2024

UNIDAD 2. PROGRAMACIÓN MULTIHILO

2ºDAM. PROGRAMACIÓN DE SERVICIOS Y PROCESOS

FRAN HUERTAS

Contenido

INTRODUCCIÓN

Un proceso es un programa ejecutándose de forma independiente y con un espacio propio de memoria. Un sistema operativo multitarea es capaz de ejecutar más de un proceso simultáneamente. Un *thread* o hilo es un flujo secuencial simple dentro de un proceso, un único proceso puede tener varios hilos ejecutándose. Por ejemplo, el programa Firefox sería un proceso, mientras que cada una de las pestañas que se pueden tener abiertas simultáneamente trayendo páginas HTML estaría formada por al menos un hilo.

Sin el uso de *threads* hay tareas que son prácticamente imposibles de ejecutar, en particular las que tienen tiempos de espera importantes entre etapas. Los *threads* o hilos de ejecución permiten organizar los recursos del ordenador de forma que pueda haber varios programas actuando en paralelo. **Un hilo de ejecución puede realizar cualquier tarea que pueda realizar un programa normal y corriente**. Bastará con indicar lo que tiene que hacer en el método *run()*, que es el que define la actividad principal de las *threads*.

CONCEPTO DE HILO

Muchos entornos tienen la llamada multitarea en su sistema operativo, esto es distinto al multihilo. En un sistema operativo multitarea a las tareas se les llama procesos pesados, mientras que en un entorno multihilo se les denomina procesos ligeros o hilos, la diferencia es que los procesos pesados están en espacios de direccionamiento distintos y por ello la comunicación entre procesos y el cambio de contexto es caro.

Por el contrario, **los hilos comparten el mismo espacio de direcciones y comparten cooperativamente el mismo proceso pesado**, cada hilo guarda su propia pila, variables locales y contador de programa, por ello, la comunicación entre hilos es muy ligera y la conmutación de contexto muy rápida. Los *threads* tendrán que "disputarse" el tiempo de ejecución que el sistema operativo le dé al proceso en el que residen.

Programa de flujo único. Un programa de flujo único o *single-threaded* utiliza un único flujo de control para controlar su ejecución. Muchos programas no necesitan la potencia o utilidad de múltiples flujos de control, sin necesidad de especificar explícitamente que se quiere un único flujo de control.

Programa de flujo múltiple. Las aplicaciones *multithreaded* utilizan muchos contextos de ejecución para cumplir su trabajo, utilizan el hecho de que muchas tareas contienen subtareas distintas e independientes, pudiendo utilizar un *thread* para cada subtarea.

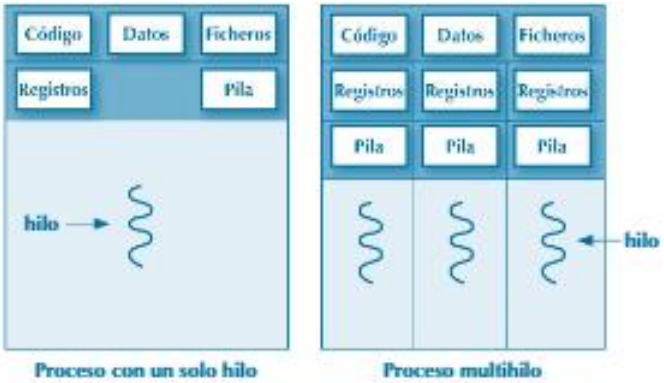


Figura 2.1
Procesos con un solo hilo y multihilo.

Algunas de las situaciones donde son de mucha utilidad los hilos son:

Programas servidores que proporcionan servicios a otros procesos, como un servidor web. Cada petición recibida se gestiona mediante un hilo dedicado.

Programas de todo tipo que muestran una interfaz gráfica de usuario a la vez que realizan procesos en segundo plano.

Juegos en los que intervienen muchos elementos que interactúan entre sí, por ejemplo, cada personaje se gestiona con un hilo diferente.

Programas de control en tiempo real. Suelen estar conectados a diferentes sensores y actuadores. La recogida de datos y las acciones a realizar en función de estos se controlan por diferentes hilos de ejecución perfectamente coordinados.

CREACIÓN DE HILOS EN JAVA

EXTENDIENDO LA CLASE THREAD

La funcionalidad básica de un *thread* esta implementada en la clase **Thread**. Cualquier clase que derive de esta puede construir un hilo de ejecución independiente, para lo cual basta con que sobrescriba el método ***run().*** En este método hay que incluir el código que deseamos que se ejecute en paralelo con otras tareas.

Para que comience la ejecución del *thread*, debe invocarse el método **start()**, que después de realizar los preparativos convenientes invoca a su vez a **run().** El siguiente ejemplo demuestra la creación de un *thread* del modo descrito:

```
public class HiloThread extends Thread {
    int i;
    public HiloThread(){
        i=0;
    }
    public HiloThread(int i){
        this.i=i;
    }
    @Override
    public void run(){
        System.out.println("Hilo "+i);
    }
    public static void main(String[] args) {

        for(int i=1;i<=10;i++){
           new HiloThread(i).start();
        }
    }
}
```

IMPLEMENTANDO LA INTERFACE RUNNABLE

Construir una clase derivada de **Thread** presenta el gran inconveniente de que para poder crear un *thread* es necesario que la clase que contiene el código que se desea ejecutar en paralelo derive de **Thread**. Esta limitación es muy importante ya que Java no dispone de herencia múltiple y por ello no podremos crear un *thread* con cualquier clase arbitraria que herede su funcionalidad de otras que sean de nuestro interés.

Para solventar este problema, puede utilizarse un segundo mecanismo para la creación de un *thread*. Este mecanismo consiste en que la clase de nuestro interés implemente el *interface* **Runnable**, y a continuación crearíamos un nuevo *thread* utilizando uno de los constructores de la clase **Thread**, que admite como argumento cualquier objeto que implemente **Runnable**.

Esta técnica no supone ninguna restricción, pues una clase puede implementar cualquier número de interfaces.

El interface **Runnable** únicamente define un método denominado **run()** y al igual que sucedía en el caso anterior, debemos colocar dentro del mismo el código que deseamos que se ejecute en paralelo con otros *threads*.

```
public class HiloRunable implements Runnable {

    int i;

    public HiloRunable() {
        i = 0;
    }
    public HiloRunable(int i) {
        this.i = i;
    }
    @Override
    public void run() {
        System.out.println("Hilo " + i);
    }
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            new Thread(new HiloRunable(i)).start();
        }
    }
}
```

Hay que dejar claro que la invocación directa del método **run()** de la clase **Thread** no produce la ejecución concurrente del hilo. La ejecución del método directo se hará el hilo principal.

USANDO UNA REFERENCIA A UN MÉTODO

Una de las formas más flexibles para iniciar la ejecución de un método (y que requiere escribir una cantidad mínima de código, a diferencia de las formas anteriores) consiste en pasar al constructor de la clase `Thread` una referencia al método que se quiere ejecutar:

```
public class HiloMetodo {

    public static void main(String[] args) {
        Thread t = new Thread(HiloMetodo::doWork);
        t.start();
    }

    public static void doWork() {
        System.out.println("Soy el thread: " + Thread.currentThread().threadId());
    }
}
```

Al emplear esta forma de creación de threads, hay que tener en cuenta lo siguiente:

- Se debe proporcionar al constructor de la clase `Thread` una referencia al método que será ejecutado por el nuevo thread. Este método debe cumplir con la interfaz funcional `Runnable`, que define un único método abstracto con la siguiente firma:

  void run()

Esto implica que, sea cual sea el método a ejecutar por un thread, su firma debe tener un tipo de retorno `void` y no puede recibir parámetros. El nombre del método no tiene por qué ser `run`, sino que puede ser cualquier identificador válido.

- Para especificar la referencia al método que debe ejecutar el thread, se utiliza la siguiente notación:

  [*<instancia>*|*<clase>*]::*<nombre_método>*

Por tanto, si queremos ejecutar el método sobre una instancia concreta, deberemos especificar el identificador de esa instancia, seguida de dos caracteres de dos puntos ('`::`') y del identificador del método.

De manera análoga, si el método a ejecutar por el *thread* es estático, deberemos especificar el identificador de la clase que lo contiene, seguida dos caracteres de dos puntos ('`::`') y del identificador del método.

Recuerda que no debes incluir nunca los paréntesis de llamada a un método cuando especifiques una referencia a un método.

ESPECIFICANDO UNA EXPRESIÓN LAMBDA

Otra forma de creación de un *thread*, equivalente al empleo de una referencia a un método, consiste en proporcionar al constructor de `Thread` una expresión lambda compatible con el método `run` definido en la interfaz `Runnable`:

```
public class HiloLambda {

    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            HiloLambda.doWork();
        });
        t.start();
    }

    public static void doWork() {
        System.out.println("Soy el thread: " + Thread.currentThread().threadId());
    }
}
```

ACTIVACIÓN DEL THREAD

Como puedes ver en los ejemplos de los apartados anteriores, una vez que el thread ha sido instanciado, es posible activarlo mediante una llamada al método `start`. Una vez invocado el método `start`, el thread se activa y pasa a una lista de procesos a la espera de recibir tiempo de CPU por parte del scheduler.

Por tanto, una llamada al método `start` no tiene por qué iniciar inmediatamente el thread, sino que depende del planificador el momento en que el thread recibe tiempo de CPU y comienza a ejecutar instrucciones.

Por otro lado, **cuando el thread finalice, no será posible volver a ejecutarlo mediante una nueva llamada a** `start`, sino que será necesario instanciar un nuevo thread. En caso de intentar reactivar un thread que ha finalizado, se emite una excepción.

LA CLASE THREAD

La clase **Thread** proporciona la funcionalidad necesaria para la creación, gestión y ejecución de hilos en Java. A continuación, vamos a ver sus diferentes miembros.

CONSTRUCTORES

Existen varios constructores de la clase **Thread** (y transferido por herencia a todas sus extensiones).

Desde el punto de vista estructural existen dos variantes básicas:

1. Las que requieren que el código del método **run()** se especifique explícitamente en la declaración de la clase. Por ejemplo: Thread(String threadName)

2. Las que requieren un parámetro de inicialización que implemente la interfaz **Runnable**. Por ejemplo: Thread(Runnable threadOb)

- **Thread()**
- **Thread(Runnable threadOb)**
- **Thread(Runnable threadOb, String threadName)**
- **Thread(String threadName)**
- **Thread(ThreadGroup groupOb, Runnable threadOb)**
- **Thread(ThreadGroup groupOb, Runnable threadOb,String threadName)**
- **Thread(ThreadGroup groupOb, String threadName)**

INICIO Y FIN

- **void run().** Contiene el código que ejecuta el *thread*.
- **void start().** Inicia la ejecución del *thread*.
- **static Thread currentThread().** Retorna el objeto Thread que ha ejecutado este método.
- **final void sleep(long milliseconds) throws InterruptedException.** Suspende la ejecución del thread por el número de milisegundos especificados.

CONTROL DE LA FINALIZACIÓN

- **final boolean isAlive().** Retorna true si el *thread* se encuentra en el estado **Alive** (en alguno de sus subestados), esto es, ya ha comenzado y aún no ha terminado.
- **final void join() throws InterruptedException.** Suspende el *thread* que invoca hasta que el thread invocado haya terminado.
- **final void join(long milliseconds) throws InterruptedException**. Suspende el *thread* que invoca hasta que el thread invocado haya terminado o hasta que hayan transcurrido los milisegundos.

INTERRUPCIÓN

- **void interrupt()**. El thread pasa a estado Interrupted. Si está en los estados Waiting, Joining o Sleeping termina y lanza la excepción InterruptedException. Si está en estado Runnable, continúa ejecutándose, aunque cambia su estado a Interrupted.
- **static boolean interrupted().** Procedimiento estático. Retorna true si el thread que lo invoca se encuentra en estado Interrupted. Si estuviese en el estado Interrupted se pasa al estado Runnable.
- **boolean isInterrupted()**. Retorna true si el objeto thread en que se invoca se encontrase en el estado Interrupted. La ejecución de este método no cambia el estado del thread.

PRIORIDADES

CONSTANTES

- **MAX_PRIORITY** //Máxima prioridad asignable al thread.
- **MIN_PRIORITY** //Mínima prioridad asignable al thread.
- **NORM_PRIORITY** //Prioridad por defecto que se asigna.

- **final void setPriority(int priority)** Establece la prioridad de Thread.
- **final int getPriority()** Retorna la prioridad que tiene asignada el *thread*.
- **static void yield()** Se invoca el planificador para que se ejecute el *thread* en espera que corresponda. Ofrece al planificador la CPU para que se la asigne al hilo invocado. No se garantiza la cesión.

DEBUGGING Y CONTROL

- **String toString()** Retorna un *string* con el nombre, prioridad y nombre del grupo del *Thread*.
- **final String getName()** Retorna el nombre del *Thread*
- **final void setName(String threadName)** Establece el nombre del *Thread*.
- **static int activeCount()** Retorna el número de *threads* activos en el grupo al que pertenece el *thread* que invoca.
- **final ThreadGroup getThreadGroup()** Retorna el grupo al que pertenece el *thread* que invoca.
- **public final long threadId()**Devuelve el identificador de este hilo.
- **public Thread.State getState()** Obtiene uno de los posibles estados en que se encuentra el thread actualmente. Todos los estados posibles están recogidos en la enumeración **State**.

DAEMON

Existen dos tipos de thread "*user*" y "*daemon* ". Su diferencia radica en su efecto sobre la finalización de la aplicación a la que pertenecen. Una aplicación termina solo si han terminado todos los *threads* de tipo *user*. Cuando todos los *threads* de tipos *user* han terminado los *thread* de tipo Daemon son terminados con independencia de cuál sea su estado y la aplicación es finalizada.

Métodos relativos al tipo de *Thread*.

- **final void setDaemon(boolean state)** Convierte un *thread* en *daemon*. Solo es aplicable después de haber creado el *thread* y antes de que se haya arrancado (start).
- **final boolean isDaemon()** Retorna true si el *thread* es de tipo *daemon*.

Los *thread* que han sido cualificado como *daemon* terminan de una forma diferente. Cuando la VM detecta que sólo permanecen en ejecución *thread* Daemon termina su ejecución. Los *daemon thread* son utilizados para ejecutar tareas auxiliares de otros *thread* normales, cuando estos han terminado aquellos no tienen función y finalizan.

Todos los *theads* creados están configurados, por defecto, en primer plano, excepto los *threads* del pool de *threads* cuya configuración por defecto es en segundo plano. El programador puede modificar este comportamiento del *thread* a través del método `setDaemon` de la clase `Thread`.

Ten en cuenta que el modo en que se establece el *thread* no está relacionado con la prioridad o el tiempo de ejecución que el planificador asigna al *thread*, sino que únicamente afecta al comportamiento del programa cuando el *thread* finaliza.

Ejemplo de uso de join y sleep.

```
import java.util.Random;
/**
 *
 * @author Fran
 */
class Hilo implements Runnable {

    private final String nombre;

    Hilo(String nombre) {
        this.nombre = nombre;
    }

    public void run() {
        System.out.printf("Soy el hilo %s\n", nombre);
        Random r = new Random();
        for (int i = 0; i < 5; i++) {
            int pausa = 10 + r.nextInt(500 - 10);
            System.out.printf("Hilo %s pausado %d ms\n", nombre, pausa);
            try {
                Thread.sleep(pausa);

            } catch (InterruptedException e) {
                System.out.printf("Hilo %s interrumpido\n", nombre);
            }
        }
        System.out.printf("Hilo %s terminado\n", nombre);
    }
}

public class HilosJoin {

    public static void main(String[] args) {
        Thread h1 = new Thread(new Hilo("H1"));
        Thread h2 = new Thread(new Hilo("H2"));
        h1.start();
        h2.start();
        try {
            h1.join();
            h2.join();
        } catch (InterruptedException e) {
            System.out.println("Hilo principal interrumpido");
        }
        System.out.println("Hilo principal finalizado");
    }
}
```

Ejemplo de uso de programa secuencial vs programa concurrente.

```
import java.math.BigInteger;

/**
 *
 * @author fran
 */
class Calculador extends Thread {

    long desde, hasta;
    Resultado r;

    public Calculador(long x, long y, Resultado r) {
        this.desde = x;
        this.hasta = y;
        this.r = r;
    }

    @Override
    public void run() {
        for (long i = desde; i <= hasta; i++) {
            if (esPrimo(i)) {
                r.sumar(i);
            }
        }
    }

    private boolean esPrimo(long n) {
        int raiz = (int) Math.sqrt((double) n);
        for (int i = 2; i <= raiz; i++) {
            if (n % i == 0) {
             return false;
            }
        }
        return true;
    }
}
public class Resultado {
    private BigInteger suma = new BigInteger("0");
    public BigInteger getSuma() {
        return suma;
    }
    public synchronized void  sumar(long n) {
        BigInteger bn = new BigInteger(String.valueOf(n));
        suma = suma.add(bn);
    }
}
```

```
public class SumaPrimosConcurrente {

    public static void main(String[] x) {
        Resultado suma = new Resultado();
        long t0 = System.currentTimeMillis();
        Calculador p1 = new Calculador(1, 2000000, suma);
        Calculador p2 = new Calculador(2000001, 4000000, suma);
        Calculador p3 = new Calculador(4000001, 6000000, suma);
        Calculador p4 = new Calculador(6000001, 8000000, suma);
        Calculador p5 = new Calculador(8000001, 10000000, suma);
        p1.start();p2.start();p3.start();p4.start();p5.start();
        try {
            p1.join();
            p2.join();
            p3.join();
            p4.join();
            p5.join();
        } catch (InterruptedException e) {}
        long t1 = System.currentTimeMillis();
        System.out.println("La suma de los números primos hasta 10000000
es: " + suma.getSuma() + " calculado en " + (t1 - t0) + " miliseg.");
    }
}
```

```
/**
 *
 * @author fran
 */
public class SumaPrimosSecuencial {

    public static void main(String[] x) {
        Resultado suma = new Resultado();
        long t0 = System.currentTimeMillis();
        Calculador p1 = new Calculador(1, 10000000, suma);
        p1.start();
        try {
            p1.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        long t1 = System.currentTimeMillis();
        System.out.println("La suma de los números primos hasta 100000000
es: " + suma.getSuma() + " calculado en " + (t1 - t0) + " miliseg.");
    }
}
```

GESTIÓN DE LA MEMORIA EN HILOS

Existen dos áreas de memoria en las que se almacenan las variables del proceso[1]:

- El ***heap*** (memoria montón) es un área de memoria **del proceso**, que almacena las **variables de vida larga** del programa, es decir, los **atributos** de una clase, que **no están definidos de manera local a un método**. Estas variables no se destruyen automáticamente al quedar fuera de alcance, como ocurre con las variables locales a los métodos, sino que se mantienen en memoria hasta que el recolector de basura libera esas posiciones de memoria. **Todos los threads del programa tienen acceso al heap del proceso**, por lo que **todos pueden leer y escribir estas variables**. Las modificaciones que realice un thread sobre una de estas variables pueden ser observada por cualquier otro thread.

- El ***stack*** (pila)es un área de memoria presente en cada uno de los threads del programa. Esta área de memoria es una estructura de tipo pila que almacena todas las **variables de vida corta** del programa, es decir, **almacena todas las variables creadas localmente, en el cuerpo de los métodos**. Cuando un thread ejecuta un método, todas las variables locales del método se almacenan en su stack de forma privada. Si dos threads ejecutan el mismo método, cada thread tendrá en su stack una copia diferente e independiente de las variables del método.

Esto quiere decir que distintos *threads* pueden ejecutarse de forma segura sobre un mismo método, puesto que cada *thread* tiene su propia pila con sus variables privadas. Sin embargo, los *threads* pueden comunicarse entre sí (e interferir) leyendo o escribiendo en los miembros de clase (campos y atributos).

En el siguiente ejemplo se muestra esta característica:

- La variable `Contador` está definida a nivel de clase (es un campo) y, por tanto, está almacenada en el **heap**. En esa ubicación todos los threads de la aplicación tienen acceso a esa variable y pueden interferir unos con otros si modifican su valor.

- La variable `i` está definida de forma local al método Test, por lo que se almacena en el **stack** de cada *thread* que ejecuta el método. Cada *thread* tendrá una variable `i` privada, que puede tener un valor distinto en cada *thread*, en función del camino de ejecución que haya recorrido cada *thread*.

```
public class TestHilo {
    // Contador indica el número total de veces que se ha ejecutado el método Test.

    static int Contador = 0;

    public static void main(String[] args) {
        Thread t = new Thread(TestHilo::test);
        t.start();
```

1 Los tipos de datos manejados por valor y las referencias se almacenan en el heap o el stack en función de su tiempo de vida esperado. No obstante, las instancias en sí, se almacenan en el heap y deben ser liberadas por el recolector de basura.

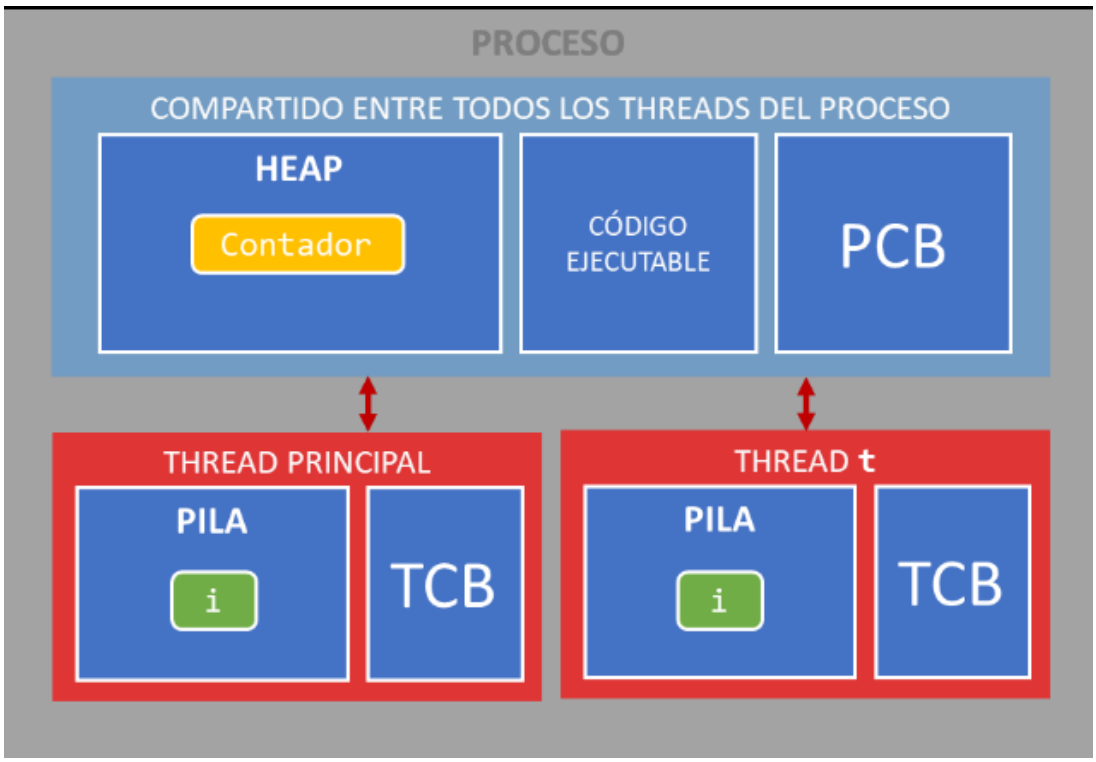
```
        test();
    }

    static void test() {
// Como la variable Contador NO es local a este método, sino que está
// definida a nivel de clase (es un campo), todos los threads "ven" ese
// campo (no tienen una copia privada)
        Contador++;
        for (int i = 0; i < 5; i++) {
            System.out.println("Escribiendo desde el thread con id "
                    + Thread.currentThread().threadId() + "\n"
                    + "En este thread i vale: " + i
                    + " y Contador vale: " + Contador + "\n");
        }
    }
}
```

Hay que tener en cuenta:

- Cada thread mantiene en su **pila** (stack) una **copia privada (aislada)** de las **variables locales declaradas en los métodos que ejecuta** en su camino de ejecución. Al mantener copias aisladas de esas variables locales, varios *threads* pueden ejecutar un mismo método sin interferir entre sí.

- El **heap** del proceso almacena todas las variables definidas a nivel clase (atributos), por lo que todos los *threads* comparten el acceso a esas variables. Esto puede resultar de utilidad como mecanismo de comunicación entre *threads*, ya que los distintos *threads* pueden intercambiar información a través de los atributos.

La organización en memoria de las variables quedaría así:



Observa que, tanto el **thread principal** como el **thread** `t` tienen, cada uno, su propia variable `i`, con valores independientes en cada thread. Esto es debido a que cada thread tiene su propia pila de memoria privada donde se crean las variables locales a los métodos. Cada thread posee su propio conjunto de variables locales al método, de modo que múltiples threads pueden ejecutar simultáneamente el método sin ocasionarse interferencias mutuamente.

Esta es una de las muchas posibles salidas que puede mostrar el programa tras su ejecución:

```
Escribiendo desde el thread con id 1
En este thread i vale: 0 y Contador vale: 1

Escribiendo desde el thread con id 1
En este thread i vale: 1 y Contador vale: 2

Escribiendo desde el thread con id 1
En este thread i vale: 2 y Contador vale: 2

Escribiendo desde el thread con id 1
En este thread i vale: 3 y Contador vale: 2

Escribiendo desde el thread con id 20
En este thread i vale: 0 y Contador vale: 2

Escribiendo desde el thread con id 1
En este thread i vale: 4 y Contador vale: 2

Escribiendo desde el thread con id 20
En este thread i vale: 1 y Contador vale: 2

Escribiendo desde el thread con id 20
En este thread i vale: 2 y Contador vale: 2

Escribiendo desde el thread con id 20
En este thread i vale: 3 y Contador vale: 2

Escribiendo desde el thread con id 20
En este thread i vale: 4 y Contador vale: 2
```

Al ejecutar el programa, comprobarás que el valor de la variable `i` es consistente en cada thread (se incrementa secuencialmente dentro de un mismo thread), ya que esta variable **es local al método** `Test` y, por tanto, **se almacena en la pila de cada uno de los threads**. Cada thread tiene su propia variable `i`, por lo que la ejecución del método por parte de un thread no interfiere con la ejecución del mismo método por parte de otro thread.

Sin embargo, si observas el valor que toma la variable `Contador`, comprobarás que, en muchas ocasiones, es errático. Esto es debido a que la variable `Contador` **no es local al método**, sino que es un campo, por lo que **existe una única variable `Contador`, y está almacenada en el heap, donde ambos threads pueden observarla y modificarla**. Esto ocasiona un comportamiento anómalo del programa. Abordaremos la forma de controlar este comportamiento al tratar la sincronización de threads.

CICLO DE VIDA DE UN HILO

Un *thread* atraviesa una serie de estados desde su instanciación hasta su finalización. El método `getState` de la clase `Thread` permite consultar en qué estado se encuentra un thread en un instante determinado:

NEW	• El *thread* inicia su ciclo de vida al ser instanciado en el código del programa. En este punto, se encuentra en estado **`NEW`** y el scheduler aún no conoce su existencia.
RUNNABLE	• Al invocar al método **`start`**, el *thread* cambia su estado a **`RUNNABLE`** y pasa a una cola de threads listos para iniciar la ejecución. A cada uno de estos *threads* se le asignará un tiempo de procesador, de acuerdo con el algoritmo de planificación utilizado por el scheduler. • Cuando un *thread* se encuentra en estado **`RUNNABLE`** y el *scheduler* le asigna tiempo de procesador, el *thread* abandona la cola y comienza la ejecución de instrucciones. • Si se invoca el método Thread.yield() el hilo seguirá en estado de **RUNNABLE**
TIMED/ WAITING	• Si, durante la ejecución del thread, se invoca a los métodos **`wait`**, **`sleep`** o **`join`**, el thread pasa a estado **`TIMED/WAITING`** y se incorpora a una cola de threads bloqueados.
BLOCKED	• Si un thread trata de acceder a una sección crítica, establecida con la instrucción `lock` o con algún monitor, sin haber podido adquirir el testigo de sincronización, el thread pasa a estado **`BLOCKED`** y se mantiene en él hasta lograr adquirir el testigo de sincronización.
TERMINATED	• Cuando un thread alcanza la última instrucción de su camino de ejecución, finaliza la ejecución y pasa a estado **`TERMINATED`**. Una vez que el thread ha alcanzado este estado, no podrá volver a ejecutarse y no será posible retornar al estado **`RUNNABLE`** mediante una llamada al método `start`.

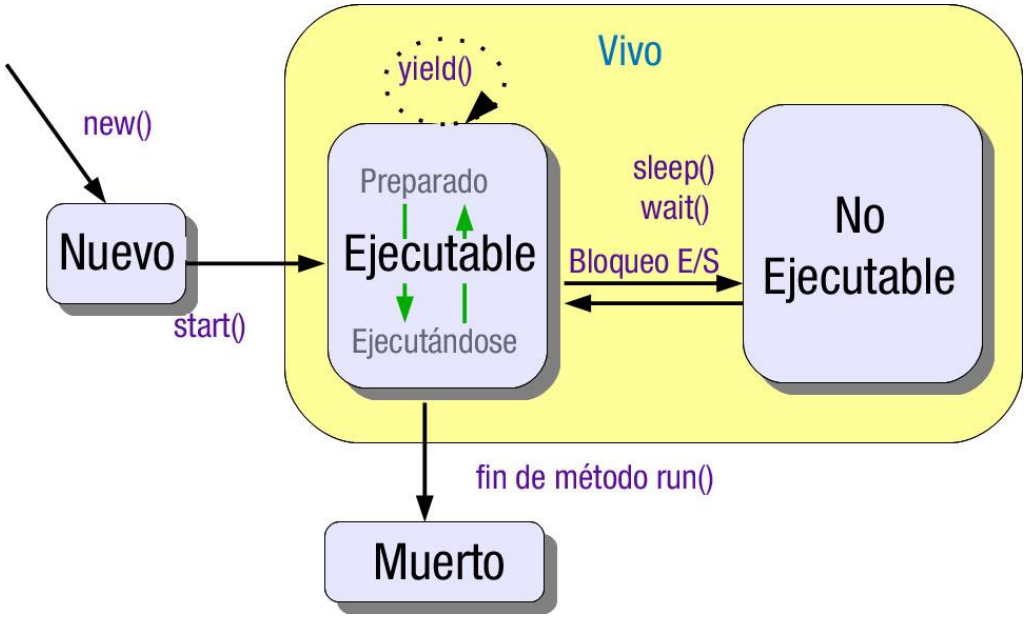
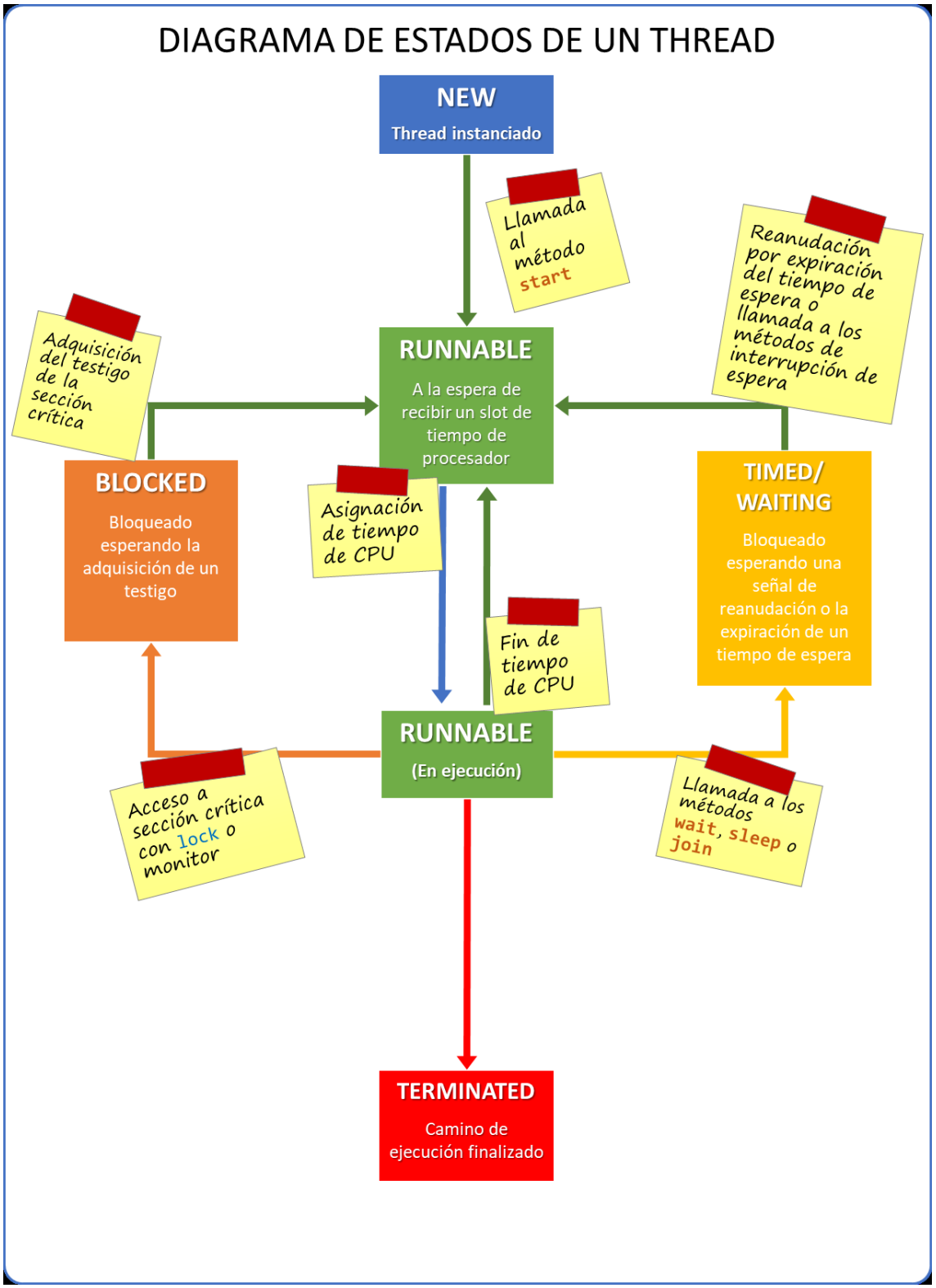


DIAGRAMA DE CICLO DE VIDA SIMPLIFICADO



SINCRONIZACIÓN DE HILOS

Ya hemos visto en el ejemplo de la suma de número primos como en la versión concurrente se producían anomalías cuando usábamos un el método de sumar sin la cláusula **synchronized**,

esto se debe a un problema llamado **condición de carrera**. En este apartado vamos a ver los diferentes problemas que se pueden dar cuando desarrollamos programas concurrentes que comparten datos o colaboran entre sí para obtener un resultado.

CONDICIÓN DE CARRERA. SECCIÓN CRÍTICA Y EXCLUSIÓN MUTUA.

En un proceso secuencial tenemos un comportamiento reproducible, asegurando que partiendo de un estado inicial y unos datos de entrada se van a obtener los mismos resultados.

En programación concurrente la anterior afirmación no siempre se cumple, si los procesos comparten recursos.

Ej. **P1 : x=x+2** **P2: x=x*2**

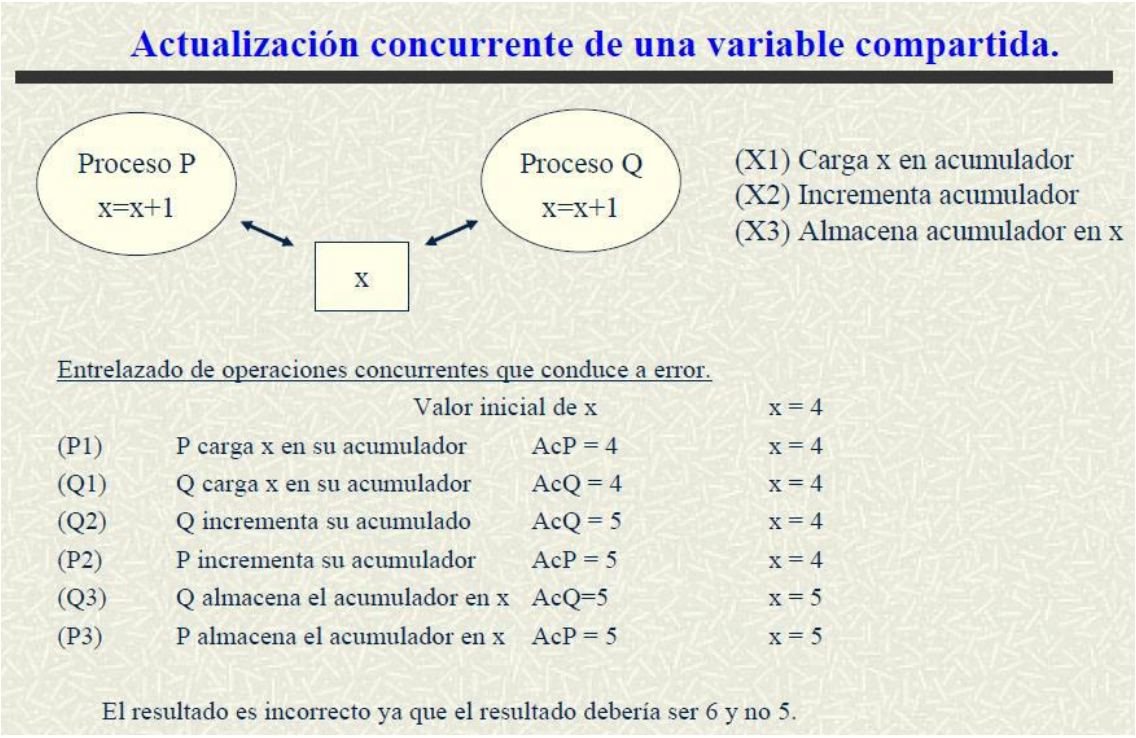
Si x tiene un valor inicial de 3, el resultado podría ser 10 u 8 en función del orden de ejecución de los procesos P1 y P2.

A esta anomalía, se le conoce como **condición de carrera**, donde la velocidad con que se ejecute un proceso u otro va a determinar el resultado, es decir, los procesos se ejecutan de forma asíncrona.

Se conoce como **sección crítica** el segmento de código que accede a un recurso compartido y que ha de ejecutarse de forma ininterrumpida para evitar errores. Ej. Dos procesos no pueden escribir en la impresora a la vez.

Se dice que una **operación** es **atómica** si se ejecuta desde su inicio hasta su fin sin interrupción.

En la siguiente imagen vemos qué puede suceder cuando una variable se actualiza por dos procesos diferentes.



Problema de las actualizaciones concurrentes: consideremos el caso de dos procesos P y Q que ejecutan de forma independiente, pero concurrentemente la sentencia x := x +1;

- Esta sentencia no es atómica. Para su realización, la CPU debe llevar a cabo la siguiente secuencia de instrucciones básicas: (1) Carga el valor de x en un acumulador de la CPU, (2) Incrementa el valor del acumulador y (3) Almacena el valor del acumulador en la variable x.
- Cuando la sentencia correspondiente a los procesos P y Q se ejecutan concurrentemente, las instrucciones básicas pueden resultar entrelazadas en cualquier combinación.
- Es fácil en este caso encontrar secuencias validas de entrelazado que no conducen a un incremento en 2 el valor de la variable x, como debe ser en un funcionamiento correcto.

Inicialmente: x = 4

1. (P1) P carga el valor de x en su acumulador PAC. PAC = 4
2. (Q1) Q carga el valor de x en su acumulador QAC. QAB= 4
3. (Q2) Q Incrementa el valor de su acumulador QAC. QAB = 5
4. (P2) P Incrementa el valor de su acumulador PAC. PAB = 5
5. (Q3) Q almacena el valor de su acumulador QAC en x. x = 5
6. (P3) P almacena el valor de su acumulador QAC en x. x = 5

Resultado: x = 5

El valor que resulta para la variable x es 5, y no 6 como debería ser si la funcionalidad hubiese sido correcta.

La solución pasa por ejecutar la sección crítica en exclusión mutua (MUTEX).

En el ejemplo de la suma de los números primos la sección crítica corresponde a la actualización del campo suma del objeto Resultado.

```
    public void  sumar(long n) {
        BigInteger bn = new BigInteger(String.valueOf(n));
        suma = suma.add(bn);
    }
```

Para garantizar la exclusión mutua en Java podemos usar la cláusula **synchronized** sobre el método anterior. Quedando de la siguiente forma:

```
    public synchronized void  sumar(long n) {
        BigInteger bn = new BigInteger(String.valueOf(n));
        suma = suma.add(bn);
    }
}
```

En Java cada objeto derivado de la clase **Object** (esto es, prácticamente todos) tienen asociado un elemento de sincronización o **lock** intrínseco, que afecta a la ejecución de los métodos definidos como **synchronized** en el objeto:

- Cuando un **thread** ejecuta un método **synchronized**, toma el **lock**, y cuando termina de ejecutarlo lo libera.
- Cuando un **thread** tiene tomado el **lock**, ningún otro **thread** puede ejecutar ningún otro método **synchronized** del mismo objeto.
- El **thread** que ejecuta un método **synchronized** de un objeto cuyo **lock** se encuentra tomado, se suspende hasta que el objeto es liberado y se le concede el acceso.

Cada clase Java derivada de **Object**, tiene también un mecanismo **lock** asociado a ella (que es independiente del asociado a los objetos de esa clase) y que afecta a los métodos estáticos declarados **synchronized**.

Consideraciones sobre el **lock** de un objeto

- El **lock** es tomado por el **thread**, por lo que mientras un **thread** tiene tomado el **lock** de un objeto puede acceder a otro método **synchronized** del mismo objeto.
- Hay un **lock** por cada instancia de la clase.
- Los métodos de clase (**static**) también pueden ser **synchronized**. Por cada clase hay un **lock** y es relativo a todos los métodos **synchronized** de la clase. Este **lock** no afecta a los accesos a los métodos **synchronized** de los objetos que son instancia de la clase.
- Cuando una clase se extiende y un método se sobrescribe, este se puede definir como **synchronized** o no, con independencia de cómo era y como sigue siendo el método de la clase madre.

Ejemplos.

Bloque sincronizado para el objeto contador.

```
public void run(){
      synchronized (contador) {
            for (int j=0; j<300; j++){
            contador.decrementa();
      }
      System.out.println(getName() + “contador = “+
      contador.getValor());

}
```

Cada vez que un hilo intenta acceder a un bloque sincronizado **le pregunta al objeto** sincronizado si hay otro hilo que lo esté usando. Si está ocupado entonces el hilo actual se suspende y espera hasta que esté libre el objeto, si está libre lo bloquea para que ningún otro hilo pueda acceder. Una vez finalizada la ejecución del bloque el objeto queda liberado para que lo pueda usar otro hilo.

Método de objeto sincronizado.

```
synchronized void RetirarDinero(int cant, String nom){
}
```

Cuando un hilo invoca un método sincronizado, trata de tomar al objeto al que pertenece, si está libre, lo toma, lo bloquea y ejecuta. Si está bloqueado por otro hilo, se suspende hasta que el objeto esté liberado. Además, cuando un hilo está ejecutando un método sincronizado de un objeto, impide que otro hilo ejecute otro método sincronizado de **ese objeto**.

Resumen del uso de synchronized en Java.

Uso de synchronized	Explicación

`public synchronized void metodo()`
`{`
`// codigo del metodo`
`}`
	Bloquea el acceso a todos los métodos sincronizados de ese objeto, para otros hilos que intenten acceder al método. Por tanto, no se podrá acceder al código del método desde el mismo objeto, pero sí desde otro objeto de la clase. Ya que cada objeto tendrá su propia instancia del método.

`public void metodo() {`
`synchronized(this) {`
`// codigo del metodo`
`}`
`}`
	Igual que el caso anterior pero aplicado al bloque synchronized desde el objeto actual o el que se especifique.

`public static synchronized void metodo() {`
`// codigo del metodo`
`}`
	Bloquea el acceso a métodos estáticos sincronizados y a bloques sincronizados con el objeto Clase.class

`public static void metodo() {`
`synchronized(MiClase.class) {`
`// codigo del metodo }`
`}`
	Igual que el anterior.

INTERBLOQUEO DE HILOS

Un interbloqueo o **deadlock** en programación concurrente es una situación en la que dos o más procesos o hilos quedan bloqueados permanentemente, cada uno esperando que el otro libere un recurso que tiene asignado, de modo que ninguno puede continuar su ejecución.

Las principales características de un interbloqueo son:

1. **Exclusión mutua:** Al menos un recurso debe estar en modo no compartido, es decir, solo un proceso a la vez puede usar el recurso.
2. **Retención y espera:** Un proceso retiene al menos un recurso mientras espera adquirir recursos adicionales que están siendo retenidos por otros procesos.
3. **No apropiación:** Los recursos no pueden ser arrebatados forzosamente de un proceso, sino que deben ser liberados voluntariamente por el proceso que los posee.
4. **Espera circular:** Existe una cadena circular de dos o más procesos, cada uno esperando un recurso que está siendo retenido por el siguiente proceso en la cadena.

El interbloqueo impide que el programa concurrente pueda continuar su ejecución, quedando en un estado de bloqueo permanente del que no puede salir por sí mismo.

Un ejemplo típico estaría representado por el siguiente código:

```
Synchronized (r1){
```

```
        Sinchronized(r2){
                (operaciones con r1 y r2)
        }
}
```

Los recursos se obtienen en el orden que aparecen los bloques synchronized. Cuando se obtiene el **lock** de r1 se queda a la espera de obtener el de r2.

Puede producirse un **deadlock** cuando dos hilos quedan mutuamente bloqueados, cada uno esperando a que el otro desbloquee el recurso que tiene bloqueado.

Una solución pasa por establecer un orden entre los recursos para que sean solicitados por los hilos siguiendo ese orden.

Un ejemplo típico de esta situación se da la realizar una transferencia entre cuentas, como podemos ver en el siguiente método.

```
public static boolean transferencia(Cuenta c1,Cuenta c2, float cantidad){
        boolean result=false;
        synchronized(c1){
            synchronized(c2){
                if (c1.getSaldo()>=cantidad){
                    c1.sacar(c2);
                    c2.ingresar(cantidad);
                    result=true;
                }
            }
        }
        return result;
    }
```

Si un hilo pretendiese realizar una transferencia de la cuenta c1 a la c2 y otro hilo lo intentase en sentido contrario (de c2 a c1) al mismo tiempo se produciría un interbloqueo. Algo que en la realidad es poco probable pero el ejemplo ilustra muy bien el problema del interbloqueo.

La forma de resolver esta situación sería estableciendo un orden en las cuentas.

Por ejemplo.

```
 public static boolean transferencia(Cuenta c1,Cuenta c2, float canti-
dad){
        Cuenta cMayor, cMenor;
        cMayor=c1; cMenor=c2;
        if (c1.getNumCuenta().compareTo(c2.getNumCuenta())<0){
            cMayor=c1;
            cMenor=c2;
        }else{
            cMayor=c2;
            cMenor=c1;
        }
```

```
        boolean result=false;
         synchronized(cMayor){
             synchronized(cMenor){
                 if (c1.getSaldo()>=cantidad){
                     c1.sacar(cantidad);
                     c2.ingresar(cantidad);
                     result=true;
                 }
             }
         }
         return result;
    }
```

BOQUEO POR ESTADO DEL RECURSO COMPARTIDO.

A veces necesitamos sincronizar a los hilos en función del estado de los recursos compartidos.

Un caso típico de esta situación es el del **productor/consumidor**. En esta situación hay unos hilos que producen algún tipo de información y otros que los consumen. Para poder consumir debe haber información en el recurso compartido y para poder producir los productores deben tener acceso al recurso y éste tener espacio para depositar la nueva producción.

En estos casos además de un mecanismo de bloqueo para acceder al recurso compartido será necesario otro mecanismo de comprobación del estado del recurso y por tanto de espera (activa) en caso de que éste no esté disponible.

Para este tipo de problemas, se utilizan varios métodos que solo se pueden ejecutar dentro de un bloque **synchronized** y sobre el objeto de bloqueo para el que se ha obtenido el **lock**.

- **wait()** interrumpe la ejecución del hilo actual y lo pone en una cola de espera. Solo se reanudará cuando otro hilo ejecute el método **notify() o notifyAll()** sobre el objeto bloqueado.
- **notify()**. Desbloquea uno de los hilos (no se sabe cuál) que están esperando sobre el objeto tras haber ejecutado el método **wait().**
- **notifyAll().** Desbloquea todos los hilos que están esperando sobre el objeto de bloqueo después de haber ejecutado **wait()** para que puedan continuar su ejecución.

El esquema típico sería algo así:

```
synchronized (objetoBloqueo) {
      while (!condicionContinuar){
            try[
            objetoBloqueo.wait();
            }catch (InterruptedException e){}
      }
      realizar_operacion;
      if (condicionContinuarOtros)objetoBloqueo.notifyAll();
```

```
        /*Si la operación realizada cambia el estado del recurso
        para que otro hilo pueda continuar se debería invocar a
        notify() o notifyAll() del objetoBloqueo*/

        }

}
```

Veamos el ejemplo del productor/consumidor

```
        /**
 *
 * @author Fran
 */
public class ProductorConsumidor {
    public static void main(String[] args) {
        Contenedor<Integer> cont = new Contenedor<Integer>();
        Thread p = new Productor(cont,"Productor");
        Thread c = new Consumidor(cont,"Consumidor");
        p.start();
        c.start();
    }
}

class Productor extends Thread{
    final Contenedor<Integer> c;
    String nombre;
    Productor(Contenedor<Integer> c, String nombre){
        this.c=c;
        this.nombre=nombre;
    }
    public void run(){
        for (int i = 1; ; i++) {
            synchronized(this.c){
                while(this.c.datoDisponible()){
                    try {
                        this.c.wait();
                    } catch (InterruptedException ex) {
                         }
                }
                int dato = new Random().nextInt(1,100);
                try {
                    sleep(500);
                } catch (InterruptedException ex) {
                }
                System.out.printf("Hilo %s produce valor %d\n",this.nombre,dato);
                this.c.put(dato);
                this.c.notify();
            }

        }
```

```
    }
}

class Consumidor extends Thread{
    final Contenedor<Integer> c;
    String nombre;
    Consumidor(Contenedor<Integer> c, String nombre){
        this.c=c;
        this.nombre=nombre;
    }
    public void run(){
        while(true){
            synchronized(this.c){
                while(!this.c.datoDisponible()){
                    try {
                        this.c.wait();
                    } catch (InterruptedException ex) {
                         }
                }
                try {
                    sleep(200);
                } catch (InterruptedException ex) {
                    Logger.getLogger(Consumidor.class.getName()).log(Level.SEVERE,
null, ex);
                }
                Integer dato = this.c.get();
                this.c.notify();
                System.out.printf("Hilo %s consume valor %d.\n",this.nombre,dato);
            }
        }
    }
}

class Contenedor<T> {
    private T dato;
    synchronized public T get(){
        T result = this.dato;
        this.dato = null;
        return result;
    }
    synchronized public void put(T dato){
        this.dato=dato;
    }
    synchronized public boolean datoDisponible(){
        return (this.dato != null);
    }
}
```

A este esquema de sincronización también se le llama **monitor** y consta de un mecanismo de acceso exclusivo al recurso (**synchronized**), un mecanismo de espera activa (**wait()**) y otro de comunicación entre hilos (**notify() y notifyAll()**).

CLASES THREAD-SAFE O SEGURAS EN APLICACIONES MULTIHILO

Una forma diferente de plantear el ejemplo anterior consiste en encapsular la sincronización dentro de la clase que representa el recurso compartido. En el caso anterior la clase Contenedor. Esta clase se convierte en una clase *Thread-safe*. Así los hilos solo tendrán que llamar a sus métodos sin más.

```
class Contenedor2<T> {
    private T dato;
    public T get() throws InterruptedException{
        T result;
        synchronized(this){
            while (!this.datoDisponible()){
                this.wait();
            }
            result= this.dato;
            this.dato = null;
            this.notifyAll();
        }
        return result;
    }
    public void put(T dato) throws InterruptedException{
        synchronized(this){
            while (this.datoDisponible()) {
                this.wait();
            }
            this.dato=dato;
            this.notifyAll();
        }
        this.dato=dato;
    }
    synchronized public boolean datoDisponible(){
        return (this.dato != null);
    }
}
```

INTERRUPCIÓN DE HILOS

A los hilos se les puede solicitar su interrupción usando el método **interrupt().**

Si el hilo se encuentra bloqueado en una llamada **sleep()** de la clase **Thread** o en espera por una llamada a **wait()** de la clase **Object**, se produce una excepción de tipo **InterruptedException**.

Los hilos también pueden detectar su estado de interrupción usando el método **isInterrupted()** de la clase **Thread**, sin necesidad de haber invocado a ninguno de los métodos anteriores.

```
/**
 *
 * @author Fran
 */
public class Interrupciones {

    public static void main(String[] args) {
        try {
            Thread h1 = new HiloInt1();
            Thread h2 = new HiloInt2();
            h1.start();
            h2.start();
            Thread.sleep(5000);
            h1.interrupt();
            h2.interrupt();
        } catch (InterruptedException ex) {
            System.out.println("Hilo principal interrumpido");
        }
    }
}

class HiloInt1 extends Thread {

    public void run() {
        try {
            while (true) {

                System.out.println("Hilo 1");
                Thread.sleep(1000);

            }
        } catch (InterruptedException ex) {
            System.out.println("Hilo 1 INTERRUMPIDO");
        }
    }
}

class HiloInt2 extends Thread {

    public void run() {
        while (!Thread.currentThread().isInterrupted()) {
            for (long i = 0; i < 10000000000L; i++) {

            }
            System.out.println("Hilo 2");
        }
```

```
        System.out.println("Hilo 2 INTERRUMPIDO");
    }
}
```

OTROS MECANISMOS DE SINCRONIZACIÓN

El paquete **java.util.concurrent** proveed diferentes mecanismos para el control de la concurrencia y la sincronización de hilos. Este paquete es muy amplio por lo que veremos solo algunos de estos mecanismos.

VARIABLES ATÓMICAS

Una variable atómica en Java es un tipo especial de variable que proporciona operaciones atómicas sin necesidad de sincronización explícita, evitando así el problema de la condición de carrera.

Se encuentran el paquete **java.util.concurrent.atomic** y algunos de los tipos son:

- **AtomicInteger**: Para enteros.
- **AtomicLong**: Para números largos.
- **AtomicBoolean**: Para valores booleanos.
- **AtomicReference**: Para referencias a objetos.

Las operaciones atómicas asociadas a estos tipos son:

- **get():** Obtiene el valor actual.
- **set(newValue):** Establece un nuevo valor.
- **getAndSet(newValue):** Obtiene el valor actual y establece uno nuevo.
- **compareAndSet(expect, update):** Compara con un valor esperado y actualiza si coincide.
- **incrementAndGet(), decrementAndGet():** Incrementa o decrementa y devuelve el nuevo valor.

COLECCIONES CONCURRENTES

La interface **BlockingQuene** define una cola FIFO que bloquea procesos cuando intentan extraer un elemento de la cola vacía y también permite especificar un número máximo de elementos de manera que no se puedan añadir más elementos de ese máximo. También permite especificar un tiempo máximo de bloqueo.

Algunas de las implementaciones de esta interface son: **ArrayBlockingQueue, LinkedBlockingQueue, SynchronousQueue,** etc.

Otra interface que permite esta funcionalidad es **ConcurrentMap** que contiene operaciones atómicas como eliminar, añadir o reemplazar una pareja clave-valor. Una clase que la implementa es **ConcurrentHashMap**, similar a **HashMap**.

SEMÁFOROS

Otro mecanismo que permite la sincronización y comunicación de hilos es el uso de semáforos.

Se pueden usar para controlar el acceso a un recurso con número finito de instancias. Está representado por una variable entera que indica en cada momento el número de instancias libres del recurso compartido y una cola de procesos bloqueados esperando para usar el recurso. Además, cuenta con dos operaciones atómicas: una para adquirir y disminuir el número de instancias libres y otra para aumentar el número de instancias. El número de instancias se define en el proceso de inicialización y el contador tomará valores comprendidos entre 0 y el valor inicial.

Podemos entender un semáforo como un portero que deja pasar a una sala tantas personas como admita el aforo de la misma. Cuando la sala esté completa no dejará pasar a nadie más y el que quiera hacerlo deberá ponerse en una cola de espera. Conforme van saliendo personas de la sala el portero va dejando pasar a las que estaban en cola, sin sobrepasar nunca el aforo.

En esta metáfora el portero hace la función del semáforo, la sala la sección crítica y las personas son los procesos que desean acceder a ella.

Cuando el valor inicial del contador del semáforo es 1 se conoce como semáforo binario.

En Java tenemos la clase **java.util.concurrent.Semaphore**

Métodos

Semaphore(int valor) Constructor del semáforo, valor es el número de permisos. Si valor=1 se dice que es un semáforo binario

Semaphore(int valor, boolean fair) El parámetro fair especifica que la adquisición del semáforo se realice de forma justa, por orden de llegada, previniendo así la inanición.

acquire(int valor) Implementa la operación de espera para acceder.

release((int valor) Implementa la operación de liberación.

El siguiente ejemplo veremos cómo gestionar un parking con aforo limitado mediante el uso de semáforos. Cada coche representa un hilo y el parking estará representado por un semáforo con tantas instancias como plazas disponibles.

Para poder aparcar cada coche (hilo) ha de obtener una instancia del semáforo, método **acquire(),** una vez finalizada su estancia liberará dicha instancia, con el método **release().**

```
/**
 *
 * @author fran
 */
class Parking {
    private int plazas;
    Parking(int n){
```

```
        plazas=n;
    }
    public int plazasLibres(){
        return plazas;
    }
    public void aparcar(){
        if (plazas>0)
            plazas--;
        else
            System.out.println("Parking Lleno");
    }
    public void salir(){
        plazas++;
    }

}
class Coche extends Thread {
    Semaphore sem;
    private int id;
    private Parking p;
    Coche (Semaphore s, Parking p, int id){
        this.id=id;
        sem = s;
        this.p = p;
    }
    public void run (){
        while(true)
        try {
            System.out.println("Esperando aparcamiento coche "+id);
            sem.acquire(1);
            System.out.println("Aparcando coche "+id);
            p.aparcar();
            sleep((int)(Math.random()*5000));
            System.out.println("Saliendo coche "+id);
            p.salir();
            sem.release(1);
        } catch (InterruptedException ex) { }
    }
}
public class ParkingSemaforo {
    public static void main(String[] args)  {
        int n = 5;
        Coche [] coches = new Coche[n*2];
        Parking p = new Parking(n);
        Semaphore s = new Semaphore(n);
        for (int i=0;i<n*2;i++){
            coches [i]=  new Coche(s,p,i);
            coches[i].start();
        }
```

```
    }
}
```

CARACTERÍSTICAS Y PROBLEMAS A EVITAR EN UN PROGRAMA MULTIHILO

Por lo general, las aplicaciones multihilo son más difíciles de desarrollar y complicadas de depurar que una aplicación secuencial o de un solo hilo.

Sin embargo, si utilizamos las librerías que aporta el lenguaje de programación nos pueden aportar algunas ventajas:

- **Facilitar la programación**. Requiere menos esfuerzo usar una clase estándar que desarrollarla para realizar la misma tarea.
- **Mayor rendimiento**. Los algoritmos utilizados han sido desarrollados por expertos en concurrencia y rendimiento.
- **Mayor fiabilidad**. Usar librerías o bibliotecas estándar, que han sido diseñadas para evitar interbloqueos (deadlocks), cambios de contexto innecesarios o condiciones de carrera, nos permiten garantizar un mínimo de calidad en nuestro software.
- **Menor mantenimiento**. El código que generemos será más legible y fácil de actualizar.
- **Mayor productividad**. El uso de una API estándar permite mejor coordinación entre desarrolladores y reduce el tiempo de aprendizaje.

Una aplicación multihilo debe reunir las siguientes **propiedades**:

- **Seguridad**. La aplicación no llegará a un estado inconsistente por un mal uso de los recursos compartidos. Esto implicará sincronizar hilos asegurando la exclusión mutua.
- **Viveza**. La aplicación no se bloqueará o provocará que un hilo no se pueda ejecutar. Esto implicará un comportamiento no egoísta de los hilos y ausencia de interbloqueos e inanición.

Por otro lado, los problemas más importantes a evitar en las aplicaciones multihilo serían:

1. Condiciones de carrera:
    - Ocurren cuando múltiples hilos acceden y modifican datos compartidos sin sincronización adecuada.
    - Pueden llevar a resultados impredecibles o corrupción de datos.
2. Interbloqueos (deadlocks):
    - Situación donde dos o más hilos se bloquean mutuamente esperando recursos que el otro tiene.
    - Resulta en un estancamiento donde ningún hilo puede avanzar.
3. Inanición (starvation):

- Ocurre cuando un hilo nunca obtiene los recursos necesarios para completar su tarea.
- Puede ser causado por una mala gestión de prioridades o recursos.

GESTIÓN DE GRUPOS DE HILOS

THREADGROUP

En las primeras versiones de Java se usaba la clase ThreadGroup, que aún está disponible, pero desaconsejada, para agrupar hilos en una estructura jerárquica. De esta manera se puede aplicar operaciones en conjunto (como interrumpir, suspender, etc.) a todos los hilos del grupo.

Además, proporciona un mecanismo para aislar hilos en grupos para una gestión más organizada.

Crear un Grupo de Hilos

```
// Crear un ThreadGroup
ThreadGroup grupo = new ThreadGroup("MiGrupo");
// Crear un hilo dentro del grupo
Thread hilo1 = new Thread(grupo, () -> {
    System.out.println("Hilo 1 ejecutándose");
}, "Hilo-1");

Thread hilo2 = new Thread(grupo, () -> {
    System.out.println("Hilo 2 ejecutándose");
}, "Hilo-2");
```

Gestión de Hilos

- activeCount(): Devuelve una estimación del número de hilos activos en el grupo.
- enumerate(Thread[] list): Copia en el array los hilos activos del grupo.
- interrupt(): Interrumpe todos los hilos del grupo.
- setMaxPriority(int pri): Establece la prioridad máxima para el grupo.

EXECUTORSERVICE EN JAVA - LA ALTERNATIVA MODERNA A THREADGROUP

Una alternativa más actualizada y adecuada a la programación actual es el uso de **ExecutorService**.

Jerarquía de clases

Executor

↓

ExecutorService

↓

Creación de ExecutorService

```java
public class CreacionExecutors {
    public static void main(String[] args) {
        // 1. Fixed Thread Pool - Número fijo de hilos
        ExecutorService fixedPool = Executors.newFixedThreadPool(4);

        // 2. Cached Thread Pool - Hilos que se crean según demanda
        ExecutorService cachedPool = Executors.newCachedThreadPool();

        // 3. Single Thread Executor - Un solo hilo secuencial
        ExecutorService singleThread = Executors.newSingleThreadExecutor();

        // 4. Scheduled Thread Pool - Para tareas programadas
        ScheduledExecutorService scheduledPool = Executors.newScheduledThreadPool(2);

        // 5. Work Stealing Pool - Usa el número de procesadores disponibles
        ExecutorService workStealingPool = Executors.newWorkStealingPool();

        System.out.println("Número de procesadores: " +
            Runtime.getRuntime().availableProcessors());
    }
}
```

Ejecución de Tareas con Runnable

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

public class TareasRunnable {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        // Enviar tareas Runnable
        for (int i = 1; i <= 6; i++) {
            final int taskId = i;
            executor.execute(() -> {
                System.out.println("Tarea " + taskId + " ejecutada por " +
                    Thread.currentThread().getName());
                try {
                    Thread.sleep(1000); // Simular trabajo
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            });
        }
```

```
        // Finalizar el executor
        executor.shutdown();
    }
}
```

BIBLIOGRAFÍA

- Programación de servicios y procesos. Ed. Síntesis.
- Apuntes de programación de servicios y procesos. Francisco Pérez.
- Documentación oficial de Java.
- Inteligencia Artificial. Perpeplexity.