rellotgeFons.setDaemon(true);    // s'ha de cridar ABANS de start()
rellotgeFons.start();

```

5. Compartició d'informació entre fils i condicions de carrera

5.1 El problema

Quan diversos fils accedeixen al mateix objecte i almenys un d'ells el modifica, poden aparèixer resultats incoherents. Vegem-ho amb un comptador compartit:

```
public class Comptador {
    private int valor = 0;

    public void incrementa() {
        valor++; // en realitat són 3 operacions: llegir, sumar 1,
        ↪ escriure
    }

    public int getValor() { return valor; }
}
```

```
public class ProvaCarrera {
    public static void main(String[] args) throws InterruptedException
    ↪ {
        Comptador c = new Comptador();

        Runnable tasca = () -> {
            for (int i = 0; i < 100_000; i++) c.incrementa();
        };

        Thread t1 = new Thread(tasca);
        Thread t2 = new Thread(tasca);
        t1.start();
        t2.start();
        t1.join();
        t2.join();

        System.out.println("Valor esperat: 200000");
        System.out.println("Valor obtingut: " + c.getValor());
    }
}
```

Si executem aquest codi diverses vegades, el valor final gairebé mai serà 200000, sinó un valor **menor i variable**. La raó és que `valor++` no és una operació atòmica: internament és “llegeix `valor`”, “suma 1” i “desa el resultat”. Si els dos fils fan la lectura abans que l'altre hagi desat el seu increment, es “perd” una actualització. Això és una **condició de carrera**.

5.2 La solució: synchronized

Java proporciona el mecanisme de **monitor** (o *lock* intrínsec) mitjançant la paraula clau `synchronized`. Cada objecte Java té associat un monitor; quan un fil entra en un bloc o mètode `synchronized` sobre un objecte, **bloqueja** aquest monitor, i cap altre fil pot entrar en cap altre bloc `synchronized` que faci servir el mateix objecte fins que el primer en surti.

Sincronitzant el mètode sencer:

```
public class ComptadorSegur {
    private int valor = 0;

    public synchronized void incrementa() {
        valor++;
    }

    public synchronized int getValor() {
        return valor;
    }
}
```

Sincronitzant només el bloc necessari (útil quan el mètode fa altres coses que no cal protegir):

```
public class ComptadorSegur {
    private int valor = 0;
    private final Object bloqueig = new Object();

    public void incrementa() {
        synchronized (bloqueig) {
            valor++;
        }
    }

    public int getValor() {
        synchronized (bloqueig) {
            return valor;
        }
    }
}
```

NOTA

La sincronització té un cost de rendiment (els fils es poden arribar a bloquejar entre ells) i, si s'abusa d'objectes de sincronització diferents que es necessiten mútuament, pot provocar **interbloquejos** (*deadlocks*). La recomanació general és: sincronitza el mínim de codi possible i, sempre que puguis, prefereix sincronitzar mètodes sencers curts abans que blocs dispersos.

5.3 Un exemple més realista: places lliures d'un pàrquing

Imaginem un pàrquing amb un nombre limitat de places, i diversos fils que representen cotxes intentant reservar-ne una alhora. Sense sincronització, dos cotxes podrien comprovar que encara queda una plaça *abans* que l'altre l'hagi ocupat, i acabar tots dos "aparcant" a la mateixa plaça inexistent:

```
public class Parquing {
    private int placesLliures;

    public Parquing(int totalPlaces) {
        this.placesLliures = totalPlaces;
    }

    public synchronized boolean reservarPlaça(String matricula) {
        if (placesLliures > 0) {
            System.out.println(matricula + " veu " + placesLliures + "
            ↪ places lliures");
            try { Thread.sleep(50); } catch (InterruptedException e) {}
            placesLliures--;
            System.out.println(matricula + " aparca -> queden " +
            ↪ placesLliures);
            return true;
        }
        System.out.println(matricula + " no troba plaça lliure");
        return false;
    }
}
```

IMPORTANT

El `Thread.sleep()` que apareix dins `reservarPlaça()` és deliberat: simula el temps que tarda un cotxe a maniobrar, per fer visible la finestra de temps en què podria aparèixer una condició de carrera si el mètode no fos `synchronized`. En codi real, però, mai no s'ha de cridar `sleep()` ni cap operació llarga (I/O, xarxa, càlcul intensiu) dins d'un bloc o mètode `synchronized`: el fil dorm però **manté el monitor**, de manera que tots els altres fils que necessitin entrar a qualsevol mètode `synchronized` del mateix objecte queden bloquejats durant tot el temps d'espera. La regla pràctica és: dins d'un bloc `synchronized` només hi ha d'haver la **mínima feina imprescindible** per mantenir l'estat consistent (llegir, comparar i modificar les variables compartides); tot el que sigui operació lenta ha d'anar fora.

Marcant `reservarPlaça()` com a `synchronized`, quan un fil (cotxe) hi entra, els altres queden esperant fins que aquest surt del mètode, i mai no es poden arribar a ocupar més places de les que realment existeixen.

6. Sincronització avançada: `wait()`, `notify()` i `notifyAll()`

`synchronized` evita que dos fils **entrin alhora** en una secció crítica, però de vegades necessitem alguna cosa més: que un fil **s'esperi** fins que es doni una determinada condició, i que un altre fil **l'avis**i quan aquesta condició es compleixi. Per això Java ofereix tres mètodes, definits a la classe `Object` (no a `Thread`):

Mètode	Efecte
<code>wait()</code>	El fil que el crida queda suspès fins que un altre fil cridi <code>notify()/notifyAll()</code> sobre el mateix objecte
<code>notify()</code>	Desperta un dels fils que esperen sobre l'objecte (l'elecció no està definida)
<code>notifyAll()</code>	Desperta tots els fils que esperen sobre l'objecte

IMPORTANT

`wait()`, `notify()` i `notifyAll()` **només es poden cridar dins d'un bloc o mètode `synchronized`** sobre el mateix objecte; en cas contrari es llança `IllegalMonitorStateException`.

6.1 El problema productor-consumidor

Un escenari clàssic de sincronització: un fil **productor** genera dades i les diposita en un magatzem compartit (una cua, un buffer...) i un fil **consumidor** les recull i les processa. Cal evitar dues coses:

- Que el consumidor llegeixi quan **encara no hi ha res** disponible.
- Que el productor sobreescriui una dada **abans que el consumidor l'hagi consumida**.

Vegem un magatzem amb capacitat per a diversos elements alhora (no només un), cosa que obliga a controlar dues condicions diferents: que no estigui ple (per al productor) i que no estigui buit (per al consumidor):

```
import java.util.LinkedList;

public class Magatzem {
    private final LinkedList<String> elements = new LinkedList<>();
    private final int capacitat;

    public Magatzem(int capacitat) {
        this.capacitat = capacitat;
    }

    public synchronized void afegir(String element) throws
        ↪ InterruptedException {
        while (elements.size() == capacitat) {    // ple: el productor
            ↪ espera
            wait();
        }
        elements.addLast(element);
    }
}
```

```

        notifyAll();                                // avisa que hi ha un
        ↪ element nou
    }

    public synchronized String treure() throws InterruptedException {
        while (elements.isEmpty()) {                // buit: el
            ↪ consumidor espera
            wait();
        }
        String element = elements.removeFirst();
        notifyAll();                                // avisa que ha quedat
        ↪ un lloc lliure
        return element;
    }
}

```

```

public class Productor extends Thread {
    private final Magatzem magatzem;

    public Productor(Magatzem m) { this.magatzem = m; }

    @Override
    public void run() {
        for (int i = 1; i <= 8; i++) {
            try {
                magatzem.afegir("peça-" + i);
                System.out.println("Productor -> afegeix peça-" + i);
                Thread.sleep(80);
            } catch (InterruptedException e) { return; }
        }
        try {
            magatzem.afegir(null); // Sentinella: indica que no hi
            ↪ ha més dades
        } catch (InterruptedException e) {
            ↪ Thread.currentThread().interrupt(); }
    }
}

public class Consumidor extends Thread {
    private final Magatzem magatzem;

    public Consumidor(Magatzem m) { this.magatzem = m; }

    @Override
    public void run() {
        try {
            while (true) {
                String element = magatzem.treure();
                if (element == null) break; // sentinella rebut:
                ↪ acabem
                System.out.println("Consumidor <- retira " + element);
                Thread.sleep(150);
            }
        }
    }
}

```

```

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

```

NOTA

Aquest exemple assumeix un sol productor i un sol consumidor amb el mateix nombre d'elements. En un cas real amb múltiples productors o un nombre d'elements indeterminat cal un mecanisme explícit de finalització: el més habitual és que el productor afegeixi un element sentinella (un valor especial, sovint null o una constant FI) que el consumidor reconeix com a senyal per deixar de llegir. Sense aquest mecanisme, el consumidor quedaria bloquejat per sempre en `wait()` un cop el magatzem s'hagués buidat definitivament.

```

public class ProvaMagatzem {
    public static void main(String[] args) {
        Magatzem m = new Magatzem(3); // capacitat per a 3 elements
        new Productor(m).start();
        new Consumidor(m).start();
    }
}

```

Com que el magatzem admet fins a 3 elements pendents, el productor pot avançar-se al consumidor sense bloquejar-se immediatament (a diferència d'un magatzem d'una sola posició), però mai no podrà afegir-ne un quart mentre n'hi hagi 3 pendents, ni el consumidor podrà treure'n cap si el magatzem és buit.

NOTA

Fixa't que la condició d'espera sempre es comprova dins d'un **bucle while**, mai amb un simple `if`. Això és una norma d'or de la programació amb `wait()/notify()`: quan un fil es desperta, ha de tornar a comprovar la condició, perquè `notifyAll()` desperta *tots* els fils en espera i no tots necessàriament trobaran la condició que buscaven certa (algú altre podria haver-la canviat abans que els toqués torn).

7. Gestió de prioritats

Cada fil de Java té una **prioritat**, un valor enter entre `Thread.MIN_PRIORITY` (1) i `Thread.MAX_PRIORITY` (10). Per defecte, un fil hereta la prioritat del fil que l'ha creat, normalment `Thread.NORM_PRIORITY` (5).

```

Thread t = new Thread(tasca);
t.setPriority(Thread.MAX_PRIORITY);
System.out.println("Prioritat actual: " + t.getPriority());

```

La prioritat és només una **pista** que es dona al planificador del sistema operatiu: en igualtat de condicions, un fil amb més prioritat tindrà preferència per rebre temps de CPU, però el

comportament exacte depèn de la plataforma (Windows, Linux...) i de la implementació concreta de la JVM. No hi ha cap garantia que un fil de prioritat màxima s'executi sempre abans que un de prioritat mínima, i en sistemes Linux sovint l'efecte és pràcticament imperceptible.

NOTA

A la pràctica professional gairebé mai es toquen les prioritats a mà. Si un disseny depèn de prioritats concretes per funcionar correctament, sol ser un símptoma que caldria repensar la sincronització (per exemple, amb estructures d'`java.util.concurrent`, com veurem al punt 8).

8. Més enllà de Thread: una mirada a `java.util.concurrent`

Tot el que hem vist fins ara (`Thread`, `synchronized`, `wait()`/`notify()`) forma part de l'API "clàssica" de Java, disponible des de les primeres versions del llenguatge. Des de Java 5, el paquet **`java.util.concurrent`** ofereix eines d'alt nivell que resolen els mateixos problemes amb menys codi i menys risc d'errors. No formen part del contingut bàsic d'aquesta unitat, però és important que en coneguis l'existència:

- **ExecutorService**: gestiona un grup ("pool") de fils reutilitzables, de manera que no cal crear i destruir fils manualment per a cada tasca.

```
ExecutorService executor = Executors.newFixedThreadPool(4);
executor.submit(() -> System.out.println("Tasca executada pel
↳ pool"));
executor.shutdown();
```

- **ReentrantLock** (paquet `java.util.concurrent.locks`): una alternativa a `synchronized` amb més flexibilitat (per exemple, permet intentar adquirir el bloqueig amb un temps màxim d'espera).
- **Col·leccions concurrents**, com `ConcurrentHashMap` o `BlockingQueue`, dissenyades des de zero per ser accedides de manera segura per diversos fils sense haver-nos de preocupar de sincronitzar-les manualment.
- **CountDownLatch** i **CyclicBarrier**: mecanismes per coordinar que un grup de fils arribin a un mateix punt abans de continuar, sense haver d'escriure `wait()`/`notify()` a mà.

CONSELL

Un `BlockingQueue` (per exemple `LinkedBlockingQueue` sense límit de capacitat, o `ArrayBlockingQueue` si es vol un buffer de mida fixa com el Magatzem del punt 6) resol el problema productor-consumidor del punt 6 amb molt menys codi: els mètodes `put()` i `take()` ja incorporen tota la lògica d'espera i notificació que hem programat a mà a la classe `Magatzem`.

9. Depuració d'aplicacions multifil

Els errors en programes multifil tenen una característica especialment incòmoda: **no sempre es reproduïxen igual**. Un programa amb una condició de carrera pot funcionar bé desenes de vegades i fallar només en condicions de càrrega concretes. Algunes recomanacions pràctiques:

- Dona **nom** als teus fils amb `setName()`; els missatges de traça i els *thread dumps* són molt més llegibles.
- Afegeix missatges de log (`System.out.println` en desenvolupament, un *logger* real en producció) indicant quin fil executa cada acció.
- Utilitza el depurador de l'IDE (NetBeans, IntelliJ, Eclipse) amb els seus panells de fils (*Threads*), que permeten veure l'estat de cada fil i posar-los en pausa individualment.
- Per detectar interbloquejos en calent, es pot demanar un *thread dump* (per exemple amb `jstack <pid>` des de línia d'ordres, o des del mateix IDE), que mostra l'estat i la pila de crides de cada fil.
- Escriu **proves de càrrega**: executar la mateixa prova centenars de vegades en bucle ajuda molt a fer aflorar condicions de carrera que amb una sola execució no es manifesten.
- Documenta sempre, en un comentari, **quin objecte protegeix quina dada** quan facis servir `synchronized`. És l'error de disseny multifil més freqüent: sincronitzar sobre objectes diferents pensant erròniament que protegeixen el mateix recurs.

Índex de figures

1	Estats d'un fil	4
---	---------------------------	---

Versions d'aquest document

- [HTML](#)
- [PDF](#)
- [ODT](#)
- [MD](#)

[Domini Públic \(CC0\)](#)