Perfecto, voy a crear un resumen completo enfocado 100% en Java con todos los ejemplos prácticos. Esto será mucho más extenso y detallado:

RESUMEN COMPLETO PARA EXAMEN - PSP UA1, UA2 y UA3

VERSIÓN JAVA - EJEMPLOS PRÁCTICOS COMPLETOS

ÍNDICE

UA1 - PROGRAMACIÓN MULTIPROCESO EN JAVA

CONCEPTOS TEÓRICOS FUNDAMENTALES

1. PROCESOS EN JAVA ¿Qué es un Proceso?

Un proceso es un programa en ejecución que incluye: - El código ejecutable - Su propio espacio de memoria (heap, stack, código, datos) - Recursos del sistema operativo (archivos abiertos, conexiones de red) - Un identificador único (PID - Process ID)

Características en Java: - Java permite **crear** y **gestionar** procesos externos - Cada proceso es **independiente** y tiene su propia JVM si es un programa Java - Los procesos **NO comparten memoria** entre sí - La comunicación requiere mecanismos especiales (pipes, streams, archivos)

---

2. ESTADOS DE UN PROCESO

```
NUEVO    (Proceso creado)
  ↓
LISTO    (Esperando CPU)
  ↓ ↑
```

```
ACTIVO   (Ejecutándose)
  ↓
BLOQUEADO    (Esperando I/O, recurso)
  ↓
TERMINADO
```

1. NUEVO (New) - El proceso está siendo creado - Se asigna memoria y recursos - En Java: después de `new ProcessBuilder()` pero antes de `start()`

2. LISTO (Ready) - El proceso está preparado para ejecutarse - Espera que el scheduler le asigne CPU - Puede haber muchos procesos en este estado

3. EN EJECUCIÓN (Running) - El proceso está usando la CPU - Sus instrucciones se están ejecutando - Solo un proceso por núcleo en este estado

4. BLOQUEADO/ESPERA (Blocked/Waiting) - El proceso espera un evento: - Operación de E/S (lectura de archivo, red) - Entrada del usuario - Respuesta de otro proceso - No puede continuar hasta que el evento ocurra

5. TERMINADO (Terminated) - El proceso ha finalizado - Puede ser: - **Normal**: Terminación controlada (`System.exit(0)`) - **Anormal**: Error o excepción (`System.exit(-1)`) - Los recursos se liberan

---

3. HILOS vs PROCESOS

Aspecto	**PROCESO**	**HILO**
Memoria	Espacio independiente y aislado	Comparten memoria del proceso
Creación	Costosa (milisegundos)	Rápida (microsegundos)
Comunicación	Compleja (IPC, streams, archivos)	Simple (variables compartidas)
Contexto	Cambio lento (mucho estado)	Cambio rápido (poco estado)
Seguridad	Aislados (fallo de uno no afecta otros)	Riesgo compartido (un error afecta a todos)
Recursos	Propios (archivos, conexiones)	Comparten recursos del proceso
Overhead	Alto	Bajo

Aspecto	PROCESO	HILO
Uso en Java	`ProcessBuilder`, `Runtime.exec()`	`Thread`, `Runnable`

¿Cuándo usar cada uno?

Usa PROCESOS cuando: - Necesitas ejecutar programas externos - Quieres aislamiento total - Necesitas seguridad (fallos no propagables) - Cada tarea es independiente

Usa HILOS cuando: - Necesitas compartir datos fácilmente - Quieres mayor rendimiento - Las tareas están relacionadas - Todo está en la misma JVM

---

4. PROGRAMACIÓN CONCURRENTE **Definición**: Capacidad de manejar múltiples tareas que progresan en períodos de tiempo solapados.

Concurrencia NO es paralelismo: - **Concurrencia**: Gestión de múltiples tareas (pueden ejecutarse en 1 núcleo intercalándose) - **Paralelismo**: Ejecución simultánea real (requiere múltiples núcleos)

```
CONCURRENCIA (1 núcleo):          PARALELISMO (2 núcleos):
Tarea A:                     Tarea A:
Tarea B:                    Tarea B:
        (se intercalan)                  (simultáneas)
```

Aplicaciones: 1. **Servidores web**: Atender múltiples peticiones 2. **Interfaces gráficas**: Mantener UI responsiva 3. **Procesamiento de datos**: Dividir trabajo 4. **Sistemas operativos**: Gestionar múltiples programas

---

5. PROGRAMACIÓN PARALELA vs DISTRIBUIDA PROGRAMACIÓN PARALELA

Definición: Ejecución simultánea en múltiples núcleos de la **misma máquina**.

Características:

```java
// Ejemplo conceptual
ExecutorService executor = Executors. newFixedThreadPool(4); // 4 núcleos
for (int i = 0; i < 1000; i++) {
    executor.submit(() -> procesarDato());
}
```

Ventajas: - Comunicación muy rápida (memoria compartida) - Baja latencia - Sincronización más sencilla - Mayor velocidad de cómputo

Inconvenientes: - Limitado por hardware físico - Costo de hardware con muchos núcleos - Punto único de fallo - No escalable más allá de la máquina

---

PROGRAMACIÓN DISTRIBUIDA

Definición: Ejecución en múltiples computadoras conectadas por **red**.

Características:

```
Servidor 1 ↔ Red ↔ Servidor 2
    ↓                    ↓
Base de datos       Base de datos
```

Ventajas: - Escalabilidad horizontal (añadir más máquinas) - Tolerancia a fallos (redundancia) - Recursos distribuidos geográficamente - Menor costo (hardware común)

Inconvenientes: - Mayor latencia de red - Sincronización compleja - Problemas de consistencia - Más difícil de depurar

---

6. PROBLEMAS CLÁSICOS DE CONCURRENCIA A. CONDICIONES DE CARRERA (Race Conditions)

Definición: El resultado depende del orden impredecible de ejecución.

Ejemplo en Java:

```java
class Contador {
    private int valor = 0;  // ¡PELIGRO!

    public void incrementar() {
        valor++;  // NO ES ATÓMICO
        // En realidad son 3 operaciones:
        // 1. Leer valor
        // 2. Sumar 1
        // 3. Escribir valor
    }
}

// Si 2 procesos/hilos ejecutan incrementar() simultáneamente:
Proceso A: lee valor (0)
Proceso B: lee valor (0)
Proceso A: suma 1 → 1
Proceso B: suma 1 → 1
Proceso A: escribe 1
Proceso B: escribe 1
// Resultado: 1 (debería ser 2)
```

Solución:

```
public synchronized void incrementar() {
    valor++;  // Ahora es seguro
}
```

B. DEADLOCK (Interbloqueo)

Definición: Dos o más procesos esperan indefinidamente por recursos retenidos mutuamente.

Ejemplo:

```
Proceso A: tiene Recurso 1, espera Recurso 2
Proceso B: tiene Recurso 2, espera Recurso 1
→ Ambos esperan eternamente
```

Las 4 condiciones de Coffman (todas deben cumplirse): 1. **Exclusión mutua**: Recursos no compartibles 2. **Retención y espera**: Retener recursos mientras espera otros 3. **No apropiación**: Recursos no pueden ser arrebatados 4. **Espera circular**: Ciclo de procesos esperando

Prevención: Romper al menos una condición.

C. INANICIÓN (Starvation)

Definición: Un proceso nunca obtiene los recursos necesarios.

Ejemplo:

```
// Sistema con prioridades
Proceso Alta Prioridad (siempre ejecutándose)
Proceso Baja Prioridad (nunca ejecuta) ← INANICIÓN
```

PROGRAMACIÓN DE PROCESOS EN JAVA

CLASES PRINCIPALES

1. **`ProcessBuilder`**: Clase moderna para crear y configurar procesos (RECOMENDADA)
2. **`Process`**: Representa un proceso en ejecución
3. **`Runtime`**: Clase legacy para ejecutar procesos (menos recomendada)

PROCESSBUILDER - LA CLASE PRINCIPAL

Constructor:

```
ProcessBuilder pb = new ProcessBuilder(String... command);
```

Métodos importantes:

Método	Descripción
`start()`	Inicia el proceso y devuelve objeto `Process`
`command(List<String>)`	Establece el comando a ejecutar
`directory(File)`	Establece el directorio de trabajo
`environment()`	Obtiene/modifica variables de entorno
`redirectInput(File)`	Redirige entrada estándar
`redirectOutput(File)`	Redirige salida estándar
`redirectError(File)`	Redirige salida de error
`inheritIO()`	Hereda los streams del proceso padre

PROCESS - REPRESENTACIÓN DEL PROCESO

Métodos importantes:

Método	Descripción	Retorno
`getInputStream()`	Obtiene salida estándar del proceso (stdout)	`InputStream`
`getOutputStream()`	Obtiene entrada estándar del proceso (stdin)	`OutputStream`
`getErrorStream()`	Obtiene salida de error del proceso (stderr)	`InputStream`
`waitFor()`	Espera a que el proceso termine	`int` (código de salida)
`waitFor(long, TimeUnit)`	Espera con timeout	`boolean`
`exitValue()`	Obtiene código de salida (sin esperar)	`int`
`destroy()`	Termina el proceso forzosamente	`void`
`isAlive()`	¿Está el proceso ejecutándose?	`boolean`

EJEMPLOS PRÁCTICOS COMPLETOS

EJEMPLO 1: Crear un Proceso Simple

```
package com.ceslopedevega. procesos;
import java.io.IOException;
```

```java
public class Ejemplo1_ProcesoSimple {
    public static void main(String[] args) throws IOException {
        // Crear un ProcessBuilder con el comando a ejecutar
        ProcessBuilder pb = new ProcessBuilder("notepad.exe");

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

        System.out.println("Proceso lanzado:  Notepad");
        System.out.println("¿Está vivo? " + p.isAlive());
    }
}
```

Explicación: - `ProcessBuilder("notepad.exe")`: Crea un constructor con el ejecutable - `pb.start()`: Inicia el proceso y devuelve objeto `Process` - El programa Java continúa su ejecución (no espera al proceso) - En Linux sería: `new ProcessBuilder("gedit")`

EJEMPLO 2: Ejecutar Comando del Sistema y Leer Salida

```java
package com.ceslopedevega.procesos;
import java.io.*;

public class Ejemplo2_LeerSalida {
    public static void main(String[] args) throws IOException {
        // Ejecutar comando DIR en Windows
        Process p = new ProcessBuilder("CMD", "/C", "DIR").start();

        try {
            // Obtener el InputStream de salida del proceso
            InputStream is = p.getInputStream();

            // Leer carácter a carácter
            int c;
            while ((c = is.read()) != -1) {
                System.out.print((char) c);
            }
            is.close();

        } catch (Exception e) {
            e.printStackTrace();
        }

        // Esperar a que termine y obtener código de salida
        try {
```

```
            int exitVal = p.waitFor();
            System.out.println("\nCódigo de salida: " + exitVal);
            // 0 = éxito, != 0 = error
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Puntos Clave: 1. `CMD /C DIR`: - `CMD`: Lanza el intérprete de comandos - `/C`: Ejecuta el comando y termina - `DIR`: Lista archivos del directorio 2. `getInputStream()`: Obtiene la SALIDA del proceso (stdout) 3. `waitFor()`: Bloquea hasta que el proceso termine 4. **Código 0**: Éxito. Cualquier otro valor indica error

---

EJEMPLO 3: Manejo de Errores del Proceso

```
package com.ceslopedevega. procesos;
import java.io.*;

public class Ejemplo3_ManejoErrores {
    public static void main(String[] args) throws IOException {
        // Comando INVÁLIDO (DIRR no existe)
        Process p = new ProcessBuilder("CMD", "/C", "DIRR").start();

        try {
            // Intentar leer salida estándar
            InputStream is = p.getInputStream();
            int c;
            while ((c = is.read()) != -1)
                System.out.print((char) c);
            is.close();

        } catch (Exception e) {
            e.printStackTrace();
        }

        // Comprobar código de salida
        try {
            int exitVal = p.waitFor();
            System.out. println("Código de Salida: " + exitVal);

            if (exitVal != 0) {
                // Leer el stream de ERROR
                InputStream er = p.getErrorStream();
```

```
                BufferedReader brer = new BufferedReader(
                    new InputStreamReader(er));
                String liner = null;
                System.out.println("\n=== ERRORES ===");
                while ((liner = brer.readLine()) != null) {
                    System.out. println("ERROR > " + liner);
                }
            }
        } catch (InterruptedException | IOException e) {
            e.printStackTrace();
        }
    }
}
```

Salida:

```
Código de Salida: 1
=== ERRORES ===
ERROR > 'DIRR' no se reconoce como un comando interno o externo...
```

Conceptos: - `getErrorStream()`: Obtiene stderr (salida de errores) - `BufferedReader`: Lee línea a línea más eficientemente - **Códigos de salida 0**: Indican error

---

EJEMPLO 4: Enviar Entrada a un Proceso Programa Hijo (EjemploLectura.java):

```
package com.ceslopedevega.procesos;
import java. io.*;

public class EjemploLectura {
    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System. in));

        System. out.println("Esperando entrada...");
        String linea = br.readLine();
        System.out.println("Has escrito: " + linea);
    }
}
```

Programa Padre (Ejemplo4_EnviarEntrada.java):

```
package com.ceslopedevega. procesos;
import java.io.*;

public class Ejemplo4_EnviarEntrada {
```

```
    public static void main(String[] args) throws IOException {
        File directorio = new File("bin");
        ProcessBuilder pb = new ProcessBuilder(
            "java", "com.ceslopedevega. procesos.EjemploLectura");
        pb.directory(directorio);

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

        // ENVIAR ENTRADA al proceso hijo
        OutputStream os = p. getOutputStream();
        os.write("Hola desde el proceso padre\n".getBytes());
        os.flush(); // ¡Importante! Vacía el buffer

        // LEER SALIDA del proceso hijo
        InputStream is = p. getInputStream();
        int c;
        while ((c = is. read()) != -1)
            System.out.print((char) c);
        is.close();

        // Esperar y verificar
        try {
            int exitVal = p.waitFor();
            System.out.println("Código de salida: " + exitVal);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Salida:

```
Esperando entrada...
Has escrito: Hola desde el proceso padre
Código de salida: 0
```

Análisis: 1. `getOutputStream()`: Obtiene el STDIN del proceso hijo 2. `write()`: Envía bytes al proceso 3. `flush()`: **MUY IMPORTANTE** - Fuerza el envío inmediato 4. Sin `flush()`, los datos pueden quedarse en buffer

---

EJEMPLO 5: Argumentos y Variables de Entorno Programa que recibe argumentos (LeerNombre.java):

```
package com.ceslopedevega.procesos;
```

```java
public class LeerNombre {
    public static void main(String[] args) {
        if (args.length != 1) {
            System. out.println("Error: Se necesita 1 argumento");
            System.exit(-1);
        }
        System.out.println("Hola, " + args[0]);
        System.exit(1); // Código de salida personalizado
    }
}
```

Programa que lo ejecuta (Ejemplo5_Argumentos.java):

```java
package com.ceslopedevega.procesos;
import java.io.*;
import java.util.*;

public class Ejemplo5_Argumentos {
    public static void main(String[] args) throws IOException {
        File directorio = new File(". \\bin");

        ProcessBuilder pb = new ProcessBuilder();

        // Ver variables de entorno
        Map<String, String> entorno = pb.environment();
        System.out.println("=== VARIABLES DE ENTORNO ===");
        System.out.println("PATH: " + entorno.get("PATH"));
        System.out. println("JAVA_HOME: " + entorno. get("JAVA_HOME"));

        // Configurar comando con argumentos
        pb.command("java", "com.ceslopedevega. procesos.LeerNombre", "Luis");

        // Ver comando configurado
        List<String> comando = pb.command();
        System.out.println("\n=== COMANDO ===");
        for (String parte : comando) {
            System. out.println(parte);
        }

        // Ejecutar
        pb.directory(directorio);
        pb.redirectOutput(ProcessBuilder.Redirect.INHERIT); // Hereda stdout
        Process p = pb.start();

        try {
            int exitVal = p.waitFor();
            System.out.println("\nCódigo de salida:  " + exitVal);
```

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

Salida:

```
=== VARIABLES DE ENTORNO ===
PATH: C:\Windows\system32;C:\Windows;...
JAVA_HOME: C:\Program Files\Java\jdk-11

=== COMANDO ===
java
com.ceslopedevega. procesos.LeerNombre
Luis

Hola, Luis
Código de salida: 1
```

Conceptos: - **`environment()`**: Accede a variables de entorno del sistema - **`command()`**: Establece/obtiene el comando a ejecutar - **`INHERIT`**: El proceso hijo hereda los streams del padre

---

EJEMPLO 6: Redirección de E/S a Archivos

```
package com.ceslopedevega.procesos;
import java.io.*;

public class Ejemplo6_Redireccion {
    public static void main(String[] args) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD");

        // Archivos para redirección
        File fBat = new File("comandos.bat");
        File fOut = new File("salida.txt");
        File fErr = new File("errores.txt");

        // Configurar redirecciones
        pb.redirectInput(fBat);      // Leer comandos desde archivo
        pb.redirectOutput(fOut);     // Escribir salida a archivo
        pb.redirectError(fErr);      // Escribir errores a archivo

        pb.start();

        System.out.println("Proceso lanzado");
```

```
        System.out.println("Revisa los archivos salida.txt y errores.txt");
    }
}
```

comandos.bat:

```
@echo off
echo Hola desde el archivo bat
dir
echo Comando inexistente:
comandoinvalido
```

Resultado: - `salida.txt`: Contendrá "Hola desde el archivo bat" y resultado de `dir` - `errores.txt`: Contendrá error de `comandoinvalido`

EJEMPLO 7: Ejecutar Otro Programa Java Programa simple (Unsaludo.java):

```
package com.ceslopedevega.procesos;

public class Unsaludo {
    public static void main(String[] args) {
        if (args.length < 1) {
            System. out.println("SE NECESITA UN SALUDO.. .");
            System.exit(1);
        }
        for (int i = 0; i < 5; i++) {
            System.out.println((i+1) + ".  " + args[0]);
        }
    }
}
```

Programa que lo ejecuta:

```
package com.ceslopedevega.procesos;
import java.io.*;

public class Ejemplo7_EjecutarJava {
    public static void main(String[] args) {
        try {
            // Configurar proceso
            ProcessBuilder pb = new ProcessBuilder(
                "java",
                "-cp", "bin",  // classpath
                "com.ceslopedevega.procesos. Unsaludo",
                "¡Hola Mundo!"  // argumento
            );
```

```
            // Heredar I/O para ver salida directamente
            pb.inheritIO();

            // Ejecutar
            Process p = pb.start();

            // Esperar
            int exitCode = p.waitFor();
            System.out.println("\nProceso terminado con código: " + exitCode);

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

Salida:

```
1. ¡Hola Mundo!
2. ¡Hola Mundo!
3. ¡Hola Mundo!
4. ¡Hola Mundo!
5. ¡Hola Mundo!

Proceso terminado con código: 0
```

TÉCNICAS AVANZADAS

EJEMPLO 8: Timeout para Procesos

```
import java.io. IOException;
import java.util. concurrent.TimeUnit;

public class Ejemplo8_Timeout {
    public static void main(String[] args) {
        try {
            ProcessBuilder pb = new ProcessBuilder("ping", "google.com", "-n", "100");
            Process p = pb.start();

            // Esperar máximo 5 segundos
            boolean terminado = p.waitFor(5, TimeUnit.SECONDS);

            if (terminado) {
                System.out.println("Proceso terminó a tiempo");
                System.out.println("Código: " + p.exitValue());
```

```
            } else {
                System. out.println("TIMEOUT - Proceso tardó demasiado");
                p.destroy();  // Forzar terminación
                System.out.println("Proceso destruido");
            }

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

Uso: Evita que procesos colgados bloqueen tu aplicación.

EJEMPLO 9: Destrucción Forzosa de Procesos

```
import java.io.IOException;

public class Ejemplo9_Destruir {
    public static void main(String[] args) {
        try {
            ProcessBuilder pb = new ProcessBuilder("notepad.exe");
            Process p = pb.start();

            System.out.println("Notepad iniciado");
            Thread.sleep(3000); // Esperar 3 segundos

            System.out.println("Destruyendo proceso...");
            p.destroy();  // Terminación "amable"

            Thread.sleep(1000);

            if (p.isAlive()) {
                System.out.println("Aún vivo, forzando.. .");
                p.destroyForcibly();  // Terminación forzosa
            }

            System.out.println("Proceso terminado");

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

Diferencia: - `destroy()`: Terminación "gentil" (permite limpieza) - `destroyForcibly()`: Terminación inmediata (puede dejar recursos sin liberar)

CÓDIGOS DE SALIDA Y SU SIGNIFICADO

```
// Convención de códigos de salida
System.exit(0);   // Éxito
System.exit(1);   // Error genérico
System.exit(2);   // Mal uso del comando
System.exit(126); // Comando no ejecutable
System.exit(127); // Comando no encontrado
System.exit(130); // Terminado por Ctrl+C
System.exit(-1);  // Error fatal
```

Verificar código de salida:

```
int exitCode = process.waitFor();
switch (exitCode) {
    case 0:
        System.out.println("Éxito");
        break;
    case 1:
        System.out. println("Error en el proceso");
        break;
    default:
        System.out.println("Código desconocido: " + exitCode);
}
```

ERRORES COMUNES Y SOLUCIONES

Error 1: No cerrar streams

```
//  MAL
InputStream is = process.getInputStream();
// ...  usar is ...
// NO cerrar → FUGA DE RECURSOS

//  BIEN
InputStream is = process.getInputStream();
try {
    // ...  usar is ...
} finally {
    is.close();
}
```

```java
//  MEJOR
try (InputStream is = process.getInputStream()) {
    // ... usar is ...
} // Se cierra automáticamente
```

Error 2: No hacer flush() al enviar datos

```java
//  MAL
OutputStream os = process.getOutputStream();
os.write("datos\n".getBytes());
// Los datos pueden quedarse en buffer

//  BIEN
OutputStream os = process.getOutputStream();
os.write("datos\n".getBytes());
os.flush(); // Fuerza el envío
```

Error 3: Deadlock por buffers llenos

```java
//  MAL - Puede bloquearse
Process p = pb.start();
p.waitFor(); // Espera a que termine
InputStream is = p.getInputStream();
// Leer salida...  ← Puede ser demasiado tarde

//  BIEN - Leer mientras ejecuta
Process p = pb.start();
InputStream is = p. getInputStream();
// Leer salida en paralelo
BufferedReader reader = new BufferedReader(new InputStreamReader(is));
String line;
while ((line = reader. readLine()) != null) {
    System.out.println(line);
}
p.waitFor();
```

RESUMEN DE CONCEPTOS CLAVE UA1

Concepto	Definición	Uso en Java
Proceso	Programa en ejecución con memoria propia	`ProcessBuilder`, `Process`
PID	Identificador único del proceso	No accesible directamente en Java <11

Concepto	Definición	Uso en Java
Fork	Crear copia de proceso (concepto Unix)	Java crea procesos nuevos, no fork
Streams	Flujos de entrada/salida del proceso	`getInputStream()`, `getOutputStream()`, `getErrorStream()`
Exit Code	Código de salida del proceso	`waitFor()`, `exitValue()`
IPC	Comunicación entre procesos	Streams, archivos, pipes del SO
Concurrencia	Múltiples tareas progresando	Procesos + Hilos
Deadlock	Bloqueo mutuo esperando recursos	Evitar con diseño correcto

---

UA2 - PROGRAMACIÓN MULTIHILO EN JAVA

CONCEPTOS TEÓRICOS FUNDAMENTALES

1. ¿QUÉ ES UN HILO (THREAD)? Definición: Un hilo es la unidad más pequeña de procesamiento que puede ser programada por un sistema operativo. Es un "subproceso" ligero dentro de un proceso.

Analogía: - **Proceso** = Empresa completa - **Hilo** = Empleado trabajando en la empresa - Múltiples empleados (hilos) trabajan en la misma empresa (proceso) compartiendo recursos (memoria)

Características de los Hilos:

```
        PROCESO


  Hilo 1   Hilo 2   Hilo 3


            ↓
     MEMORIA COMPARTIDA
  [Heap, Variables, Objetos]

  Cada hilo tiene:
  - Stack propio
  - Program Counter propio
  - Registros propios
```

Ventajas de los Hilos: 1. **Compartir memoria**: Acceso directo a variables compartidas 2. **Menos overhead**: Creación y cambio de contexto más rápido 3. **Comunicación fácil**: No necesitan IPC 4. **Mejor rendimiento**: Aprovechan múltiples núcleos

Desventajas: 1. **Complejidad**: Sincronización necesaria 2. **Riesgo de errores**: Race conditions, deadlocks 3. **Depuración difícil**: Comportamiento no determinista 4. **Seguridad**: Un error puede afectar todo el proceso

---

2. ESTADOS DE UN HILO EN JAVA

```
     NEW
      ↓
   RUNNABLE ←
      ↓
   RUNNING
      ↓
   BLOCKED
      ↓
   WAITING
      ↓
TIMED_WAITING
      ↓
  TERMINATED
```

Estados Detallados:

1. **NEW (Nuevo)**

```
Thread t = new Thread();  // Estado:  NEW
// El hilo ha sido creado pero NO iniciado
```

2. **RUNNABLE (Ejecutable)**

```
t.start();  // Estado: RUNNABLE
// El hilo está listo para ejecutarse o ejecutándose
// Incluye dos subestados:
// - Ready:  Esperando CPU
// - Running: Ejecutándose
```

3. **BLOCKED (Bloqueado)**

```
synchronized(obj) {  // Si otro hilo tiene el lock
    // Este hilo pasa a BLOCKED
}
// Esperando adquirir un monitor lock
```

4. **WAITING (Esperando)**

```
obj.wait();      // Estado: WAITING
t.join();        // Estado: WAITING
// Esperando indefinidamente a que otro hilo lo notifique
```

5. **TIMED_WAITING (Espera con Tiempo)**

```
Thread.sleep(1000);      // Estado: TIMED_WAITING
obj.wait(1000);          // Estado: TIMED_WAITING
t.join(1000);            // Estado: TIMED_WAITING
// Esperando por un tiempo específico
```

6. **TERMINATED (Terminado)**

```
// El método run() ha finalizado
// El hilo no puede ser reiniciado
```

Obtener el estado:

```
Thread.State estado = t.getState();
System.out.println("Estado: " + estado);
```

---

3. CREACIÓN DE HILOS EN JAVA Existen **3 formas principales** de crear hilos:

FORMA 1: Extender la clase Thread

```
class MiHilo extends Thread {
    @Override
    public void run() {
        // Código que ejecutará el hilo
        System.out.println("Hilo ejecutándose:  " + Thread.currentThread().getName());
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        MiHilo hilo = new MiHilo();
        hilo.start();  // ¡IMPORTANTE! start(), NO run()
    }
}
```

Ventajas: - Sintaxis simple - Acceso directo a métodos de Thread

Desventajas: - No permite herencia múltiple - Mezcla la tarea con el mecanismo

---

FORMA 2: Implementar Runnable (RECOMENDADO)

```java
class MiTarea implements Runnable {
    @Override
    public void run() {
        System.out.println("Ejecutando tarea");
    }
}

// Uso
public class Main {
    public static void main(String[] args) {
        MiTarea tarea = new MiTarea();
        Thread hilo = new Thread(tarea);
        hilo.start();
    }
}
```

Ventajas: - Deja libre la herencia - Separa tarea de mecanismo de ejecución - Más flexible (se puede pasar a ExecutorService)

FORMA 3: Lambda o Clase Anónima (Java 8+)

```java
// Con Lambda
Thread hilo = new Thread(() -> {
    System.out.println("Ejecutando con lambda");
});
hilo.start();

// Con Clase Anónima
Thread hilo2 = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Clase anónima");
    }
});
hilo2.start();
```

EJEMPLOS COMPLETOS DE HILOS

EJEMPLO 1: Hilo Simple Extendiendo Thread

```java
package com.ceslopedevega. hilos;

// Definición del hilo
class HiloSimple1 extends Thread {
    private int numHilo;
```

```
    private int repeticiones;

    public HiloSimple1(int numHilo, int repeticiones) {
        this.numHilo = numHilo;
        this. repeticiones = repeticiones;
    }

    @Override
    public void run() {
        for (int i = 1; i <= repeticiones; i++) {
            System.out.println("Hilo " + numHilo + " - Iteración " + i);
            try {
                Thread.sleep(500); // Dormir 500ms
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        System.out.println("Hilo " + numHilo + " TERMINADO");
    }
}

// Programa principal
public class LanzaHiloSimple1 {
    public static void main(String[] args) {
        System.out.println("=== INICIANDO HILOS ===");

        HiloSimple1 hilo1 = new HiloSimple1(1, 10);
        HiloSimple1 hilo2 = new HiloSimple1(2, 4);
        HiloSimple1 hilo3 = new HiloSimple1(3, 5);

        // Lanzar hilos
        hilo1.start();
        hilo2.start();
        hilo3.start();

        System.out.println("=== HILOS LANZADOS ===");
        System.out.println("Main continúa ejecutándose...");
    }
}
```

Salida (no determinista):

```
=== INICIANDO HILOS ===
=== HILOS LANZADOS ===
Main continúa ejecutándose...
Hilo 1 - Iteración 1
Hilo 2 - Iteración 1
```

```
Hilo 3 - Iteración 1
Hilo 1 - Iteración 2
Hilo 3 - Iteración 2
Hilo 2 - Iteración 2
...
```

Análisis: - Los hilos se ejecutan **concurrentemente** - El orden NO es predecible - `main()` NO espera a que terminen - Cada hilo tiene su propio stack y variables locales

---

EJEMPLO 2: Implementando Runnable

```java
package com.ceslopedevega.hilos;

// Definición de la tarea
class HiloSimple2 implements Runnable {
    private int numHilo;
    private int repeticiones;

    public HiloSimple2(int numHilo, int repeticiones) {
        this.numHilo = numHilo;
        this.repeticiones = repeticiones;
    }

    @Override
    public void run() {
        for (int i = 1; i <= repeticiones; i++) {
            System.out.printf("[Hilo %d] Iteración %d - Thread: %s\n",
                numHilo, i, Thread.currentThread().getName());
            try {
                Thread.sleep(300);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

// Programa principal
public class LanzaHiloSimple2 {
    public static void main(String[] args) {
        // Crear tareas
        HiloSimple2 tarea1 = new HiloSimple2(1, 10);
        HiloSimple2 tarea2 = new HiloSimple2(2, 4);
        HiloSimple2 tarea3 = new HiloSimple2(3, 5);
```

```
        // Crear hilos con las tareas
        Thread t1 = new Thread(tarea1, "MiHilo-1");
        Thread t2 = new Thread(tarea2, "MiHilo-2");
        Thread t3 = new Thread(tarea3, "MiHilo-3");

        // Lanzar
        t1.start();
        t2.start();
        t3.start();

        System.out.println("Todos los hilos lanzados");
    }
}
```

Diferencia clave:
- La clase `HiloSimple2` puede extender de otra clase si fuera necesario - Separación clara entre la **tarea** (Runnable) y el **mecanismo** (Thread)

EJEMPLO 3: Método join() - Esperar a que Hilos Terminen

```
public class EjemploJoin {
    public static void main(String[] args) {
        System.out. println("Main:  Iniciando");

        Thread t1 = new Thread(() -> {
            try {
                System.out.println("T1: Iniciando trabajo largo.. .");
                Thread.sleep(3000);
                System.out. println("T1: Trabajo completado");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread t2 = new Thread(() -> {
            try {
                System.out.println("T2: Trabajo rápido");
                Thread.sleep(1000);
                System.out.println("T2: Terminado");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
```

```
        t1.start();
        t2.start();

        try {
            System.out.println("Main: Esperando a T1.. .");
            t1.join();  // Espera a que T1 termine
            System.out.println("Main: T1 ha terminado");

            System.out.println("Main: Esperando a T2...");
            t2.join();  // Espera a que T2 termine
            System. out.println("Main: T2 ha terminado");

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

        System.out.println("Main: Todos los hilos han terminado");
    }
}
```

Salida:

```
Main: Iniciando
Main: Esperando a T1...
T1: Iniciando trabajo largo...
T2: Trabajo rápido
T2: Terminado
T1: Trabajo completado
Main: T1 ha terminado
Main:  Esperando a T2...
Main: T2 ha terminado
Main: Todos los hilos han terminado
```

Concepto: - `join()`: Bloquea el hilo actual hasta que el hilo especificado termine - Uso típico: Esperar resultados antes de continuar

---

SINCRONIZACIÓN - EL PROBLEMA

PROBLEMA: Condiciones de Carrera (Race Conditions)

```
class Contador {
    private int c = 0;  // ¡PELIGRO! Variable compartida

    public void incrementa() {
        c++;  // ¡NO ES ATÓMICO!
    }
```

```
    public void decrementa() {
        c--;
    }

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

¿Por qué c++ NO es atómico?

```
c++ en realidad son 3 operaciones:
1. LEER el valor de c desde memoria
2. SUMAR 1
3. ESCRIBIR el nuevo valor en memoria

Si 2 hilos hacen c++ simultáneamente:
Hilo A: LEE c (0)
Hilo B: LEE c (0)      ← Ambos leen 0
Hilo A: SUMA → 1
Hilo B: SUMA → 1       ← Ambos calculan 1
Hilo A: ESCRIBE 1
Hilo B: ESCRIBE 1      ← Resultado: 1 (debería ser 2)
```

EJEMPLO 4: Demostración del Problema (SIN Sincronización)

```
// EJEMPLO.  Hilos:  Bloques NO Sincronizados
// Este ejemplo DEMUESTRA EL PROBLEMA

class Contador {
    private int c = 0;

    Contador(int c) {
        this.c = c;
    }

    public void incrementa() {  // ¡SIN synchronized!
        c++;
    }

    public void decrementa() {  // ¡SIN synchronized!
        c--;
    }
```

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

class HiloSumador extends Thread {
    private Contador contador;

    public HiloSumador(String nombre, Contador c) {
        setName(nombre);
        contador = c;
    }

    public void run() {
        for (int j = 0; j < 300; j++) {
            contador. incrementa();
        }
        System.out.println(getName() + " - contador vale " + contador.valor());
    }
}

class HiloRestador extends Thread {
    private Contador contador;

    public HiloRestador(String nombre, Contador c) {
        setName(nombre);
        contador = c;
    }

    public void run() {
        for (int j = 0; j < 300; j++) {
            contador.decrementa();
        }
        System. out.println(getName() + " - contador vale " + contador.valor());
    }
}

public class BloquesNoSincronizados {
    public static void main(String[] args) {
        System.out.println("--------------------------------");
        System.out.println("Hilos: Bloques NO Sincronizados");
        System.out.println("--------------------------------");

        Contador cont = new Contador(100);
        HiloSumador hiloSuma = new HiloSumador("Hilo Sumador", cont);
        HiloRestador hiloResta = new HiloRestador("Hilo Restador", cont);
```

```
        System.out.println("Valor inicial: " + cont.valor());
        System.out.println("Comienza la ejecución de los hilos .. .");

        hiloSuma.start();
        hiloResta.start();

        try {
            hiloSuma.join();
            hiloResta.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("---------------------------------------");
        System.out.println("Valor final: " + cont.valor());
        System.out.println("ESPERADO: 100");
        System.out.println("¿Coincide? " + (cont.valor() == 100 ? "SÍ" :  "NO  "));
    }
}
```

Posibles salidas (NO determinista):

```
Ejecución 1:
Valor final: 102

Ejecución 2:
Valor final: 97

Ejecución 3:
Valor final: 100  ← Por suerte

Ejecución 4:
Valor final: 105
```

El resultado VARÍA porque hay condiciones de carrera.

SOLUCIONES: SINCRONIZACIÓN

SOLUCIÓN 1: Métodos Sincronizados

```
class Contador {
    private int c = 0;

    Contador(int c) {
        this.c = c;
```

```
    }

    //  synchronized hace que solo 1 hilo pueda ejecutar a la vez
    public synchronized void incrementa() {
        c++;
    }

    public synchronized void decrementa() {
        c--;
    }

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

¿Cómo funciona synchronized?

```
1.  Cada objeto en Java tiene un "lock" o "monitor"
2. Cuando un hilo entra en un método synchronized:
   - Adquiere el lock del objeto
   - Otros hilos que intenten entrar ESPERAN (BLOCKED)
3. Cuando el hilo sale del método:
   - Libera el lock
   - Otro hilo puede adquirirlo
```

EJEMPLO 5: Con Métodos Sincronizados (SOLUCIÓN)

```
// EJEMPLO. Hilos: Métodos Sincronizados

class Contador {
    private int c = 0;

    Contador(int c) {
        this.c = c;
    }

    public synchronized void incrementa() {  //  SINCRONIZADO
        c++;
    }

    public synchronized void decrementa() {  //  SINCRONIZADO
        c--;
    }
```

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

// Los hilos son iguales que antes...
// [HiloSumador y HiloRestador sin cambios]

public class MetodosSincronizados {
    public static void main(String[] args) {
        System.out. println("----------------------------");
        System.out.println("Hilos: Métodos Sincronizados");
        System.out.println("----------------------------");

        Contador cont = new Contador(100);
        HiloSumador hiloSuma = new HiloSumador("Hilo Sumador", cont);
        HiloRestador hiloResta = new HiloRestador("Hilo Restador", cont);

        System.out.println("Valor inicial: " + cont.valor());

        hiloSuma.start();
        hiloResta.start();

        try {
            hiloSuma.join();
            hiloResta.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("------------------------------------");
        System.out. println("Valor final: " + cont.valor());
        System.out.println("ESPERADO:  100");
        System.out.println("¿Coincide? " + (cont.valor() == 100 ? "SÍ  " : "NO"));
    }
}
```

Salida (SIEMPRE):

```
Valor final: 100
ESPERADO:  100
¿Coincide? SÍ
```

Ahora el resultado es SIEMPRE correcto.

SOLUCIÓN 2: Bloques Sincronizados

```
class HiloSumador extends Thread {
    private Contador contador;

    public HiloSumador(String nombre, Contador c) {
        setName(nombre);
        contador = c;
    }

    public void run() {
        synchronized(contador) {  // Sincronizar solo este bloque
            for (int j = 0; j < 300; j++) {
                contador. incrementa();
            }
            System.out.println(getName() + " - contador vale " + contador.valor());
        }
    }
}
```

Ventaja: Más flexible, puedes sincronizar solo la parte crítica.

Sintaxis:

```
synchronized(objeto) {
    // Solo 1 hilo puede estar aquí para este objeto
    // Código protegido
}
```

COMUNICACIÓN ENTRE HILOS: wait() y notify()

PROBLEMA: Productor-Consumidor

- **Productor**: Genera datos y los pone en una cola
- **Consumidor**: Toma datos de la cola y los procesa
- **Problema**: ¿Qué pasa si el consumidor intenta leer cuando no hay datos?

EJEMPLO 6: Productor-Consumidor SIN Sincronización (PROBLEMA)

```
class Cola {
    private int datos = 0;
    private boolean disponible = false;

    public int get() {  // ¡SIN synchronized!
        if (disponible) {
```

```
            disponible = false;
            return datos;
        } else {
            return -1;  // ¡No hay datos!
        }
    }

    public void put(int valor) {  // ¡SIN synchronized!
        datos = valor;
        disponible = true;
    }
}

class Productor extends Thread {
    private Cola cola;
    private int idProductor;

    public Productor(Cola c, int n) {
        cola = c;
        this.idProductor = n;
    }

    public void run() {
        for (int i = 0; i < 5; i++) {
            cola.put(i);
            System.out.println("Productor " + idProductor + " produce: " + i);
            try {
                sleep(100);
            } catch (InterruptedException e) {}
        }
    }
}

class Consumidor extends Thread {
    private Cola cola;
    private int idConsumidor;

    public Consumidor(Cola c, int n) {
        cola = c;
        this.idConsumidor = n;
    }

    public void run() {
        int valor = 0;
        for (int i = 0; i < 5; i++) {
            valor = cola.get();
```

```
            System. out.println("Consumidor " + idConsumidor + " consume: " + valor);
            try {
                sleep(100);
            } catch (InterruptedException e) {}
        }
    }
}

public class ProductorConsumidor {
    public static void main(String[] args) {
        Cola cola = new Cola();

        Productor p = new Productor(cola, 1);
        Consumidor c = new Consumidor(cola, 1);

        p.start();
        c.start();
    }
}
```

Problemas: - El consumidor puede leer `-1` (no hay datos) - El productor puede sobrescribir datos no consumidos - No hay coordinación

---

EJEMPLO 7: Productor-Consumidor CON Sincronización (SOLUCIÓN)

```
class Cola {
    private int datos = 0;
    private boolean disponible = false;

    public synchronized int get() {  //  synchronized
        while (! disponible) {  // Mientras NO haya datos
            try {
                wait();  // Esperar a que haya datos
            } catch (InterruptedException e) {}
        }
        System.out.println("Se consume: " + datos);
        disponible = false;
        notify();  // Notificar al productor
        return datos;
    }

    public synchronized void put(int valor) {  //  synchronized
        while (disponible) {  // Mientras haya datos sin consumir
            try {
```

```
                wait();  // Esperar a que se consuman
            } catch (InterruptedException e) {}
        }
        datos = valor;
        disponible = true;
        System.out. println("Se produce: " + datos);
        notify();  // Notificar al consumidor
    }
}

// Productor y Consumidor sin cambios...

public class ProductorConsumidorSincronizado {
    public static void main(String[] args) {
        Cola cola = new Cola();

        Productor p = new Productor(cola, 1);
        Consumidor c = new Consumidor(cola, 1);

        p.start();
        c.start();
    }
}
```

Salida (ordenada correctamente):

```
Se produce: 0
Se consume: 0
Se produce: 1
Se consume:  1
Se produce: 2
Se consume: 2
...
```

Métodos de comunicación:

Método	Descripción	Debe estar en
`wait()`	Libera el lock y espera	Bloque synchronized
`notify()`	Despierta UN hilo en espera	Bloque synchronized
`notifyAll()`	Despierta TODOS los hilos en espera	Bloque synchronized

Diagrama de flujo:

`Productor:` `Consumidor:`

```
1. Adquiere lock              1. Adquiere lock
2. ¿Hay espacio?              2. ¿Hay datos?
   NO → wait()                   NO → wait()
   SÍ ↓                          SÍ ↓
3. Pone dato                  3. Toma dato
4. notify()                   4. notify()
5. Libera lock                5. Libera lock
```

PRIORIDADES DE HILOS

```java
public class EjemploPrioridades {
    public static void main(String[] args) {
        Thread t1 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System.out.println("T1 (MAX): " + i);
            }
        });

        Thread t2 = new Thread(() -> {
            for (int i = 0; i < 5; i++) {
                System. out.println("T2 (MIN): " + i);
            }
        });

        // Establecer prioridades
        t1.setPriority(Thread.MAX_PRIORITY);   // 10
        t2.setPriority(Thread.MIN_PRIORITY);   // 1

        t1.start();
        t2.start();
    }
}
```

Prioridades disponibles: - `Thread.MIN_PRIORITY` = 1 - `Thread.NORM_PRIORITY` = 5 (predeterminado) - `Thread.MAX_PRIORITY` = 10

IMPORTANTE: - Las prioridades son **sugerencias** al scheduler - NO garantizan orden de ejecución - Dependen del sistema operativo - **Mejor práctica**: NO depender de prioridades

---

HILOS DEMONIO (Daemon Threads)

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

```
        Thread demonio = new Thread(() -> {
            while (true) {
                System. out.println("Demonio trabajando...");
                try {
                    Thread.sleep(500);
                } catch (InterruptedException e) {}
            }
        });

        // Establecer como demonio ANTES de start()
        demonio.setDaemon(true);
        demonio.start();

        Thread usuario = new Thread(() -> {
            for (int i = 0; i < 3; i++) {
                System. out.println("Usuario: " + i);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {}
            }
        });

        usuario.start();

        try {
            usuario.join();
        } catch (InterruptedException e) {}

        System.out. println("Main termina");
        // El demonio termina automáticamente
    }
}
```

Salida:

```
Demonio trabajando...
Usuario: 0
Demonio trabajando...
Demonio trabajando...
Usuario: 1
Demonio trabajando...
Demonio trabajando...
Usuario: 2
Demonio trabajando...
Main termina
```

Características: - **Hilos demonio**: Servicios en segundo plano - **Ter-**

minan automáticamente: Cuando todos los hilos no-demonio terminan - **Usos**: Garbage collector, monitoreo, tareas periódicas - **Importante**: `setDaemon(true)` ANTES de `start()`

POOLS DE HILOS (ExecutorService)

Crear y destruir hilos es costoso. Los **pools** reutilizan hilos.

```
import java.util.concurrent.*;

public class EjemploExecutorService {
    public static void main(String[] args) {
        // Crear pool de 3 hilos
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Enviar 10 tareas al pool
        for (int i = 1; i <= 10; i++) {
            final int numTarea = i;
            executor.submit(() -> {
                System.out.println("Tarea " + numTarea +
                    " ejecutada por " + Thread.currentThread().getName());
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            });
        }

        // Cerrar el executor
        executor.shutdown(); // No acepta nuevas tareas

        try {
            // Esperar a que terminen todas las tareas
            if (!executor.awaitTermination(60, TimeUnit.SECONDS)) {
                executor.shutdownNow(); // Forzar cierre
            }
        } catch (InterruptedException e) {
            executor.shutdownNow();
        }

        System.out.println("Todas las tareas completadas");
    }
}
```

Salida:

```
Tarea 1 ejecutada por pool-1-thread-1
Tarea 2 ejecutada por pool-1-thread-2
Tarea 3 ejecutada por pool-1-thread-3
Tarea 4 ejecutada por pool-1-thread-1  ← Reutiliza thread-1
Tarea 5 ejecutada por pool-1-thread-2
...
```

Tipos de Pools:

```java
// 1. Pool de tamaño fijo
ExecutorService fixed = Executors.newFixedThreadPool(5);

// 2. Pool de 1 solo hilo (ejecución secuencial)
ExecutorService single = Executors.newSingleThreadExecutor();

// 3. Pool con cache (crea hilos según demanda)
ExecutorService cached = Executors.newCachedThreadPool();

// 4. Pool programado (tareas periódicas)
ScheduledExecutorService scheduled = Executors.newScheduledThreadPool(2);
```

TAREAS PERIÓDICAS

```java
import java.util.concurrent.*;

public class EjemploTareasPeriodicas {
    public static void main(String[] args) {
        ScheduledExecutorService scheduler =
            Executors.newScheduledThreadPool(1);

        // Tarea que se ejecuta UNA VEZ después de 2 segundos
        scheduler.schedule(() -> {
            System.out.println("Tarea única ejecutada");
        }, 2, TimeUnit.SECONDS);

        // Tarea que se ejecuta cada 1 segundo
        scheduler.scheduleAtFixedRate(() -> {
            System.out.println("Tarea periódica:  " +
                System.currentTimeMillis());
        }, 0, 1, TimeUnit.SECONDS); // delay inicial=0, período=1s

        // Dejar ejecutar por 10 segundos
        try {
            Thread.sleep(10000);
```

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

        scheduler.shutdown();
    }
}
```

Métodos importantes:

Método	Descripción
`schedule(task, delay, unit)`	Ejecuta una vez después de delay
`scheduleAtFixedRate(task, initialDelay, period, unit)`	Ejecuta periódicamente con período fijo
`scheduleWithFixedDelay(task, initialDelay, delay, unit)`	Ejecuta con delay fijo entre ejecuciones

Diferencia entre FixedRate y FixedDelay:

```
scheduleAtFixedRate (período = 2s):
Inicio: 0s → 2s → 4s → 6s → 8s
(Independiente de la duración de la tarea)

scheduleWithFixedDelay (delay = 2s):
Tarea(1s) → Espera(2s) → Tarea(1s) → Espera(2s)
Inicio: 0s → 3s → 6s → 9s
(Espera a que termine la tarea + delay)
```

OTROS MECANISMOS DE SINCRONIZACIÓN

1. ReentrantLock (Locks Explícitos)

```
import java.util.concurrent.locks.*;

class ContadorConLock {
    private int contador = 0;
    private Lock lock = new ReentrantLock();

    public void incrementar() {
        lock.lock(); // Adquirir lock
        try {
```

```
            contador++;
        } finally {
            lock.unlock(); // SIEMPRE liberar en finally
        }
    }

    public int getValor() {
        lock.lock();
        try {
            return contador;
        } finally {
            lock.unlock();
        }
    }
}

public class EjemploReentrantLock {
    public static void main(String[] args) {
        ContadorConLock contador = new ContadorConLock();

        // Crear 100 hilos que incrementan
        Thread[] hilos = new Thread[100];
        for (int i = 0; i < 100; i++) {
            hilos[i] = new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    contador.incrementar();
                }
            });
            hilos[i].start();
        }

        // Esperar a todos
        for (Thread t : hilos) {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        System.out.println("Valor final: " + contador.getValor());
        System.out.println("Esperado: 100000");
    }
}
```

Ventajas de ReentrantLock sobre synchronized: - Más flexible (tryLock,

lockInterruptibly) - Permite timeout en adquisición - Condiciones múltiples (Condition) - Puede verificar si está bloqueado

```
// Intento de adquirir lock con timeout
if (lock.tryLock(5, TimeUnit. SECONDS)) {
    try {
        // Sección crítica
    } finally {
        lock.unlock();
    }
} else {
    System.out.println("No pude adquirir el lock");
}
```

---

2. Semaphores (Semáforos)

```
import java.util.concurrent. Semaphore;

public class EjemploSemaforo {
    // Semáforo con 3 permisos (máximo 3 hilos concurrentes)
    private static Semaphore semaforo = new Semaphore(3);

    public static void main(String[] args) {
        // Crear 10 hilos
        for (int i = 1; i <= 10; i++) {
            final int id = i;
            new Thread(() -> {
                try {
                    System.out.println("Hilo " + id + " esperando permiso.. .");
                    semaforo. acquire(); // Adquirir permiso
                    System.out.println("Hilo " + id + " TIENE permiso");

                    // Simular trabajo
                    Thread.sleep(2000);

                    System.out.println("Hilo " + id + " LIBERA permiso");
                    semaforo.release(); // Liberar permiso

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

Salida:

```
Hilo 1 esperando permiso...
Hilo 1 TIENE permiso
Hilo 2 esperando permiso...
Hilo 2 TIENE permiso
Hilo 3 esperando permiso...
Hilo 3 TIENE permiso
Hilo 4 esperando permiso...  ← Espera
Hilo 5 esperando permiso...  ← Espera
...
(después de 2 segundos)
Hilo 1 LIBERA permiso
Hilo 4 TIENE permiso  ← Ahora puede entrar
```

Usos: - Limitar acceso a recursos limitados (conexiones a BD, pool de conexiones) - Implementar productores-consumidores con buffer limitado

---

3. CountDownLatch (Barrera de Cuenta Regresiva)

```java
import java.util.concurrent.CountDownLatch;

public class EjemploCountDownLatch {
    public static void main(String[] args) {
        int numHilos = 5;
        CountDownLatch latch = new CountDownLatch(numHilos);

        System.out.println("Iniciando hilos de trabajo...");

        for (int i = 1; i <= numHilos; i++) {
            final int id = i;
            new Thread(() -> {
                try {
                    System.out. println("Hilo " + id + " trabajando...");
                    Thread. sleep((long)(Math.random() * 3000));
                    System.out.println("Hilo " + id + " TERMINADO");
                    latch.countDown(); // Decrementar contador
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }

        try {
            System.out.println("Main esperando a que todos terminen...");
```

```
            latch. await(); // Espera a que el contador llegue a 0
            System. out.println("¡Todos los hilos han terminado!");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Uso: Esperar a que múltiples hilos completen su trabajo antes de continuar.

---

4. CyclicBarrier (Barrera Cíclica)

```
import java.util.concurrent.CyclicBarrier;

public class EjemploCyclicBarrier {
    public static void main(String[] args) {
        int numHilos = 4;

        // Acción a ejecutar cuando todos lleguen a la barrera
        Runnable accionBarrera = () -> {
            System.out.println("\n=== ¡Todos llegaron!  Continuando...  ===\n");
        };

        CyclicBarrier barrera = new CyclicBarrier(numHilos, accionBarrera);

        for (int i = 1; i <= numHilos; i++) {
            final int id = i;
            new Thread(() -> {
                try {
                    System.out.println("Hilo " + id + " - Fase 1");
                    Thread.sleep((long)(Math.random() * 2000));
                    System.out.println("Hilo " + id + " esperando en barrera...");

                    barrera.await(); // Esperar a que todos lleguen

                    System.out.println("Hilo " + id + " - Fase 2");

                } catch (Exception e) {
                    e. printStackTrace();
                }
            }).start();
        }
    }
}
```

Diferencia con CountDownLatch: - **CountDownLatch**: De un solo uso, cuenta regresiva - **CyclicBarrier**: Reutilizable, todos esperan hasta que todos lleguen

---

COLECCIONES CONCURRENTES

Las colecciones estándar (`ArrayList`, `HashMap`) **NO son thread-safe**.

```
import java.util.concurrent.*;

public class EjemploColeccionesConcurrentes {
    public static void main(String[] args) {
        // 1. Lista concurrente
        CopyOnWriteArrayList<String> lista = new CopyOnWriteArrayList<>();

        // 2. Mapa concurrente
        ConcurrentHashMap<String, Integer> mapa = new ConcurrentHashMap<>();

        // 3. Cola concurrente
        BlockingQueue<String> cola = new LinkedBlockingQueue<>();

        // Ejemplo con BlockingQueue (productor-consumidor)
        Thread productor = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    String item = "Item-" + i;
                    cola.put(item); // Se bloquea si la cola está llena
                    System.out.println("Producido: " + item);
                    Thread.sleep(500);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });

        Thread consumidor = new Thread(() -> {
            try {
                for (int i = 1; i <= 5; i++) {
                    String item = cola.take(); // Se bloquea si la cola está vacía
                    System.out. println("Consumido: " + item);
                    Thread.sleep(1000);
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
```

```
        });

        productor.start();
        consumidor.start();
    }
}
```

Colecciones concurrentes disponibles:

Clase	Descripción
`ConcurrentHashMap`	HashMap thread-safe, mejor que Hashtable
`CopyOnWriteArrayList`	ArrayList thread-safe (copia en escritura)
`BlockingQueue`	Cola bloqueante (productor-consumidor)
`LinkedBlockingQueue`	Cola enlazada bloqueante
`ArrayBlockingQueue`	Cola con array de tamaño fijo
`PriorityBlockingQueue`	Cola de prioridad bloqueante

RESUMEN DE CONCEPTOS CLAVE UA2

Concepto	Definición	Uso en Java
Hilo	Unidad ligera de ejecución dentro de un proceso	`Thread`, `Runnable`
start()	Inicia un nuevo hilo de ejecución	`thread.start()`
run()	Método que contiene el código del hilo	Sobrescribir `run()`
join()	Espera a que un hilo termine	`thread.join()`
sleep()	Pausa el hilo actual	`Thread.sleep(ms)`
synchronized	Protege sección crítica	`synchronized method/block`
wait()	Libera lock y espera notificación	En bloque synchronized
notify()	Despierta un hilo en espera	En bloque synchronized
Lock	Sincronización explícita	`ReentrantLock`
Semaphore	Controla acceso a recursos limitados	`Semaphore`
ExecutorService	Pool de hilos reutilizables	`Executors.newFixedThreadPool()`

Concepto	Definición	Uso en Java
Race Condition	Resultado depende del orden de ejecución	Resolver con sincronización
Deadlock	Bloqueo mutuo entre hilos	Evitar con diseño cuidadoso

---

UA3 - PROGRAMACIÓN DE COMUNICACIONES EN RED

CONCEPTOS TEÓRICOS FUNDAMENTALES

1. PROTOCOLOS DE COMUNICACIÓN Modelo OSI simplificado:

```
APLICACIÓN      ← HTTP, FTP, SMTP

TRANSPORTE      ← TCP, UDP

   RED          ← IP

ENLACE/FÍSICA   ← Ethernet, WiFi
```

PROTOCOLO IP (Internet Protocol) Función: Direccionamiento y enrutamiento de paquetes.

Características: - **Sin conexión**: Cada paquete es independiente - **No confiable**: No garantiza entrega - **Mejor esfuerzo**: Intenta entregar pero puede perder paquetes

Direcciones IP:

```
// IPv4: 32 bits (4 bytes)
192.168.1.100  // Ejemplo

// IPv6: 128 bits (16 bytes)
2001:0db8:85a3:0000:0000:8a2e:0370:7334  // Ejemplo
```

PROTOCOLO TCP (Transmission Control Protocol) Características: - **Orientado a conexión**: Establece conexión antes de transmitir -

Confiable: Garantiza entrega y orden - **Control de flujo**: Ajusta velocidad de transmisión - **Control de errores**: Detecta y retransmite paquetes perdidos - **Mayor overhead**: Más lento que UDP

Proceso de conexión TCP (3-way handshake):

```
Cliente                    Servidor

      SYN          >

   <   SYN-ACK

      ACK          >

      Conexión establecida
   <   DATOS        >
```

Casos de uso: - HTTP/HTTPS (navegación web) - FTP (transferencia de archivos) - SMTP (correo electrónico) - SSH (conexión remota) - Bases de datos

---

PROTOCOLO UDP (User Datagram Protocol) Características: - **Sin conexión**: No establece conexión previa - **Rápido**: Menor overhead - **Eficiente**: Ideal para transmisión en tiempo real - **No confiable**: Puede perder paquetes - **Sin orden**: Los paquetes pueden llegar desordenados - **Sin control de flujo**: No ajusta velocidad

Proceso de comunicación UDP:

```
Cliente                    Servidor

      DATAGRAMA      >
        (puede perderse)
   <   DATAGRAMA
        (puede perderse)
```

Casos de uso: - Streaming de video/audio (Netflix, Spotify) - Videojuegos online - VoIP (llamadas por internet) - DNS (resolución de nombres) - Broadcast/Multicast

---

COMPARACIÓN TCP vs UDP

Aspecto	TCP	UDP
Conexión	Orientado a conexión	Sin conexión
Confiabilidad	Garantiza entrega	No garantiza

Aspecto	TCP	UDP
Orden	Mantiene orden	No mantiene orden
Velocidad	Más lento	Más rápido
Overhead	Alto	Bajo
Control de flujo	Sí	No
Control de errores	Sí	No
Uso de ancho de banda	Mayor	Menor
Casos de uso	Transferencias críticas	Transmisión en tiempo real

---

2. PUERTOS **Definición**: Número que identifica un proceso o servicio en una máquina.

Rangos:

```
// Puertos bien conocidos (0-1023)
HTTP:    80
HTTPS:  443
FTP:    21
SSH:    22
SMTP:   25
DNS:    53

// Puertos registrados (1024-49151)
MySQL:   3306
PostgreSQL: 5432
MongoDB: 27017

// Puertos dinámicos/privados (49152-65535)
// Asignados temporalmente por el SO
```

Socket = IP + Puerto:

```
192.168.1.100:8080  ← Dirección del socket
      ↑        ↑
      IP     Puerto
```

---

3. MODELOS DE COMUNICACIÓN **MODELO CLIENTE/SERVIDOR**

```
     SERVIDOR  ← Siempre activo, espera conexiones
```

```
CLI1   CLI2  CLI3 ← Clientes inician conexión
```

Características: - **Servidor**: - Siempre activo - Dirección IP fija conocida - Espera solicitudes - Provee servicios - **Cliente**: - Inicia comunicación - Puede estar inactivo - IP puede variar - Solicita servicios

Ejemplo: Navegación web

```
Cliente (navegador) → Solicita página → Servidor web
Cliente            ← Envía HTML      ← Servidor
```

MODELO P2P (Peer-to-Peer)

```
   P1 ←  → P2


     → P3 ←
```

Características: - Cada nodo es **cliente Y servidor** - No hay servidor central - Descentralizado - Escalable horizontalmente

Ejemplos:
- BitTorrent - Blockchain - Skype (parcialmente)

MODELO HÍBRIDO

```
    SERVIDOR  ← Índice/Coordinación
    (Tracker)




 P1 ←→P2 ←→P3  ← P2P para datos
```

Ejemplos: - BitTorrent con tracker - Spotify (servidor central + P2P)

---

4. SOCKETS **Definición**: Punto final de comunicación bidireccional entre dos programas en red.

Tipos de Sockets:

1. **Stream Sockets (TCP)**
   - Flujo continuo de datos
   - Confiable, ordenado
   - Clase Java: `Socket`, `ServerSocket`
2. **Datagram Sockets (UDP)**
   - Paquetes independientes
   - No confiable, rápido
   - Clase Java: `DatagramSocket`, `DatagramPacket`

Identificación de un Socket:

```
Socket = {Protocolo, IP Local, Puerto Local, IP Remota, Puerto Remoto}

Ejemplo:
{TCP, 192.168.1.5, 3000, 142.250.185.46, 80}
```

CLASES JAVA PARA REDES

CLASE InetAddress - Direcciones IP

```java
import java.net.*;

public class EjemploInetAddress {
    public static void main(String[] args) {
        try {
            // Obtener dirección por nombre de host
            InetAddress google = InetAddress.getByName("www.google.com");
            System.out.println("Nombre:  " + google.getHostName());
            System.out.println("IP: " + google.getHostAddress());

            // Obtener localhost
            InetAddress local = InetAddress.getLocalHost();
            System.out. println("\nLocalhost:");
            System.out.println("Nombre: " + local.getHostName());
            System.out. println("IP: " + local. getHostAddress());

            // Obtener todas las IPs de un host
            InetAddress[] todas = InetAddress.getAllByName("www.google.com");
            System.out.println("\nTodas las IPs de Google:");
            for (InetAddress ip : todas) {
                System.out.println("  " + ip.getHostAddress());
            }
```

```
            // Verificar alcanzabilidad
            boolean alcanzable = google.isReachable(5000); // 5 segundos
            System.out.println("\n¿Google alcanzable?  " + alcanzable);

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

Salida:

```
Nombre: www.google.com
IP: 142.250.185.46

Localhost:
Nombre: MI-PC
IP: 192.168.1.10

Todas las IPs de Google:
  142.250.185.46
  142.250.185.36
  142.250.185.68

¿Google alcanzable? true
```

CLASE URL - Uniform Resource Locator

```
import java.net.*;
import java.io.*;

public class EjemploURL {
    public static void main(String[] args) {
        try {
            // Crear URL
            URL url = new URL("https://www.example.com:8080/path/file.html? param=value#sect

            // Obtener componentes
            System.out.println("URL completa: " + url.toString());
            System.out.println("Protocolo: " + url.getProtocol());
            System.out.println("Host: " + url.getHost());
            System.out.println("Puerto: " + url.getPort());
            System.out.println("Puerto por defecto: " + url.getDefaultPort());
            System.out.println("Archivo: " + url.getFile());
```

```
            System.out.println("Ruta: " + url.getPath());
            System. out.println("Query:  " + url.getQuery());
            System.out.println("Referencia: " + url.getRef());
            System.out.println("Autoridad: " + url.getAuthority());

            // Leer contenido de una URL
            System.out.println("\n=== Leyendo contenido ===");
            URL ejemplo = new URL("http://www.example.com");
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(ejemplo. openStream()));

            String linea;
            while ((linea = reader.readLine()) != null) {
                System.out.println(linea);
            }
            reader.close();

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

Salida:

```
URL completa: https://www.example.com:8080/path/file.html?param=value#section
Protocolo: https
Host: www.example.com
Puerto: 8080
Puerto por defecto: 443
Archivo: /path/file.html? param=value
Ruta: /path/file.html
Query: param=value
Referencia: section
Autoridad: www.example.com:8080
```

CLASE URLConnection

```
import java.net.*;
import java.io.*;

public class EjemploURLConnection {
    public static void main(String[] args) {
        try {
            URL url = new URL("https://www.google.com");
```

```
            URLConnection conexion = url.openConnection();

            // Información de la conexión
            System.out. println("Content-Type: " + conexion. getContentType());
            System. out.println("Content-Length: " + conexion.getContentLength());
            System.out.println("Date: " + new java.util.Date(conexion.getDate()));
            System. out.println("Last-Modified: " + new java.util.Date(conexion.getLastModi

            // Leer contenido
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(conexion.getInputStream()));

            String linea;
            int contador = 0;
            while ((linea = reader.readLine()) != null && contador < 5) {
                System.out.println(linea);
                contador++;
            }
            reader.close();

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

PROGRAMACIÓN CON SOCKETS TCP

ARQUITECTURA CLIENTE-SERVIDOR TCP

```
SERVIDOR                              CLIENTE

    1. Crear ServerSocket(puerto)

    2. accept() ← BLOQUEA

                                 3. new Socket(ip, puerto)

    ← Conexión establecida       >

    4. getInputStream()           4. getOutputStream()
       getOutputStream()             getInputStream()

    5. Leer/Escribir datos  ←  →  Escribir/Leer datos
```

6. close() 6. close()

EJEMPLO COMPLETO: SERVIDOR TCP SIMPLE

```
import java.io.*;
import java.net.*;

public class ServidorTCPSimple {
    public static void main(String[] args) {
        final int PUERTO = 6000;
        ServerSocket servidor = null;
        Socket clienteConectado = null;

        try {
            // 1. Crear ServerSocket
            servidor = new ServerSocket(PUERTO);
            System. out.println("Servidor iniciado en puerto " + PUERTO);
            System.out.println("Esperando cliente.. .");

            // 2. Esperar conexión (BLOQUEA hasta que un cliente se conecte)
            clienteConectado = servidor.accept();
            System.out.println("¡Cliente conectado!");

            // Información del cliente
            System.out. println("IP cliente: " + clienteConectado.getInetAddress().getHostAd
            System.out. println("Puerto remoto: " + clienteConectado.getPort());
            System.out.println("Puerto local: " + clienteConectado.getLocalPort());

            // 3. Crear flujos de entrada/salida
            DataInputStream flujoEntrada = new DataInputStream(
                clienteConectado.getInputStream());
            DataOutputStream flujoSalida = new DataOutputStream(
                clienteConectado. getOutputStream());

            // 4. Recibir mensaje del cliente
            String mensajeCliente = flujoEntrada.readUTF();
            System. out.println("Cliente dice: " + mensajeCliente);

            // 5. Enviar respuesta al cliente
            flujoSalida.writeUTF("Hola cliente, mensaje recibido");

            // 6. Cerrar conexiones
            flujoEntrada.close();
```

```
            flujoSalida.close();
            clienteConectado.close();
            servidor.close();

            System.out.println("Servidor cerrado");

        } catch (IOException e) {
            System.err.println("Error: " + e.getMessage());
            e.printStackTrace();
        } finally {
            // Asegurar cierre de recursos
            try {
                if (clienteConectado != null && !clienteConectado.isClosed())
                    clienteConectado.close();
                if (servidor != null && !servidor.isClosed())
                    servidor.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

EJEMPLO COMPLETO: CLIENTE TCP SIMPLE

```
import java.io.*;
import java.net.*;

public class ClienteTCPSimple {
    public static void main(String[] args) {
        final String HOST = "localhost";
        final int PUERTO = 6000;
        Socket socket = null;

        try {
            System.out.println("Conectando al servidor...");

            // 1. Crear Socket y conectar al servidor
            socket = new Socket(HOST, PUERTO);
            System.out.println("¡Conectado al servidor!");

            // Información de la conexión
            System.out. println("IP local: " + socket.getLocalAddress().getHostAddress());
            System.out.println("Puerto local: " + socket.getLocalPort());
```

```
            System.out.println("IP remota: " + socket.getInetAddress().getHostAddress());
            System.out.println("Puerto remoto: " + socket.getPort());

            // 2. Crear flujos
            DataOutputStream flujoSalida = new DataOutputStream(
                socket.getOutputStream());
            DataInputStream flujoEntrada = new DataInputStream(
                socket.getInputStream());

            // 3. Enviar mensaje al servidor
            flujoSalida.writeUTF("Hola servidor, soy el cliente");
            System.out.println("Mensaje enviado");

            // 4. Recibir respuesta
            String respuesta = flujoEntrada.readUTF();
            System. out.println("Servidor dice: " + respuesta);

            // 5. Cerrar conexiones
            flujoSalida. close();
            flujoEntrada.close();
            socket.close();

            System.out.println("Cliente cerrado");

        } catch (UnknownHostException e) {
            System.err.println("Host desconocido: " + HOST);
        } catch (IOException e) {
            System.err.println("Error de I/O: " + e.getMessage());
            e.printStackTrace();
        } finally {
            try {
                if (socket != null && !socket.isClosed())
                    socket.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Ejecución:

```
Terminal 1 (Servidor):
$ java ServidorTCPSimple
Servidor iniciado en puerto 6000
Esperando cliente...
¡Cliente conectado!
```

```
IP cliente: 127.0.0.1
Puerto remoto: 54321
Puerto local: 6000
Cliente dice: Hola servidor, soy el cliente
Servidor cerrado

Terminal 2 (Cliente):
$ java ClienteTCPSimple
Conectando al servidor...
¡Conectado al servidor!
IP local: 127.0.0.1
Puerto local: 54321
IP remota: 127.0.0.1
Puerto remoto: 6000
Mensaje enviado
Servidor dice:  Hola servidor, mensaje recibido
Cliente cerrado
```

EJEMPLO: SERVIDOR QUE CONVIERTE A MAYÚSCULAS
Servidor:

```java
import java.io.*;
import java.net.*;

public class ServidorMayusculas {
    public static void main(String[] args) {
        final int PUERTO = 6000;

        try (ServerSocket servidor = new ServerSocket(PUERTO)) {
            System. out.println("Servidor de Mayúsculas en puerto " + PUERTO);
            System.out.println("Esperando cliente...");

            try (Socket cliente = servidor.accept();
                 BufferedReader entrada = new BufferedReader(
                     new InputStreamReader(cliente.getInputStream()));
                 PrintWriter salida = new PrintWriter(
                     cliente.getOutputStream(), true)) {

                System.out.println("Cliente conectado desde:  " +
                    cliente.getInetAddress().getHostAddress());

                String mensajeCliente;
                while ((mensajeCliente = entrada.readLine()) != null) {
                    System. out.println("Recibido: " + mensajeCliente);
```

```
                    if (mensajeCliente.equalsIgnoreCase("FIN")) {
                        System. out.println("Cliente solicitó terminar");
                        break;
                    }

                    String mayusculas = mensajeCliente. toUpperCase();
                    salida.println(mayusculas);
                    System.out.println("Enviado: " + mayusculas);
                }

                System.out.println("Conexión cerrada");
            }

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

Cliente:

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

public class ClienteMayusculas {
    public static void main(String[] args) {
        final String HOST = "localhost";
        final int PUERTO = 6000;

        try (Socket socket = new Socket(HOST, PUERTO);
             PrintWriter salida = new PrintWriter(
                 socket.getOutputStream(), true);
             BufferedReader entrada = new BufferedReader(
                 new InputStreamReader(socket.getInputStream()));
             Scanner teclado = new Scanner(System. in)) {

            System.out.println("Conectado al servidor");
            System.out.println("Escribe mensajes (escribe 'FIN' para terminar):");

            String mensaje;
            while (true) {
                System.out.print("> ");
                mensaje = teclado.nextLine();

                // Enviar al servidor
```

```
                salida.println(mensaje);

                if (mensaje.equalsIgnoreCase("FIN")) {
                    System.out.println("Cerrando conexión...");
                    break;
                }

                // Recibir respuesta
                String respuesta = entrada.readLine();
                System.out. println("Servidor:  " + respuesta);
            }

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

Ejecución:

```
Cliente:
> hola mundo
Servidor:  HOLA MUNDO
> java es genial
Servidor:  JAVA ES GENIAL
> FIN
Cerrando conexión...
```

EJEMPLO: SERVIDOR MULTICLIENTE (CON HILOS)

```
import java.io.*;
import java.net.*;

public class ServidorMulticliente {
    private static int contadorClientes = 0;

    public static void main(String[] args) {
        final int PUERTO = 6000;

        try (ServerSocket servidor = new ServerSocket(PUERTO)) {
            System.out. println("Servidor Multicliente iniciado en puerto " + PUERTO);

            while (true) {
                System.out.println("Esperando cliente...");
                Socket cliente = servidor.accept();
```

```
                contadorClientes++;

                System.out.println("Cliente #" + contadorClientes + " conectado");

                // Crear un hilo para atender al cliente
                Thread hiloCliente = new Thread(new ManejadorCliente(cliente, contadorClient
                hiloCliente.start();
            }

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

class ManejadorCliente implements Runnable {
    private Socket socket;
    private int idCliente;

    public ManejadorCliente(Socket socket, int id) {
        this.socket = socket;
        this.idCliente = id;
    }

    @Override
    public void run() {
        try (BufferedReader entrada = new BufferedReader(
                 new InputStreamReader(socket.getInputStream()));
             PrintWriter salida = new PrintWriter(
                 socket.getOutputStream(), true)) {

            salida.println("Bienvenido, eres el cliente #" + idCliente);

            String mensaje;
            while ((mensaje = entrada.readLine()) != null) {
                System.out.println("[Cliente " + idCliente + "]:  " + mensaje);

                if (mensaje.equalsIgnoreCase("ADIOS")) {
                    salida.println("Hasta luego!");
                    break;
                }

                salida.println("[Eco]:  " + mensaje);
            }

            System.out.println("Cliente #" + idCliente + " desconectado");
```

```
            socket.close();

        } catch (IOException e) {
            System.err.println("Error con cliente #" + idCliente);
            e.printStackTrace();
        }
    }
}
```

Ventajas: - Puede atender múltiples clientes simultáneamente - Cada cliente tiene su propio hilo - El servidor continúa aceptando nuevas conexiones

---

PROGRAMACIÓN CON SOCKETS UDP

ARQUITECTURA CLIENTE-SERVIDOR UDP

SERVIDOR	CLIENTE
1. new DatagramSocket(puerto)	1. new DatagramSocket()
2. new DatagramPacket(buffer)	2. new DatagramPacket(datos, ip, puerto)
3. receive(packet) ← BLOQUEA	3. send(packet)
4. Procesar datos	4. receive(response)
5. new DatagramPacket(respuesta)	5. Procesar respuesta
6. send(packet)	
7. close()	6. close()

EJEMPLO COMPLETO: SERVIDOR UDP

```
import java.net.*;
import java.io.*;

public class ServidorUDP {
    public static void main(String[] args) {
        final int PUERTO = 9876;
        DatagramSocket serverSocket = null;

        try {
```

```
        // 1. Crear DatagramSocket
        serverSocket = new DatagramSocket(PUERTO);
        System.out.println("Servidor UDP iniciado en puerto " + PUERTO);

        byte[] bufferRecepcion = new byte[1024];
        byte[] bufferEnvio = new byte[1024];

        System.out.println("Esperando datagramas...");

        while (true) {
            // 2. Preparar paquete para recibir
            DatagramPacket paqueteRecibido = new DatagramPacket(
                bufferRecepcion, bufferRecepcion.length);

            // 3. Recibir datagrama (BLOQUEA)
            serverSocket.receive(paqueteRecibido);

            // 4. Procesar datos recibidos
            String mensajeCliente = new String(paqueteRecibido.getData(),
                0, paqueteRecibido.getLength()).trim();

            InetAddress ipCliente = paqueteRecibido.getAddress();
            int puertoCliente = paqueteRecibido.getPort();

            System.out.println("Recibido de " + ipCliente + ":" + puertoCliente);
            System.out.println("Mensaje: " + mensajeCliente);

            // 5. Convertir a mayúsculas
            String respuesta = mensajeCliente. toUpperCase();
            bufferEnvio = respuesta. getBytes();

            // 6. Enviar respuesta
            DatagramPacket paqueteEnvio = new DatagramPacket(
                bufferEnvio, bufferEnvio.length, ipCliente, puertoCliente);
            serverSocket.send(paqueteEnvio);

            System.out.println("Respuesta enviada:  " + respuesta);
            System.out.println("---");

            // Limpiar buffer
            bufferRecepcion = new byte[1024];
        }

    } catch (IOException e) {
        e.printStackTrace();
    } finally {
```

```
            if (serverSocket != null && !serverSocket.isClosed()) {
                serverSocket.close();
            }
        }
    }
}
```

EJEMPLO COMPLETO: CLIENTE UDP

```
import java.io.*;
import java.net.*;

public class ClienteUDP {
    public static void main(String[] args) {
        final String HOST = "localhost";
        final int PUERTO = 9876;
        DatagramSocket clientSocket = null;

        try {
            // 1. Crear DatagramSocket (puerto automático)
            clientSocket = new DatagramSocket();

            // 2. Obtener dirección del servidor
            InetAddress ipServidor = InetAddress.getByName(HOST);

            // 3. Preparar mensaje
            BufferedReader teclado = new BufferedReader(
                new InputStreamReader(System.in));

            System.out.print("Escribe un mensaje: ");
            String mensaje = teclado.readLine();

            byte[] datosEnvio = mensaje.getBytes();

            // 4. Crear paquete de envío
            DatagramPacket paqueteEnvio = new DatagramPacket(
                datosEnvio, datosEnvio.length, ipServidor, PUERTO);

            // 5. Enviar paquete
            clientSocket.send(paqueteEnvio);
            System.out.println("Mensaje enviado");

            // 6. Preparar recepción de respuesta
            byte[] datosRecepcion = new byte[1024];
```

```
            DatagramPacket paqueteRecepcion = new DatagramPacket(
                datosRecepcion, datosRecepcion.length);

            // 7. Recibir respuesta
            clientSocket.receive(paqueteRecepcion);

            // 8. Procesar respuesta
            String respuesta = new String(paqueteRecepcion.getData(),
                0, paqueteRecepcion.getLength());

            System.out. println("Respuesta del servidor: " + respuesta);

            // 9. Cerrar socket
            clientSocket.close();

        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (clientSocket != null && !clientSocket.isClosed()) {
                clientSocket.close();
            }
        }
    }
}
```

Ejecución:

```
Cliente:
Escribe un mensaje: hola mundo udp
Mensaje enviado
Respuesta del servidor:  HOLA MUNDO UDP
```

EJEMPLO: UDP CON TIMEOUT (Detectar Paquetes Perdidos)

```
import java.net.*;
import java. io.*;

public class ClienteUDPconTimeout {
    public static void main(String[] args) {
        final String HOST = "localhost";
        final int PUERTO = 9876;
        final int TIMEOUT = 5000; // 5 segundos

        try (DatagramSocket socket = new DatagramSocket()) {
```

```
            // Establecer timeout
            socket.setSoTimeout(TIMEOUT);
            System.out.println("Timeout configurado:  " + TIMEOUT + "ms");

            // Preparar y enviar mensaje
            String mensaje = "Mensaje de prueba";
            byte[] envio = mensaje.getBytes();

            InetAddress ip = InetAddress.getByName(HOST);
            DatagramPacket paqueteEnvio = new DatagramPacket(
                envio, envio.length, ip, PUERTO);

            socket.send(paqueteEnvio);
            System.out.println("Paquete enviado");

            // Intentar recibir respuesta
            byte[] recepcion = new byte[1024];
            DatagramPacket paqueteRecepcion = new DatagramPacket(
                recepcion, recepcion.length);

            try {
                socket.receive(paqueteRecepcion);
                String respuesta = new String(paqueteRecepcion.getData(),
                    0, paqueteRecepcion.getLength());
                System.out.println("Respuesta recibida: " + respuesta);

            } catch (SocketTimeoutException e) {
                System.out.println("TIMEOUT: No se recibió respuesta");
                System.out.println("El paquete puede haberse perdido");
            }

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

SERIALIZACIÓN DE OBJETOS

¿Por qué Serialización? Para enviar objetos complejos por red, deben convertirse a bytes.

```
import java.io.*;
import java. net.*;
```

```java
// Clase serializable
class Persona implements Serializable {
    private static final long serialVersionUID = 1L;

    private String nombre;
    private int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String getNombre() { return nombre; }
    public int getEdad() { return edad; }

    @Override
    public String toString() {
        return "Persona{nombre='" + nombre + "', edad=" + edad + "}";
    }
}
```

SERVIDOR QUE RECIBE OBJETOS

```java
import java.io.*;
import java.net.*;

public class ServidorObjetos {
    public static void main(String[] args) {
        final int PUERTO = 6000;

        try (ServerSocket servidor = new ServerSocket(PUERTO)) {
            System. out.println("Servidor de objetos en puerto " + PUERTO);

            Socket cliente = servidor.accept();
            System.out.println("Cliente conectado");

            // Recibir objeto
            ObjectInputStream entrada = new ObjectInputStream(
                cliente.getInputStream());

            Persona persona = (Persona) entrada.readObject();
            System.out.println("Objeto recibido: " + persona);

            // Enviar respuesta
```

```
            ObjectOutputStream salida = new ObjectOutputStream(
                cliente. getOutputStream());

            Persona respuesta = new Persona("Servidor", 100);
            salida.writeObject(respuesta);
            System.out.println("Objeto enviado:  " + respuesta);

            entrada.close();
            salida. close();
            cliente.close();

        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

CLIENTE QUE ENVÍA OBJETOS

```
import java.io.*;
import java.net.*;

public class ClienteObjetos {
    public static void main(String[] args) {
        final String HOST = "localhost";
        final int PUERTO = 6000;

        try (Socket socket = new Socket(HOST, PUERTO)) {
            System. out.println("Conectado al servidor");

            // Enviar objeto
            ObjectOutputStream salida = new ObjectOutputStream(
                socket.getOutputStream());

            Persona persona = new Persona("Juan", 25);
            salida.writeObject(persona);
            System.out.println("Objeto enviado: " + persona);

            // Recibir objeto
            ObjectInputStream entrada = new ObjectInputStream(
                socket.getInputStream());

            Persona respuesta = (Persona) entrada.readObject();
            System.out.println("Objeto recibido: " + respuesta);
```

```
            salida.close();
            entrada. close();

        } catch (IOException | ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

ERRORES COMUNES Y BUENAS PRÁCTICAS

1. No cerrar Sockets

```
//  MAL
Socket socket = new Socket(HOST, PORT);
// ...  usar socket ...
// Olvidar cerrar → FUGA DE RECURSOS

//  BIEN - try-with-resources
try (Socket socket = new Socket(HOST, PORT)) {
    // ... usar socket ...
} // Se cierra automáticamente
```

2. No usar trim() en UDP

```
//  MAL
String mensaje = new String(packet.getData());
// Incluye bytes basura del buffer

//  BIEN
String mensaje = new String(packet. getData(), 0, packet.getLength()).trim();
```

3. Olvidar flush() en streams

```
//  MAL - Los datos pueden quedarse en buffer
PrintWriter out = new PrintWriter(socket.getOutputStream());
out.println("mensaje");

//  BIEN - Forzar envío
PrintWriter out = new PrintWriter(socket.getOutputStream(), true); // autoFlush
// O manualmente:
out.println("mensaje");
out.flush();
```

4. No manejar excepciones específicas

```
//  MAL
try {
    Socket s = new Socket(HOST, PORT);
} catch (Exception e) {
    // Demasiado genérico
}

//  BIEN
try {
    Socket s = new Socket(HOST, PORT);
} catch (UnknownHostException e) {
    System.err.println("Host desconocido");
} catch (ConnectException e) {
    System.err.println("Conexión rechazada");
} catch (IOException e) {
    System.err.println("Error de I/O");
}
```

BATERÍA DE PREGUNTAS Y RESPUESTAS

UA1 - PROGRAMACIÓN MULTIPROCESO

P1: ¿Cuál es la diferencia fundamental entre Process Builder y Runtime. exec()? R: `ProcessBuilder` es más moderno, flexible y recomendado. Permite: - Configurar directorio de trabajo con `directory()` - Modificar variables de entorno con `environment()` - Redirigir entrada/salida/error fácilmente con `redirectInput/Output/Error()` - API más clara y fácil de usar `Runtime.exec()` es más antiguo y menos flexible, considerado legacy.

P2: ¿Por qué es importante usar flush() al enviar datos a un proceso? R: Porque los datos se almacenan en un buffer antes de enviarse. Sin `flush()`, los datos pueden quedarse en memoria y no llegar al proceso destino, causando bloqueos donde el proceso espera datos que nunca llegan.

P3: Explica getInputStream() vs getOutputStream() en Process R: - `getInputStream()`: Obtiene la **salida estándar (stdout)** del proceso hijo. Lo que el hijo imprime, el padre lo lee por aquí. - `getOutputStream()`: Obtiene la **entrada estándar (stdin)** del proceso hijo. El padre escribe aquí para enviar datos al hijo. Es confuso porque desde la perspectiva del padre, "lee" de un InputStream la "salida" del hijo.

P4: ¿Qué indica un código de salida diferente de 0? R: Por convención Unix/Linux: - `0`: Éxito, ejecución correcta - `!= 0`: Error. Cada valor puede indicar un tipo de error específico (ej: 1=error genérico, 2=uso incorrecto, 127=comando no encontrado). Se obtiene con `process.waitFor()` o

`process.exitValue()`.

P5: ¿Qué hace waitFor() y por qué es importante? R: `waitFor()` bloquea el hilo actual hasta que el proceso hijo termine. Es importante porque: - Sin él, el padre puede terminar antes que el hijo (proceso huérfano) - Permite obtener el código de salida - Sincroniza la ejecución padre-hijo - Evita procesos zombies

P6: ¿Cuál es la diferencia entre destroy() y destroyForcibly()? R: - `destroy()`: Solicita terminación "gentil", permite al proceso hacer limpieza - `destroyForcibly()`: Terminación inmediata y forzosa, puede dejar recursos sin liberar En Windows ambos suelen tener el mismo efecto (terminación inmediata).

P7: ¿Qué riesgo existe si no lees el InputStream de un proceso? R: Deadlock por buffers llenos. Si el proceso hijo genera mucha salida y el padre no la lee, el buffer se llena y el hijo se bloquea esperando espacio, mientras el padre espera que el hijo termine. Solución: leer la salida en paralelo con un hilo separado.

P8: ¿Por qué los procesos NO comparten variables? R: Porque cada proceso tiene su propio espacio de memoria aislado por el sistema operativo. Esto proporciona seguridad y estabilidad, pero complica la comunicación, requiriendo IPC (Inter-Process Communication) mediante streams, pipes, archivos compartidos o sockets.

P9: ¿Qué es más eficiente, procesos o hilos? ¿Por qué? R: Hilos son más eficientes porque: - Comparten memoria (no necesitan copiarla) - Creación más rápida (microsegundos vs milisegundos) - Cambio de contexto más rápido - Menor overhead del sistema Pero los procesos ofrecen mejor aislamiento y seguridad.

P10: ¿Qué significa que un proceso esté en estado BLOCKED? R: El proceso está esperando un evento externo y no puede continuar: - Operación de E/S (lectura de disco, red) - Entrada del usuario - Señal de otro proceso - Recurso ocupado No consume CPU en este estado, el SO lo mueve a una cola de espera.

UA2 - PROGRAMACIÓN MULTIHILO

P11: ¿Por qué NUNCA debes llamar directamente a run()? R: Porque `run()` es un método normal que se ejecuta en el hilo actual (el que hace la llamada), NO crea un nuevo hilo. `start()` es el que: 1. Crea un nuevo hilo 2. Inicializa los recursos del hilo 3. Invoca `run()` en el nuevo hilo

P12: ¿Qué pasa si llamas a start() dos veces en el mismo Thread? R: Lanza `IllegalThreadStateException`. Un hilo solo puede iniciarse una vez.

Una vez que termina (TERMINATED), no puede reiniciarse. Debes crear un nuevo objeto Thread.

P13: Explica la diferencia entre BLOCKED y WAITING R: - **BLOCKED**: El hilo está esperando adquirir un lock/monitor que otro hilo tiene. Es involuntario, causado por sincronización. - **WAITING**: El hilo llamó explícitamente a `wait()`, `join()` sin timeout, o `LockSupport.park()`. Espera una notificación específica de otro hilo.

P14: ¿Por qué c++ NO es una operación atómica? R: Porque en realidad son 3 operaciones:

```
1.  LEER c desde memoria → registro
2. SUMAR 1 al registro
3. ESCRIBIR registro → memoria
```

Si 2 hilos ejecutan esto simultáneamente, pueden intercalarse causando que ambos lean el mismo valor y solo se incremente una vez en lugar de dos (race condition).

P15: ¿Qué hace realmente synchronized? R: Adquiere el **monitor/lock** del objeto. Solo un hilo puede tener el lock a la vez. Otros hilos que intenten entrar al mismo bloque/método synchronized sobre el mismo objeto quedarán BLOCKED hasta que el lock se libere.

P16: ¿Cuál es la diferencia entre synchronized method y synchronized block? R: - **Método**: Bloquea todo el método, el lock es sobre `this` (o la clase si es estático) `java public synchronized void metodo() { }` - **Bloque**: Puedes sincronizar solo parte del código y elegir el objeto del lock `java synchronized(objetoEspecifico) { }` Más flexible y eficiente.

P17: ¿Qué hacen wait(), notify() y notifyAll()? R: - **wait()**: Libera el lock y pone el hilo en WAITING. Debe estar en bloque synchronized. - **notify()**: Despierta UN hilo que esté en wait() sobre el mismo objeto. - **notifyAll()**: Despierta TODOS los hilos en wait() sobre ese objeto. El hilo despertado debe re-adquirir el lock antes de continuar.

P18: ¿Por qué usar while en lugar de if con wait()? R: Por **spurious wakeups** (despertares espurios). Un hilo puede despertar sin que se haya llamado notify(), o la condición puede cambiar entre el notify() y la re-adquisición del lock. `while` verifica la condición nuevamente:

```java
while (!condicion) {
    wait(); // Vuelve a esperar si la condición sigue siendo falsa
}
```

P19: ¿Qué es un hilo demonio y cuándo termina? R: Un hilo demonio (daemon) es un hilo de servicio en segundo plano. Características: - Se marca con `setDaemon(true)` ANTES de `start()` - Termina automáticamente cuando todos los hilos no-demonio terminan - La JVM no espera a que terminen para finalizar - Usos: garbage collector, timers, monitorización

P20: ¿Qué ventaja tiene ExecutorService sobre crear hilos manualmente? R: - **Reutilización de hilos**: No crea/destruye hilos constantemente (costoso) - **Gestión automática**: Controla el número de hilos activos - **Mejor rendimiento**: Pool de hilos listos para usar - **Simplifica el código**: No necesitas gestionar ciclo de vida - **Future**: Facilita obtener resultados de tareas - **Separación de conceptos**: La tarea (Runnable/Callable) está separada del mecanismo de ejecución

P21: Diferencia entre scheduleAtFixedRate y scheduleWithFixedDelay R: - **scheduleAtFixedRate**: Período fijo entre **inicios** de ejecución. Si la tarea tarda 1s y el período es 3s → ejecuta a los 0s, 3s, 6s, 9s… - **scheduleWithFixedDelay**: Delay fijo entre **fin de una ejecución e inicio de la siguiente**. Si tarea=1s y delay=3s → 0s, 4s (1+3), 8s (4+1+3)…

P22: ¿Qué es un deadlock y cómo se previene? R: Deadlock es cuando dos o más hilos esperan indefinidamente por recursos retenidos mutuamente. **Condiciones de Coffman** (las 4 deben cumplirse): 1. Exclusión mutua 2. Retención y espera 3. No apropiación 4. Espera circular

Prevención: Romper al menos una condición: - Orden consistente al adquirir locks - Timeout al intentar adquirir - Usar `tryLock()` en lugar de `lock()` - Evitar nested locks

P23: ¿Qué es starvation (inanición)? R: Un hilo nunca obtiene CPU/recursos necesarios para ejecutarse. Causas: - Hilos de alta prioridad monopolizan CPU - Algoritmo de scheduling injusto - Lock siempre adquirido por los mismos hilos **Solución**: Algoritmos de scheduling justos (fair scheduling), evitar dependencia excesiva de prioridades.

P24: ¿Por qué ReentrantLock es mejor que synchronized en algunos casos? R: - **tryLock()**: Intenta adquirir lock sin bloquearse - **tryLock(timeout)**: Intenta con timeout - **lockInterruptibly()**: Puede ser interrumpido mientras espera - **Fairness**: Puede garantizar orden FIFO - **Múltiples Conditions**: Más de una condición de espera - **isLocked()**: Verifica si está bloqueado Pero `synchronized` es más simple y suficiente para la mayoría de casos.

P25: ¿Qué es una race condition y cómo se soluciona? R: Condición donde el resultado depende del orden impredecible de ejecución de hilos que acceden a recursos compartidos.
Ejemplo: Dos hilos incrementan un contador compartido, ambos leen 0, ambos calculan 1, ambos escriben 1 → resultado 1 en lugar de 2. **Solución**: - Sincronización con `synchronized` - Locks explícitos (ReentrantLock) - Variables atómicas (AtomicInteger) - Colecciones concurrentes

P26: ¿Qué diferencia hay entre ConcurrentHashMap y Hashtable? R: - **Hashtable**: Sincroniza TODO el mapa en cada operación (un lock global) → lento en concurrencia - **ConcurrentHashMap**: Usa "lock striping" (múltiples locks para diferentes segmentos) → múltiples hilos pueden escribir simultánea-

mente en diferentes partes → mucho más rápido Nunca usar Hashtable, ConcurrentHashMap es superior en todos los aspectos.

P27: ¿Qué hace join() y cuándo usarlo? R: `join()` hace que el hilo actual espere a que el hilo especificado termine. **Uso típico**: Esperar resultados antes de continuar

```
Thread worker = new Thread(() -> calcularResultado());
worker.start();
// ... hacer otras cosas ...
worker.join(); // Esperar a que termine
// Ahora puedo usar el resultado
```

También existe `join(millis)` para esperar con timeout.

P28: ¿Por qué CopyOnWriteArrayList es thread-safe? R: Porque cada operación de **escritura** crea una **copia nueva** del array interno. Las lecturas no requieren locks porque leen del array inmutable actual.
- Ideal cuando hay muchas lecturas y pocas escrituras - Costoso si hay muchas escrituras (copia todo el array) - Iteradores nunca lanzan ConcurrentModificationException

P29: Explica el patrón Producer-Consumer R: Patrón donde: - **Productor(es)**: Generan datos y los ponen en un buffer/cola - **Consumidor(es)**: Toman datos del buffer y los procesan - **Sincronización necesaria**: - Productor espera si el buffer está lleno - Consumidor espera si el buffer está vacío **Solución en Java**: - `wait()/notify()` con synchronized - `BlockingQueue` (más simple)

P30: ¿Qué es un semáforo y para qué sirve? R: Semáforo es un contador que controla acceso a recursos limitados. Tiene "permisos":

```
Semaphore sem = new Semaphore(3); // 3 permisos
sem.acquire(); // Toma un permiso (bloquea si no hay)
// ... usar recurso ...
sem.release(); // Devuelve permiso
```

Uso: Limitar número de hilos que acceden a un recurso (ej: pool de conexiones a BD con máximo 10 conexiones).

UA3 - PROGRAMACIÓN DE COMUNICACIONES EN RED

P31: Diferencias fundamentales entre TCP y UDP R:

Aspecto	TCP	UDP
Conexión	Orientado a conexión (3-way handshake)	Sin conexión
Confiabilidad	Garantiza entrega y orden	No garantiza nada
Velocidad	Más lento (overhead)	Más rápido

Aspecto	TCP	UDP
Control de flujo	Sí	No
Retransmisión	Sí (paquetes perdidos se reenvían)	No
Uso típico	HTTP, FTP, Email	Streaming, Gaming, DNS

P32: ¿Qué es un socket? **R:** Un socket es un endpoint (punto final) de comunicación bidireccional entre dos programas en red. Se identifica por: `{Protocolo, IP, Puerto}`. Es una abstracción que permite comunicación de red como si fuera lectura/escritura de archivos.

P33: ¿Qué hace ServerSocket. accept()? **R:** `accept()` es un método **bloqueante** que: 1. Espera a que un cliente se conecte 2. Cuando un cliente conecta, crea y devuelve un **nuevo Socket** para esa conexión 3. El ServerSocket sigue escuchando en el puerto original Es el punto donde el servidor "se duerme" esperando clientes.

P34: ¿Por qué accept() es bloqueante y qué implica? **R:** Es bloqueante porque detiene la ejecución del hilo hasta que llegue una conexión. **Implicaciones**: - El servidor no puede hacer nada más mientras espera - Para atender múltiples clientes, necesitas crear un **hilo nuevo** por cada `accept()` - Patrón típico:

```
while(true) {
    Socket cliente = servidor.accept(); // Bloquea aquí
    new Thread(new ManejadorCliente(cliente)).start(); // Hilo propio
}
```

P35: ¿Cuál es la diferencia entre Socket y ServerSocket? **R:** - **ServerSocket**: Usado por el **servidor**, solo escucha en un puerto esperando conexiones. No transmite datos. Su `accept()` devuelve un Socket. - **Socket**: Representa una **conexión establecida**. Tanto cliente como servidor usan Socket para comunicarse (leer/escribir datos).

P36: Explica getInputStream() y getOutputStream() en Socket **R:** - **getInputStream()**: Obtiene flujo para **leer datos** que llegan del otro extremo - **getOutputStream()**: Obtiene flujo para **enviar datos** al otro extremo

```
// Servidor
InputStream in = socket.getInputStream(); // Lee lo que envía el cliente
OutputStream out = socket.getOutputStream(); // Envía al cliente

// Cliente
OutputStream out = socket.getOutputStream(); // Envía al servidor
InputStream in = socket.getInputStream(); // Lee respuesta del servidor
```

P37: ¿Qué es el 3-way handshake de TCP? **R:** Proceso de establecimiento de conexión TCP: 1. **Cliente → Servidor: SYN** (Solicitud de sincronización) 2. **Servidor → Cliente: SYN-ACK** (Confirmación + sincronización) 3.

Cliente → Servidor: ACK (Confirmación final) Después de esto, la conexión está establecida y pueden intercambiar datos.

P38: ¿Por qué UDP no necesita accept()? R: Porque UDP **no establece conexión**. Es "connectionless". TCP necesita `accept()` para: - Aceptar solicitud de conexión - Crear Socket dedicado para esa conexión - Mantener estado de conexión UDP simplemente envía/recibe datagramas independientes sin concepto de "conexión establecida".

P39: ¿Qué es un DatagramPacket? R: Un DatagramPacket es un paquete UDP que encapsula: - **Datos** (array de bytes) - **Longitud** de los datos - **Dirección IP** de destino/origen - **Puerto** de destino/origen Cada paquete es independiente y puede llegar en cualquier orden o perderse.

P40: ¿Por qué es importante usar trim() al leer DatagramPacket? R: Porque el buffer del DatagramPacket tiene tamaño fijo (ej: 1024 bytes), pero el mensaje real puede ser más pequeño (ej: 50 bytes). Sin `trim()`:

```
// Buffer de 1024, mensaje de 50 bytes
String msg = new String(packet.getData()); // 1024 chars con basura

// CORRECTO
String msg = new String(packet.getData(), 0, packet.getLength()).trim();
```

P41: ¿Qué hace setSoTimeout() y por qué es útil? R: Establece un timeout máximo para operaciones bloqueantes (receive(), read()).

```
socket.setSoTimeout(5000); // 5 segundos
socket.receive(packet); // Lanza SocketTimeoutException si no recibe en 5s
```

Utilidad: - Evita bloqueos indefinidos - Detecta paquetes perdidos en UDP - Permite reintentos - Mejora robustez

P42: ¿Qué información puedo obtener de un Socket conectado? R:

```
// Información remota
socket.getInetAddress();      // IP del otro extremo
socket.getPort();             // Puerto remoto

// Información local
socket.getLocalAddress();     // Mi IP
socket.getLocalPort();        // Mi puerto

// Estado
socket.isConnected();         // ¿Conectado?
socket.isClosed();            // ¿Cerrado?
socket.isBound();             // ¿Vinculado a puerto?

// Configuración
socket.getSoTimeout();        // Timeout configurado
```

```
socket.getKeepAlive();        // Keep-alive activo
socket.getTcpNoDelay();       // Algoritmo de Nagle
```

P43: ¿Qué es Serializable y por qué es necesario? R: `Serializable` es una interfaz marcador que indica que un objeto puede convertirse a bytes (serialización). **Necesario porque**: - Los objetos en memoria no pueden enviarse por red directamente - Hay que convertirlos a secuencia de bytes - Al recibir, se reconstruye el objeto (deserialización)

```
class MiClase implements Serializable {
    private static final long serialVersionUID = 1L;
    // ...
}
```

Sin `Serializable`, lanza `NotSerializableException`.

P44: Diferencia entre DataInputStream/DataOutputStream y ObjectInputStream/ObjectOutputStream R: - DataInputStream/DataOutputStream: Para tipos primitivos y Strings `java dos.writeInt(123); dos.writeUTF("texto"); int n = dis.readInt(); String s = dis.readUTF();` - **ObjectInputStream/ObjectOutputStream**: Para objetos completos (requiere Serializable) `java oos.writeObject(miObjeto); MiClase obj = (MiClase) ois.readObject();`

P45: ¿Cómo implementar un servidor multicliente? R: Usando **hilos**:

```
ServerSocket servidor = new ServerSocket(puerto);
while(true) {
    Socket cliente = servidor.accept(); // Acepta conexión
    new Thread(new ManejadorCliente(cliente)).start(); // Hilo dedicado
}
```

Cada cliente es atendido en su propio hilo, permitiendo múltiples clientes simultáneos. Alternativa moderna: ExecutorService con pool.

P46: ¿Por qué cerrar streams antes que el socket? R: Porque: 1. Los streams pueden tener datos en buffer que necesitan enviarse 2. Cerrar el socket abruptamente puede perder datos en tránsito 3. Los streams pueden necesitar hacer limpieza 4. Cerrar el socket cierra automáticamente los streams asociados, pero es buena práctica cerrarlos explícitamente **Orden correcto**:

```
flujoEntrada.close();
flujoSalida.close();
socket.close();
```

P47: ¿Qué son los puertos bien conocidos? R: Puertos en el rango 0-1023 reservados para servicios estándar: - HTTP: 80 - HTTPS: 443 - FTP: 21 - SSH: 22 - SMTP: 25 - DNS: 53 Requieren privilegios de administrador para usar en servidores.

P48: ¿Qué hace flush() en streams de red? R: Fuerza el **envío inmediato**

de datos que están en el buffer. Sin flush(), los datos pueden quedarse en memoria esperando: - Que se llene el buffer - Que se cierre el stream Esto puede causar: - Deadlocks (el receptor espera datos que nunca llegan) - Delays innecesarios

```
out.write(datos);
out.flush(); // ¡IMPORTANTE!
```

P49: Explica el modelo Cliente/Servidor R: - Servidor: - Proceso que siempre está activo - Tiene IP/dominio conocido - Escucha en un puerto específico - Espera y responde a peticiones - Provee servicios - **Cliente**: - Inicia la comunicación - Puede estar inactivo la mayor parte del tiempo - IP puede ser dinámica - Solicita servicios **Ejemplo**: Navegación web (cliente=navegador, servidor=servidor web).

P50: ¿Cuándo usar TCP y cuándo UDP? R:

Usar TCP cuando: - La confiabilidad es crítica (transferencias de archivos, emails) - Necesitas garantizar orden de datos - No importa latencia ligeramente mayor - Ejemplo: HTTP, FTP, bases de datos

Usar UDP cuando: - La velocidad es más importante que la confiabilidad - Puedes tolerar pérdida de paquetes - Transmisión en tiempo real - Overhead debe ser mínimo - Ejemplo: Streaming, videojuegos, VoIP, DNS

CONSEJOS FINALES PARA EL EXAMEN

Para UA1 - Procesos:

1. Domina `ProcessBuilder` y sus métodos principales
2. Conoce la diferencia entre getInputStream/getOutputStream/getErrorStream
3. Practica enviar/recibir datos de procesos
4. Siempre controla códigos de salida con waitFor()
5. Recuerda que los procesos NO comparten memoria
6. Usa try-with-resources o finally para cerrar recursos
7. No olvides flush() al enviar datos

Para UA2 - Hilos:

1. NUNCA llames a run() directamente, siempre start()
2. Entiende bien synchronized (método y bloque)
3. Practica wait()/notify() para productor-consumidor
4. Preferir Runnable sobre extends Thread
5. Conoce los estados de un hilo (NEW, RUNNABLE, BLOCKED, WAITING, TERMINATED)
6. Usa join() cuando necesites esperar resultados
7. Recuerda que variables compartidas necesitan sincronización

8. ExecutorService para pools de hilos reutilizables
9. while con wait(), no if

Para UA3 - Redes:

1. Conoce diferencias TCP vs UDP a fondo
2. ServerSocket.accept() es bloqueante → multicliente con hilos
3. En UDP usa trim() al leer datos
4. DataInputStream/DataOutputStream para tipos primitivos
5. ObjectInputStream/ObjectOutputStream para objetos (requiere Serializable)
6. Siempre cerrar: streams primero, socket después
7. flush() después de escribir datos importantes
8. try-with-resources para gestión automática de recursos
9. Maneja excepciones específicas (UnknownHostException, ConnectException)
10. setSoTimeout() para evitar bloqueos indefinidos

Errores Críticos a Evitar:

Llamar a run() en lugar de start() No sincronizar acceso a variables compartidas Olvidar flush() al enviar datos No cerrar sockets/streams (fugas de recursos) Confundir getInputStream (salida del proceso) con entrada No usar trim() en UDP No manejar SocketTimeoutException en UDP Mezclar puertos locales con remotos No hacer waitFor() cuando necesitas código de salida Usar if en lugar de while con wait()

RESUMEN DE CLASES IMPORTANTES

UA1:

- `ProcessBuilder` - Crear y configurar procesos
- `Process` - Representación de proceso en ejecución
- `InputStream/OutputStream` - Comunicación con procesos
- `BufferedReader/PrintWriter` - Lectura/escritura de texto

UA2:

- `Thread` - Representación de hilo
- `Runnable` - Interfaz para tareas
- `ExecutorService` - Pool de hilos
- `Semaphore` - Control de acceso a recursos
- `ReentrantLock` - Lock explícito
- `CountDownLatch` - Barrera de cuenta regresiva
- `CyclicBarrier` - Barrera cíclica
- `ConcurrentHashMap` - Mapa thread-safe

- `BlockingQueue` - Cola bloqueante

UA3:

- `InetAddress` - Direcciones IP
- `URL/URLConnection` - URLs y conexiones HTTP
- `Socket` - Cliente TCP
- `ServerSocket` - Servidor TCP
- `DatagramSocket` - UDP
- `DatagramPacket` - Paquetes UDP
- `DataInputStream/DataOutputStream` - Tipos primitivos
- `ObjectInputStream/ObjectOutputStream` - Objetos

PLANTILLAS DE CÓDIGO ÚTILES

Servidor TCP Multicliente:

```
ServerSocket servidor = new ServerSocket(puerto);
while(true) {
    Socket cliente = servidor.accept();
    new Thread(() -> {
        try (BufferedReader in = new BufferedReader(
                new InputStreamReader(cliente.getInputStream()));
             PrintWriter out = new PrintWriter(
                cliente.getOutputStream(), true)) {
            // Atender cliente
        } catch (IOException e) {
            e.printStackTrace();
        }
    }).start();
}
```

Cliente TCP Simple:

```
try (Socket socket = new Socket(host, puerto);
     PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
     BufferedReader in = new BufferedReader(
        new InputStreamReader(socket.getInputStream()))) {
    out.println("mensaje");
    String respuesta = in.readLine();
} catch (IOException e) {
    e.printStackTrace();
}
```

Hilo con Runnable:

```
Thread hilo = new Thread(() -> {
    // Código del hilo
});
hilo.start();
hilo.join(); // Esperar
```

Synchronized Básico:

```
private Object lock = new Object();
synchronized(lock) {
    // Sección crítica
}
```

¡MUCHA SUERTE EN EL EXAMEN!

Este resumen cubre **TODOS** los conceptos teóricos y prácticos de las UA1, UA2 y UA3 con ejemplos completos en Java. Estudia especialmente: - Los ejemplos de código completos - Las diferencias entre conceptos similares - Las preguntas y respuestas teóricas - Los errores comunes a evitar

¡Éxito!