

# Creación y Control de *Threads* en Java

## Tema 3 - Programación Concurrente y de Tiempo Real

Antonio J. Tomeu<sup>1</sup>   Manuel Francisco<sup>2</sup>

<sup>1</sup>Departamento de Ingeniería Informática  
Universidad de Cádiz

<sup>2</sup>Alumno colaborador de la asignatura  
Universidad de Cádiz

PCTR, 2015

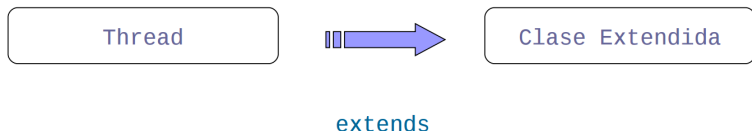
1. Revisión del Concepto de Hilo.
2. Técnicas de Creación de Hilos.
3. Ciclo de Vida. Control de Hilos.
4. Prioridades.
5. Hilos y Sistemas Operativos.
6. Ejecutores y *Pool de Threads*.

# Revisión del Concepto de Hilo (*Thread*) I

- ▶ Dentro de un proceso, el control suele seguir un hilo de ejecución, que comienza con `main`, continúa con el resto de las instrucciones, y termina con el proceso.
- ▶ Java soporta varios hilos de ejecución y por tanto, los programas de Java pueden crear dentro de sí mismos varias secuencias de ejecución concurrentes.
- ▶ A diferencia de los procesos concurrentes, que son independientes, los hilos de un mismo proceso **comparten el espacio de direcciones virtuales, y los recursos del sistema operativo**.
- ▶ Por tanto, cada hilo tiene acceso a los datos y procedimientos del proceso, pero poseen su **propio contador de programa y pila de llamadas** a procedimientos.

# Revisión del Concepto de Hilo (*Thread*) II

- ▶ Los problemas que aparecen con una concurrencia multihilo son los habituales: exclusión mutua y sincronización, y con menor importancia, esquema de prioridades e interbloqueos.
- ▶ Se pueden tener hilos de dos formas: herencia de la clase `Thread` o implementación de la interfaz `Runnable`.

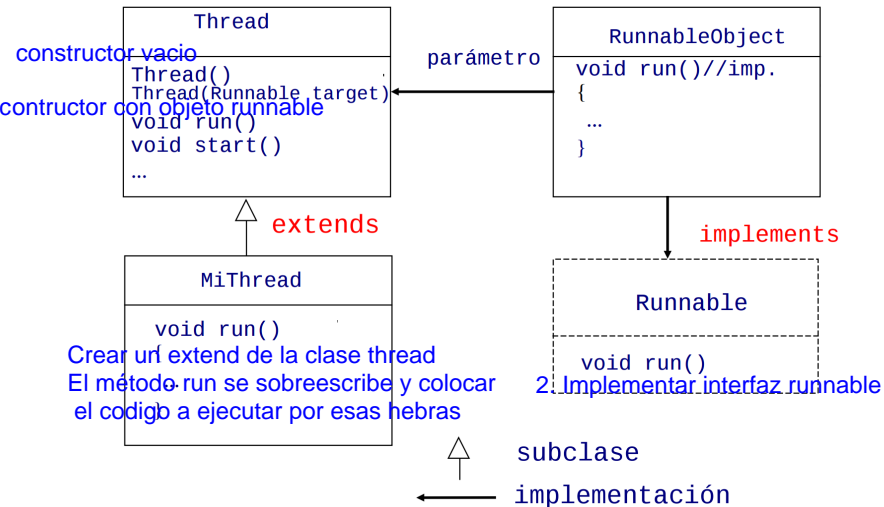


# Razones para usar hilos

- ▶ Estamos en programación concurrente... y toca usar hilos.
- ▶ Se optimiza el uso de la CPU.
- ▶ Se modelan mejor determinados problemas (o no).
- ▶ El problema no admite otra solución razonable, por ejemplo:
  - ▶ Programar un servidor decente.
  - ▶ Diseñar un GUI interactivo.

# API de Java para *Threads*: Marco General

Existen 3 formas de crear hilos en java



# Clase *Thread*: API Básica

```
1  public class Thread extends Object implements Runnable {  
2      public Thread();  
3      public Thread(String name);  
4      public Thread(Runnable target);  
5      public Thread(Runnable target, String name);  
6      public Thread(Runnable target, String name, long stackSize);  
7      public void run();  
8      public void start();  
9      public void join();  
10     ...  
11 }
```

# Concurrencia con Hilos por Herencia de la clase *Thread* I

## Código 1: codigos\_t3/Ejemplo\_Hilos1.java

```
1  class Ejemplo_Hilos1 extends Thread {
2      public Ejemplo_Hilos1(int Tope) //constructor
3      {
4          T = Tope;
5      }
6
7      public void run() //sobreescritura del metodo run
8      { Run se realizará de manera concurrente con el hilo principal
9          for (int i = 1; i <= T; i++)
10             System.out.println(i); //aqui comportamiento del
11         } //hilo deseado
12
13     private int T;
14 }
```



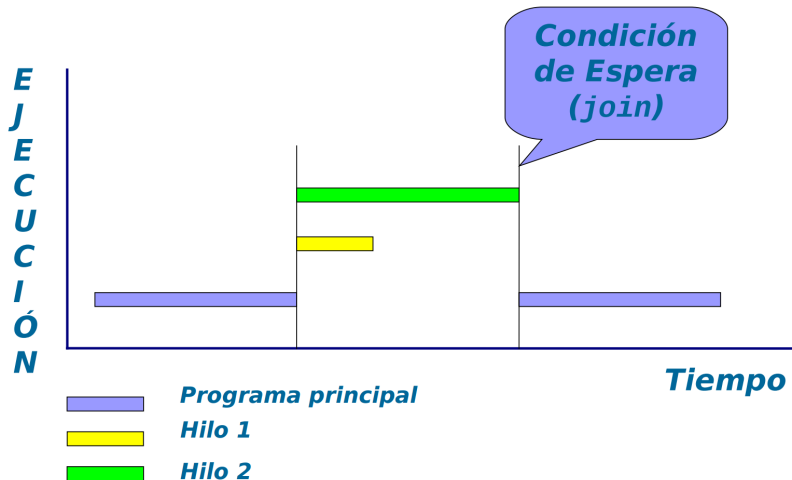
# Concurrencia con Hilos por Herencia de la clase *Thread* II

## Código 2: codigos\_t3/Prueba\_Hilo1.java

```
1  class Prueba_Hilo1 //Hace uso de la clase anterior
2  {
3      public static void main(String[] args) throws
           InterruptedException {
4          Ejemplo_Hilos1 Hilo1 = new Ejemplo_Hilos1(5);
5          Ejemplo_Hilos1 Hilo2 = new Ejemplo_Hilos1(15);
6          cada objeto ejemplohilos es un posible hilo
7          Hilo1.start(); //Ahora se lanzan ambos hilos...
8          Hilo2.start(); //con apertura de co-rutina
9  El metodo start coloca los hilos en una cola para ejecutar de forma concurrente
10         Hilo1.join();
11         Hilo2.join(); // y cierre de co-rutina
12
13         System.out.println("Hilos terminados");
14     }
15 }
```

No se puede saber con certeza cuando se empezaran a ejecutarse los hilos.

# Secuencia Temporal de Co-rutina con Hilos



- ▶ Dado el código anterior, incremente el número de hilos y el número de vueltas que cada hilo da. Recompile y ejecute.
- ▶ ¿Observa entrelazado en la salida?
- ▶ Continúe aumentando el número de hilos (por ejemplo definiendo un vector de hilos).
- ▶ ¿Dónde está el límite práctico al número de hilos (% uso de CPU próximo al 100%)?
- ▶ Escriba un “hola mundo” concurrente.

### Código 3: codigos\_t3/Hola\_Adios.java

```
1 public class Hola_Adios extends Thread {
2     public Hola_Adios(String Palabra) {
3         Cadena = Palabra;
4     }
5
6     private void otrometodo() {
7         System.out.println("otro metodo");
8     }
9
10    public void run() {
11        for (;;) {
12            System.out.println(Cadena);
13            this.otrometodo(); // run puede invocar otros metodos de
                               // la clase
14            Integer p = new Integer(3); //o crear los objetos que
                               // necesita
15        }
16    }
17
18    public static void main(String[] args) {
19        new Hola_Adios("Hola").start();
20        new Hola_Adios("Hola").start();
21        new Hola_Adios("Hola").start();
```

```
22     new Hola_Adios("Adios").start();
23 }
24
25 private String Cadena;
26 }
```

# Revisitando incConcurrente

- ▶ Descargue `incConcurrente.java`.
- ▶ Array de Threads.
- ▶ Dato común: `n`.
- ▶ Variables static **permiten compartir memoria entre threads.**
- ▶ Secciones críticas.
- ▶ Ausencia de control.
- ▶ Resultados no consistentes.

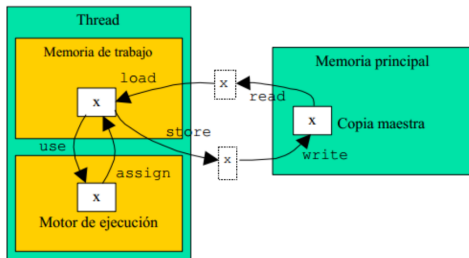


Figura: Modelo de Memoria de Java

- ▶ Escriba un hilo que muestre pares o impares, según se indique en el constructor, un número dado de veces, que también se indicará en el constructor. Llame a la clase `ParImpar.java`.
- ▶ Escriba ahora un código que haga uso de la clase anterior. Llámelo `Usa_ParImpar.java`. Observe el entrelazado.
- ▶ Aloje una variable compartida en una clase llamada `Critica.java`. Provea métodos para incrementar la variable y para mostrar su contenido. ¿Habría condiciones de concurso?
- ▶ Escriba ahora hilos que utilicen un objeto común de esa clase.
- ▶ Láncelos en un código aparte. ¿Observa algo raro?
- ▶ Aunque no lo observe, ¿qué puede ocurrir potencialmente?
- ▶ Realmente ¿hacía falta la clase `Critica.java`?

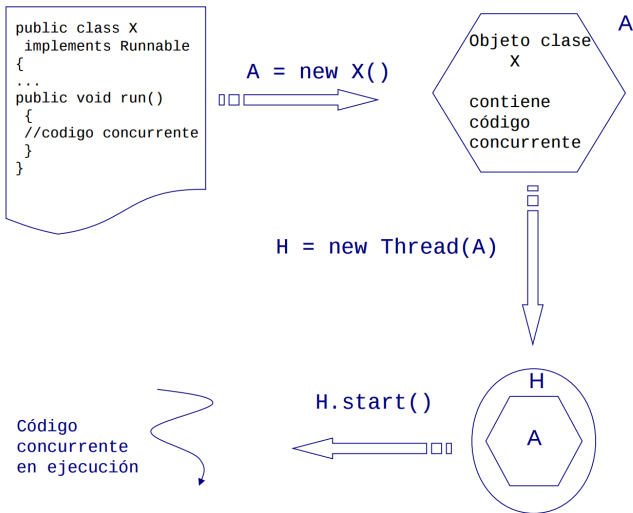
# Concurrencia con Hilos por Implementación de la Interfaz Runnable I

```
1 public interface Runnable {  
2     public void run();  
3 }
```

- ▶ Es una interfaz de `java.lang`.
- ▶ Cualquier clase X que la implemente **expresa ejecución concurrente**.
- ▶ Los objetos de la clase X son parámetros del constructor de `Thread`.



# Concurrencia con Hilos por Implementación de la Interfaz Runnable II



#### Código 4: codigos\_t3/Usorunnable.java

```
1 public class Usorunnable implements Runnable {
2     private String Cadena;
3
4     public Usorunnable(String Palabra) {
5         Cadena = Palabra;
6     }
7
8     public void run() {
9         for (;;)
10             System.out.println(Cadena);
11     }
12
13     public static void main(String[] args) {
14         Runnable Hilo1 = new Usorunnable("Hola");
15         Runnable Hilo2 = new Usorunnable("Adios");
16         new Thread(Hilo1).start();
17         new Thread(Hilo2).start();
18     }
19 }
```

Se crea una clase de Usorunnable, que implementa el objeto runnable.

Nos obliga a utilizar un metodo run.

Para crear nuevos hilos creamos objetos de tipo thread pasandoles un objeto de tipo runnable por argumentos.

Para ejecutarlo, tenemos que pasarselo como argumento a un objeto de tipo thread, para despues aplicarle un metodo start.

## Código 5: codigos\_t3/Usorunnable2.java

```
1  /*
2   * Otra forma de crear hilos concurrentes dandoles nombre
3   * @author Antonio J. Tomeu
4   */
5  public class Usorunnable2 implements Runnable {
6      private int Iter;
7
8      public Usorunnable2(int Dato) {
9          Iter = Dato;
10     }
11
12     public void run() {
13         for (int i = 1; i <= Iter; i++)
14             System.out.println("Trabajando");
15     }
16     public static void main(String[] args) throws
17         InterruptedException {
18         Runnable HiloA = new Usorunnable2(100);
19         Runnable HiloB = new Usorunnable2(200);
20         Runnable HiloC = new Usorunnable2(100);
21
22         //version del constructor Thread crea hilo con un nombre
23         Thread A = new Thread(HiloA, "Mi Hilo");
24         Thread B = new Thread(HiloB, "Tu Hilo"); //sin nombre
```

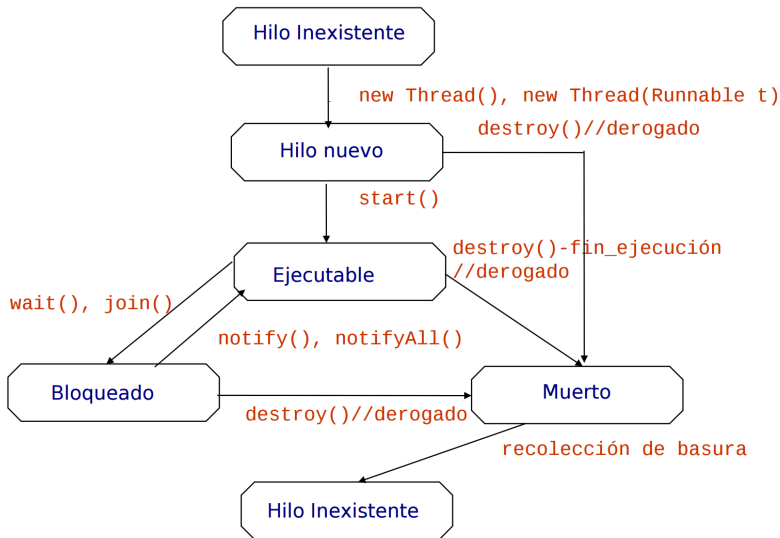
```
24     Thread C = new Thread(HiloC);
25
26     A.start();
27     B.start();
28     A.join();
29     B.join();
30     C.join();
31
32     //metodo getName() de objetos de la clase Thread devuelve
        el nombre
33     //del hilo
34     System.out.println(A.getName());
35     System.out.println(B.getName());
36     //no tenia nombre, pero se le dio uno en tiempo de
        ejecucion.
37     System.out.println(C.getName());
38 }
39 }
```

# Comparativa de Métodos de *Multithreading*

- ▶ Inconveniente de heredar de Thread: No permite heredar de otras clases.
- ▶ **Alternativa** de creación de hilos: implementación de la interfaz Runnable.
- ▶ Sobreescibir siempre el método run.
- ▶ Los objetos que implementan Runnable deben ser lanzados **explícitamente**.
- ▶ Se hace creando un objeto Thread cuyo parámetro es un objeto Runnable.
- ▶ Luego se llama al método start del objeto Thread creado.
- ▶ **Técnica recomendada: implementar la interfaz.**

- ▶ Escriba ahora código de hilo que implemente la interfaz Runnable. Déle al hilo el comportamiento que estime oportuno. Llámelo hiloRunn.java.
- ▶ Escriba un programa que haga uso de los hilos anteriores. Llámelo usahilRunn.java.
- ▶ Inspeccione el API de la clase Thread. En C/C++, el API de pthread.h incluye herramientas de sincronización mediante mutex. ¿Puede decirse lo mismo de la clase Thread en Java?
- ▶ Finalmente, desarrolle una versión en Java del algoritmo de Lamport para dos procesos utilizando hilos heredados de Thread.

# Objetos *Thread*: Ciclo de Vida



# Clase *Thread*: API de Control

Método	Comportamiento
<code>t1.checkAccess()</code>	Determina si <code>t0</code> tiene permiso para controlar a <code>t1</code> .
<code>t1.start()</code>	Lanza el hilo <code>t1</code> .
<code>t1.stop()</code>	Mata al hilo <code>t1</code> . <b>DEROGADO</b>
<code>t1.isAlive()</code>	Comprueba si el hilo <code>t1</code> está vivo.
<code>t1.suspend()</code>	Envía a <code>t1</code> de listo/en ejecución a bloqueado.
<code>t1.resume()</code>	Hace que <code>t1</code> vaya de bloqueado a listo. <b>DEROGADO</b>
<code>t1.interrupt()</code>	Envía una señal a <code>t1</code> .
<code>t1.sleep(int t)</code>	Suspende a <code>t1</code> durante <code>t</code> milisegundos.
<code>t1.yield()</code>	Pasa a <code>t1</code> de en ejecución a listo.
<code>t1.join()</code>	Suspende a <code>t0</code> y espera que termine <code>t1</code> .
<code>t1.destroy()</code>	Mata a <code>t1</code> . <b>DEROGADO</b>



# Control de *Threads*: Ejemplo con Métodos Derogados I

## Código 6: codigos\_t3/Control.java

```
1  import java.io.*;
2  Import java.util.*;
3
4  public class Control extends Thread {
5      //No declara constructor explicito. Usa el disponible por
        defecto
6      public void run() {
7          for (;;)
8              System.out.println("Trabajando");
9      }
10
11     public static void main(String[] args) throws IOException {
12         int c;
13         Control Hilo = new Control(); //usando el constructor
            implicito
14         Hilo.start();
15
16         for (int i = 1; i <= 100; i++) //entrelazado de
            instrucciones
```

# Control de *Threads*: Ejemplo con Métodos Derogados II

```
17         System.out.println("Hola soy el padre");
18
19     Hilo.suspend(); //USO DE METODO DEROGADO, HILO PADRE
        SUSPENDE A HIJO .
20     System.out.println("Hijo suspendido");
21     //Ahora reactivamos al hijo, que pasa a listo.
22     System.out.println("Pulsa 1 para despertar al hijo");
23
24     do {
25         c = System.in.read();
26     } while (c != -1);
27
28     Hilo.resume(); //USO DE METODO DEROGADO, PASA A LISTO A HIJO
29     //un poquito de interfoliacion otra vez.
30
31     for (int i = 1; i <= 100; i++)
32         System.out.println("Hola soy el padre");
33
34     Hilo.stop(); //USO DE METODO DEROGADO, PADRE PARA AL HIJO
35 }
36 }
```

# Control de *Threads*: Ejemplo de Replanificación Voluntaria (sleep) I

## Código 7: codigos\_t3/AutoControl.java

```
1  import java.io.*;
2
3  public class AutoControl extends Thread {
4      private int Vueltas;
5
6      public AutoControl(int Dato) {
7          Vueltas = Dato;
8      }
9
10     public void run() { //el uso de sleep exige capturar la
11         //posible excepcion.
12         try {
13             for (int i = 1; i <= Vueltas; i++) {
14                 System.out.println(i);
15                 if (i == 25) { //los hilos se suspenden en la iteracion
16                     //25
17                     System.out.println("Suspension durante dos segundos");
18                 }
19             }
20         } catch (InterruptedException e) {
21             //Manejo de excepcion
22         }
23     }
24 }
```

# Control de *Threads*: Ejemplo de Replanificación Voluntaria (sleep) II

```
16         int timeout = 1000;
17         sleep(timeout);
18         System.out.println("Continuando");
19     } //if
20 } //for
21 } catch (InterruptedException e) {
22     return;
23 }
24 }
25
26 public static void main(String[] args) {
27     new AutoControl(50).start();
28     new AutoControl(150).start();
29 }
30 }
```

# Control de *Threads*: Ejemplo de Cesión de Prioridad Voluntaria (yield) I

## Código 8: codigos\_t3/replaniYield.java

```
1  public class replaniYield extends Thread {
2      private boolean hY; //indicara si el hilo cede prioridad o
        no..
3      private int v;
4
5      public replaniYield(boolean hacerYield, int vueltas) {
6          hY = hacerYield;
7          v = vueltas;
8      }
9
10     public void run() {
11         for (int i = 0; i < v; i++)
12             if (i == 20 && hY == true) {
13                 this.yield();
14             } //indica cesion de prioridad...
15         else System.out.println("Hilo " + this.getName() + " en
            iteracion " + i);
```

# Control de *Threads*: Ejemplo de Cesión de Prioridad Voluntaria (yield) II

```
16     }
17
18     public static void main(String[] args) {
19         replaniYield h0 = new replaniYield(false, 50);
20         replaniYield h1 = new replaniYield(false, 50);
21         replaniYield h2 = new replaniYield(true, 50); //cedera
                prioridad y
22
23         h0.setName("1-NoYield"); //sera o no considerada
24         h1.setName("2-NoYield");
25         h2.setName("3-SIYield");
26
27         h0.start();
28         h1.start();
29         h2.start();
30     }
31 }
```

- ▶ Prioridad hilo hijo igual a la de hilo padre.
- ▶ La prioridad tiene sentido **exclusivamente** en el ámbito de la JVM, aunque se mapea a los hilos de sistema (**Ojo: El *mapping* NO es riguroso  $\implies$  Inversiones de Prioridad**).
- ▶ Clase Thread: esquema de diez niveles de prioridad.
- ▶ Ver la prioridad de un hilo:
  - ▶ `public int getPriority()`.
- ▶ Alterar la prioridad de un hilo:
  - ▶ `public void setPriority(int p) ( $1 \leq p \leq 10$ )`.

# Clase *Thread*: API de Control de Prioridad

```
1 package java.lang;
2 public class Thread implements Runnable {
3     public final static int Thread.MIN_PRIORITY;
4     public final static int Thread.NORM_PRIORITY;
5     public final static int Thread.MAX_PRIORITY;
6     public void setPriority(int prioridad);
7     public int getPriority();
8 }
```



- ▶ Valor prioridad en JVM indica al planificador del SO qué hilos van primero.
- ▶ Pero **no es un *contrato absoluto*** entre ambos ya que depende:
  - ▶ de la implementación de la JVM.
  - ▶ del SO subyacente.
  - ▶ del mapping prioridad JVM-Prioridad SO.

# Planificación Basada en Prioridades II

## Código 9: codigos\_t3/Prioridades.java

```
1  public class Prioridades extends Thread {
2      private long dato;
3      private static int prio = 4; //atributo de clase comun a
        instancias
4
5      private long fac(long n) {
6          if (n == 0) return 0;
7          else if (n == 1) return 1;
8          else return (fac(n - 1) * n);
9      }
10
11
12     public void run() {
13         //this.setPriority(prio++); //ejecutar con y sin el ajuste
            de prioridad
14         System.out.println("El factorial de " + dato + " es " +
            fac(dato));
15     }
16
17
18
```

# Planificación Basada en Prioridades III

```
19     public static void main(String[] args) {
20         new Prioridades(10).start(); //orden lanzamiento no es
            igual al orden
21         new Prioridades(20).start(); //de ejecucion... pero
22         new Prioridades(30).start(); //ajustando las prioridades?
23         new Prioridades(40).start();
24         new Prioridades(50).start();
25         new Prioridades(60).start();
26     }
27 }
```

# Planificación Basada en Prioridades IV

## Código 10: codigos\_t3/Trabajo.java

```
1  import java.util.*;
2  import java.text.*;
3
4  public class Trabajo implements Runnable {
5      long n;
6      String id;
7
8      private long fib(long n) {
9          if (n == 0) return 0L;
10         if (n == 1) return 1L;
11
12         return fib(n - 1) + fib(n - 2);
13     }
14
15     public Trabajo(long n, String id) {
16         this.n = n;
17         this.id = id;
18     }
19
20
21
```

# Planificación Basada en Prioridades V

```
22     public void run() {
23         Date d = new Date();
24         DateFormat df = new SimpleDateFormat("HH:mm:ss:SSS");
25         long startTime = System.currentTimeMillis();
26         d.setTime(startTime);
27         System.out.println("Iniciando trabajo " + id + " a las " +
28                             df.format(d));
29
30         fib(n);
31
32         long endTime = System.currentTimeMillis();
33         d.setTime(endTime);
34         System.out.println("Acabando trabajo " + id + " a las " +
35                             df.format(d) + " tras " + (endTime - startTime) + "
36                             milliseconds");
37     }
38 }
```

## Código 11: codigos\_t3/ThreadTest.java

```
1  public class ThreadTest {
2      public static void main(String[] args) {
3          int nHilos = Integer.parseInt(args[0]);
4          long n = Long.parseLong(args[1]);
5
6          Thread t[] = new Thread[nHilos];
7          for (int i = 0; i < t.length; i++) {
8              t[i] = new Thread(new Trabajo(n, "Trabajo " + i));
9              t[i].start();
10         }
11
12         for (int i = 0; i < t.length; i++) {
13             try {
14                 t[i].join();
15             } catch (InterruptedException ie) {}
16         }
17     }
18 }
```

## Código 12: codigos\_t3/ThreadTestNuevaPrioridad.java

```
1  public class ThreadTestNuevaPrioridad {
2      public static void main(String[] args) {
3          int nHilos = Integer.parseInt(args[0]);
4          long n = Long.parseLong(args[1]);
5
6          Thread t[] = new Thread[nHilos];
7          for (int i = 0; i < t.length; i++) {
8              t[i] = new Thread(new Trabajo(n, "Trabajo " + i));
9              t[i].setPriority((i % 10) + 1);
10             t[i].start();
11         }
12
13         for (int i = 0; i < t.length; i++) {
14             try {
15                 t[i].join();
16             } catch (InterruptedException ie) {}
17         }
18     }
19 }
```

- ▶ Compile y ejecute los códigos anteriores.
- ▶ ¿Observa inversiones de prioridad?
- ▶ ¿A qué cree que se deben?
- ▶ Desarrolle ahora código de hilo sincronizado basado en prioridades (p.e. un hilo incrementa un dato y otro lo muestra, en ese orden).
- ▶ ¿Es una estrategia válida de sincronización?
- ▶ ¿Y con una co-rutina?



- ▶ El SO conoce el número de hilos que usa la JVM.
- ▶ Se aplican uno-a-uno (JVM a Win).
- ▶ El secuenciamiento de hilos Java está sujeto al del SO.
- ▶ Se aplican 10 prioridades en la JVM sobre 7 en el SO + 5 prioridades de secuenciamiento.

# Mapping de Hilos JVM-Hilos Nativos de Win32 II

Prioridad Java	Prioridad Windows (Java 6)
1 (Thread.MIN_PRIORITY)	THREAD.PRIORITY_LOWEST
2	THREAD.PRIORITY_LOWEST
3	THREAD.PRIORITY_BELOW_NORMAL
4	THREAD.PRIORITY_BELOW_NORMAL
5 (Thread.NORM_PRIORITY)	THREAD.PRIORITY_NORMAL
6	THREAD.PRIORITY_NORMAL
7	THREAD.PRIORITY_ABOVE_NORMAL
8	THREAD.PRIORITY_ABOVE_NORMAL
9	THREAD.PRIORITY_HIGHEST
10 (Thread.MAX_PRIORITY)	THREAD.PRIORITY_HIGHEST

- ▶ Núcleos recientes implementan Native Posix Thread Library.
- ▶ Aplican hilos JVM a hilos del núcleo uno-a-uno bajo el modelo de Solaris.
- ▶ La prioridad Java es un factor muy pequeño en el cálculo global del secuenciamiento.

## Mapping de Hilos de JVM-Hilos Nativos de Linux II

Prioridad Java	Prioridad Linux (nice)
1 (Thread.MIN_PRIORITY)	4
2	3
3	2
4	1
5 (Thread.NORM_PRIORITY)	0
6	-1
7	-2
8	-3
9	-4
10 (Thread.MAX_PRIORITY)	-5

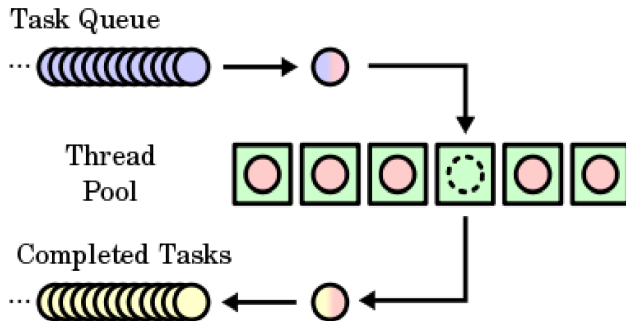
- ▶ La actual especificación de la JVM no establece un modelo de planificación por prioridades válido.
- ▶ El comportamiento puede y debe variar en diferentes máquinas.
- ▶ En secuencias de tareas estrictas, no es posible planificar con prioridades.
- ▶ Aunque sí con co-rutinas, a nivel básico.

- ▶ Crear y destruir hilos tiene **latencias**.
- ▶ Creación y control son **responsabilidad del programador**.
- ▶ El programador debe pensar en hilos en lugar de concentrarse en las tareas.
- ▶ Determinadas aplicaciones crean hilos de forma masiva (*web server*) para las cuales el modelo de threads es **ineficiente**.

## Pool de Threads: Definición

- ▶ Es una **reserva de hilos** a la espera de recibir tareas para ejecutarlas.
- ▶ Sólo se **crea una vez** (reduce latencias).
- ▶ **Reutiliza los hilos** una y otra vez.
- ▶ Efectúa de forma automática la gestión del ciclo de vida de las tareas.
- ▶ Recibe las tareas a ejecutar mediante objetos Runnable.

## Pool de Threads: Patrón Gráfico





# La Interfaz java.util.concurrent.Executor

```
1 package java.util.concurrent;
2
3 public interface Executor {
4     public void execute(Runnable tarea);
5 }
```

# La Interfaz java.util.concurrent.ExecutorService

```
1 package java.util.concurrent.*;
2
3 public interface ExecutorService extends Executor {
4     boolean awaitTermination(long timeout, TimeUnit unit);
5     boolean isShutdown();
6     List<Runnable> shutdownNow();
7     boolean isTerminated();
8     void shutdown();
9 }
```

# Ejecutores Predefinidos: la Clase Executors

```
1 package java.util.concurrent.*;
2
3 public class Executors {
4     static ExecutorService newCachedThreadPool();
5     static ExecutorService newFixedThreadPool(int nThreads);
6     static ExecutorService newSingleThreadExecutor();
7 }
```

# Ejecutores Predefinidos: Ejecutor de Hilo Único I

## Código 13: codigos\_t3/EjecutorHiloSimple.java

```
1  import java.util.concurrent.*;
2
3  class Tarea implements Runnable {
4      public static int cont = 0;
5
6      public Tarea() {}
7
8      public void run() {
9          for (int i = 0; i < 10000000; i++)
10             cont++;
11     }
12 }
13
14 public class EjecutorHiloSimple {
15
16     public static void main(String[] args) throws Exception {
17         ExecutorService ejecutor =
18             Executors.newSingleThreadExecutor();
```

# Ejecutores Predefinidos: Ejecutor de Hilo Único II

```
19     for (int i = 0; i < 1000; i++)
20         ejecutor.execute(new Tarea());
21
22     ejecutor.shutdown();
23     while (!ejecutor.isTerminated());
24
25     System.out.println(Tarea.cont);
26 }
27 }
```

# Ejecutores Predefinidos: Ejecutor de Capacidad Fija I

## Código 14: codigos\_t3/EjecutorHiloTamanoFijo.java

```
1  import java.util.concurrent.*;
2
3  class Tarea implements Runnable {
4      public static int cont = 0;
5
6      public Tarea() {}
7
8      public void run() {
9          for (int i = 0; i < 10000000; i++)
10             cont++;
11     }
12 }
13
14 public class EjecutorHiloTamanoFijo {
15
16     public static void main(String[] args) throws Exception {
17         ExecutorService ejecutor =
18             Executors.newFixedThreadPool(500);
19     }
20 }
```

# Ejecutores Predefinidos: Ejecutor de Capacidad Fija II

```
19     for (int i = 0; i < 1000; i++)
20         ejecutor.execute(new Tarea());
21
22     ejecutor.shutdown();
23     while (!ejecutor.isTerminated());
24
25     System.out.println(Tarea.cont);
26 }
```

# Ejecutores Predefinidos: Ejecutor de Capacidad Fija I

## Código 15: codigos\_t3/EjecutorHiloTamanoVariable.java

```
1  import java.util.concurrent.*;
2
3  class Tarea implements Runnable {
4      public static int cont = 0;
5
6      public Tarea() {}
7
8      public void run() {
9          for (int i = 0; i < 10000000; i++)
10             cont++;
11     }
12 }
13
14 public class EjecutorHiloTamanoVariable {
15     public static void main(String[] args) throws Exception {
16         ExecutorService ejecutor = Executors.newCachedThreadPool();
17
18         for (int i = 0; i < 1000; i++)
19             ejecutor.execute(new Tarea());
20     }
21 }
```



# Ejecutores Predefinidos: Ejecutor de Capacidad Fija II

```
20
21     ejecutor.shutdown();
22     while (!ejecutor.isTerminated());
23
24     System.out.println(Tarea.cont);
25 }
```

# Ejecutores Altamente Configurables: Clase ThreadPoolExecutor I

- ▶ Alta configurabilidad por programador
- ▶ Implementa a `ExecutorService`
- ▶ El constructor:

```
1 public ThreadPoolExecutor(int corePoolSize, int  
    maximumPoolSize, long keepAliveTime, TimeUnit unit,  
    BlockingQueue<Runnable> workQueue);
```

# Ejecutores Altamente Configurables: Clase ThreadPoolExecutor II

## Código 16: codigos\_t3/Tarea.java

```
1  public class Tarea implements Runnable {
2      int numTarea;
3
4      public Tarea(int n) {
5          numTarea = n;
6      }
7
8      public void run() {
9          for (int i = 1; i < 100; i++) {
10             System.out.println("Esta es la tarea numero: " + numTarea
11                                 + "imprimiendo" + i);
12         }
13     }
```

# Ejecutores Altamente Configurables: Clase ThreadPoolExecutor III

Código 17: codigos\_t3/pruebaThreadPool.java

```
1  import java.util.concurrent.*;
2
3  public class pruebaThreadPool {
4
5      public static void main(String[] args) {
6
7          int nTareas = Integer.parseInt(args[0]);
8          int tamPool = Integer.parseInt(args[1]);
9
10         ThreadPoolExecutor miPool = new ThreadPoolExecutor(tamPool,
11             tamPool, 60000L, TimeUnit.MILLISECONDS, new
12             LinkedBlockingQueue < Runnable > ());
13
14         Tarea[] tareas = new Tarea[nTareas];
15
16         for (int i = 0; i < nTareas; i++) {
17             tareas[i] = new Tarea(i);
18             miPool.execute(tareas[i]);
19         }
20     }
21 }
```

# Ejecutores Altamente Configurables: Clase ThreadPoolExecutor IV

```
18  
19     miPool.shutdown();  
20 }  
21 }
```

# En el próximo tema...

- ▶ Modelos teóricos de control de la exclusión mutua.
- ▶ Algoritmos con variables compartidas.
- ▶ Semáforos.
- ▶ Regiones Críticas.
- ▶ Monitores.
- ▶ Equivalentes Reales en Java y C++11.



Eckel, B.

*Thinking in Java*

Prentice Hall, 2006



Goetz et al.

*Java Concurrency in Practice*

Addison Wesley, 2006



Oaks and Wong.

*Java Threads*

O'Reilly, 2004