

# Programación Avanzada en Java

## Hilos(Threads)

---

Dra. María Elena Lárraga Ramírez

M.C. Fernando Reyes Gómez

Ing. Laura Evelyn Gómez Suárez

Esp. Israel Velázquez Gutiérrez

Noviembre, 2022



# Agenda

1. Antecedentes de la Concurrency.
2. Concurrency.
3. Concurrency en Java (Hilos).
4. Mecanismos de interrupción.

Parte 1

5. Mecanismos de sincronización.
6. La API de concurrency.
7. Patrones de concurrency
  - A. Productor – Consumidor.
  - B. Manager – Workers.

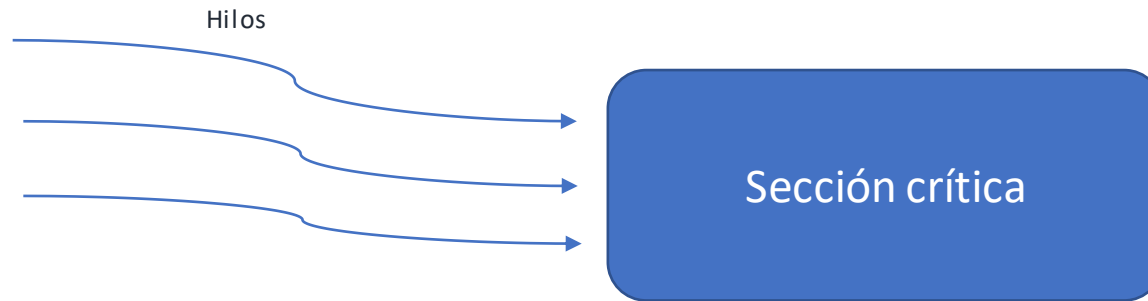
Parte 2

8. Proyecto final

# Mecanismos de Sincronización

# La Sección Crítica

Es el fragmento de código donde se modifica cualquier recurso compartido entre dos o más hilos, normalmente se trata de variables compartidas.



## Problemática

Sí más de un hilo ingresa simultáneamente a modificar los valores de la sección crítica, ésta se corrompe y el resultado final es incierto.

## Objetivo

Garantizar que solo un hilo a la vez, pueda emplear esta sección, para evitar incoherencias.

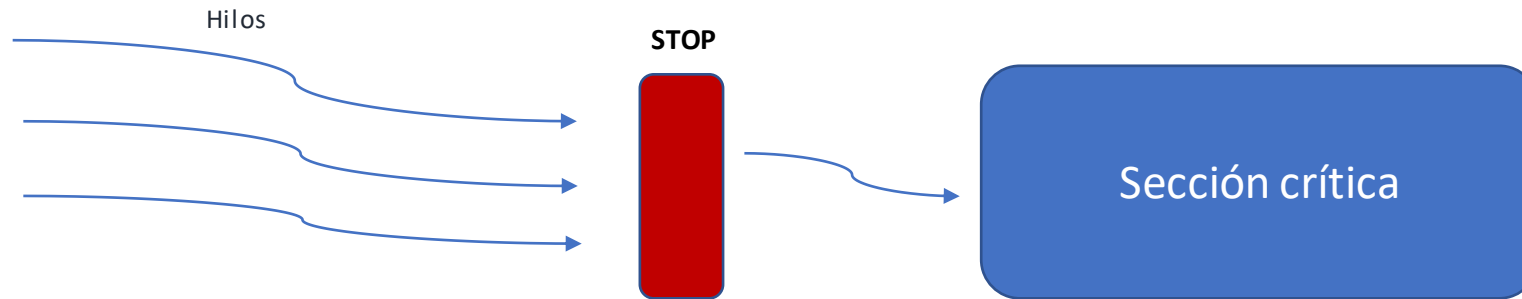
## ¿Cómo podemos proteger la sección crítica?

# Propuesta de solución

En 1965 E. W. Dijkstra, identificó que cuando 2 o mas procesos son cooperantes concurrentemente, se presentaban escenarios donde se comprometía el recurso compartido, y propuso un mecanismo de sincronización, al que llamo “**Exclusión Mutua**”



Edsger Wybe Dijkstra



La **exclusión mutua**, es un mecanismo de interrupciones para asegurar que el **Scheduler** otorgue exclusividad de la **sección crítica** a un solo hilo en un tiempo determinado  $t$ , donde  $t \geq 1$  ciclo de reloj del CPU.

# Mecanismos de sincronización

Mecanismos para implementar exclusión mutua:

- ☐ Monitores

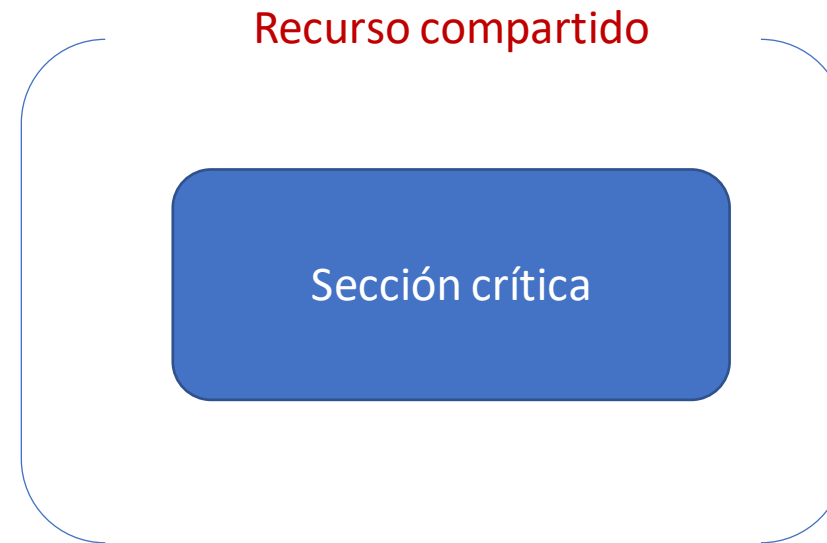
- ☐ Semáforos

- ☐ Candados

# Monitores

Los **Monitores**, son directivas a nivel bloque de código, que protegen a la **sección crítica** con mecanismos de bloqueo para lograr la sincronización y la exclusividad de acceso, es decir; es una forma de implementar exclusión mutua.

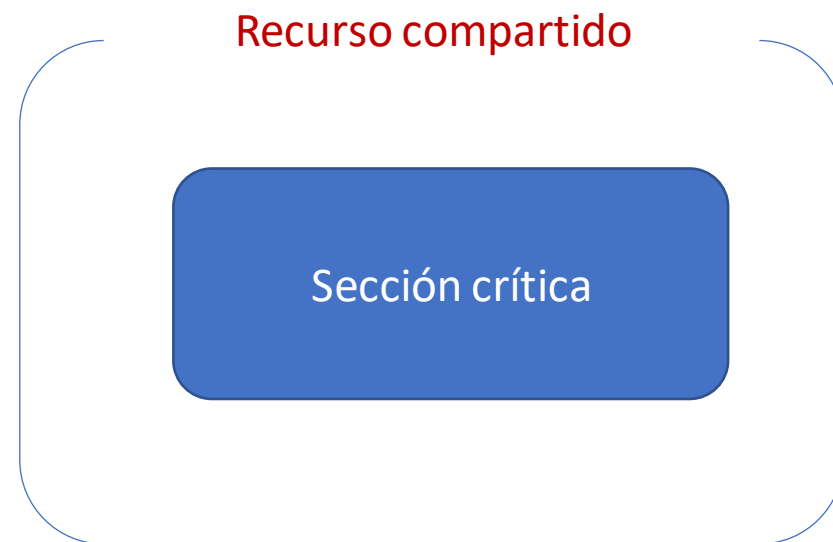
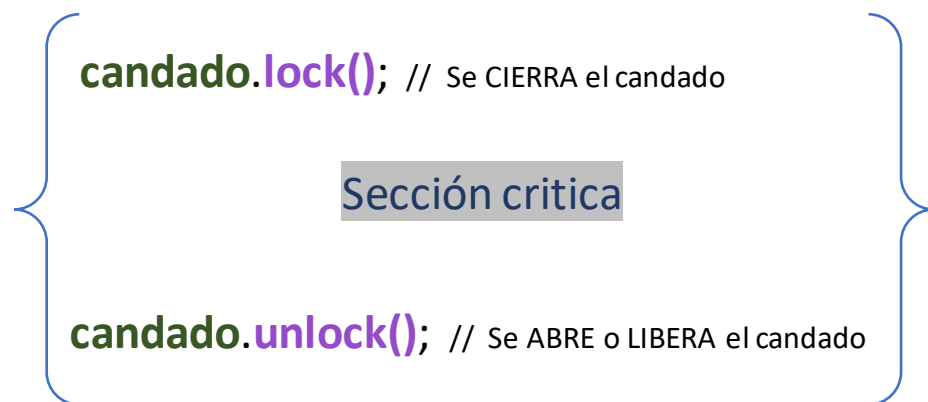
```
synchronized void método () {  
    // Sección crítica  
}
```



# Candados

Los **Candados**, son objetos que protegen a la **sección crítica** con llamadas a funciones de bloqueo(**cerrar-abrir**) desde dentro de un bloque, para lograr la sincronización y la exclusividad de acceso, es decir, es otra forma de implementar exclusión mutua.

```
java.util.concurrent.locks.ReentrantLock  
private static Lock candado = new ReentrantLock();
```





## Ejercicio 0 (Evidenciando la problemática)

En el siguiente ejercicio, se muestra un ejemplo clásico de cómo se corrompe el valor de una variable compartida cuando dos hilos no sincronizados modifican su valor.

```
public class El_CuentaNoProtegida implements El_ICuentaBancaria {  
  
    private int saldoActual = 0;  
  
    @Override  
    public void incremento() {  
        this.saldoActual++;  
    }  
  
    @Override  
    public void decremento() {  
        this.saldoActual--;  
    }  
  
    @Override  
    public int saldoActual() {  
        return this.saldoActual;  
    }  
  
    static {  
        System.out.println("#####");  
        System.out.println("# Cuenta no protegida #");  
        System.out.println("# ----- #\n");  
    }  
}
```

Sección crítica

Sección crítica

## Ejercicio 0 (Evidenciando la problemática)

```
public class El_SeccionCritica implements Runnable{

    El_ICuentaBancaria nomina;

    public El_SeccionCritica(El_ICuentaBancaria s) {
        this.nomina = s;
    }

    @Override
    public void run() {
        for (int i = 0; i < El_ICuentaBancaria.GASTOS; i++) {
            nomina.decremento();
        }
    }
}
```

Ingresa a la variable de la clase **E1\_CuentaNoProtegida** que se convierte en la sección crítica

## Ejercicio 0 (Evidenciando la problemática)

```
public class Test_El_CuentaBancaria {  
  
    public static void main(String r[]) {  
        El_ICuentaBancaria objetoCuenta = new El_CuentaNoProtegida();  
        Runnable objetoRunnable = new El_SeccionCritica(objetoCuenta);  
        Thread objetoHilo = new Thread(objetoRunnable);  
        objetoHilo.start();  
        for (int i = 0; i < El_ICuentaBancaria.INGRESOS; i++) {  
            objetoCuenta.incremento();  
        }  
        try {  
            objetoHilo.join();  
        } catch (InterruptedException e) { /* Lo que quieran aqui */ }  
  
        System.out.printf("El saldo final es: %d \t \n\n", objetoCuenta.saldoActual());  
    }  
}
```

Ingresa a la variable de la clase **E1\_CuentaNoProtegida** que se convierte en la sección crítica

## Ejercicio 0 (Evidenciando la problemática)

run:

El valor final es: 4718

BUILD SUCCESSFUL (total time: 0 seconds)

run:

El valor final es: 4264

BUILD SUCCESSFUL (total time: 0 seconds)

run:

El valor final es: 5242

BUILD SUCCESSFUL (total time: 0 seconds)

Obsérvese las inconsistencias en el resultado final

## Ejercicio 1 (Implementando mecanismos de interrupción)

```
public class Test_El_CuentaBancaria {  
  
    public static void main(String r[]) {  
        El_ICuentaBancaria objetoCuenta = new El_CuentaNoProtegida();  
        Runnable objetoRunnable = new El_SeccionCritica(objetoCuenta);  
        Thread objetoHilo = new Thread(objetoRunnable);  
        objetoHilo.start();  
        for (int i = 0; i < El_ICuentaBancaria.INGRESOS; i++) {  
            objetoCuenta.incremento();  
        }  
        try {  
            objetoHilo.join();  
        } catch (InterruptedException e) { /* Lo que quieran aqui */ }  
  
        System.out.printf("El saldo final es: %d \t \n\n", objetoCuenta.saldoActual());  
    }  
}
```

Obsérvese la posición dónde se encuentra la llamada al método **join()**, anterior a esto, el hilo principal **main()** también está accediendo a la sección crítica en el bloque **for**, por lo tanto tenemos dos **hilos no sincronizados** compitiendo por el mismo recurso.

## Ejercicio 1 (Implementando mecanismos de interrupción)

Sí cambiamos el orden, significa qué el hilo principal **main** NO puede continuar su ejecución mientras el hilo **objetoHilo** se encuentre con trabajo pendiente.

```
try {  
    objetoHilo.join();  
} catch (InterruptedException e) {  
    /* Lo que quieran aqui */  
}  
for (int i = 0; i < El_ICuentaBancaria.INGRESOS; i++) {  
    objetoCuenta.incremento();  
}
```

## Ejercicio 1 (Implementando mecanismos de interrupción)

run:

El valor final es: 5000

BUILD SUCCESSFUL (total time: 0 seconds)

run:

El valor final es: 5000

BUILD SUCCESSFUL (total time: 0 seconds)

run:

El valor final es: 5000

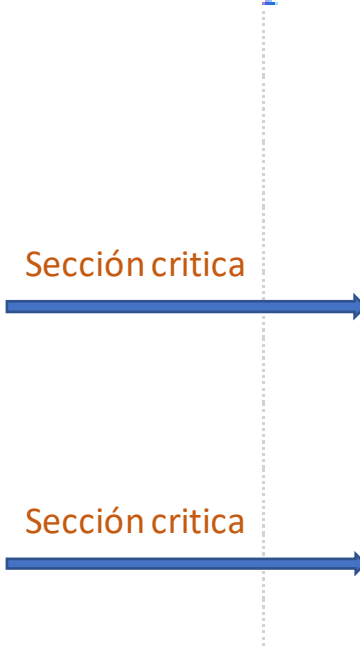
BUILD SUCCESSFUL (total time: 0 seconds)

Pero **CUIDADO**, esto no significa que estén sincronizados

## Ejercicio 2 (Implementando sincronización con monitores)

En el siguiente ejercicio, se muestra el mismo ejemplo anterior, pero en esta ocasión vamos a proteger la sección crítica empleando **monitores**.

```
public class El_CuentaProtegidaMonitor implements El_ICuentaBancaria {  
    private int saldoActual = 0;  
  
    @Override  
    public synchronized void incremento() {  
        this.saldoActual++;  
    }  
  
    @Override  
    public synchronized void decremento() {  
        this.saldoActual--;  
    }  
}
```





## Ejercicio 2 (Implementando sincronización con monitores)

```
public class Test_El_CuentaBancaria {  
  
    public static void main(String r[]) {  
  
        El_ICuentaBancaria objetoCuenta = new El_CuentaProtegidaMonitor();  
  
        Runnable objetoRunnable = new El_SeccionCritica(objetoCuenta);  
        Thread objetoHilo = new Thread(objetoRunnable);  
        objetoHilo.start();  
        for (int i = 0; i < El_ICuentaBancaria.INGRESOS; i++) {  
            objetoCuenta.incremento();  
        }  
        try {  
            objetoHilo.join();  
        } catch (InterruptedException e) { /* Lo que quieran aqui */ }  
  
        System.out.printf("El saldo final es: %d \t \n\n", objetoCuenta.saldoActual());  
    }  
}
```

## Ejercicio 2 (Implementando sincronización con monitores)

```
run:
#####
# Cuenta protegida (synchronized) #
# ----- #

El saldo final es: 5000

BUILD SUCCESSFUL (total time: 0 seconds)
```

```
run:
#####
# Cuenta protegida (synchronized) #
# ----- #

El saldo final es: 5000

BUILD SUCCESSFUL (total time: 0 seconds)
```

Ahora sí se están ejecutando **hilos sincronizados**

### Ejercicio 3 (Implementando sincronización con candados)

En el siguiente ejercicio, se muestra el mismo ejemplo anterior, pero en esta ocasión vamos a proteger la sección crítica empleando **candados**.

### Ejercicio 3 (Implementando sincronización con candados)

```
public class El_CuentaProtegidaCandado implements El_ICuentaBancaria {  
  
    private static final Lock objetoCANDADO = new ReentrantLock(); //  
    private int saldoActual = 0;  
  
    @Override  
    public void incremento() {  
        objetoCANDADO.lock(); // Se CIERRA el candado  
        try {  
            this.saldoActual++;  
        } finally {  
            objetoCANDADO.unlock(); // Se ABRE o LIBERA el candado  
        }  
    }  
  
    @Override  
    public void decremento() {  
        objetoCANDADO.lock(); // Se CIERRA el candado  
        try {  
            this.saldoActual--;  
        } finally {  
            objetoCANDADO.unlock(); // Se ABRE o LIBERA el candado  
        }  
    }  
}
```

Sección crítica

Sección crítica

### Ejercicio 3 (Implementando sincronización con candados)

```
public class Test_El_CuentaBancaria {  
  
    public static void main(String r[]) {  
  
        El_ICuentaBancaria objetoCuenta = new El_CuentaProtegidaCandado();  
  
        Runnable objetoRunnable = new El_SeccionCritica(objetoCuenta);  
        Thread objetoHilo = new Thread(objetoRunnable);  
        objetoHilo.start();  
        for (int i = 0; i < El_ICuentaBancaria.INGRESOS; i++) {  
            objetoCuenta.incremento();  
        }  
        try {  
            objetoHilo.join();  
        } catch (InterruptedException e) { /* Lo que quieran aqui */ }  
  
        System.out.printf("El saldo final es: %d \t \n\n", objetoCuenta.saldoActual());  
    }  
}
```

### Ejercicio 3 (Implementando sincronización con candados)

```
run:
#####
# Cuenta protegida (ReentrantLock) #
# ----- #

El saldo final es: 5000

BUILD SUCCESSFUL (total time: 0 seconds)
```

```
run:
#####
# Cuenta protegida (ReentrantLock) #
# ----- #

El saldo final es: 5000

BUILD SUCCESSFUL (total time: 0 seconds)
```

Nuevamente tenemos **hilos sincronizados**



**java.util.concurrent**

# Hilos en Java

java.lang

La clase **Thread**

La interface **Runnable**

JDK 1.0

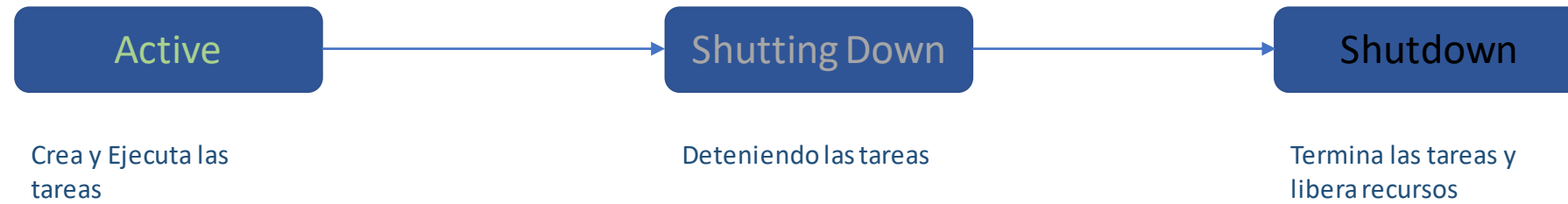
java.util.concurrent

Recursos de multihilos

JDK 5.0



# Ciclo de vida de un pool de Hilos



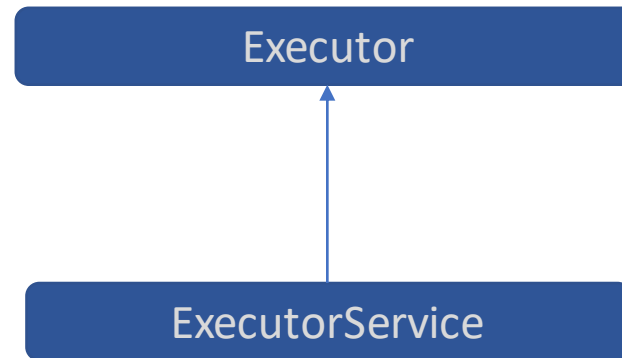
# La API `java.util.concurrent`

## `java.util.concurrent.Executor`

Para ejecutar pools de tareas

## `java.util.concurrent.ExecutorService`

Para controlar y administrar el pool de tareas de tipo `Executor`



# La clase `Executors`

`java.util.concurrent.Executors`

## Métodos de Clase

- `newCachedThreadPool(): ExecutorService`
- `newFixedThreadPool(numero ObjRunnables): ExecutorService`

} Los 2 más empleados para crear pool de hilos

# La clase `Executors`

`java.util.concurrent.Executors`

Crear un pool fijo de hilos

```
ExecutorService poolHilos = Executors.newFixedThreadPool(int numeroObjetosHilo);
```

## Activamos el pool de hilos

Por cada objeto hilo, se debe realizar esta operación, comúnmente se hace con ciclo `for()`

```
poolHilos.execute(objetoHilo);
```

# La clase `Executors`

`java.util.concurrent.Executors`

Crear un pool dinámico de hilos

```
ExecutorService poolHilos = Executors.newCachedThreadPool();
```

## Activamos el pool de hilos

Por cada objeto hilo, se debe realizar esta operación, comúnmente se hace con ciclo `for()`

```
poolHilos.execute(objetoHilo);
```

# Ejercicio

`java.util.concurrent.Executors`

A continuación, se mostrará un ejemplo de la implementación de cada método, el punto donde se debe prestar atención es en el rendimiento de cada uno de ellos.

El alumno deberá practicar con distintos escenarios, por ejemplo, considérense las pruebas siguientes:

1. Pool de hilos fijos y menor al numero de CPUs físicos que lee la JVM.
2. Pool de Hilos fijos y una carga mucho mayor de trabajo.
3. Pool de hilos dinámicos con incremento paulatino de la carga de trabajo.

## Ejercicio 4 (`java.util.concurrent.Executors`)

En el siguiente ejercicio, se muestra como crear un pool de hilos de ambas formas, con un tamaño fijo y después dinámico, iniciamos suponiendo que tenemos un objeto Runnable llamado Taquero, y debe atender órdenes de los clientes.

#### Ejercicio 4 (java.util.concurrent.Executors)

```
public class E2_Taqueros implements Runnable {

    private static int ordenesDespachadas = 0;
    private static int tiempoTotal = 0;
    private final char orden;
    private final int cantidad;
    private final int numeroOrden;

    public E2_Taqueros(char producto, int cantidad) {
        this.orden = producto;
        this.cantidad = cantidad;
        this.numeroOrden = ++ordenesDespachadas;
        tiempoTotal += cantidad;
    }

    @Override
    public void run() {
        atenderOrden();
    }
}
```



## Ejercicio 4 (java.util.concurrent.Executors)

```
public class Test_E2_Taqueros_PoolFijo {  
    public static void main(String Argumentos[]) {  
        int cantidadTaqueros;  
        // Obtenemos el numero de CPUs que lee la JVM  
        int CPUs = Runtime.getRuntime().availableProcessors();  
  
        // Crear el pool de hilos con un numero Fijo  
        ExecutorService poolTaqueros = Executors.newFixedThreadPool(CPU);  
  
        /* Vamos a simular que la CARGA de trabajo es que los  
        Taqueros preparen ordenes de algun producto  
        */  
        poolTaqueros.execute(new E2_Taqueros('T', 6)); // Orden 1: (Produc  
        poolTaqueros.execute(new E2_Taqueros('Q', 4));  
        poolTaqueros.execute(new E2_Taqueros('P', 6));  
        poolTaqueros.execute(new E2_Taqueros('S', 5));  
        poolTaqueros.execute(new E2_Taqueros('E', 5));  
        poolTaqueros.execute(new E2_Taqueros('T', 6));  
        //Obtenemos el numero de Hilos que se activaron  
        //para atender la carga de trabajo, restando el hilo main  
        cantidadTaqueros = Thread.activeCount() -1;  
  
        // Se debe detener el pool  
        poolTaqueros.shutdown();  
        while (!poolTaqueros.isTerminated()) {  
        }  
    }  
}
```

## Ejercicio 4 (java.util.concurrent.Executors) newFixedThreadPool

```
public class Test_E2_Taqueros_PoolDinamico {  
    public static void main(String Argumentos[]) {  
        int cantidadTaqueros;  
        // Crear el pool de hilos con un numero Dinamico,  
        // depende del numeo de objetos Runnables  
        ExecutorService poolTaqueros = Executors.newCachedThreadPool();  
  
        // Activamos el pool de hilos enviandoles tareas  
        for (int i = 0; i < 30; i++) {  
            poolTaqueros.execute(new E2_Taqueros('T', 10));  
        }  
        //Obtenemos el numero de Hilos que se activaron  
        //para atender la carga de trabajo, restando el hilo main  
        cantidadTaqueros = Thread.activeCount() - 1;  
  
        // Se debe detener el pool  
        poolTaqueros.shutdown();  
        while (!poolTaqueros.isTerminated()) {  
        } // Trampa de espera }  
        E2_Taqueros.imprimirResumen();  
        System.out.println("# Hilos (Taqueros trabajando): \t" + cantidadTaqueros);  
        System.out.println("\n\nMetodo empleado:\t Executors.newCachedThreadPool()");  
    }  
}
```

#### Ejercicio 4 (`java.util.concurrent.Executors`)

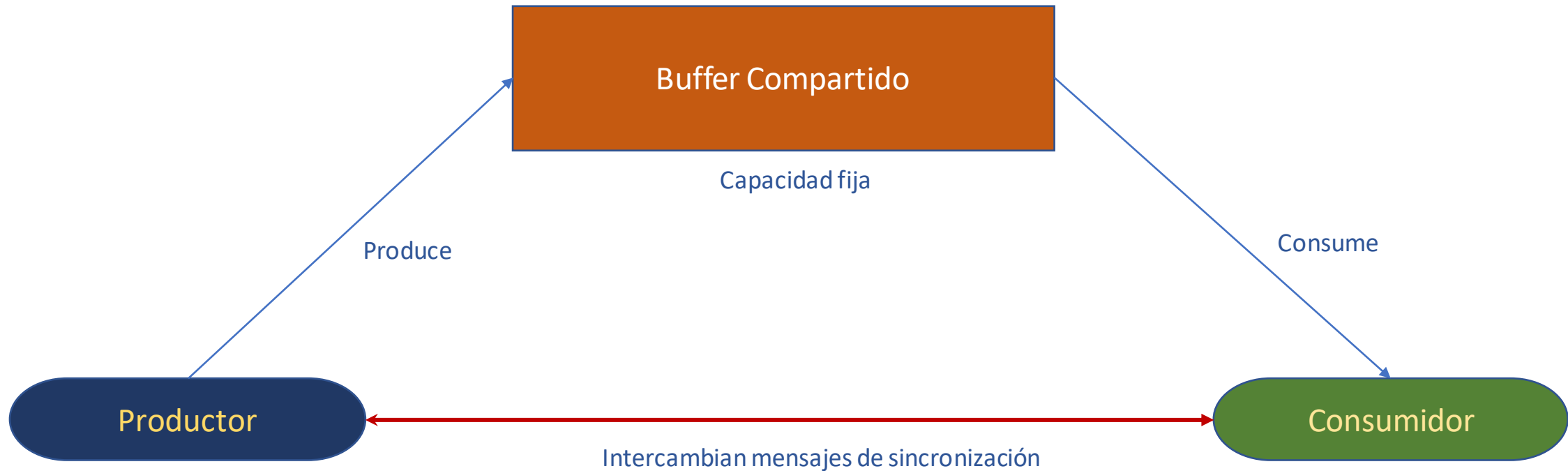
```
#####  
#                               #  
#-----#  
# Ordenes Atendidas:           360  
# Tiempo serial (1 Taquero):    3600 segundos  
# Hilos (Taqueros trabajando): 360
```

Metodo empleado: `Executors.newCachedThreadPool()`

BUILD SUCCESSFUL (total time: 10 seconds)

# Patrones de concurrencia y paralelismo

# El patron Productor Consumidor



Si el Buffer está lleno, Productor se duerme y le envía un mensaje a Consumidor.  
Si el Buffer está vacío, Consumidor se duerme y le envía un mensaje a Productor

## Ejercicio 5 (Productor-Consumidor)

# Productor

```
public class E4_Productor implements Runnable {

    E4_MesaBuffet mesaBuffet;

    public E4_Productor(E4_MesaBuffet mesaBuffet) {
        this.mesaBuffet = mesaBuffet;
    }

    @Override
    public void run() {
        int i = 1;
        //System.out.println("Cocinero cocina: " );
        while (true) {
            //System.out.println("Cocinero cocina: " + i);
            System.out.println("Cocinero produce: " + i);
            // Simulamos la produccion de un producto
            mesaBuffet.producir(i++);
            try {
                /* Lo dormimos un tiempo a leatoria en cada ciclo*/
                Thread.sleep((int) (Math.random() * 1000));
            } catch (InterruptedException ex) {
                /* La logica que cada quien quiera....*/
            }
        }
    }
}
```

## Ejercicio 5 (Productor-Consumidor)

# Consumidor

```
public class E4_Consumidor implements Runnable {

    E4_MesaBuffet mesaBuffet;

    public E4_Consumidor(E4_MesaBuffet mesaBuffet) {
        this.mesaBuffet = mesaBuffet;
    }

    @Override
    public void run() {
        try {
            while (true) {
                System.out.println("\t\t\tComelon consume: " + mesaBuffet.consumir());
                Thread.sleep((int) (Math.random() * 1000));
            }
        } catch (InterruptedException ex) {
            // La logica que cada quien quiera....
        }
    }
}
```

# MesaBuffet

```
public class E4_MesaBuffet {  
  
    // Tenemos que definir un tamaño de mesaBuffet  
    // es donde se colocaran los productos del Productor  
    private final static int TAMANYO;  
    private final LinkedList<Integer> mesaBuffet = new LinkedList<>();  
    // Creamos el candado  
    private final static Lock candado;  
    // Se deben de crear las condiciones de señalizacion entre los procesos  
    private final static Condition hayPlatillos;  
    private final static Condition noHayPlatillos;  
}
```



# MesaBuffet

```
public void producir(int cantidadPlatillos) {  
    candado.lock(); // Cerramos el candado  
    try {  
        while (mesaBuffet.size() == TAMANYO) {  
            System.out.print("# MesaBuffet (LLENA) ");  
            System.out.println("\tCOCINERO despierta a COMELON\n ");  
            noHayPlatillos.await();  
        }  
        mesaBuffet.offer(cantidadPlatillos);  
        hayPlatillos.signal(); // Señal de condicion hayPlatillos  
    } catch (InterruptedException ex) {  
    } finally {  
        candado.unlock(); // Liberamos el candado  
    }  
}
```

# MesaBuffet

```
public int consumir() {  
    int valor = 0;  
    candado.lock(); // Cerramos el candado  
    try {  
        while (mesaBuffet.isEmpty()) {  
            System.out.print("# MesaBuffet (VACIA) ");  
            System.out.println("\tCOMELON despierta a COCINERO\n ");  
            hayPlatillos.await();  
        }  
        valor = mesaBuffet.remove();  
        noHayPlatillos.signal(); // señal de condicion noHayPlatillos  
    } catch (InterruptedException ex) {  
    } finally {  
        candado.unlock(); // Liberamos el candado  
    }  
    return valor;  
}
```

## Ejercicio 5 (Productor-Consumidor)

# Test

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class Test_E4_ProductorConsumidor {

    private static E4_MesaBuffet mesaBuffet = new E4_MesaBuffet();

    public static void main(String[] args) {
        // Creamos el pool de hilos
        ExecutorService cocineros = Executors.newFixedThreadPool(2);
        ExecutorService comelones = Executors.newFixedThreadPool(5);

        cocineros.execute(new E4_Productor(mesaBuffet));

        comelones.execute(new E4_Consumidor(mesaBuffet));
        cocineros.shutdown();
        comelones.shutdown();
    }
}
```

## Ejercicio 5 (Productor-Consumidor)

run:

Cocinero produce: 1

# MesaBuffet (VACIA)

Cocinero produce: 2

Cocinero produce: 3

Cocinero produce: 4

Cocinero produce: 5

Cocinero produce: 6

Cocinero produce: 7

Cocinero produce: 8

Cocinero produce: 9

Cocinero produce: 10

Comelon consume: 1

COMELON despierta a COCINERO

Comelon consume: 2

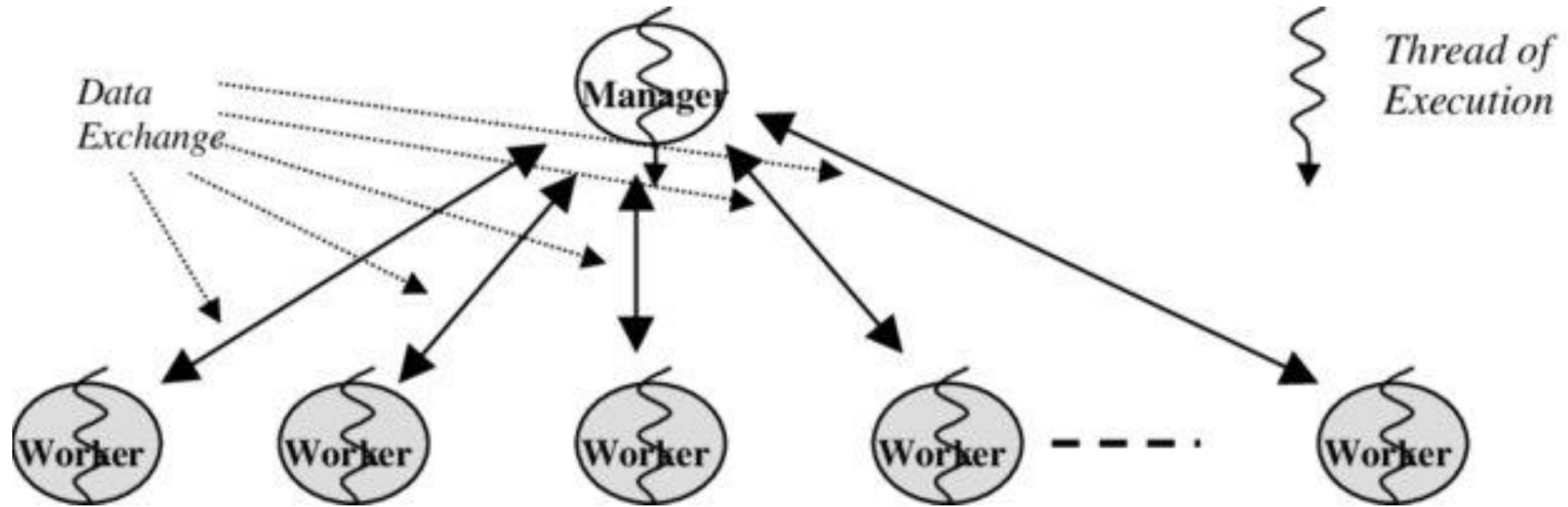
Comelon consume: 3

Comelon consume: 4

Comelon consume: 5

Comelon consume: 6

# El patron Manager - Worker



Divide y vencerás

Fuente: <https://lya.fciencias.unam.mx/jloa/patrones/MW.html>

# Manager - Worker

¿Cómo puedo saber si es factible a implementar en mi trabajo?

- Se emplea cuando los **datos** de una tarea se pueden dividir en varias partes.
- Todas las partes deben pasar por el **mismo cálculo** (**función**) para dar un resultado congruente y que sumará al resultado final.

# Manager - Worker

## Comunicación entre los procesos o hilos

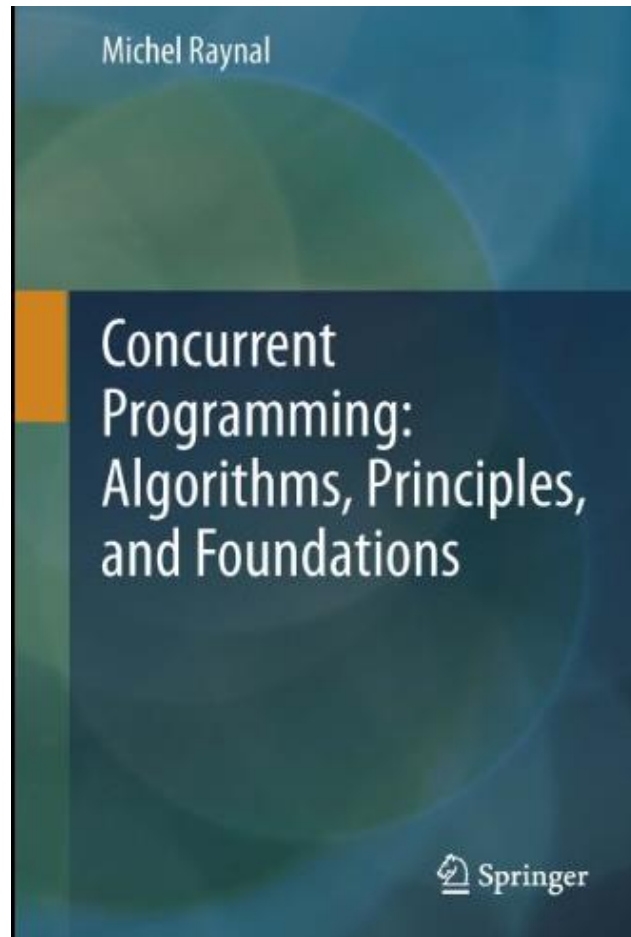
- La única comunicación es entre el manager y el worker.
- No se comunican entre los workers(trabajadores).
- El usuario final (Persona) sólo se comunica con el manager para asignar el trabajo requerido.
- El manager tiene que mantener un registro de cómo se han distribuido los datos divididos, y recabar los resultados parciales de cada uno de los trabajadores.

# Manager - Worker

¿Dónde se emplea ?



Lectura recomendada.

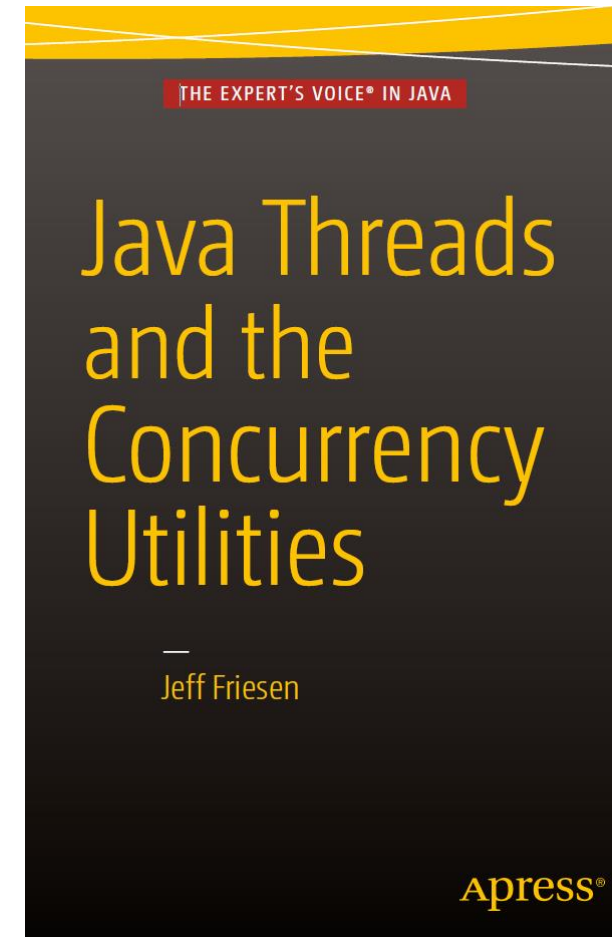


Jorge Luis Ortega-Arjona

## PATTERNS FOR PARALLEL SOFTWARE DESIGN



WILEY SERIES IN  
SOFTWARE DESIGN PATTERNS



# Documentación Oficial de Java

# Documentación oficial

Tutoriales en línea

<https://docs.oracle.com/javase/tutorial/>

Documentación del API

<https://docs.oracle.com/javase/8/docs/api/index.html>

Para documentación del lenguaje Java y de la JVM

<https://docs.oracle.com/javase/specs/index.html>