Programación de Servicios y Procesos

BLOQUE 2:
PROGRAMACIÓN MULTIHILO
U3-Threads

ÍNDICE

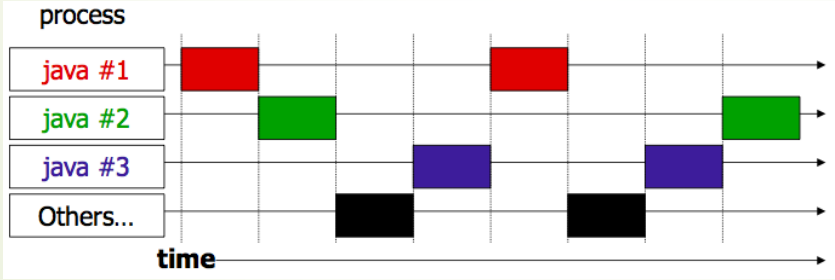
- 1. OBJETIVOS
- 2. INTRODUCCIÓN
- 3. BENEFICIOS Y RIESGOS
- 4. HILOS EN JAVA
- 5. CICLO DE VIDA DEL HILO

1. OBJETIVOS

- Comprender la importancia de los hilos en los sistemas actuales
- Aprender los beneficios y los riesgos de utilizar hilos.
- Aprender como se implementa la programación multihilo en Java para lograr la programación paralela o la programación asíncrona.
- Crear clases definidias por el programador hacienda uso de las características de los hilos.
- Escribir aplicaciones multihilo.

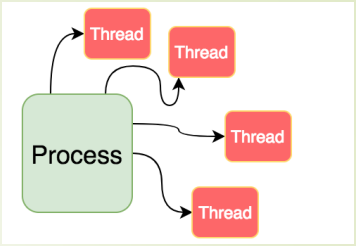
2. INTRODUCCIÓN

- **SO Monotarea**: ejecuta sólo una tarea al mismo tiempo y esa tiene acceso a todos los recursos de la máquina
- **SO Multitatera**: El tiempo de uso de la CPU se comparte por los procesos en ejecución.

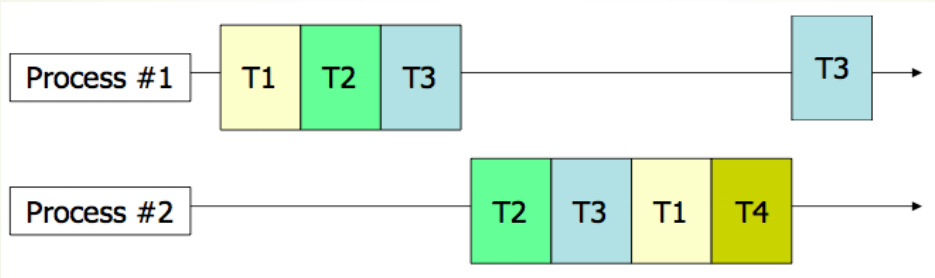


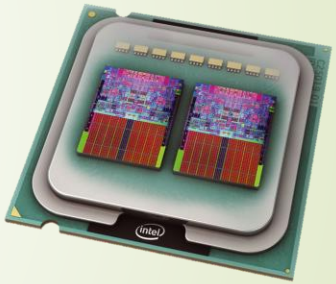
- Retos para los desarrolladores:
  - Compartir el tiempo de CPU
  - Compartir la memoria y otros recursos del PC.

2. INTRODUCTION



- **Multi-threading:** Múltiples hilos de ejecución dentro del mismo proceso.





- Los hilos se ejecutan dentro del espacio de memoria del proceso **compartiendo** sus recursos.
- Más **dificultades** para los desarrolladores: Los threads pueden leer y escribir simultáneamente en el mismo recurso del proceso.

1. INTRODUCCIÓN

Consideraciones

- *Un hilo **lee** una dirección de memoria mientras otro la **escribe**. ¿Qué valor leerá finalmente el primer hilo?*
  - *¿El valor viejo?*
  - *¿El valor escrito por el segundo hilo?*
  - *¿O un valor que es una mezcla de los dos?*
- *Y ¿ Si dos hilos están escribiendo en la misma posición de memoria simultánemanete, que valor habrá en la dirección cuando finalicen?*
  - *¿El valor escrito por el primer hilo?*
  - *¿El valor escrito por el segundo?*
  - *¿Un mix entre los dos?*

1. INTRODUCTION

Consideraciones

- Sin las precauciones adecuadas **cualquiera** de los resultados anteriores es posible.
- **No** se puede **predecir** el comportamiento
- El resultado final pdría **cambiar** en cada ejecución.

- Los desarrolladores de Java se suelen enfrentar a estos problemas.
- Java tiene integradas capacidades de multithreading que ayudan a lidiar con estos problemas.

3. BENEFICIOS Y RIESGOS

- La JVM es capaz de realizar **múiltiples** tareas en un único proceso mediante uno o más hilos para cada tarea: manejo de memoria, colector de basura…
- Utilizando hilos podemos llegar a conseguir estos beneficions:
  - Mejor utilización de los recursos.
  - Diseños de programas más simples en algunas situaciones.
  - Programas más interactivos.

- En cambio, los desarrolladores tiene que tener cuidado con los problemas relaciondos con **thread safety**, **liveness** and **performance.**

Actividad:

Lee el artículo que encontrarás en la documentación de la unidad: ***"Threads are everywhere".***

Y contesta a las siguientes preguntas:

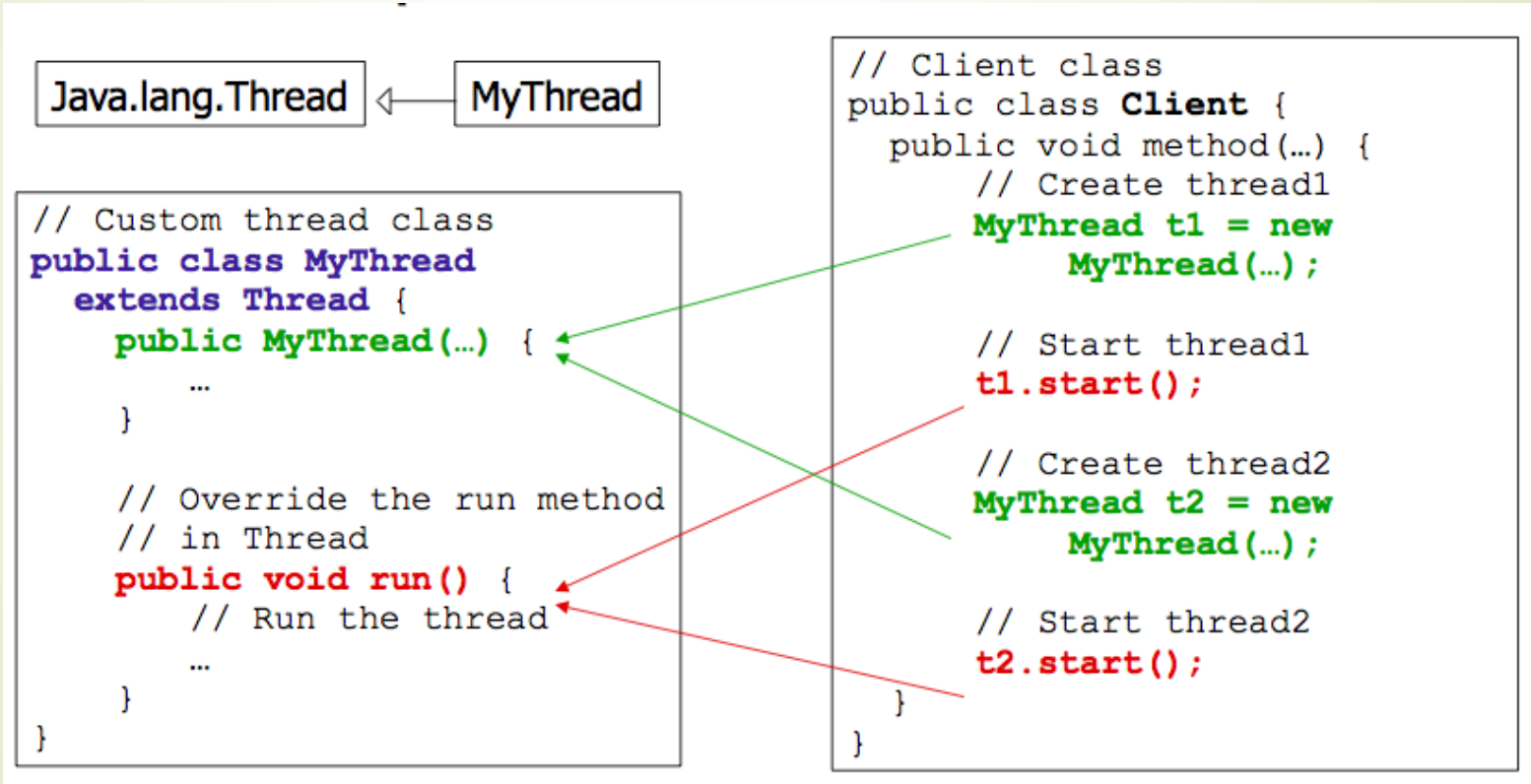
1. **¿Por qué es importante que las clases sean "thread-safe" cuando se trabaja con frameworks que crean hilos automáticamente?**
2. **Explica cómo el uso de un temporizador (Timer) puede complicar un programa que originalmente era secuencial. ¿Qué medidas se deben tomar para garantizar que el código sea seguro para hilos?**
3. **¿De qué manera asegura Swing la seguridad de hilos en aplicaciones de GUI? ¿Por qué es necesario ejecutar las operaciones de interfaz en el hilo de eventos?**

4. HILOS EN JAVA

- Cuando un programa de Java se ejectua, la JVM lanza un hilo que corra el método main(): **"hilo principal"**
- Los hilos en Java son objetos como cualquier otro objeto en java con la habilidad de ejecutar código independiente y en concurrencia con otros hilos del propio proceso.
- Cada hilo es una instancia de una clase que:
  - **Heredando** de la clase thread **Thread**
  - **Implementa** la interfaz **Runnable**
- El nuevo objeto es un **objeto ejecutable** (también conocido como **objeto activo**).

4. HILOS JAVA

Crear hilos extendiendo la clase Thread



4. HILOS JAVA

Crear hilos extendiendo la clase Thread

Exercise 1:

Escribe un programa que arranque **tres** hilos independientes.

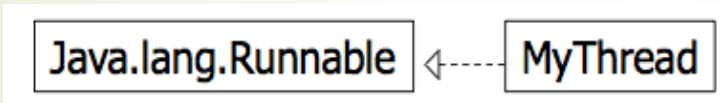
- Hilo uno: Imprime la letra 'A' 50 veces.
- Hilo dos: Imprime la letra 'T' 50 veces.
- Hilo tres: Imprime los números enteros desde el 1 hasta el 50.

- *Consejo: Crea dos clases: **PrintChar**, se encargue de ejecutar los dos primeros casos y otra clase, **PrintNumber**, para el tercer hilo.*

- Ejecuta el programa y mira la salida. ¿Que podemos decir de ella?
- Ahora cambia el número de 50 a 5000 y ejecuta el programa. ¿Es distinta la salida? ¿Por qué?

4. HILOS JAVA

Creando hilos implementando la interfaz Runnable




```
// Custom thread class
public class MyThread
  implements Runnable {
    public MyThread(…) {
        …
    }

    // Override the run method
    // in Thread
    public void run() {
        // Run the thread
        …
    }
}
```

```
// Client class
public class Client {
  public void method(…) {
      // Create thread1
      Thread t1 = new Thread (
          new MyThread(…));

      // Start thread1
      t1.start();

      // Create thread2
      Thread t2 = new Thread (
          new MyThread(…));

      // Start thread2
      t2.start();
  }
}
```

4. HILOS JAVA

Cambiando y leyendo el nombre del thread

```
MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable, "New Thread");

thread.start();
System.out.println(thread.getName());
```

```
String threadName = Thread.currentThread().getName();
```

```
public class MyThread extends Thread {

    public MyThread(String name) {
        super(name);
    }
```

4. HILOS JAVA

Creando hilos implementando la interfaz Runnable

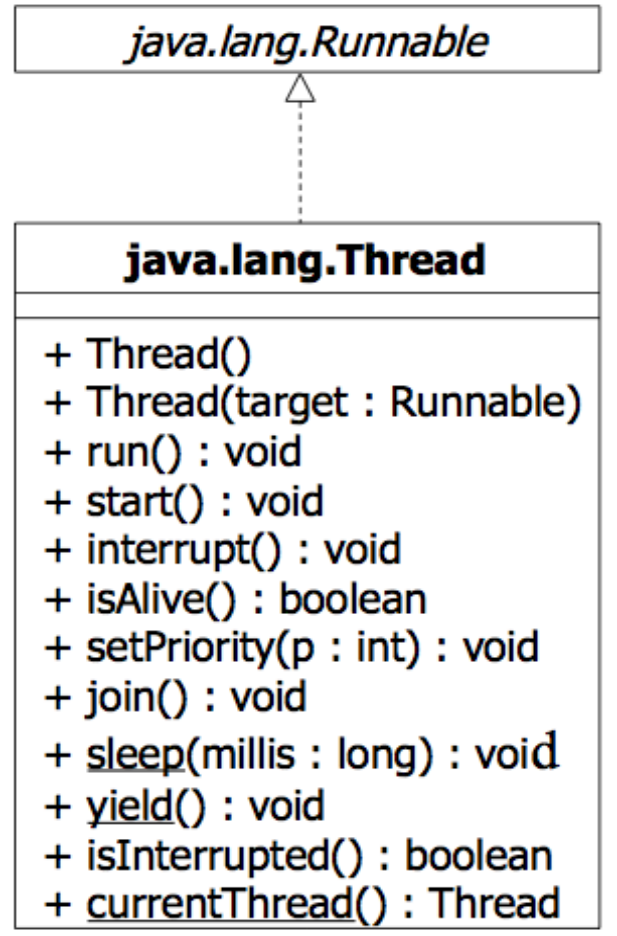
Ejercicio 2:

- Modifica las clases thread del Ejercicio 1 para que implementen la interfaz **Runnable**.
  - No ha haber ningún cambio en la ejecución.
  - Dale un **nombre** a cada **hilo** y imprímelo por pantalla.

- ¿Cómo se le daría nombre a un thread cuando se hereda de la case Thread?

4. HILOS JAVA

Métodos de la Clase Thread



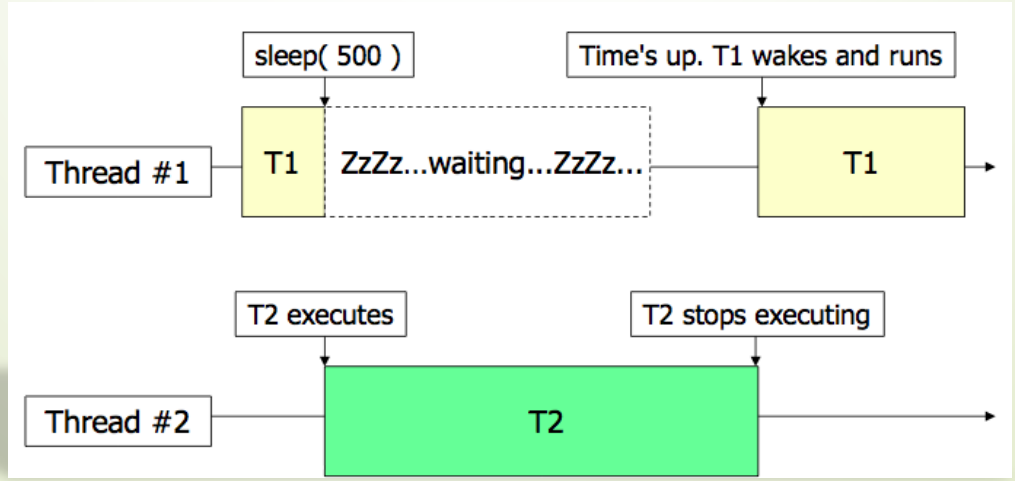
Underlined methods are static

4. HILOS JAVA

Control del Hilo: *sleep*

- Pausa la ejecución del hilo actual por un period de tiempo determinado(*ms*):

```
try {
     Thread.sleep(5000);
}
catch (InterruptedException ex) {
}
```



4. HILOS JAVA
Control del Hilo: *sleep*

Ejercicio 3:

- Modifica el Ejercicio 1 para que el hilo *PrintChar* ***duerma n*** segundos en cada paso del bucle.
  - El valor de la ***n*** ha de ser 0 en el caso que no hayan **argumentos** en la línea de comandos. (Modifica el proyecto para que ejecute con parámetros de la línea de comandos)
- La salida debe ser algo similar a esto:

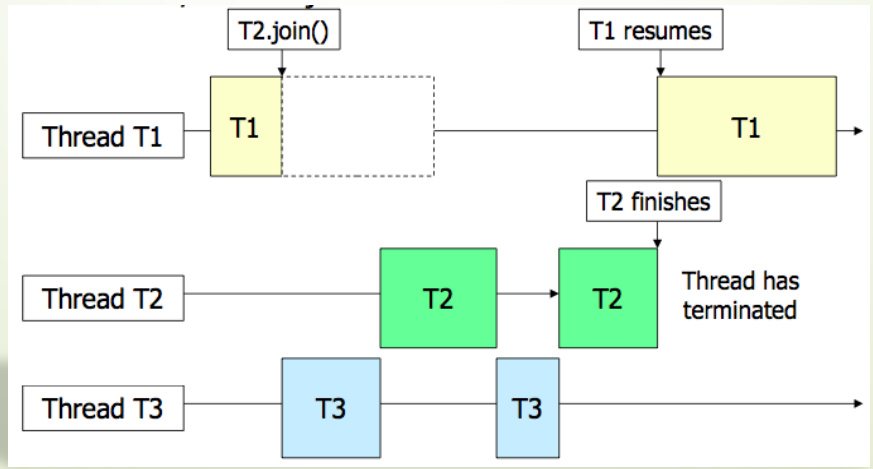
```
run:
If there are no cmd line args, PrintChar threads won't sleep
Sleeping PrintChar to 1000 milliseconds.
TEST SLEEP BEGINNING
 A T 0 1 2 3 4 5 6 7 8 9 10
MAIN FINISHED

 A T A T A T A T A T A T A T A T A T A TBUILD SUCCESSFUL (total time: 11 seconds)
```

4. HILOS JAVA

- Hace que un hilo espere a que finalice el otro:

Control del Hilo: *join*

```
// waits until waitOnThread finishes
try {
    waitOnThread.join();
} catch (InterruptedException ex) {
}
```



4. HILOS JAVA
Control del Hilo: *join*

Ejercicio 4:

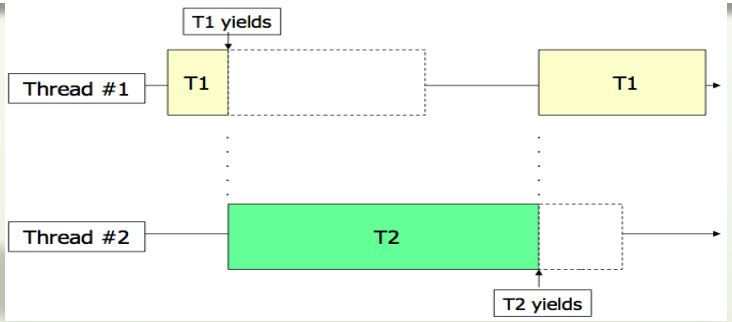
- Modifica el Ejercicio 2 para que el hilo PrintNumber espere a que el Segundo hilo finalice cuando la **mitad** de los números se hayan impreso.
  - *Consejo: Incluye un nuevo parámetro en el Constructor de la clase PrintNumber en el que se pase la referencia al segundo Thread.*
    - Hilo uno imprime la latra "A" 500 veces.
    - Hilo dos imprime la latra "T" 500 veces.
    - Hilo tres imprime enteros del 1 al 75.
- *Nota: Inclue los comentarios y los mensajes adecuados para ver que está pasando.*

4. HILOS JAVA

Control de Hilos: *yield*

- Hace que un hilo libere la CPU.
- Hace que la ejecución se entrelace más.

```
@Override
public void run() {
    for (int i = 0; i <= lastNumber; i++) {
        System.out.print(" " + i);

        // Let other threads run if told to
        if (yield) {
            Thread.yield();
        }
    }
}
```



*Añadir atributo boolean yield a la clase del hilo

4. HILOS JAVA

Control de Hilos: *yield*

Cuando un hilo llama a `Thread.yield()`, básicamente dice al planificador:

- "Ya he hecho suficiente trabajo por ahora; si hay otros hilos de igual o mayor prioridad, puedes darles la CPU".

Sin embargo:

- Si no hay otros hilos de igual o mayor prioridad, **el hilo actual puede continuar ejecutándose**.
- En algunos sistemas operativos, **`yield()` puede no hacer nada** dependiendo de cómo el planificador maneja los hilos.

4. HILOS JAVA

Diferencias: yield vs join

Aspecto	`yield`	`join`
Propósito	Permitir que otros hilos de igual o mayor prioridad se ejecuten.	Obligar al hilo actual a esperar a que otro hilo termine su ejecución.
Estado del hilo	Pasa a "listo" (runnable) y puede reanudar en cualquier momento.	Pasa a "espera" (waiting) hasta que el hilo al que llama `join` termine.
Uso	Optimización y mejora de multitarea.	Sincronización entre hilos, asegurando el orden de ejecución.
Bloqueo	No bloquea el hilo.	Bloquea el hilo actual hasta que otro hilo termine.

4. JAVA THREADS
Thread control: *yield*

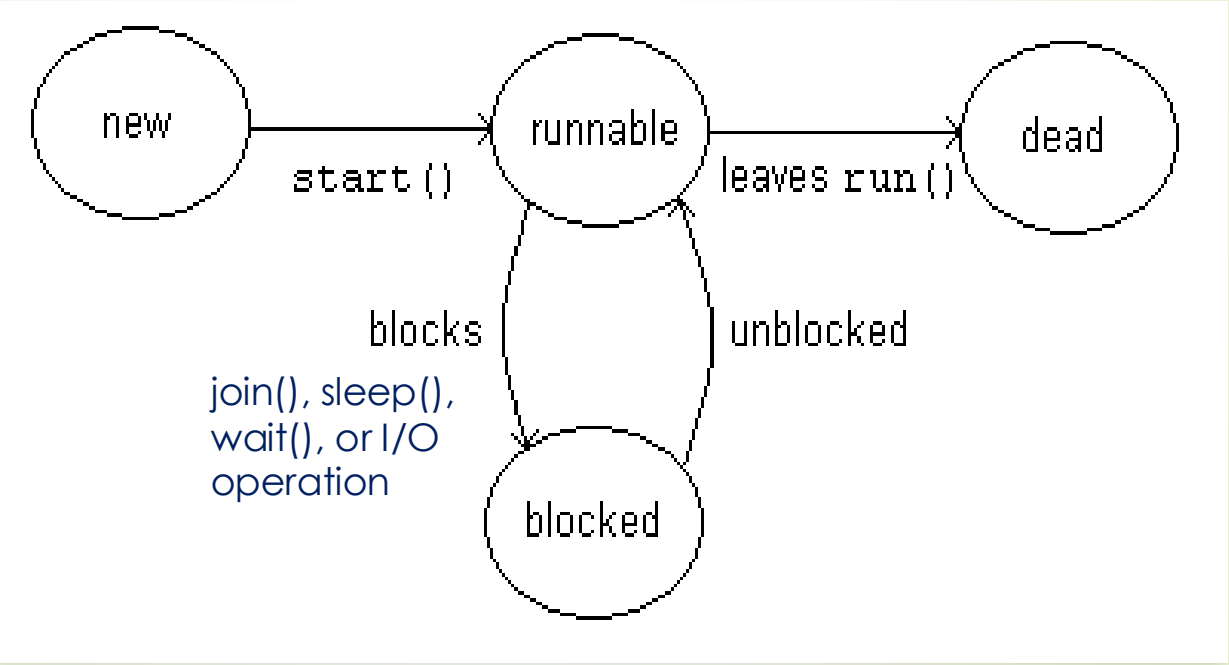
Ejercicio 5:

- Modifica el ejercicio #1 para hacer que los hilos *PrintChar* and *PrintNumber* hagan ***yield*** solo si no se introducen argumentos en la línea de comandos.
  - *Tip: Añade un nuevo parámetro al constructor de la clase del hilo en el que se indique si está habilitada o no.*
    - *Thread uno imprime el caracter 'A' 10 veces.*
    - *Thread dos imprime el caracter 'T' 10 veces.*
    - *Thread tres imprime enteros del 1 al 10.*
- *Note: No olvides incluir commandos para mostrar que está ocurriendo.*

```
run:
TEST YIELD BEGINNING. YIELD true
 A A A A T 1 2 3 A 4 A 5 A 6 T A 7 T A 8 T A 9 T A 10 T T T T
MAIN FINISHED
 T TBUILD SUCCESSFUL (total time: 0 seconds)
```

5. CICLO DE VIDA DEL HILO

- El ciclo de vida de los hilos en Java es muy similar al ciclo de vida de los procesos que se ejecutan en un sistema operativo.



https://docs.oracle.com/javase/8/docs/api/index.html?java/lang/Thread.State.html

5. CICLO DE VIDA DEL HILO

```java
// Clase que implementa Runnable
class MiRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("Estado dentro del run: " + Thread.currentThread().getState());
        try {
            Thread.sleep(2000); // El hilo pasa a TIMED_WAITING
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public class CicloDeVidaHilo {
    public static void main(String[] args) {
        // Crear una instancia de MiRunnable
        MiRunnable miRunnable = new MiRunnable();

        // Crear un hilo usando la instancia de MiRunnable
        Thread thread = new Thread(miRunnable);

        System.out.println("Estado antes de start(): " + thread.getState()); // NEW

        thread.start();
        System.out.println("Estado después de start(): " + thread.getState()); // RUNNABLE

        try {
            Thread.sleep(500); // Esperar a que el hilo cambie de estado
            System.out.println("Estado durante la ejecución: " + thread.getState()); // RUNNABLE o TIMED_WAITING
            thread.join();
            System.out.println("Estado después de join(): " + thread.getState()); // TERMINATED
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

5. CICLO DE VIDA DEL HILO

- *Al mismo tiempo, el hilo sólo puede estar en **un** estado.*
  - Los estados son **estados de JVM**, ya que no están vinculados a los estados de procesos del sistema operativo.
- *El planificador de hilos JVM es responsable de administrar los hilos java.*
  - La especificación de la API de Java dice: "Cada hilo tiene una prioridad. Los hilos con más prioridad se ejecutan preferentemente a los hilos con menos prioridad"
  - Pero tened en cuenta que el resultado puede diferir entre los sistemas operativos y las JVM.

5. CICLO DE VIDA DEL HILO

Thread vs. Runnable

```
public class Program {
  public static void main (String[] args) {
    Runner r = new Runner();
    Thread t1 = new Thread(r, "Thread A");
    Thread t2 = new Thread(r, "Thread B");
    Thread s1 = new Strider("Thread C");
    Thread s2 = new Strider("Thread D");
    t1.start();
    t2.start();
    s1.start();
    s2.start();
  }
}
class Strider extends Thread {
  private int counter;
  Strider(String name)    {
    super(name);
  }
  public void run()   {
    try {
      for (int i = 0; i != 2; i++) {
        System.out.println(Thread.currentThread().getName() + ": "
            + counter++);
        Thread.sleep(1000);
      }
    }
    catch(InterruptedException e)     {
       e.printStackTrace();
    }
  }
}
```

Muchos hilos pueden compartir la misma instancia de objeto Runnable

No se puede extender otra clase

```
class Runner implements Runnable {
  private int counter;
  public void run() {
    try {
      for (int i = 0; i != 2; i++) {
        System.out.println(Thread.currentThread().getName() + ": "
            + counter++);
        Thread.sleep(1000);
      }
    }
    catch(InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

5. CICLO DE VIDA DEL HILO
Thread ***vs. Runnable***

- **Heredar de la clase Thread**: Cuando el programa necesita un control total sobre el ciclo de vida del hilo (sobreescribiendo los métodos de clase Thread deseados star(), join(), etc) ya que con Runnable solo puedes sobreescribir run().

- **Implementar la Interfaz Runnable**: Cuando el programa necesita heredar de otra clase base.

- Si nada de lo anterior está presente, cualquiera de los dos se puede usar. Aunque **usar `Runnable` es generalmente una mejor práctica** por su flexibilidad.

5. CICLO DE VIDA DEL HILO

Interrupciones

- Una **interrupción** es una señal a un hilo que indica detener lo que está haciendo y hacer otra cosa.
- Un subproceso envía una interrupción invocando el método **interrupt()** en el objeto de subproceso deseado.

```
try {
    Thread.sleep(4000);
} catch (InterruptedException e) {
    // We've been interrupted: no more messages.
    return;
}
```

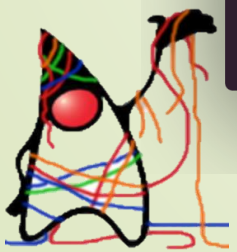
- Métodos como **sleep()** y **join()** lanzan *InterruptedException*.
  - El desarrollador debe decidir qué hacer cuando se detecta la InterruptedException: es muy común terminar el hilo.

- Además, el hilo puede comprobar la llamada de interrupción al método estático Thread.interrupted() o isInterrupted())

5. CICLO DE VIDA DEL HILO

Interrupciones

```java
class TareaInterrumpible implements Runnable {
    @Override
    public void run() {
        while (!Thread.interrupted()) { // Verifica y limpia el estado de interrupción
            System.out.println("Tarea en ejecución...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                System.out.println("Interrumpido durante sleep. Deteniendo tarea.");
                break; // Salir del bucle
            }
        }
        System.out.println("Tarea finalizada.");
    }
}

public class EjemploThreadInterrupted {
    public static void main(String[] args) throws InterruptedException {
        Thread hilo = new Thread(new TareaInterrumpible());
        hilo.start();

        Thread.sleep(3000); // Espera 3 segundos antes de interrumpir
        hilo.interrupt(); // Envía la señal de interrupción
    }
}
```



5. CICLO DE VIDA DEL HILO
Interrupciones

Ejercicio 6: Probando las Interrupciones

- Modifica el Ejercicio #1:
  - *Clase PrintNumber*: Incluye un sleep(10) para cada paso.
  - Desde el método *main()*:
    - Interrumpe el hilo que imprime "A"
    - Interrumpe el hilo que imprime numeros.
    - Si el hilo no está vivo, imprime mensajes del tipo: "*No puede interrumpirse el hilo Thread-XXX porque no está vivo, ha finalizado, está muerto*".
  - Cuando un thread se interrumpe, imprime el mensaje: "Thread-0 interrumpido en #0" y continua el trabajo de imprersión.
  - Ejecuta el programa y revisa los resultados.

5. CICLO DE VIDA DEL HILO

Prioridad de los Hilos

- En Java, todas las instancias de la clase Thread comparen la misma prioridad(***NORM_PRIORITY***)
  - Por lo que se planifican para ejecutarse de un modo (FIFO like)
- Pero es posible contrar la prioridad del thread utilizando el método ***setPriority(value)***. Valores:

  MIN_PRIORITY *(=1)* ***NORM_PRIORITY*** *(=5)* ***MAX_PRIORITY*** *(=10)*

- **Atención**:
  - La prioridad de subprocesos es solo una sugerencia para el pplanificador del sistema operativo y depende del sistema operativo subyacente.
  -
  - Los métodos: ***stop()***, ***resume()*** y ***suspend()*** se han marcado como deprecated desde Java 1.4 ya que **no son seguros**. Se recomienda utilizar interrupciones.

5. CICLO DE VIDA DEL HILO

Prioridad de los Hilos

```java
public class PrioridadHilos {
    public static void main(String[] args) {
        // Crear instancias de Runnable
        MiTarea tarea1 = new MiTarea("Hilo-1");
        MiTarea tarea2 = new MiTarea("Hilo-2");
        MiTarea tarea3 = new MiTarea("Hilo-3");

        // Crear hilos con las tareas
        Thread hilo1 = new Thread(tarea1);
        Thread hilo2 = new Thread(tarea2);
        Thread hilo3 = new Thread(tarea3);

        // Establecer prioridades
        hilo1.setPriority(Thread.MIN_PRIORITY); // Prioridad mínima (1)
        hilo2.setPriority(Thread.NORM_PRIORITY); // Prioridad normal (5)
        hilo3.setPriority(Thread.MAX_PRIORITY); // Prioridad máxima (10)

        // Iniciar los hilos
        hilo1.start();
        hilo2.start();
        hilo3.start();
    }
}
```