Taller de Lenguajes II

Tema de hoy: Concurrencia en JAVA

¿Qué es un *thread*?

Creación y gerenciamiento de *threads*

- La clase *thread* y el método run()
- Los métodos sleep(), join() y yield()
- La interface **Runnable**

El ciclo de vida de un thread

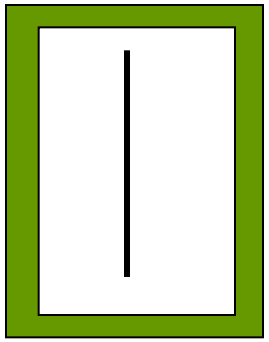
Prioridades en threads

Ejemplos

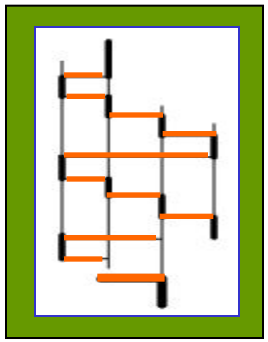
La clase TimerTask

Concurrencia
Threads

- Un **thread** es un flujo de control secuencial dentro de un proceso. A los threads también se los conoce como **procesos livianos** (requiere menos recursos crear un thread nuevo que un proceso nuevo) o **contextos de ejecución**.
- Un **thread** es similar a un programa secuencial: tiene un comienzo, una secuencia de ejecución, un final y en un instante de tiempo dado hay un único punto de ejecución. Sin embargo, **un thread no es un programa**. Un **thread** se ejecuta dentro de un programa.
- Los novedoso es el uso de múltiples **threads** dentro de un mismo programa, ejecutándose simultáneamente y realizando tareas diferentes:



Programa *singleThread*



Programa *multiThread*

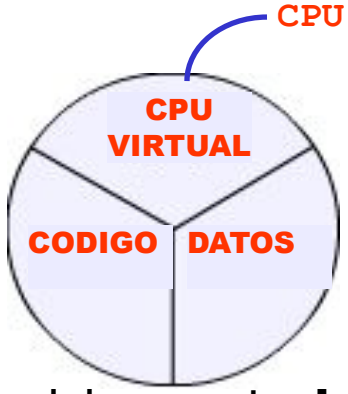
En un programa *multithread*, más de un thread se ejecuta en forma concurrente. El control de ejecución es transferido entre los diferentes threads, cada uno de los cuáles es responsable de distintas tareas.

- En el modelo de multithreading la CPU asigna a cada thread un tiempo para que se ejecute; cada thread "tiene la percepción" que dispone de la CPU constantemente, sin embargo el tiempo de CPU está dividido entre todos los threads.

Threads

- Un **thread** se ejecuta dentro del contexto de un programa (o proceso) y **comparte** los recursos asignados al **programa**. Asimismo los **threads** toman algunos **recursos** del ambiente de ejecución del programa como **propios**: tienen su propia **pila de ejecución**, **contador de programa**, **código** y **datos**. Como un **thread** solamente se ejecuta dentro de un contexto, a un thread también se lo llama **contexto de ejecución**.
- La plataforma JAVA soporta programas **multhreading** a través del lenguaje, de librerías y del sistema de ejecución. A partir de la versión 5.0, la plataforma JAVA incluye librerías de concurrencia de más alto nivel.

Múltiples-threads comparten el mismo código cuando se ejecutan a partir de instancias de la misma clase.



Múltiples-threads comparten datos cuando acceden a objetos comunes (podría ser a partir de códigos diferentes).

- La **clase Thread** forma parte del paquete **java.lang** y provee una implementación de threads independiente del sistema de ejecución. Hay **dos estrategias** para usar objetos **Threads:**
  - **Directamente controlar la creación y el gerenciamiento** instanciando un thread cada vez que la aplicación requiere iniciar una tarea concurrente.
  - **Abstraer el gerenciamiento** de threads pasando la tarea concurrente a un **ejecutor** para que la administre y ejecute.

Creación y Gerenciamiento de Threads

- La **clase Thread** provee el **comportamiento genérico** de los threads JAVA: arranque, ejecución, interrupción, asignación de prioridades, etc.
- El método **run()** es el más importante de la clase **Thread**: implementa la funcionalidad del thread, es el código que se ejecutará "simultáneamente" con otros threads del programa. El método **run()** predeterminado provisto por la clase Thread no hace nada.
- La **plataforma JAVA** es **multithread**: siempre hay un thread ejecutándose junto con las aplicaciones de los usuarios, por ejemplo el **garbage collector** es un thread que se ejecuta en background; las GUI´s **recolectan los eventos** generados por el usuario en threads separados, etc.
- Una **aplicación JAVA** siempre se ejecuta en un **thread** llamado **main thread.** Este **thread** ejecuta secuencialmente las sentencias del cuerpo del método main() de la clase. En otros programas JAVA como **servlets** que no tienen método main(), la ejecución del **main thread** comienza con el método main() de su contenedor, que es el encargado de invocar los métodos del ciclo de vida de dichas componentes.

El método run() de la clase Thread

Es un método de la clase **Thread**. Es el lugar donde el **thread** comienza su ejecución.

```
public class SimpleThread extends Thread {
     private int contador=10;
     public SimpleThread(int nro) {
      super("" +nro);
     }
     public String toString() {
      return "#" + getName()+ ":" +contador--;
     }
     public void run()  {
       for (int i = contador; i > 0; i --)
           System.out.println(this);

       System.out.println("Termino!" + this );
     }
}
```

Invoca al constructor de Thread con un argumento String que es el nombre del thread

Un thread termina cuando finaliza el método run()

```
public class TestCincoThread {
 private static int nroThread=0;
 public static void main(String[] args) {
  for (int i=0; i<5;i++)
   new SimpleThread(++ nroThread).start();
 }
}
```

Inicializa el objeto Thread e invoca al método run(). El thread pasa a estado "vivo"

Se recupera el nombre del thread con el método getName() de la clase Thread

Cuando el método **start()** retorna, hay 2 threads ejecutándose en paralelo: el thread que invocó al **start()**, en nuestro caso el main thread y el thread que está ejecutando el método **run()**.

Tenemos 5 tareas concurrentes, cada una de ellas imprime en pantalla 10 veces su nombre. Además tenemos el **main thread.**

¿Cuál es la salida del programa TestCincoThread?

La salida de una ejecución del programa es diferente a la salida de otra ejecución del mismo programa, dado que el mecanismo de ***scheduling*** de threads no es determinístico.

Métodos de la clase Thread

Sleep

Suspende temporalmente la ejecución del **thread** que se está ejecutando. Afecta solamente al **thread** que ejecuta el **sleep()**, no es posible decirle a otro thread que "se duerma". Es un método de clase. El tiempo de suspensión se expresa en milisegundos.

```
import java.util.concurrent.TimeUnit;
public class SimpleThread extends Thread {
 private int contador=10;
 public SimpleThread(int nro) {
   super("" +nro);
 }
 public String toString() {
   return "#" + getName()+ ":" +contador--;
 }
 public void run()  {
   for (int i = contador; i > 0; i --){
    System.out.println(this);
    try {
      //Antes de JSE 5:
     // sleep(100);
     // Estilo JSE 5
        TimeUnit.MILLISECONDS.sleep(100);
     } catch (InterruptedException e){
        throw new RuntimeException();
     }
   }
  System.out.println("Termino!" + this );
 }
}
```

- El método **sleep()** está encerrado en un bloque **try** dado que podría ser interrumpido antes que el tiempo se agote (por ej. se invoca al método **interrupt()** sobre el objeto thread).
- **sleep()** es un método sobrecargado que permite especificar el tiempo de espera en milisegundos y en nanosegundos. En la mayoría de las implementaciones de la JVM, este tiempo se redondea a la cantidad de milisegundos más próxima (en general un múltiplo de 20 milseg o 50 milseg).
- Los threads se ejecutan en cualquier orden. El método **sleep()** no permite controlar el orden de ejecución de los threads; suspende la ejecución del thread por un tiempo dado.
- En nuestro ejemplo, la única garantía que se tiene es que el thread suspenderá su ejecución por al menos 100 milisegundos, pero podría tomar más tiempo antes de retomar la ejecución.

Métodos de la clase Thread

Join

El método **join()** permite que un **thread** espere a que otro termine de ejecutarse.
El **objetivo** del método **join()** es esperar por un evento específico: la terminación de un **thread**. El **thread** que invoca al **join()** sobre otro **thread** se bloquea hasta que dicho **thread** termine su método **run()**. Una vez que dicho **thread** completa su **run()**, el método **join()** retorna inmediatamente.

```
public class SimpleThreadTest2 {
    public static void main(String args[]) {
        SimpleThread t=new SimpleThread(1);
        t.start();
        while (t.isAlive()) {
          System.out.println("esperando...");
          try {
              t.join();
           } catch(InterruptedException e) {
                 System.out.println(t.getName() + "join interrupido");
             }
          System.out.println(t.getName() + " join completado");
        }
    }
}
```

- En este caso el **main thread** se bloquea en espera que el **thread t** termine de ejecutarse.
- El método **join()** es sobrecargado, permite especificar el tiempo de espera. Sin embargo, de la misma manera que el **sleep()** no se puede asumir que este tiempo sea preciso. Como el método **sleep()**, el **join()** podría finalizar con la interrupción **InterruptedException.**

Métodos de la clase Thread

Yield

Permite indicarle al mecanismo de *scheduling* (planificación) que el thread ya hizo suficiente trabajo y que podría cederle tiempo de CPU a otro thread. Su efecto es dependiente del SO sobre el que se ejecuta la JVM. Permite implementar **multithreading cooperativo.**

```
public class SimpleThread extends Thread {
  private int contador=10;
  public SimpleThread(int nro) {
    super("" +nro);
  }
  public String toString() {
    return "#" + getName()+ ":" +contador--;
  }
  public void run()  {
    for (int i = contador; i > 0; i --){
     System.out.println(this);
     Thread.yield();
    }
   System.out.println("Termino!" + this );
  }
}
```

SimpleThread de esta manera realizaría un procesamiento mejor distribuido entre varias tareas SimpleThread.

La interface Runnable

- Es posible escribir **threads** implementando la interface **Runnable**.
- La interface **Runnable** solamente especifica que se debe implementar el método **run()**.

```
package java.lang;

public interface Runnable {
  public void run();
}
```

```
package java.lang;

public class Thread implements Runnable {
//Código de la clase Thread
}
```

- Si una clase implementa la **interface Runnable** simplemente significa que tiene un método **run()**, pero NO tiene **ninguna habilidad de threading**. Para generar un thread a partir de un objeto **Runnable** es necesario crear un **objeto Thread** y pasarle el objeto **Runnable** al constructor. Luego, se invoca al método **start()** sobre el **thread** creado, NO sobre el objeto **Runnable**.

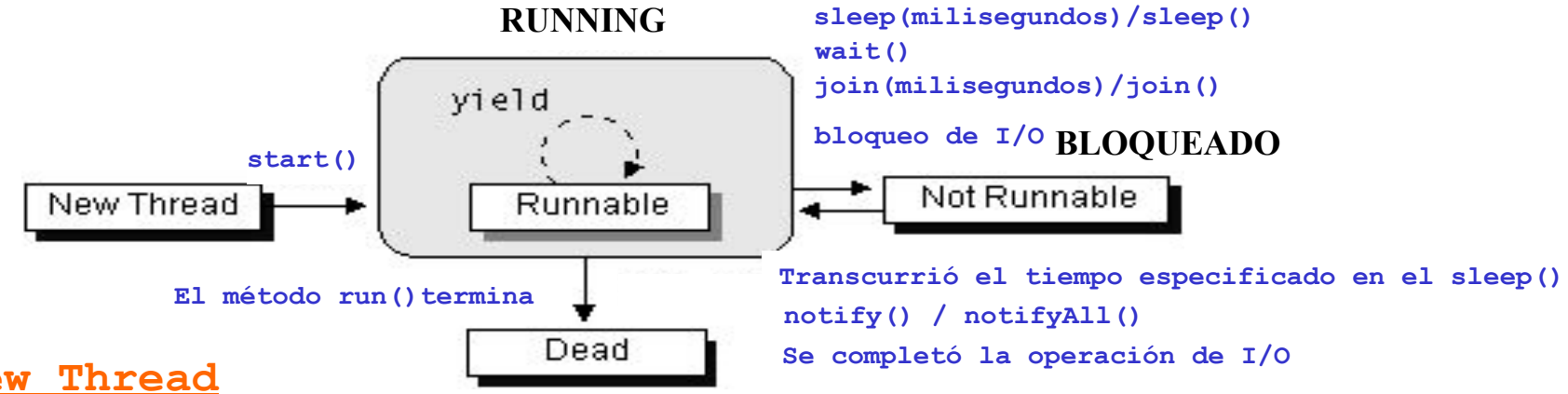
Identifica el thread ejecutándose

```
public class SimpleTask implements Runnable{
 private int contador=10;
 public String toString() {
  return "#" + Thread.currentThread().getName()+ ":" +contador--;
 }
 public void run(){
  for (int i = contador; i > 0; i --)
      System.out.println(this);
  System.out.println("Termino!" + this );
 }
}
```

Se obtiene una referencia al thread en ejecución

```
public class TestCincoThread {
 private static int nroThread=0;
 public static void main(String[] args) {
  for (int i=0; i<5;i++)
     new Thread(new SimpleTask(),""+i).start();
  }
}
```

Ciclo de vida de un Thread



Estado New Thread

Inmediatamente después que un thread es creado pasa a estado **New Thread**, pero aún no ha sido arrancado, por lo tanto no puede ejecutarse. Se debe invocar al método **start()**

Estado Running/Runnable

Después de ejecutar el método **start()**, el thread pasa al estado **Runnable**. Un thread arrancado con **start()** podría o no comenzar a ejecutarse (Running). No hay nada que evite que el thread se ejecute. La JVM implementa una estrategia (scheduling) que permite compartir la CPU entre todos los threads en estado Runnable.

Estado Not Runnable o Blocked

Un thread pasa a estado **Not Runnable o Bloqueado** cuando ocurren algunos de los siguientes eventos: se invoca al método **sleep()**, al **wait()**, **join()** o **el thread está bloqueado en espera de una operación de E/S**, **el thread invoca a un método synchronized sobre un objeto y el lock del objeto no está disponible**. Cada entrada al estado **Not Runnable** tiene una forma de salida correspondiente. Cuando un thread está en estado bloqueado, el *scheduler* lo saltea y no le da ningún *slice* de CPU para ejecutarse.

Estado Dead

Los **threads** definen su finalización implementando un **run()** que termine naturalmente.

La interface Runnable
Algunas conclusiones

- Una ventaja de implementar la **interface Runnable** es que todo el **código pertenece a la misma clase** y de esta manera es simple combinar la clase base con otras interfaces. Es posible acceder a cualquier objeto y métodos de la clase evitándose mantener referencias en objeto separados.
- La **interface Runnable** permite **separar la implementación** de la **tarea** del **thread** que la ejecuta. Es más **flexible**.
- También es importante considerar que JAVA provee un conjunto de clases que gestionan **multithreading** (por ej. pool de threads). En estos casos las tareas deben ser objetos que implementan la **interface Runnable**.

Las clases Timer y TimerTask

- Las clases **Timer** y **TimerTask** son útiles para lanzar tareas cada cierto intervalo de tiempo. Ambas residen en el paquete **java.util**. Ejemplos: animaciones que necesitan actualizar la pantalla cada cierto tiempo; enviar información de estado a un servidor para no desconectarnos o lanzar una tarea a una hora determinada.

- La clase **Timer**, es un **temporizador**, se usa para **planificar la ejecución de una tarea una vez** o **repetidamente cada cierto intervalo de tiempo**. Cada objeto **Timer usa un thread** en el que se ejecutan todas las tareas del temporizador.

- La clase **TimerTask** es una clase abstracta que implementa la interface **Runnable** y que se usa para implementar la tarea que será puesta en ejecución por el **Timer**.

Ejemplo: TIC-TOC

```
package taller2;
import java.util.Timer;
import java.util.TimerTask;
public class TicToC {
  public static void main(String[] args) {
        Timer timer;
        timer = new Timer();
        TimerTask task = new MiTimerTask();
        // Empezamos dentro de 10ms y
        // luego lanzamos la tarea cada 1000ms
        timer.schedule(task, 10, 1000);
  }
}
```

El temporizador programará la ejecución de la tarea cada 1 segundo.

Podríamos terminar el temporizador invocando a **timer.cancel();**

```
package taller2;
import java.util.TimerTask;
public class MiTimerTask extends TimerTask {
int tic = 0;
@Override
public void run() {
   if (tic % 2 == 0)
        System.out.println("TIC");
    else
        System.out.println("TOC");
   tic++;
   if (tic % 10 == 0)
        System.out.println("Llegamos a 10!!");
}
}
```

El **método run()** del **objeto TimerTask** define la tarea que ejecutará el temporizador cada un intervalo de tiempo dado, en este caso cada 1000 ms. (1 seg)