UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Programación concurrente usando Threads

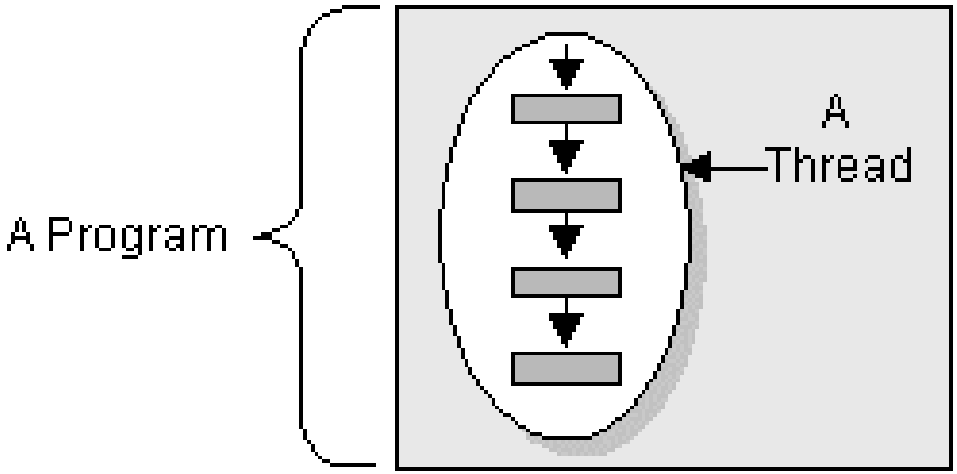
En los sistemas operativos modernos, la computadora puede tener un solo procesador, pero hacer muchas cosas de una vez.

A estos SO se les llama de multiprocesamiento porque existen muchas actividades separadas ejecutándose de una vez, todas compartiendo el mismo procesador.

Un programa secuencial tiene un comienzo, una secuencia de ejecución y un final. Un thread es similar al programa descripto. Tiene un comienzo, una secuencia y un final en cualquier momento mientras se encuentra en runtime. Sin embargo, un thread no es un programa; no puede correr por sí mismo. Corre dentro de un programa.

Programación concurrente usando Threads

La figura muestra la relación entre un thread y un programa

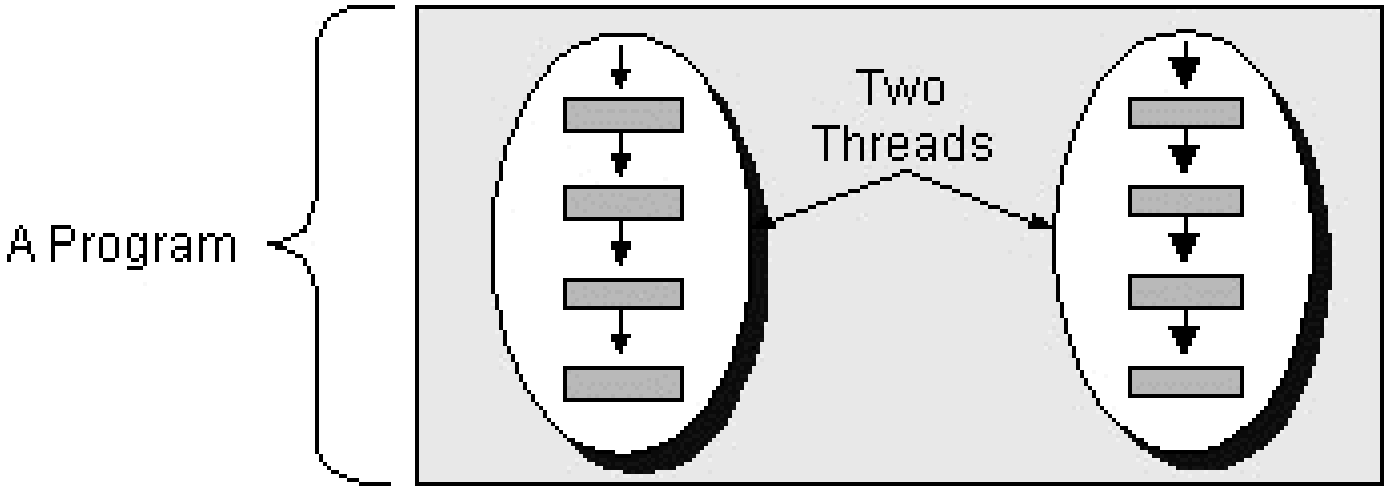


Definición: Un thread es un flujo único secuencial de control dentro de un programa

No habría nada novedoso en el concepto de thread único. Lo interesante es la posibilidad del uso de múltiples threads en un programa, corriendo al mismo tiempo y ejecutando diferentes tareas.

UNTDF
Universidad Nacional de Tierra del Fuego, Antártida e Islas del Atlántico Sur

Programación concurrente usando Threads



Hacia finales de los 80s surgieron los threads (hilos) también llamados “lightweight processes”. Un thread es similar a un proceso, en el sentido en que como tal, un thread es un camino de ejecución separado. Pero a diferencia de los procesos, no incorporan built-in protection entre un proceso y otro. Por otra parte un thread corre dentro del contexto de un programa y por lo tanto toma ventaja de los recursos asignados a ese programa y de su ambiente. Pero debe tener algunos recursos propios dentro del programa (su pila de ejecución y su Program counter, por ej.)

Programación concurrente usando Threads

La ventaja de los threads es que pueden iniciarse rápidamente.

La forma más fácil de hacer un nuevo thread es extender la clase Thread y crear una instancia de la subclase.
El constructor permite especificar:

- un nombre (en un String),
- un ThreadGroup: un grupo de threads que comparten información sobre sí mismos.
- un target: es una implementación de una interface Runnable.

Programación concurrente usando Threads

Los threads soportan una serie de métodos, entre los cuales se pueden citar:

currentThread(): retorna una referencia al thread que actualmente se está ejecutando.

getName(): retorna el nombre del Thread.

getPriority(): retorna la prioridad del Thread. Por defecto, todo thread tiene la prioridad del thread que lo comienza (start).

interrupt(): interrumpe el thread especificado.

run(): comienza a ejecutar el objeto Runnable target del Thread, si hay.

setName(): cambia el nombre del thread.

sleep(): causa que el thread actual entregue el procesador y temporalmente deje de ejecutar por un tiempo especificado.

start(): Llama a este thread para comenzar la ejecución. – JVM llama al método run() del thread.

toString(): Retorna la representación string del thread con su nombre, prioridad y ThreadGroup.

Compitiendo por el procesador

Tan pronto como los threads comienzan a correr, compiten por el procesador con otros threads que están corriendo en la misma JVM.

Existen native threads (basados en mecanismos del SO nativo) y green threads (implementados completamente en la JVM). Los primeros son más eficientes, pero los segundos son independientes de la plataforma.

Para crear un thread, se escribe:
Thread myThread = new Thread();

Existen diferentes constructores, de acuerdo a que se deba especificar un String (usado como nombre del nuevo thread), un Threadgroup, un Runnable.

Programación concurrente usando Threads

Todos los threads tienen una prioridad. Es un número entero entre: java.lang.Thread.MIN_PRIORITY y java.lang.Thread.MAX_PRIORITY.
Se usa setPriority() para ajustar la prioridad.

Hay dos tipos de Threads en Java.

- La mayor parte de ellos son User threads: tiene una interface de usuario y existen en orden a dar servicio a esa interface.
- Los threads que corren sólo en background se llaman daemon threads y se especifican invocando al método setDaemon()

Subclassing Thread y overriding run()

La primera forma de personalizar lo que hace un thread cuando se ejecuta, es crear una subclase de Thread (del paquete java.lang, que implementa la interfaz Runnable) y sobreescribir su método run vacío para que haga algo. Por ejemplo:

```
public class SimpleThread extends Thread {
    public SimpleThread(String str) {
        super(str);
    }
    public void run() {
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            try {
                sleep((long)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }
}
```

Programación concurrente usando Threads

El método run es el corazón de cualquier Thread y donde toma lugar la acción del Thread. En este caso contiene un loop que itera 10 veces. En cada iteración muestra el número de iteración y el nombre del Thread, luego espera (“duerme”) por un intervalo aleatorio de hasta 1 seg.
Después que finaliza el loop, el método run imprime DONE!, con el nombre del Thread.
La clase TwoThreadsTest provee un método main() que crea dos threads SimpleThread: uno de nombre “Jamaica” y otro de nombre “Fiji”.

```
public class TwoThreadsTest {
    public static void main (String[] args) {
        new SimpleThread("Jamaica").start();
        new SimpleThread("Fiji").start();
    }
```

Programación concurrente usando Threads

Compilando y corriendo el programa puede verse el siguiente resultado:

0 Jamaica
0 Fiji
1 Fiji
1 Jamaica
2 Jamaica
2 Fiji
3 Fiji
3 Jamaica
…..
8 Fiji
9 Fiji
8 Jamaica
DONE! Fiji
9 Jamaica
DONE! Jamaica

Programación concurrente usando Threads

Notar cómo el output de cada thread está entremezclado con el otro. Esto es porque ambos SimpleThread están corriendo concurrentemente.
Entonces, ambos métodos run se están ejecutando al mismo tiempo y cada thread está mostrando su salida al mismo tiempo que la otra.

Ejercicio: Cambiar el programa agregando un tercer thread con el nombre “Tolhuin”

Implementando la interface Runnable

El siguiente ejemplo usa una técnica diferente para proveer el método run para su thread. En lugar de extender Thread, el applet Clock implementa la interface Runnable (del paquete java.lang y por lo tanto el método run definido en ésta).
Clock luego crea un Thread y se provee a sí mismo como un argumento al constructor de Thread.
Cuando se crea de esta manera, el Thread toma su método run del objeto pasado en el constructor.
El Applet Clock del ejemplo muestra la hora y actualiza su display cada segundo. Se pueden ejecutar otras tareas mientras el clock continúa, ya que el código que actualiza el display corre en su propio thread.

Programación concurrente usando Threads

```
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.applet.Applet;

public class Clock extends Applet implements Runnable {
    private Thread clockThread = null;
    public void start() {
        if (clockThread == null) {
            clockThread = new Thread(this, "Clock");
            clockThread.start();
        }
    }
    public void run() {
        Thread myThread = Thread.currentThread();
        while (clockThread == myThread) {
            repaint();
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e){
            // the VM doesn't want us to sleep anymore,
// so get back to work
            }
        }
    }
```

Programación concurrente usando Threads

```
    public void paint(Graphics g) {
        // get the time and convert it to a date
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        // format it and display it
        DateFormat dateFormatter = DateFormat.getTimeInstance();
        g.drawString(dateFormatter.format(date), 5, 10);
    }
    // overrides Applet's stop method, not Thread's
    public void stop() {
        clockThread = null;
    }
}
```

El método run del Applet Clock ejecutará el loop hasta que el browser le requiera parar. Durante cada iteración del loop, el Clock repinta su display.

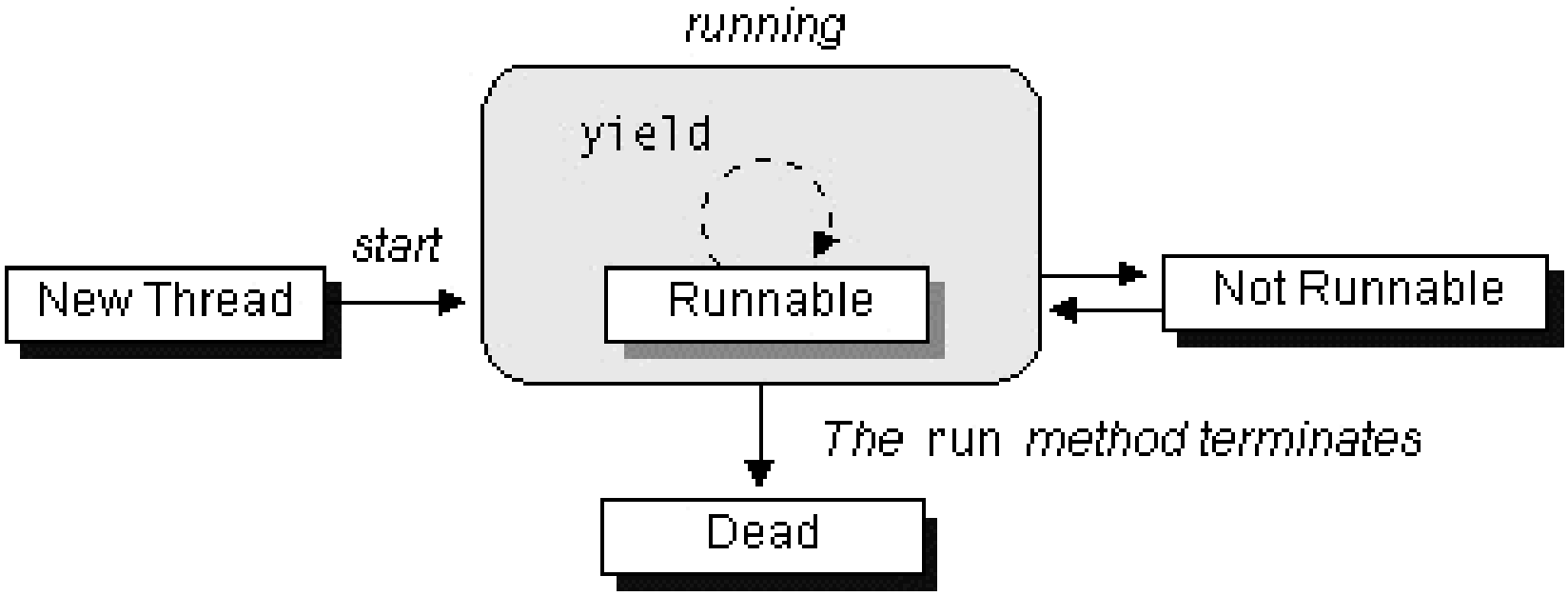
Decidiendo qué opción usar

Hay buenas razones para elegir cualquiera de las dos opciones. Sin embargo, en la mayoría de los casos, la siguiente regla será la guía para la mejor opción:

Regla: Si su clase debe ser subclase de alguna otra (el ejemplo más común es Applet), se debe usar Runnable como se describió en la segunda opción.
Para ser ejecutada en el entorno de un browser, la clase Clock debe ser subclase de Applet. Como Java no soporta herencia múltiple, no puede ser subclase de Applet y de Thread, por ello la clase Clock debe usar la Interface Runnable.

El Ciclo de vida de un Thread

Se trata de: cómo crear y comenzar (start) un thread, algunas de las cosas especiales que puede hacer mientras está corriendo y cómo finalizarlo.
El siguiente diagrama muestra los estados en que puede estar un thread de Java durante su vida.
También ilustra cuáles llamadas a métodos causan la transición a otro estado.



Creando un Thread

La aplicación en la cual está corriendo un applet llama al método start cuando un usuario visita la página del applet. El Applet Clock crea un Thread, clockThread, en su start como se muestra en negrita:

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

Después de ejecutar esa sentencia, clockThread está en el estado New Thread. En este estado, es simplemente un objeto Thread vacío; no han sido asignados recursos del sistema para éste. Cuando un thread está en este estado, sólo se puede comenzar (start) el thread.

Si se invoca cualquier otro método que no sea start() en ese estado del thread causa: IllegalThreadStateException.

Notar que la instancia de Clock es el primer argumento al constructor de thread. El primer argumento al constructor de thread debe implementar la interface Runnable y provee al thread con su método run. El segundo argumento es un nombre para el thread.

Iniciando un Thread

Ahora consideremos la siguiente línea de código en el método start:

```
public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}
```

El método start() crea los recursos del sistema necesarios para ejecutar el thread, y llama al método run del thread. El método run de clockThread es el definido en la clase Clock.

Después que el método start ha retornado, el thread está “corriendo”.

Un thread que ha sido comenzado (started) está en estado Runnable.

Programación concurrente usando Threads

Muchas computadoras tienen procesador único haciendo imposible correr todos los threads al mismo tiempo. El sistema runtime de Java debe implementar un esquema para compartir todos los “running” threads.
En un instante dado, un running thread puede estar esperando por su turno en la CPU.

```
public void run() {
   Thread myThread = Thread.currentThread();
   while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e){
            // the VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

El método run de Clock itera hasta que la condición clockThread == myThread es true.
Dentro del loop, el applet se repinta y luego le dice al thread que duerma (sleep) por un segundo.
El método repaint del applet llama el método paint del applet, el cual hace la actualización del área de display del applet. El método paint de Clock obtiene la hora, su formato y la muestra:

```
public void paint(Graphics g) {
    // get the time and convert it to a date
    Calendar cal = Calendar.getInstance();
    Date date = cal.getTime();
    // format it and display it
    DateFormat dateFormatter =
    DateFormat.getTimeInstance();
    g.drawString(dateFormatter.format(date), 5, 10);
}
```

Haciendo un Thread No-Runnable

Un thread se convierte en Not Runnable cuando ocurre uno de los siguientes eventos:

- Se invoca su método sleep().
- El thread llama al método wait para que sea satisfecha una condición específica.
- El thread está bloqueado en I/O.

El clockThread en el applet Clock se convierte en Not Runnable cuando el método run() llama a sleep en el thread actual:

Programación concurrente usando Threads

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e){
            // the VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

Durante el segundo en que el clockThread está dormido(sleep), el thread no corre, aún si el procesador se vuelve disponible. Después que finaliza el segundo, el thread se vuelve Runnable otra vez y, si el procesador se vuelve disponible, el thread comienza a correr nuevamente.

Deteniendo un Thread

Un programa no termina un thread como lo hace con un applet (llamando a un applet). El thread maneja su propia muerte teniendo un método run que termina naturalmente. Por ejemplo, el loop while en el siguiente método run es un loop finito (itera 100 veces y luego sale).

```
public void run() {
    int i = 0;
    while (i < 100) {
        i++;
        System.out.println("i = " + i);
    }
}
```

Un thread con este método run muere naturalmente cuando se completa el loop y se sale del método run. Miremos cómo el thread del applet Clock maneja su propia muerte.

Programación concurrente usando Threads

```
public void run() {
    Thread myThread = Thread.currentThread();
    while (clockThread == myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e){
            // the VM doesn't want us to sleep anymore,
            // so get back to work
        }
    }
}
```

La condición de salida para este método run es la condición de salida para el loop while.

```
while (clockThread == myThread) {
```

Esta condición indica que el loop saldrá cuando el actual thread en ejecución no sea igual a clockThread. Cuándo podría suceder este caso?

Cuando se deje la página, la aplicación en la cual el applet está corriendo, llama al método stop() del applet. Este método luego establece el clockThread a null, diciendo al loop principal en el método run que finalice:

```
public void stop() {    // applets' stop method
    clockThread = null;
}
```

Si se revisita la página, el método start se invoca de nuevo y el clock comienza otra vez con un nuevo thread.