Programación Interactiva
Hilos

Escuela de Ingeniería de Sistemas y Computación
Facultad de Ingeniería
Universidad del Valle

THREADS: PROGRAMAS MULTITAREA

- Los procesadores y los Sistemas Operativos modernos permiten la ***multitarea***
- En la realidad, un ordenador con una sola CPU no puede realizar dos actividades a la vez.
- Sin embargo los Sistemas Operativos actuales son capaces de ejecutar varios programas "simultáneamente" aunque sólo se disponga de una CPU

THREADS: PROGRAMAS MULTITAREA

- En ordenadores con dos o más procesadores la multitarea es real, ya que cada procesador puede ejecutar un ***hilo*** o ***thread*** diferente.

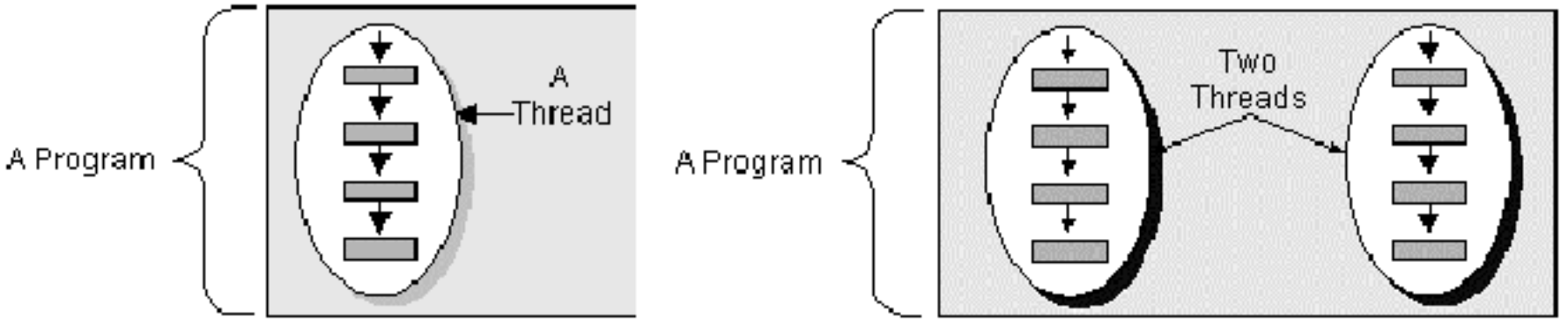


Figura 6.1. Programa con 1 y con 2 threads o hilos.

THREADS: PROGRAMAS MULTITAREA

- 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.

THREADS: PROGRAMAS MULTITAREA

- run()
- wait() –Espera evento
- stop() –Deadlock
- start() –Llama a run
- yield() –Retorna control
- sleep(long ms) -Tiempo

java.lang.Thread
java.lang.Runnable

- Daemon - Background
- no daemon – Por Defecto

CREACIÓN DE THREADS

- En ***Java*** hay dos formas de crear nuevos ***threads***.
- Crear una nueva clase que herede de la clase ***java.lang.Thread*** y sobrecargar el método ***run()***.
- Ó, declarar una clase que implemente la interface ***java.lang.Runnable***, la cual declarará el método ***run()***; posteriormente se crea un objeto de tipo ***Thread*** pasándole como argumento al constructor el objeto creado de la nueva clase

Clase Thread

```
public class SimpleThread extends Thread {
        public SimpleThread (String str) {          // constructor
        super(str);
    }
public void run() {             // redefinición del método run()
        for(int i=0;i<10;i++)
            System.out.println("Este es el thread : " + getName());
    }
}
```

```
SimpleThread miThread = new SimpleThread(“Hilo de prueba”);
miThread.start();
```

Interface Runnable

```
public class SimpleRunnable implements Runnable {
    String nameThread;              // se crea una variable
    // constructor
    public SimpleRunnable (String str) {
        nameThread = str;
    }
    // definición del método run()
    public void run() {
        for(int i=0;i<10;i++)
                System.out.println("Este es el thread: " + nameThread);
    }
}
```

Interface Runnable

```
SimpleRunnable p = new SimpleRunnable("Hilo de prueba");
// se crea un objeto de la clase Thread pasándolo el objeto
    Runnable como argumento
Thread miThread = new Thread(p);
// se arranca el objeto de la clase Thread
miThread.start();
```

Interface Runnable

```
class ThreadRunnable extends Applet implements Runnable {
    private Thread runner=null;
    // se redefine el método start() de Applet
    public void start() {
        if (runner == null) {
                runner = new Thread(this);
                runner.start(); // se llama al método start() de Thread
        }
    }
    public void stop(){       // se redefine el método stop() de Applet
        runner = null;        // se libera el objeto runner
    }
}
```

Interface Runnable

La elección de una u otra forma -derivar de ***Thread*** o implementar ***Runnable***- depende del tipo de clase que se vaya a crear.

Así, si la clase a utilizar ya hereda de otra clase (por ejemplo un ***applet***, que siempre hereda de ***Applet***), no quedará más remedio que implementar ***Runnable***, aunque normalmente es más sencillo heredar de ***Thread***.

Ciclo de Vida de un Thread

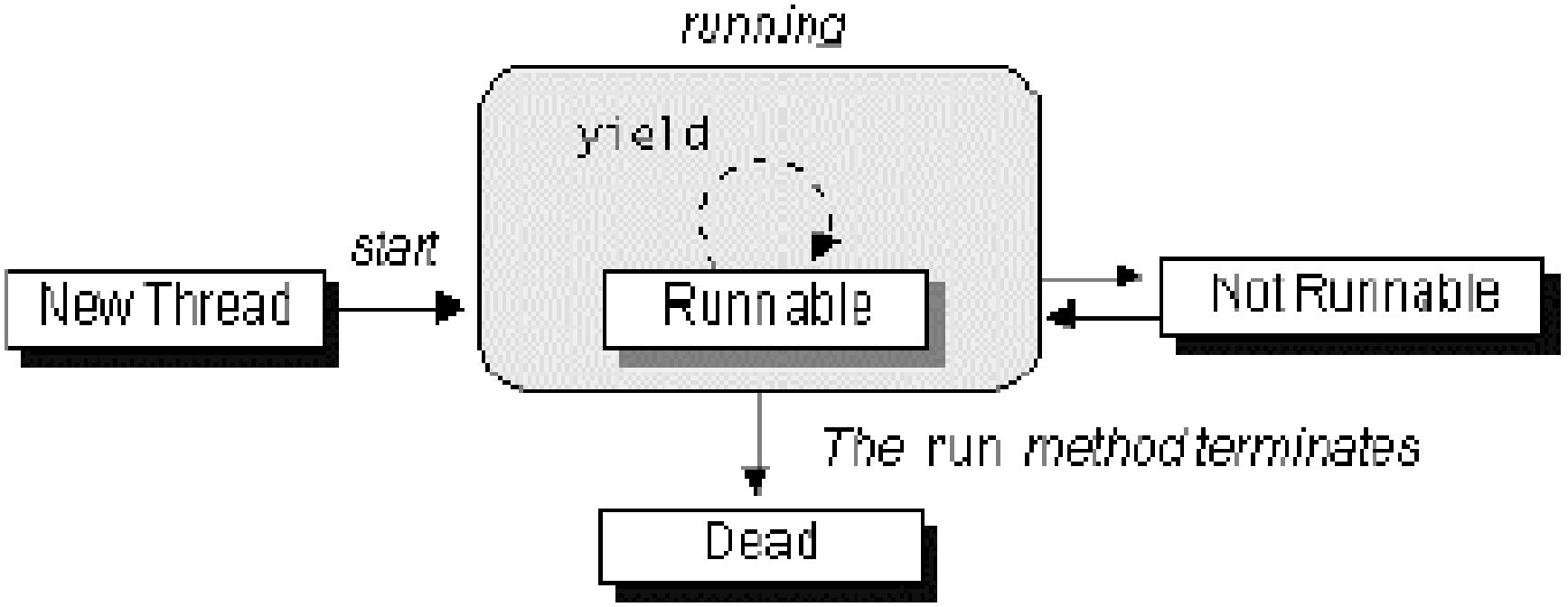


Figura 6.2. Ciclo de vida de un Thread.

Ejecución de un nuevo Thread

- La creación de un nuevo ***thread*** no implica necesariamente que se empiece a ejecutar algo.

- Hace falta iniciarlo con el método ***start()***, ya que de otro modo, cuando se intenta ejecutar cualquier método del ***thread*** -distinto del método ***start()***- se obtiene en tiempo de ejecución el error ***IllegalThreadStateException***.

Bloque Temporal de un Thread

- Ejecutando el método ***sleep()*** de la clase ***Thread***. (desde el método run())
- Ejecutando el método ***wait()*** heredado de la clase ***Object.***
- Operaciones de Entrada/Salida o Input/Output (E/S ó I/O).
- Cuando el ***thread*** está tratando de llamar a un método ***synchronized*** de un objeto, y dicho objeto está bloqueado por otro ***thread***
- **Recuperación : notify, notifyAll**

sleep()

```
public static void sleep(long millis) throws InterruptedException
public static void sleep(long millis, int nanosecons) throws
    InterruptedException
System.out.println ("Contador de segundos");
int count=0;
public void run () {
    try {
        sleep(1000);
        System.out.println(count++);
    } catch (InterruptedException e){}
}
```

Detener un Thread

La forma preferible de detener temporalmente un ***thread*** es la utilización conjunta de los métodos ***wait()*** y ***notifyAll()***.

La principal ventaja del método ***wait()*** frente a los métodos anteriormente descritos es que libera el bloqueo del objeto. por lo que el resto de threads que se encuentran esperando para actuar sobre dicho objeto pueden llamar a sus métodos.

wait()

Hay dos formas de llamar a ***wait()***:

public final void wait() throws InterruptedException

Sin argumentos, en cuyo caso el ***thread*** permanece parado hasta que sea reinicializado explícitamente mediante los métodos ***notify()*** o ***notifyAll()***.

wait()

public final void wait(long timeout) throws InterruptedException

public final void wait(long timeout, int nanos) throws InterruptedException

Indicando el tiempo máximo que debe estar parado (en *milisegundos* y con la opción de indicar también *nanosegundos*), de forma análoga a ***sleep()***. A diferencia del método ***sleep()***, que simplemente detiene el ***thread*** el tiempo indicado, el método ***wait()*** establece el tiempo máximo que debe estar parado.

Finalización de un Thread

```
public class MyApplet extends Applet implements Runnable {
     private Thread AppletThread; // se crea una referencia tipo Thread
    public void start() {                  // método start() del Applet
        if(AppletThread == null){ // si no tiene un obj Thread asociado
                AppletThread = new Thread(this, "El propio Applet");
                AppletThread.start(); //arranca el thread y llama a run()
        }
    }
    public void stop() {                // método stop() del Applet
        AppletThread = null;        // iguala la referencia a null
    }
```

Finalización de un Thread

```
public class MyApplet extends Applet implements Runnable {
    ....
    public void run() {  // método run() por implementar
    Runnable
        Thread myThread = Thread.currentThread();
        while (myThread == AppletThread) {
                //hasta !stop() de Thread ...
        }
    }
} // fin de la clase MyApplet
```

Sincronización

- La ***sincronización*** nace de la necesidad de evitar que dos o más ***threads*** traten de acceder a los mismos recursos al mismo tiempo. Así, por ejemplo, si un ***thread*** tratara de escribir en un fichero, y otro ***thread*** estuviera al mismo tiempo tratando de borrar dicho fichero, se produciría una situación no deseada.

  Otra situación en la que hay que sincronizar ***threads*** se produce cuando un ***thread*** debe esperar a que estén preparados los datos que le debe suministrar el otro ***thread***.

Sincronización

public **synchronized** void metodoSincronizado() {

...// accediendo por ejemplo a las variables de un objeto

...

}

- ¿Cuál es el problema con las variables?
- ¿Qué pasan si son publicas, privadas o protegidas?
- ¿Dos metodos sincronizados pueden acceder al mismo objeto? ¿Cómo lo soluciona Java?

Sincronización

- Metodos sincronizados
  - ***synchronized***
- Clases sincronizadas
  - Un metodo ***synchronized static***
- Sleep()
  - Dentro de Sincronizacion, esperar aunque no haya problema
- Wait()
  - Dentro de Sincronizacion, Desbloquea el objeto.

Sincronización get()

```
public synchronized int get() {
    while (available == false) {
        try {
        // Espera a que set() asigne el valor y lo comunique con
        // notify()
        wait();
        } catch (InterruptedException e) { }
    }
    available = false;
    notifyAll();                // notifica que el valor ha sido leído
    return contents;            // devuelve el valor
}
```

Sincronización set()

```
public synchronized void set(int value) {
    while (available == true) {
        try {
         // Espera a que get() lea el valor disponible antes de darle
         // otro
                wait();
        } catch (InterruptedException e) { }
    }
contents = value;   // ofrece un nuevo valor y lo declara disponible
available = true;    // notifica que el valor ha sido cambiado
notifyAll();
}
```

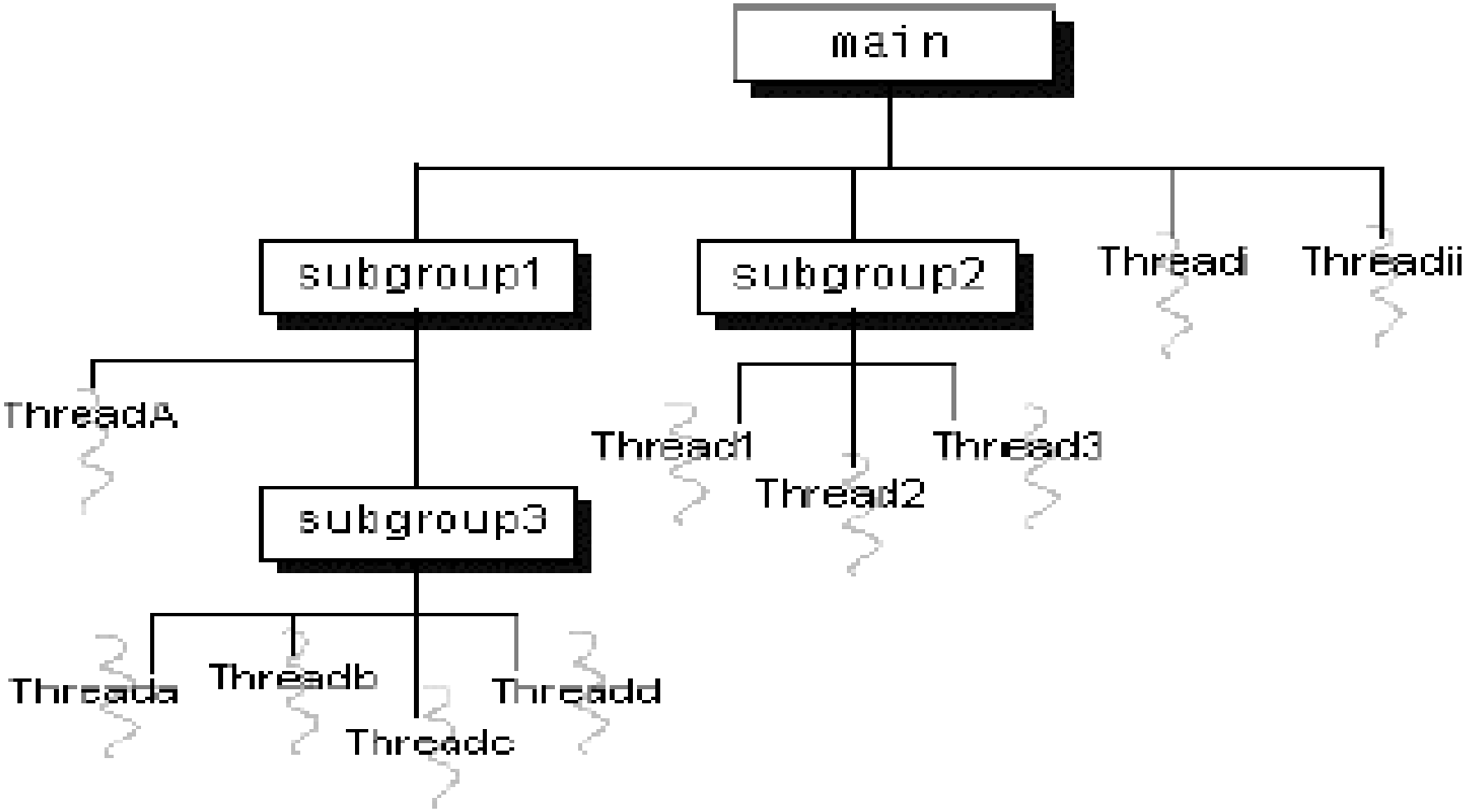
Sincronización (una parte)

```
    public void run() {
        while(true) {
                ...
                syncronized(this) { // sincroniza el propio thread
                ... // Código sincronizado
        }
        try {
                sleep(500); // Se detiene el thread durante 0.5 segundos
    // pero el objeto es accesible por otros threads al no estar sincronizado
        } catch(InterruptedException e) {}
    }
}
```

Prioridades

- Con el fin de conseguir una correcta ejecución de un programa se establecen ***prioridades*** en los ***threads***, de forma que se produzca un reparto más eficiente de los recursos disponibles.
- ***MAX_PRIORITY*** ***(10)*** ***getPriority()***
- ***MIN_PRIORITY*** ***(1)*** ***setPriority()***
- ***NORMAL_PRIORITY*** ***(5)***

Grupos de Threads



Grupos de Threads

- Todo hilo de ***Java*** debe formar parte de un grupo de hilos (***ThreadGroup***). Puede pertenecer al grupo por defecto o a uno explícitamente creado por el usuario.
- Los grupos de ***threads*** proporcionan una forma sencilla de manejar múltiples ***threads*** como un solo objeto. Así, por ejemplo es posible parar varios ***threads*** con una sola llamada al método correspondiente. Una vez que un ***thread*** ha sido asociado a un ***threadgroup***, no puede cambiar de grupo.

Grupos de Threads

- Cuando se arranca un programa, el sistema crea un ***ThreadGroup*** llamado ***main***. Si en la creación de un nuevo ***thread*** no se especifica a qué grupo pertenece, automáticamente pasa a pertenecer al ***threadgroup*** del ***thread*** desde el que ha sido creado (conocido como ***current thread group*** y ***current thread***, respectivamente).
- Si en dicho programa no se crea ningún ***ThreadGroup*** adicional, todos los ***threads*** creados pertenecerán al grupo ***main*** (en este grupo se encuentra el método ***main()***).

Grupos de Threads

- Para conseguir que un **thread** pertenezca a un grupo concreto, hay que indicarlo al crear el nuevo ***thread***, según uno de los siguientes constructores:

public Thread (ThreadGroup grupo, Runnable destino)
public Thread (ThreadGroup grupo, String nombre)
public Thread (ThreadGroup grupo, Runnable destino, String nombre)

Grupos de Threads

- A su vez, un ***ThreadGroup*** debe pertenecer a otro ***ThreadGroup***. Como ocurría en el caso anterior, si no se especifica ninguno, el nuevo grupo pertenecerá al ***ThreadGroup*** desde el que ha sido creado (por defecto al grupo ***main***). La clase ***ThreadGroup*** tiene dos posibles constructores:

```
ThreadGroup(ThreadGroup parent, String nombre);
ThreadGroup(String name);
```

Grupos de Threads

- ***getMaxPriority()***,
- ***setMaxPriority()***,
- ***getName()***,
- ***getParent()***,
- ***parentOf()***.

```
ThreadGroup miThreadGroup = new
   ThreadGroup("Mi Grupo de Threads");
Thread miThread = new Thread(miThreadGroup,
   ”un thread para mi grupo");
```