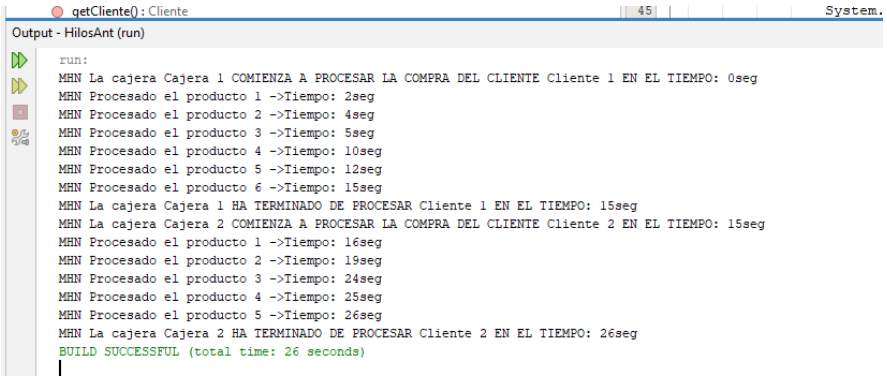
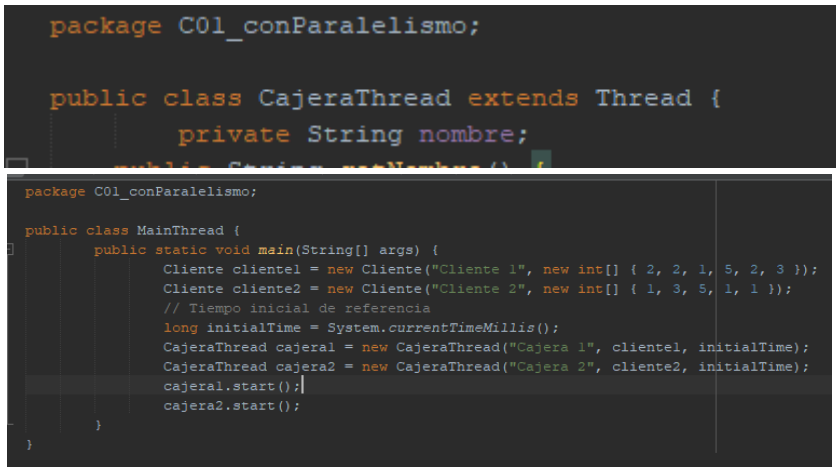
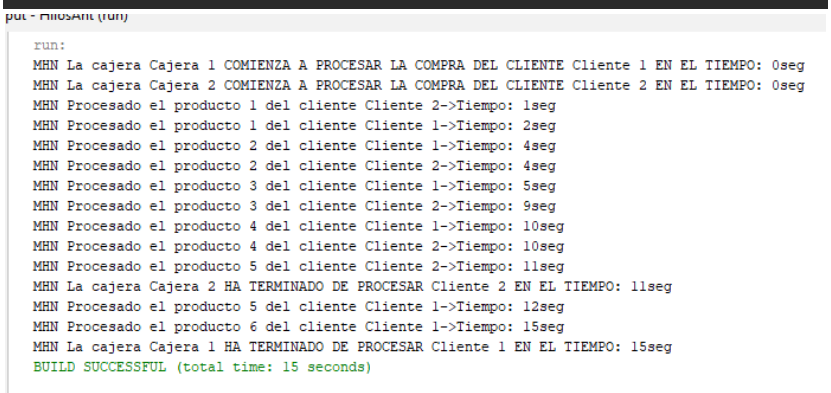
1. Códigos con y sin paralelismo

Código sin paralelismo: Al ejecutar el código se ve como la cajera atiende hasta el final el cliente 1 y una vez ha finalizado con él, comienza con el cliente 2. Los procesos no se mezclan en ningún momento ya que primero acaba un proceso y después empieza con el otro. El total del tiempo de todo el proceso es de 26 segundos.



Código con paralelismo: En cambio en este código podemos ver que las 2 cajeras atienden a los 2 clientes a la vez y el proceso es mucho más corto, durando 15 segundos. Esto es debido a que los 2 procesos se están ejecutando a la vez en 2 hilos distintos. Esto lo consigue heredando de la clase Thread.





MainRunable: En este otro caso, básicamente se consigue el mismo resultado que con el Código con Paralelismo. Simplemente se crean los hilos de forma distinta, pero con una característica fundamental que lo cambia todo y es que, cuando extendemos de la clase Thread, no se pueden heredar características de ninguna clase más, mientras que con runnable y el método run(), sí que podemos tener múltiples herencias. En vez de hacer un extends de Thread para que herede de esa clase, se usa Runnable para crear 2 tareas y ejecutarlas en 2 hilos con new Thread(procesoX).start();

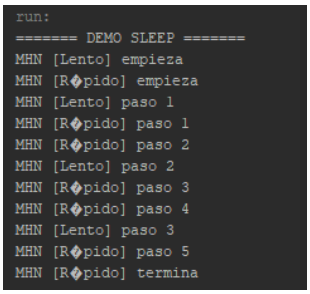
```java
package C01_conParalelismo_MainRunneable;

import C01_conParalelismo_MainRunneable.Cajera;
import C01_conParalelismo_MainRunneable.Cliente;

public class MainRunnable implements Runnable {


    private Cliente cliente;
    private Cajera cajera;
    private long initialTime;

    public MainRunnable(Cliente cliente, Cajera cajera, long initialTime) {
        this.cliente = cliente;
        this.cajera = cajera;
        this.initialTime = initialTime;
    }

    public static void main(String[] args) {

        Cliente clientel = new Cliente("Cliente 1", new int[]{2, 2, 1, 5, 2, 3});
        Cliente cliente2 = new Cliente("Cliente 2", new int[]{1, 3, 5, 1, 1});

        Cajera cajera1 = new Cajera("Cajera 1");
        Cajera cajera2 = new Cajera("Cajera 2");

        long initialTime = System.currentTimeMillis();

        Runnable proceso1 = new MainRunnable(clientel, cajera1, initialTime);
        Runnable proceso2 = new MainRunnable(cliente2, cajera2, initialTime);

        new Thread(proceso1).start();
        new Thread(proceso2).start();
    }

    @Override
    public void run() {
        this.cajera.procesarCompra(this.cliente, this.initialTime);
    }
}
```

```
Output - HilosAnt (run)
run:
MHN La cajera Cajera 2 COMIENZA A PROCESAR LA COMPRA DEL CLIENTE Cliente 2 EN EL TIEMPO: 0seg
MHN La cajera Cajera 1 COMIENZA A PROCESAR LA COMPRA DEL CLIENTE Cliente 1 EN EL TIEMPO: 0seg
MHN Procesado el producto 1 ->Tiempo: 1seg
MHN Procesado el producto 1 ->Tiempo: 2seg
MHN Procesado el producto 2 ->Tiempo: 4seg
MHN Procesado el producto 2 ->Tiempo: 4seg
MHN Procesado el producto 3 ->Tiempo: 5seg
MHN Procesado el producto 3 ->Tiempo: 9seg
MHN Procesado el producto 4 ->Tiempo: 10seg
MHN Procesado el producto 4 ->Tiempo: 10seg
MHN Procesado el producto 5 ->Tiempo: 11seg
MHN La cajera Cajera 2 HA TERMINADO DE PROCESAR Cliente 2 EN EL TIEMPO: 11seg
MHN Procesado el producto 5 ->Tiempo: 12seg
MHN Procesado el producto 6 ->Tiempo: 15seg
MHN La cajera Cajera 1 HA TERMINADO DE PROCESAR Cliente 1 EN EL TIEMPO: 15seg
BUILD SUCCESSFUL (total time: 15 seconds)
```

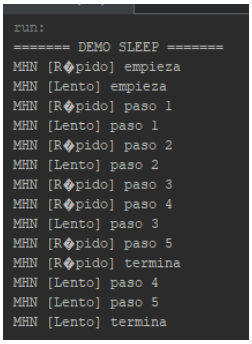
2. Código Sleep Yield set Proprity

Sleep: es un método para poner a “dormir” un hilo durante un tiempo determinado. Es decir, pausa temporalmente la ejecución y nosotros definimos ese tiempo de pausa.
Join es un método que lo que hace es decirle al hilo principal/main, que se detenga y espere a que el hilo que ha hecho el join, acabe su ejecución.
Así que cuando se comenta el s1.join() que es el hilo lento (es el que duerme más tiempo con Sleep), el DEMO SLEEP acaba cuando el hilo rápido (Sleep más corto) termina sin dejar que el hilo lento acabe su ejecución. En cambio, cuando se comenta el s2.join(), el DEMO SLEEP se acaba cuando termina el lento, pero el rápido también tiene tiempo de acabar. Así que, podemos decir que comentar el s2.join() no tiene ningún efecto.

- Comentando s1.join():

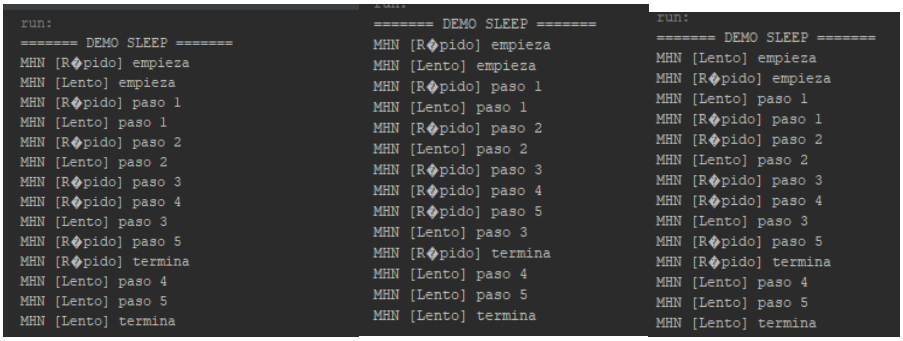


- Comentando s2.join():



No siempre empieza el [Lento]. El hilo que empieza puede ser uno u otro indistintamente ya que ninguno tiene prioridad y los 2 comienzan a la vez, sin ningún delay al inicio.

Tampoco sale el mismo orden de pasos en cada ejecución.



Yield: El yield es un método que sugiere ceder el turno a otro hilo si hay varios ejecutándose al mismo tiempo, pero al ser una sugerencia, no es algo obligatorio, así que puede o no hacer caso. Como se ve, termina antes el que no tiene Yield ya que este nunca cede el turno.

```
======= DEMO YIELD =======
MHN {SinYield} empieza
MHN {ConYield} empieza
MHN {SinYield} i=1
MHN {ConYield} i=1
MHN {SinYield} i=2
MHN {ConYield} i=2
MHN {SinYield} i=3
MHN {ConYield} i=3
MHN {SinYield} i=4
MHN {SinYield} i=5
MHN {ConYield} hace yield()
MHN {ConYield} i=4
MHN {SinYield} i=6
MHN {ConYield} i=5
MHN {SinYield} i=7
MHN {ConYield} i=6
MHN {SinYield} i=8
MHN {ConYield} hace yield()
MHN {ConYield} i=7
MHN {ConYield} i=8
MHN {ConYield} i=9
MHN {SinYield} i=9
MHN {SinYield} i=10
MHN {SinYield} i=11
MHN {SinYield} i=12
MHN {SinYield} i=13
MHN {SinYield} i=14
MHN {SinYield} i=15
MHN {SinYield} i=16
MHN {SinYield} i=17
MHN {ConYield} hace yield()
MHN {SinYield} i=18
MHN {SinYield} i=19
MHN {SinYield} i=20
MHN {ConYield} i=10
MHN {ConYield} i=11
MHN {ConYield} i=12
MHN {ConYield} hace yield()
MHN {ConYield} i=13
MHN {ConYield} i=14
MHN {SinYield} termina
MHN {ConYield} i=15
MHN {ConYield} hace yield()
MHN {ConYield} i=16
MHN {ConYield} i=17
MHN {ConYield} i=18
MHN {ConYield} hace yield()
MHN {ConYield} i=19
MHN {ConYield} i=20
MHN {ConYield} termina
```

```
======= DEMO YIELD =======
MHN {ConYield} empieza
MHN {SinYield} empieza
MHN {ConYield} i=1
MHN {SinYield} i=1
MHN {ConYield} i=2
MHN {SinYield} i=2
MHN {ConYield} i=3
MHN {SinYield} i=3
MHN {SinYield} i=4
MHN {ConYield} hace yield()
MHN {ConYield} i=4
MHN {SinYield} i=5
MHN {ConYield} i=5
MHN {SinYield} i=6
MHN {ConYield} i=6
MHN {SinYield} i=7
MHN {ConYield} hace yield()
MHN {ConYield} i=7
MHN {SinYield} i=8
MHN {ConYield} i=8
MHN {SinYield} i=9
MHN {ConYield} i=9
MHN {SinYield} i=10
MHN {ConYield} hace yield()
MHN {ConYield} i=10
MHN {SinYield} i=11
MHN {SinYield} i=12
MHN {SinYield} i=13
MHN {SinYield} i=14
MHN {SinYield} i=15
MHN {SinYield} i=16
MHN {SinYield} i=17
MHN {SinYield} i=18
MHN {ConYield} i=11
MHN {ConYield} i=12
MHN {SinYield} i=19
MHN {ConYield} hace yield()
MHN {SinYield} i=20
MHN {ConYield} i=13
MHN {SinYield} termina
MHN {ConYield} i=14
MHN {ConYield} i=15
MHN {ConYield} hace yield()
MHN {ConYield} i=16
MHN {ConYield} i=17
MHN {ConYield} i=18
MHN {ConYield} hace yield()
MHN {ConYield} i=19
MHN {ConYield} i=20
MHN {ConYield} termina
```

Priority: este método sí que otorga prioridades distintas a cada uno de los hilos. Debido a esta prioridad, lógicamente debería acabar antes el hilo con más prioridad, aunque como en los casos anteriores, no sigue siempre el mismo orden. Esto es debido a que por temas de CPU o del sistema el orden puede variar, aunque por lo general en las pruebas realizadas, la mayoría de las veces, el de prioridad alta termina primero.
Cuando comentamos el p2.setPriority para ver lo que sucedería al hacerlo, se puede ver que no hay ninguna diferencia perceptible. Esto es porque la prioridad asignada a ese hilo es de 5, que es la normal. Es decir que le asignamos la que hereda por defecto del hilo principal, por lo que básicamente al comentarla, no hacemos ningún cambio.

```
======= DEMO PRIORITY =======
MHN Prioridad_Baja = 1
MHN Prioridad_Media = 5
MHN Prioridad_Alta = 10
MHN [Prioridad_Baja] i=0
MHN [Prioridad_Media] i=0
MHN [Prioridad_Alta] i=0
MHN [Prioridad_Alta] i=10000000
MHN [Prioridad_Baja] i=10000000
MHN [Prioridad_Media] i=10000000
MHN [Prioridad_Media] i=20000000
MHN [Prioridad_Alta] i=20000000
MHN [Prioridad_Baja] i=20000000
MHN [Prioridad_Alta] i=30000000
MHN [Prioridad_Media] i=30000000
MHN [Prioridad_Baja] i=30000000
MHN [Prioridad_Alta] i=40000000
MHN [Prioridad_Media] i=40000000
MHN [Prioridad_Baja] i=40000000
MHN [Prioridad_Alta] termina en 61 ms (sum=1249999975000000)
MHN [Prioridad_Media] termina en 69 ms (sum=1249999975000000)
MHN [Prioridad_Baja] termina en 71 ms (sum=1249999975000000)

MHN Fin de todas las demos
BUILD SUCCESSFUL (total time: 2 seconds)
```

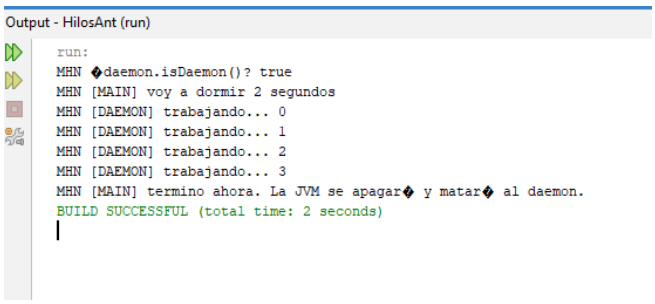
```
======= DEMO PRIORITY =======
MHN Prioridad_Baja = 1
MHN Prioridad_Media = 5
MHN Prioridad_Alta = 10
MHN [Prioridad_Baja] i=0
MHN [Prioridad_Alta] i=0
MHN [Prioridad_Media] i=0
MHN [Prioridad_Media] i=10000000
MHN [Prioridad_Alta] i=10000000
MHN [Prioridad_Baja] i=10000000
MHN [Prioridad_Media] i=20000000
MHN [Prioridad_Alta] i=20000000
MHN [Prioridad_Baja] i=20000000
MHN [Prioridad_Alta] i=30000000
MHN [Prioridad_Media] i=30000000
MHN [Prioridad_Baja] i=30000000
MHN [Prioridad_Alta] i=40000000
MHN [Prioridad_Media] i=40000000
MHN [Prioridad_Baja] i=40000000
MHN [Prioridad_Alta] termina en 68 ms (sum=1249999975000000)
MHN [Prioridad_Media] termina en 70 ms (sum=1249999975000000)
MHN [Prioridad_Baja] termina en 74 ms (sum=1249999975000000)
```

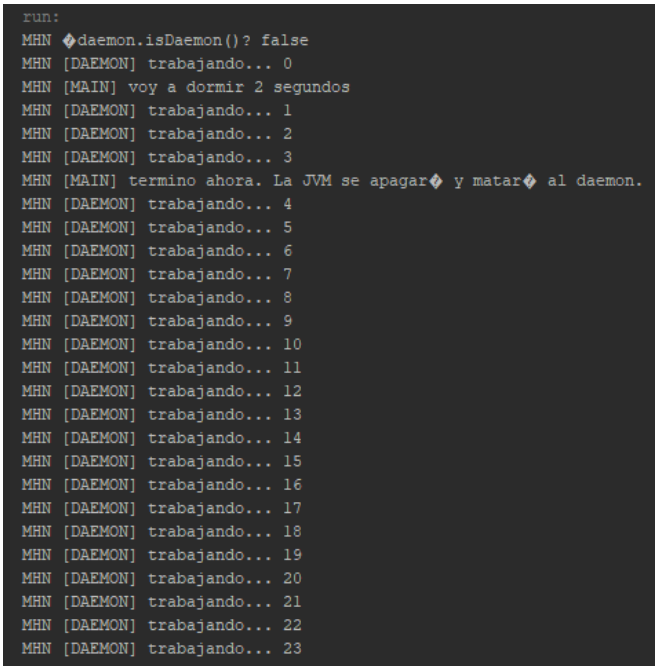
```
======= DEMO PRIORITY =======
MHN Prioridad_Baja = 1
MHN Prioridad_Media = 5
MHN Prioridad_Alta = 10
MHN [Prioridad_Baja] i=0
MHN [Prioridad_Alta] i=0
MHN [Prioridad_Media] i=0
MHN [Prioridad_Media] i=10000000
MHN [Prioridad_Alta] i=10000000
MHN [Prioridad_Baja] i=10000000
MHN [Prioridad_Media] i=20000000
MHN [Prioridad_Alta] i=20000000
MHN [Prioridad_Baja] i=20000000
MHN [Prioridad_Media] i=30000000
MHN [Prioridad_Alta] i=30000000
MHN [Prioridad_Baja] i=30000000
MHN [Prioridad_Alta] i=40000000
MHN [Prioridad_Media] i=40000000
MHN [Prioridad_Baja] i=40000000
MHN [Prioridad_Baja] termina en 72 ms (sum=1249999975000000)
MHN [Prioridad_Media] termina en 68 ms (sum=1249999975000000)
MHN [Prioridad_Alta] termina en 66 ms (sum=1249999975000000)
```

3. Demonios

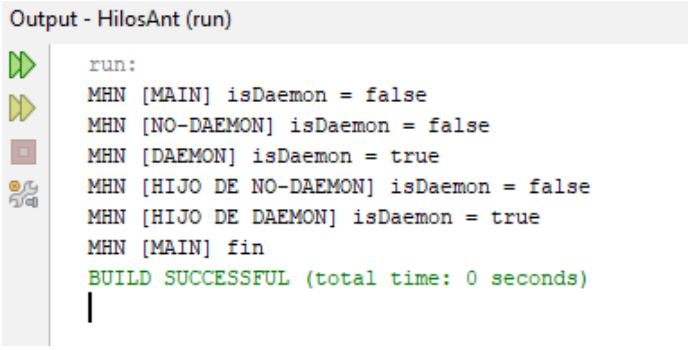
Para empezar un hilo Daemon es un hilo de baja prioridad que se ejecuta en segundo plano. Puede ser para tareas en segundo plano para soporte o servicios del sistema, por ejemplo. Se usa para cosas que al cortarse la ejecución no afecte en nada.

- DaemonBasico: aquí se está ejecutando un bucle infinito que hace un print cada medio segundo y el hilo main duerme a los 2 segundos y se termina el proceso. Como el hilo Daemon está marcado como true, si el hilo main termina, el Damon también se detiene automáticamente porque la JVM se apaga. Pero si comentamos el true o lo ponemos en false, el Daemon entra en un bucle infinito que no se corta al terminar el proceso del hilo main. Esto sucede porque el true lo que indica, es que ese hilo es un hilo de secundario de servicio. Así que cuando está en true, el Daemon se corta al final la ejecución del main y si está en false, el Daemon se sigue ejecutando indefinidamente.





- DaemonHerencia: en este caso lo que está sucediendo es que creamos 2 hilos, uno normal y el otro es un hilo Daemon. Después cada uno de ellos tiene un hijo que hereda las características de cada uno de los padres. Es decir, el hijo del Daemon, hereda la característica de ser Daemon sin que tengamos que asignarlo de forma manual, ya que hereda directamente esa característica del padre. Y con el noDaemon pasa lo mismo, hereda la característica de ser un hilo normal del padre.



Ahora bien, nosotros indicamos en el main cuando un hilo es Daemon o no con true o false. Pero una vez iniciado no se puede cambiar sus propiedades o petará.

```
daemon.setDaemon(true); // esto es lo que indica sies daemon o no
noDaemon.start();
daemon.start();
daemon.setDaemon(false); // cambio la caracteristica después de iniciarlo a true
noDaemon.join();
daemon.join();
System.out.println(m + "[MAIN] fin");
```

```
run:
MHN [MAIN] isDaemon = false
Exception in thread "main" java.lang.IllegalThreadStateException
MHN [NO-DAEMON] isDaemon = false
        at java.base/java.lang.Thread.setDaemon(Thread.java:2178)
        at C03_Demonios.DaemonHerencia.main(DaemonHerencia.java:34)
MHN [DAEMON] isDaemon = true
MHN [HIJO DE NO-DAEMON] isDaemon = false
C:\Users\Lucifer\AppData\Local\NetBeans\Cache\23\executor-snippets\run.xml:111: The following error occurred while executing this line:
C:\Users\Lucifer\AppData\Local\NetBeans\Cache\23\executor-snippets\run.xml:68: Java returned: 1
BUILD FAILED (total time: 0 seconds)
```

4. Grupos hilos

Los grupos de hilos son una forma de organizar los hilos, como si estuvieran en carpetas para ordenarlos de alguna forma y dentro de esa carpeta, puede haber más carpetas para ser más detallado a la hora de ordenarlo/organizarlo todo.
En este código se hacen 2 grupos de hilos con cada uno una serie de hilos, tareas y prioridades.

```
run:
MHN Grupo padre: Grupo Padre
MHN Subgrupo: Grupo Hijo
MHN Padre del subgrupo: Grupo Padre
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci□n 1
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci□n 1
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci□n 1

--- Info de grupos mientras los hilos est□n activos ---
MHN Hilos activos en grupoPadre (solo ese grupo): 3
MHN Subgrupos activos dentro de grupoPadre: 1

--- Hilos encontrados en grupoPadre (sin subgrupos):
MHN - Hilo-A (padre) (grupo: Grupo Padre)
MHN - Hilo-B (padre) (grupo: Grupo Padre)

Subgrupos dentro de grupoPadre:
MHN - Grupo Hijo

--- Prioridades y grupos ---
MHN Prioridad m□xima inicial del subgrupo: 10
MHN Nueva prioridad m□xima del subgrupo: 4
MHN Prioridad solicitada para Hilo-Prioridad: 10
MHN Prioridad real de Hilo-Prioridad (limitada por el grupo): 4
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci□n 1
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci□n 2
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci□n 2
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci□n 2
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci□n 2
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci□n 3
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci□n 3
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci□n 3
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci□n 3
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci□n 4
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci□n 4
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci□n 4
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci□n 4
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci□n 5
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci□n 5
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci□n 5
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci□n 5
MHN [Grupo Hijo] Hilo-C (hijo) ha terminado.
MHN [Grupo Padre] Hilo-B (padre) ha terminado.
MHN [Grupo Padre] Hilo-A (padre) ha terminado.
MHN [Grupo Hijo] Hilo-Prioridad ha terminado.

--- Info de grupos tras terminar los hilos ---
MHN Hilos activos en grupoPadre: 0
MHN Subgrupos activos en grupoPadre: 1

MHN Fin de main().
BUILD SUCCESSFUL (total time: 2 seconds)
```

Si se pone true en enumerate en vez de salir solo el Hilo-A y el B del grupo padre, sale también el Hilo-C del Grupo hijo porque ahora enumerate incluye a los subgrupos con ese true.
Así que si queremos que se enumeren todos los hilos en conjunto y no solo los del grupo padre pondremos true.

```
--- Hilos encontrados en grupoPadre (sin subgrupos):
MHN - Hilo-A (padre) (grupo: Grupo Padre)
MHN - Hilo-B (padre) (grupo: Grupo Padre)
MHN - Hilo-C (hijo) (grupo: Grupo Hijo)

Subgrupos dentro de grupoPadre:
MHN - Grupo Hijo
```

El Thread.sleep se pone para dar tiempo a los hilos a arrancar antes de pedirle información de cuantos hay, así que, si no se pone, podría darse el caso de que el main procese tan rápido la secuencia, que llegue a poner 0 hilos activos, aunque en mi caso esto no ha sucedido nunca.

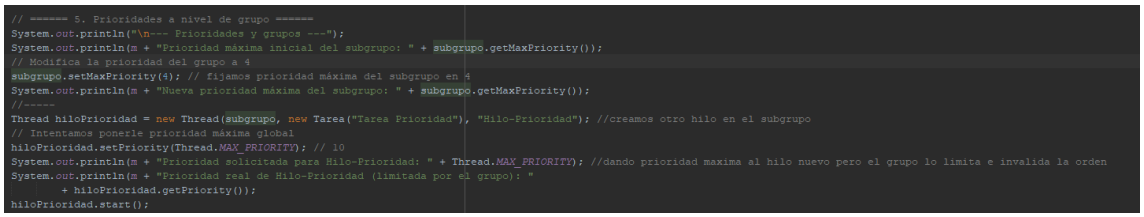
```
// Esperamos un poco para asegurarnos de que ya han arrancado
//Thread.sleep(200);
// ====== 4. Información de grupos mientras los hilos están activos ======
System.out.println("\n--- Info de grupos mientras los hilos están activos ---");
System.out.println(m + "Hilos activos en grupoPadre (solo ese grupo): " + grupoPadre.activeCount());
System.out.println(m + "Subgrupos activos dentro de grupoPadre: " + grupoPadre.activeGroupCount());
// Lista de hilos dentro de grupoPadre (sin recorrer subgrupos)
```

```
--- Info de grupos mientras los hilos est�n activos ---
MHN Hilos activos en grupoPadre (solo ese grupo): 3
MHN Subgrupos activos dentro de grupoPadre: 1
```

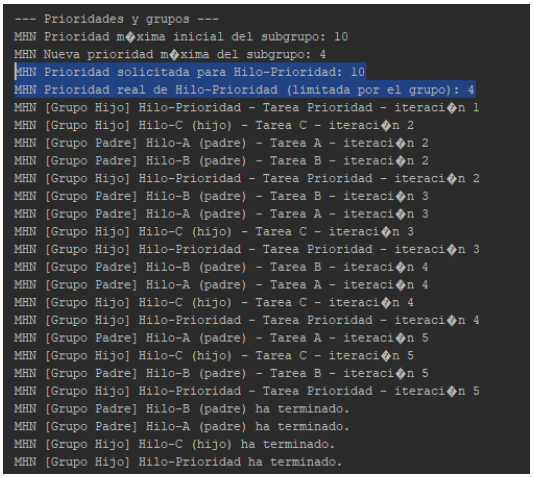
Si creamos un hilo sin especificar un grupo, siempre se asignará al grupo por defecto que es el main. Por lo tanto no aparece en el grupoPadre ni en el Hijo

```
run:
MHN Grupo padre: Grupo Padre
MHN Subgrupo: Grupo Hijo
MHN Padre del subgrupo: Grupo Padre
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci�n 1
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci�n 1
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci�n 1
MHN [main] Hilo-Default - Tarea Default - iteraci�n 1

--- Info de grupos mientras los hilos est�n activos ---
MHN Hilos activos en grupoPadre (solo ese grupo): 3
MHN Subgrupos activos dentro de grupoPadre: 1

--- Hilos encontrados en grupoPadre (sin subgrupos):
MHN - Hilo-A (padre) (grupo: Grupo Padre)
MHN - Hilo-B (padre) (grupo: Grupo Padre)

Subgrupos dentro de grupoPadre:
MHN - Grupo Hijo

--- Hilo sin Grupo
MHN - Hilo default: main
```

Lo interesante aquí también es que podemos limitar la prioridad de todo un grupo. Si ponemos el máximo del grupo en 4 e intentamos que un hilo dentro de él tenga prioridad 10, el grupo invalida la orden y lo limita a 4, manteniendo el control jerárquico."



```
// ====== 5. Prioridades a nivel de grupo ======
System.out.println("\n--- Prioridades y grupos ---");
System.out.println(m + "Prioridad máxima inicial del subgrupo: " + subgrupo.getMaxPriority());
// Modifica la prioridad del grupo a 4
subgrupo.setMaxPriority(4); // fijamos prioridad máxima del subgrupo en 4
System.out.println(m + "Nueva prioridad máxima del subgrupo: " + subgrupo.getMaxPriority());
//-----
Thread hiloPrioridad = new Thread(subgrupo, new Tarea("Tarea Prioridad"), "Hilo-Prioridad"); //creamos otro hilo en el subgrupo
// Intentamos ponerle prioridad máxima global
hiloPrioridad.setPriority(Thread.MAX_PRIORITY); // 10
System.out.println(m + "Prioridad solicitada para Hilo-Prioridad: " + Thread.MAX_PRIORITY); //dando prioridad maxima al hilo nuevo pero el grupo lo limita e invalida la orden
System.out.println(m + "Prioridad real de Hilo-Prioridad (limitada por el grupo): "
        + hiloPrioridad.getPriority());
hiloPrioridad.start();
```

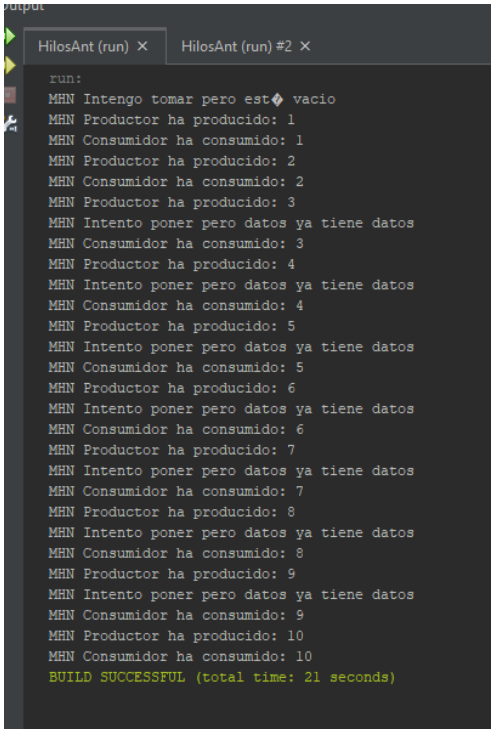


```
--- Prioridades y grupos ---
MHN Prioridad m◆xima inicial del subgrupo: 10
MHN Nueva prioridad m◆xima del subgrupo: 4
MHN Prioridad solicitada para Hilo-Prioridad: 10
MHN Prioridad real de Hilo-Prioridad (limitada por el grupo): 4
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci◆n 1
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci◆n 2
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci◆n 2
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci◆n 2
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci◆n 2
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci◆n 3
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci◆n 3
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci◆n 3
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci◆n 3
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci◆n 4
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci◆n 4
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci◆n 4
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci◆n 4
MHN [Grupo Padre] Hilo-A (padre) - Tarea A - iteraci◆n 5
MHN [Grupo Hijo] Hilo-C (hijo) - Tarea C - iteraci◆n 5
MHN [Grupo Padre] Hilo-B (padre) - Tarea B - iteraci◆n 5
MHN [Grupo Hijo] Hilo-Prioridad - Tarea Prioridad - iteraci◆n 5
MHN [Grupo Padre] Hilo-B (padre) ha terminado.
MHN [Grupo Padre] Hilo-A (padre) ha terminado.
MHN [Grupo Hijo] Hilo-C (hijo) ha terminado.
MHN [Grupo Hijo] Hilo-Prioridad ha terminado.
```

5. Sincronización

ProductorConsumidorSimple: Aquí se puede ver que el código lo que hace es poner datos y consumirlos en nuestra clase Buffer que es una memoria temporal compartida para los 2 hilos y la variable dato, es la que representa el contenido de ese buffer. Pero si ya hay 1 dato, no se puede poner otro y si ya se ha tomado el dato, no se puede tomar otro porque está vacío. Esto generalmente se usa para recursos compartidos a los que tienen acceso varios hilos y quieren acceder al mismo tiempo. Esto causa problemas de inconsistencias en los datos o los corrompe directamente. La suma no atómica del ejercicio siguiente es un buen ejemplo de ello.

Se usa wait() para hacer dormir al hilo que tenga el recurso para liberarlo y que otro hilo pueda tomarlo, notify() para despertar un hilo al azar y que este se ponga a trabajar, pero al ser al azar, no podemos controlar qué va a ejecutarse en ese momento y finalmente notifyAll() que despierta todos los hilos y que uno de ellos se ejecute.



Si quitamos el syncronized del método tomar, sin tocar nada más, nos dará una excepción porque seguimos usando wait() y notifyAll() y estos solo pueden ejecutarse si el hilo posee lo que se conoce como monitor, que es, por decirlo así, la llave de la cerradura al recurso que se intenta acceder, pero al quitar el syncronized, ya no hay cerradura, así que se genera el error.

```
run:
MHN Intento tomar pero est vacio
Exception in thread "Consumidor" java.lang.IllegalMonitorStateException: current thread is not owner
        at java.base/java.lang.Object.wait0(Native Method)
        at java.base/java.lang.Object.wait(Object.java:378)
        at java.base/java.lang.Object.wait(Object.java:352)
        at C05_Sincronizacion.ProductorConsumidorSimple$Buffer.tomar(ProductorConsumidorSimple.java:26)
        at C05_Sincronizacion.ProductorConsumidorSimple$Consumidor.run(ProductorConsumidorSimple.java:66)
        at java.base/java.lang.Thread.run(Thread.java:1575)
MHN Productor ha producido: 1
MHN Intento poner pero datos ya tiene datos
```

Si eliminamos los wait() y los notiffAll() ya no se producirá la excepción, pero en cambio la operación pierde todo control e intenta acceder a recursos que no están o puede sobrescribir datos continuamente entrando en un bucle sin fin.

```
run:
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
MHN Intento tomar pero est vacio
```

SincronizaciónContador: En este caso tenemos un contador que es el recurso compartido de los 2 hilos. Este recurso llamado valor, no es un número atómico, lo cual quiere decir que no es una sola cosa sino un conjunto de varias acciones o pasos. Estos pasos son de lectura, suma y guardado del valor en la variable. Y como 2 hilos a la vez están haciendo estas acciones al mismo tiempo, en ocasiones suman 1 a un número que ya se ha incrementado pero que en el momento de la lectura aún no se había sumado. Por lo tanto, esto va a hacer que alguna suma se pierda porque están guardando el mismo resultado de una suma a la vez.
Y como se puede ver en la captura, hay una diferencia bien perceptible entre lo esperado y el valor real que ha ido incrementándose.

```
public void incrementar() {
    valor++; // NO es atómico
} // lo de atómico es que se hace lo de leer, sumar y guardar a la vez. Que es un paso indivisible y en este caso no lo es
// lo que puede hacer que los 2 hilos lo llamen a la vez y lo sumen a la vez y se puierda uno de los incrementos por ejemplo
```

```
run:
MHN Valor esperado: 2000000
MHN Valor real:     1926120
MHN Tiempo: 12 ms
BUILD SUCCESSFUL (total time: 0 seconds)
```

Ahora usando syncronized, que es como un guardia o una cerradura que controla el acceso a el método de poner, se puede ver que ya no hay pérdidas en el incremento y el valor esperado y el valor real son el mismo. Esto es porque mientras un hilo está dentro de incrementar, no permite que el otro entre y hace que espere su turno.

```
// Versión CON sincronización
public synchronized void incrementar() {
    valor++; // ahora este incremento es atómico respecto a otros hilos
}
```

```
run:
MHN Valor esperado: 2000000
MHN Valor real:     2000000
MHN Tiempo: 35 ms
BUILD SUCCESSFUL (total time: 0 seconds)
```

Si le sumamos más hilos a la ecuación, podemos observar que, entre el valor esperado y el valor real, se abre una brecha más grande ya que son más hilos intentando acceder al mismo valor, por lo que el número de sumas que se pierden es mayor también. Y eso sin haber ajustado el número de repeticiones, que al ser 5 hilos ahora, deberían de ser 5 * 1000000, es decir 5000000 y no 2000000.

```
public static void main(String[] args) throws InterruptedException {
    Contador contador = new Contador();
    int repeticiones = 1_000_000;
    Thread t1 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-1");
    Thread t2 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-2");
    Thread t3 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-3");
    Thread t4 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-4");
    Thread t5 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-5");
    long inicio = System.currentTimeMillis();
    t1.start();
    t2.start();
    t3.start();
    t4.start();
    t5.start();
    t1.join(); // esto es para que no siga con el programa hasta que hayan terminado los 2 hilos
    t2.join();
    t3.join();
    t4.join();
    t5.join();
```

```
run:
MHN Valor esperado: 2000000
MHN Valor real:     1308843
MHN Tiempo: 14 ms
BUILD SUCCESSFUL (total time: 0 seconds)
```

Si ponemos las repeticiones que corresponden por el número de hilos, la diferencia es abismal.

```
run:
MHN Valor esperado: 5000000
MHN Valor real:     1177605
MHN Tiempo: 12 ms
BUILD SUCCESSFUL (total time: 0 seconds)
```

En cambio, al usar de nuevo syncronized, podemos ver que otra vez se ajusta al valor esperado.

```java
public static void main(String[] args) throws InterruptedException {
    Contador contador = new Contador();
    int repeticiones = 1_000_000;
    Thread t1 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-1");
    Thread t2 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-2");
    Thread t3 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-3");
    Thread t4 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-4");
    Thread t5 = new Thread(new TareaIncremento(contador, repeticiones), "Hilo-5");
    long inicio = System.currentTimeMillis();
    t1.start();
    t2.start();
    t3.start();
    t4.start();
    t5.start();
    t1.join(); // esto es para que no siga con el programa hasta que hayan terminado los 2 hilos
    t2.join();
    t3.join();
    t4.join();
    t5.join();
    long fin = System.currentTimeMillis();
    System.out.println(m + "Valor esperado: " + (5 * repeticiones));
    System.out.println(m + "Valor real:     " + contador.getValor());
    System.out.println(m + "Tiempo: " + (fin - inicio) + " ms");
```

```
run:
MHN Valor esperado: 5000000
MHN Valor real:     5000000
MHN Tiempo: 100 ms
BUILD SUCCESSFUL (total time: 0 seconds)
```