Creación y Control de *Threads* en Java

Tema 2 - Programación Concurrente y de Tiempo Real

Departamento de Ingeniería Informática
Universidad de Cádiz

PCTR, 2022

Contenido

1. Revisión del Concepto de Hilo[1].
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*.
7. Ejecución Asíncrona a Futuro.
8. Ecuación de Subramanian.

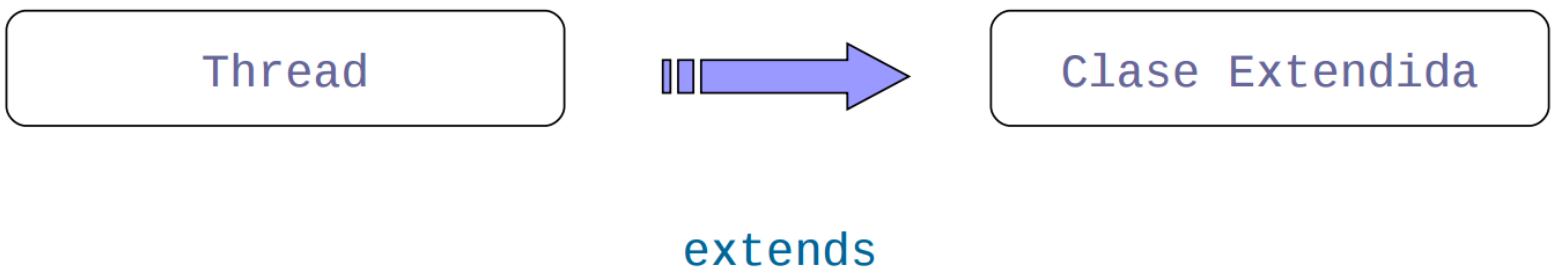
[1] En adelante, utilizaremos los términos hilo, hebra o thread de forma indistinta.

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 por proceso 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

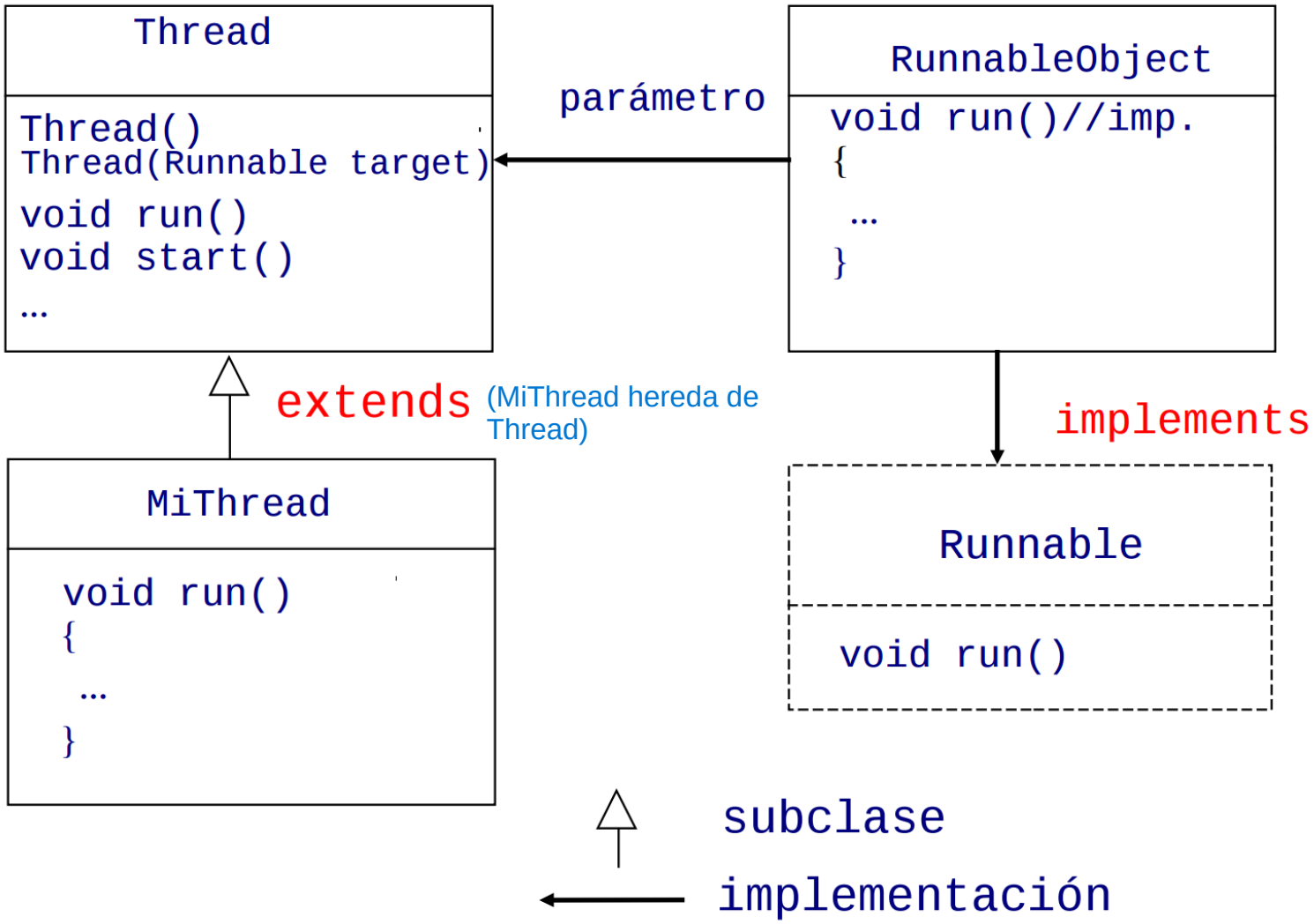
- Los problemas que aparecen con una concurrencia multihilo en Java son los habituales: exclusión mutua, interbloqueos y sincronización, etc.
- Desde un punto de vista elemental, se pueden tener hilos de dos formas: herencia de la clase Thread o implementación de la interfaz Runnable.



Razones para usar hilos

- Estamos en programación concurrente... y toca usar hilos.
- Se optimiza el uso de las actuales CPU, y de los múltiples núcleos que ofrecen.
- 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



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 }
```

Recordar que con join quedamos a la espera de que el hilo que iniciamos con start acabe

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

Siempre que creamos una clase que extiende de thread, tenemos que redefinir la función run. Esta función se ejecuta con cada hilo que iniciemos

Código: codigos_t2/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    {
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 }
```

En este caso, ambos hilos no se pisan al imprimir por pantalla porque println en java el sdk (¿?) es threadsave, está protegido (Como si fuera atómica). Pero no viene en la documentación, por lo que no podemos fiarnos y habría que tratarlo como condición de concurso.

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

Código: codigos_t2/Prueba_hilo1.java

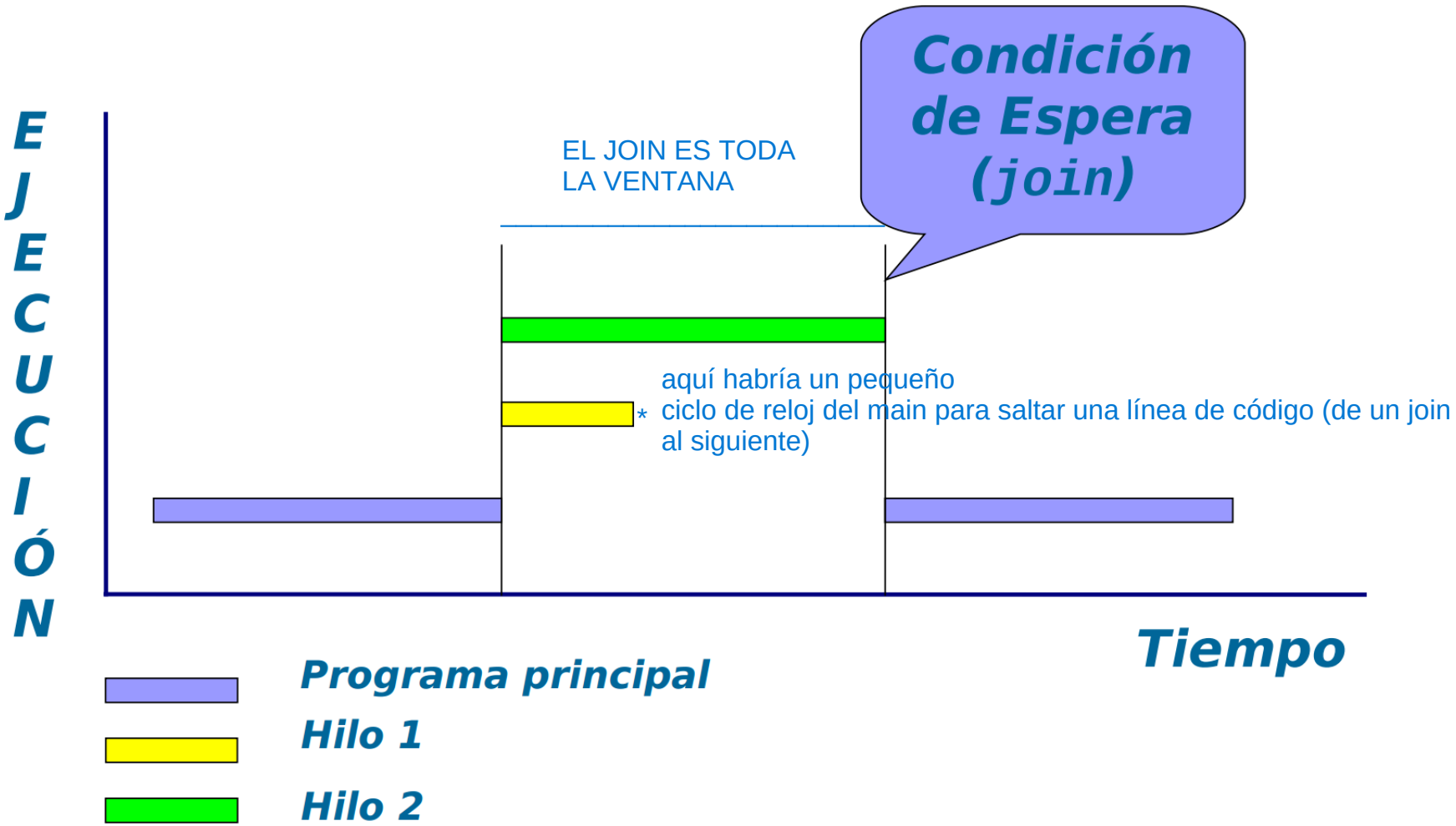
```
1  class Prueba_Hilo1 //Hace uso de la clase anterior
2  {
3    public static void main(String[] args) throws
         InterruptedException {
4      Ejemplo_Hilos1 Hilo1 = new Ejemplo_Hilos1(5);
5      Ejemplo_Hilos1 Hilo2 = new Ejemplo_Hilos1(15);
6
7      Hilo1.start(); //Ahora se lanzan ambos hilos...
8      Hilo2.start();
9  En este instante hay tres hilos coexistiendo: el main, hilo1 e hilo2.
10     Hilo1.join();
11     Hilo2.join(); // y el programa principal los espera...
12
13     System.out.println("Hilos terminados");
14   }
15 }
```

Cada hilo tiene su T, porque son objetos de la clase que hemos creado antes heredando de thread

hilo1 e hilo2 se lanzan, empiezan a iterar por el bucle de "su función run" en un orden no determinista (1234512345,1231234545..). El hilo main sigue ejecutándose hasta que llega a un join.

Por tanto, el join hace que el hilo main (o el hilo en el que se coloca el main) espere a que termine el hilo con el que se llama a join (en este caso, primero espera a que acabe hilo1 y luego a que acabe hilo2)

Secuencia Temporal: Espera por `join`



Ejercicios

- 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: codigos_t2/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();
```

A excepción del println que es " polémico", aquí no hay condiciones de concurso porque no se comparte ningún otro recurso. Se crean objetos distintos aunque sus cadenas sean todas " Hola"

```
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.

FOTO IMPORTANTE (Porque trabajamos en Java y por esto se crea entrelazado)

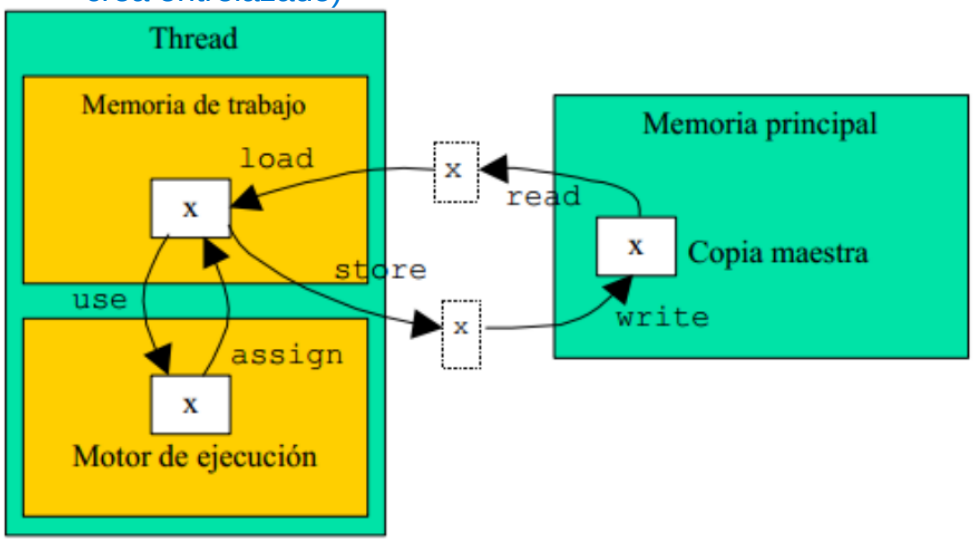


Figura: Modelo de Memoria de Java

Ejercicios

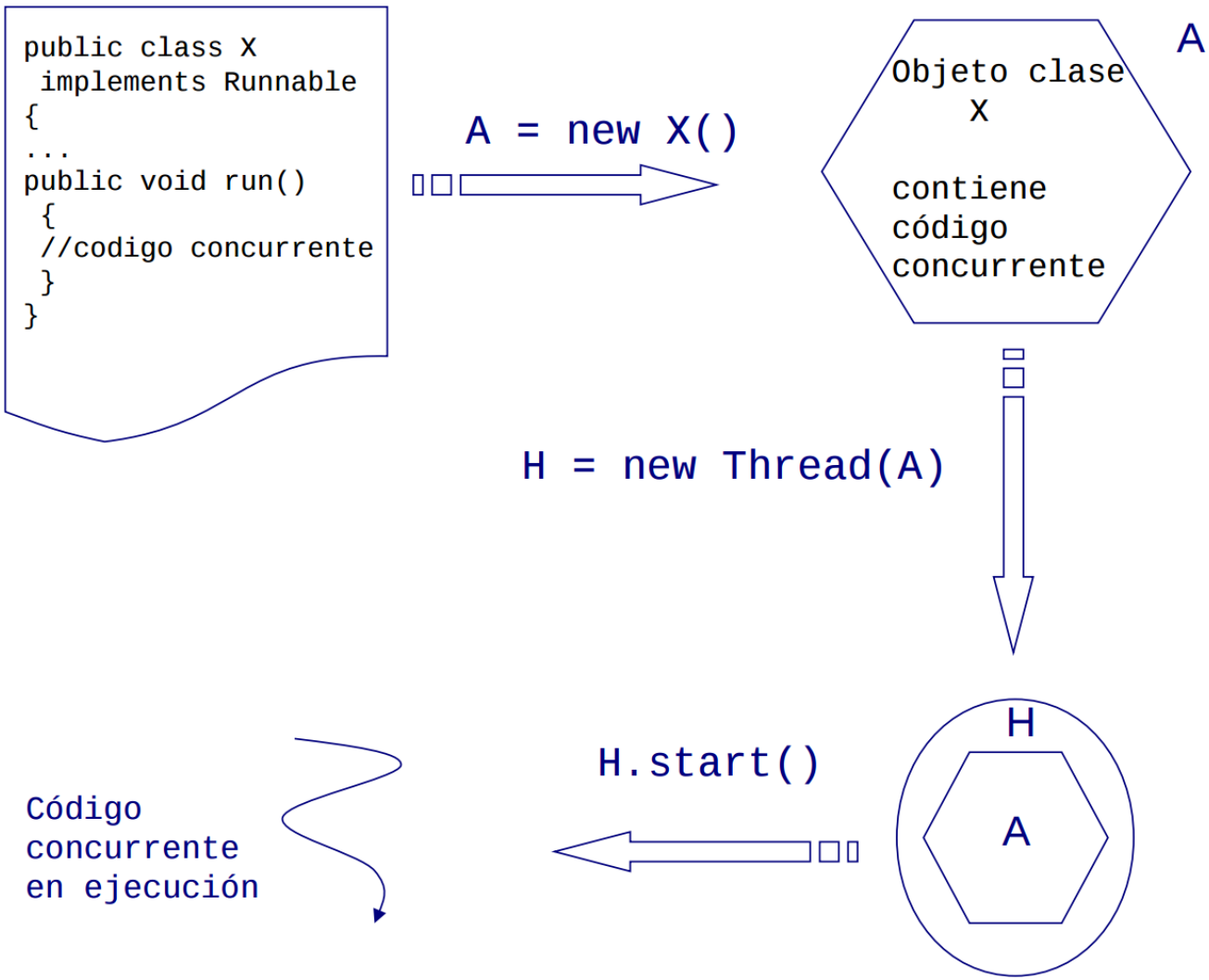
- Escriba un hilo que muestre pares o impares, según se indique en el constructor, un número dado de veces, que también se indicará en el constructor. Llame a la clase `ParImpar.java`.
- Escriba ahora un código que hago uso de la clase anterior. Llámelo `Usa_ParImpar.java`. Observe el entrelazado.
- 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: codigos_t2/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 (;;)      ;; -> true, bucle infinito
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 }
```

Ojo: si usamos runnable en lugar de extender a thread los hilos son de tipo runnable, NO del nombre de la clase que extiende a runnable, como si ocurre cuando extendemos thread

También hay que escribir *Thread(nombrehilo).start() en lugar de nombrehilo.start

*O sea, llamar al constructor de thread

Código: codigos_t2/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
         InterruptedException {
17     Runnable HiloA = new UsoRunnable2(100);
18     Runnable HiloB = new UsoRunnable2(200);
19     Runnable HiloC = new UsoRunnable2(100);
20
21     //version del constructor Thread crea hilo con un nombre
22     Thread A = new Thread(HiloA, "Mi Hilo");
23     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  }
```

Concurrencia con Runnable y Expresiones λ I

- Java soporta expresiones λ desde Java 8.0
- Es posible especificar la concurrencia que los objetos de una clase que implementa a Runnable encapsulan utilizando expresiones λ
- Aconsejable cuando el contenido del método run() es muy pequeño.

Plantilla Básica de Uso de Expresiones λ con Runnable

Código: codigos_t2/lambda.java

```
1  Runnable r = () -> System.out.println("Hello World!");
2  Thread th = new Thread(r);
3  th.start();
```

que es equivalente a:

Plantilla Básica de Uso de Expresiones λ con Runnable

Código: codigos_t2/equivale.java

```
1  Runnable r = new Runnable() {
2      @Override
3      public void run() {
4     System.out.println("Hello World!");
5      }
6  };
7  Thread th = new Thread(r);
8  th.start();
```

Un Ejemplo Completo de Uso de Expresiones λ I

Ojo, una función lambda se desarrolla en tiempo de ejecución

Código: codigos_t2/RunnableLambdaExample.java

```
1  public class RunnableLambdaExample {
2
3      public static void main(String[] args) {
4          System.out.println(Thread.currentThread().getName() +
               ": RunnableTest");
5          // Anonymous Runnable
6           Runnable task1 = new Runnable(){
7            @Override
8            public void run(){
9              System.out.println(Thread.currentThread().getName()
                   + " is running");
10           }
11         };
12
13          // Passing a Runnable when creating a new thread
14         Thread thread2 = new Thread(new Runnable() {
15             @Override
16             public void run(){
```

Un Ejemplo Completo de Uso de Expresiones λ II

```
17                  System.out.println(Thread.currentThread().getName()
                        + " is running");
18              }
19          });
20           // Lambda Runnable
21          Runnable task3 = () -> {
22              System.out.println(Thread.currentThread().getName()
                        + " is running");
23          };
24
25          Thread thread1 = new Thread(task1);
26
27          thread1.start();
28          thread2.start();
29
30          new Thread(task3).start();
31       }
32      }
```

Comparativa de Métodos de *Multithreading*

Ojo: faltan las diapositivas de hebras. Es lo que cambia con respecto a python y se da en el seminario 3

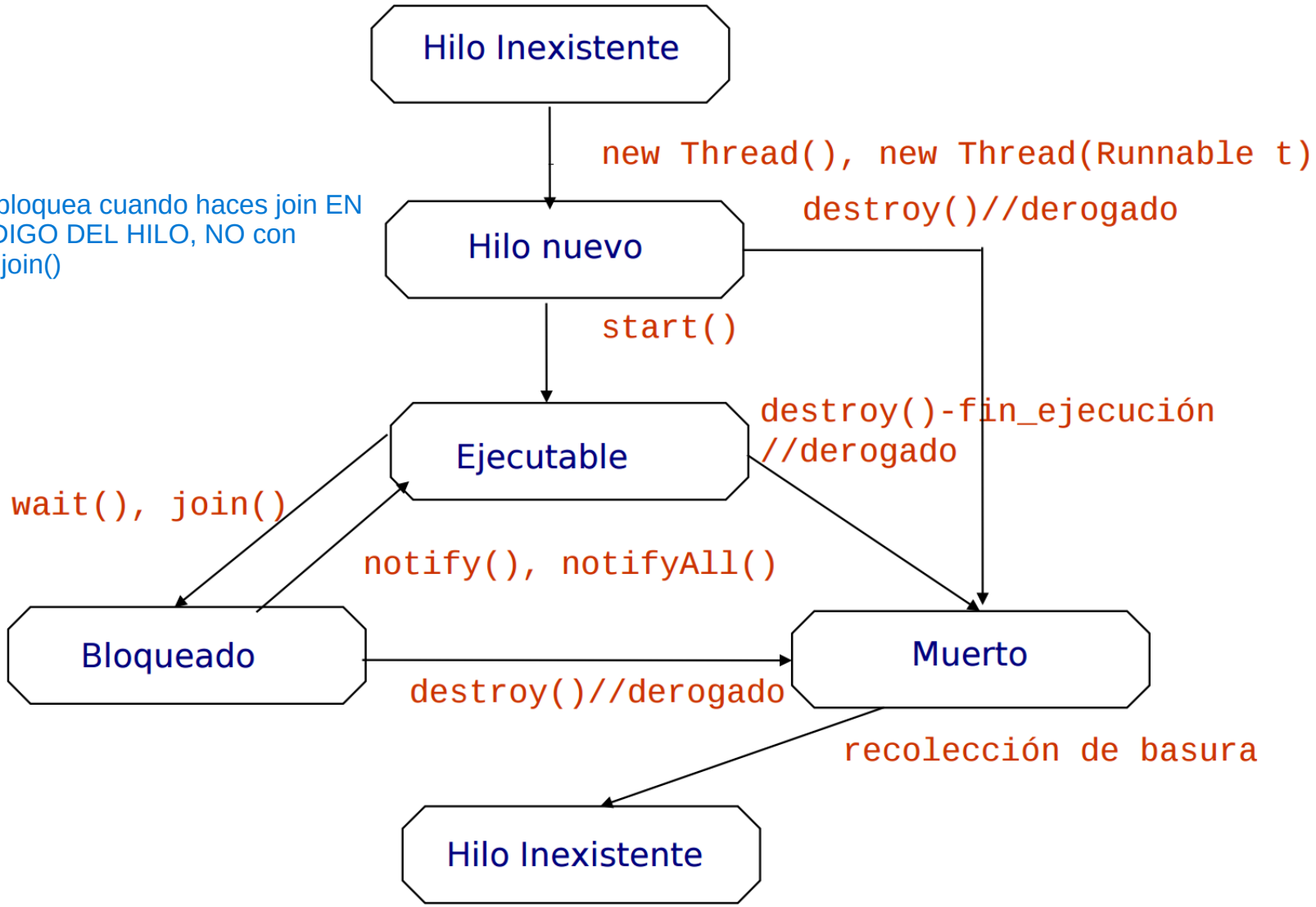
- Inconveniente de heredar de `Thread`: No permite heredar de otras clases.
- Alternativa de creación de hilos: implementación de la interfaz `Runnable`.
- Sobreescribir o implementar siempre el método `run`.
- Los objetos que implementan `Runnable` deben ser ayudados explícitamente para ejecutar el código que encapsulan.
- 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.

Ejercicios

- 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

ojo, se bloquea cuando haces join EN EL CÓDIGO DEL HILO, NO con esehilo.join()



Clase *Thread*: API de Control

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

Control de *Threads*: Ejemplo con Métodos Derogados I

Código: codigos_t2/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: codigos_t2/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
         posible excepcion.
11     try {
12       for (int i = 1; i <= Vueltas; i++) {
13         System.out.println(i);
14         if (i == 25) { //los hilos se suspenden en la iteracion
               25
15           System.out.println("Suspension durante dos segundos");
```

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

```
16              int timeout = 1000;
17              sleep(timeout);  HACER ESTO ES SUSPENDER
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: codigos_t2/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);
```

Yield no funciona siempre, se supone que no entra

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 considerarda
24        h1.setName("2-NoYield");
25        h2.setName("3-SIYield");
26
27        h0.start();
28        h1.start();
29        h2.start();
30      }
31  }
```

Control de Prioridad

- Prioridad hilo hijo igual a la de hilo padre. (empieza siendo)
  Pero la prioridad del hijo se puede modificar después de su nacimiento.
- 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 $\Longrightarrow$ 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)` $(1 \leq p \leq 10)$.

Java tiene niveles de prioridad que se adaptan a las del sistema operativo. (Esa adaptación de prioridades es el mapeo)

Ojo que ha caído en exámenes

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  }
```

Planificación Basada en Prioridades I

- 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.

Si el sistema operativo tiene menos niveles de prioridad, los niveles que se dan en Java con nuestro código pueden fusionarse en el mapeo. Incluso aunque el mapeo salga bien no se garantiza que el proceso prioritario se ejecute antes

Planificación Basada en Prioridades II

Código: codigos_t2/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
```

Planificación Basada en Prioridades III

```
18
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: codigos_t2/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
```

Planificación Basada en Prioridades V

```
21
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 " +
              df.format(d));
28
29        fib(n);
30
31        long endTime = System.currentTimeMillis();
32        d.setTime(endTime);
33        System.out.println("Acabando trabajo " + id + " a las " +
              df.format(d) + " tras " + (endTime - startTime) + "
              milliseconds");
34      }
35    }
```

Planificación Basada en Prioridades VI

Código: codigos_t2/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 }
```

Planificación Basada en Prioridades VII

Código: codigos_t2/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 }
```

Ejercicios

- 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 usando el método `join()` de forma adecuada?

Ver el comportamiento teórico es casi imposible, aunque funcione bien muchas veces no hay garantía de que establecer las prioridades así funcionen

Podría dar mejores resultados pero sigue sin ser viable

Mapping de hilos JVM-hilos Nativos de Win32 I

- 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

Mapping de hilos de JVM-hilos Nativos de Linux I

- 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

Aunque este mapeo puede hacerse bien, no da garantías

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

Control de Prioridades: Conclusiones

- La actual especificación de la JVM no establece un modelo de planificación por prioridades útil.
- El comportamiento puede y debe variar en diferentes máquinas.
- En secuencias de tareas con orden estricto, no es posible planificar con prioridades.
- Aunque se puede establecer orden de ejecución a un nivel básico, utilizando el método `join()`.

Ejecutores y *Pool de Threads*

Según Juan Carlos Pools cae seguro

Se le asigna una pila y un contador, recordar el dibujo importante

- Crear y destruir hilos tiene latencias y consume tiempo y recursos.
- Creación y control son estricta 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.
- Se necesita una aproximación alternativa para gestionar los hilos.

Pool de Threads: Definición

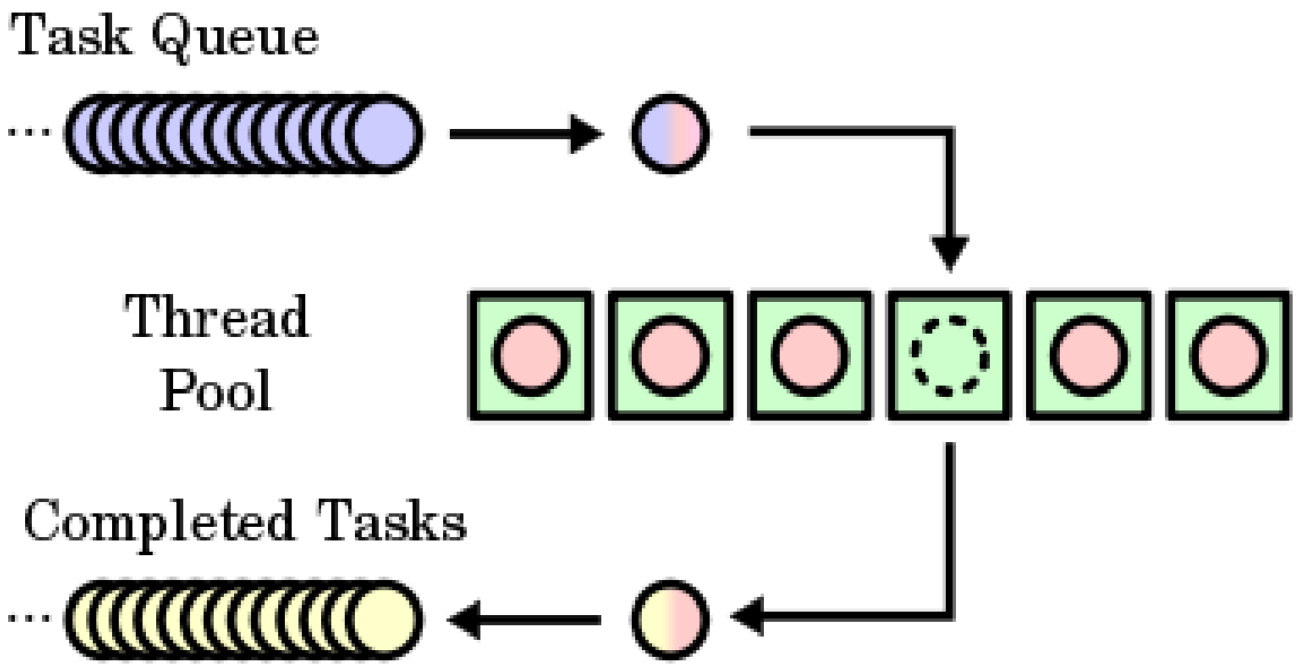
Un pool de threads es un puñado de hilos que se puede crear para ser usados cuando nos de la gana y como nos de la gana (cada uno puede hacer una cosa distinta)

Cada hilo hace una sola tarea, luego muere. La tarea puede ser muy grande. Si el hilo pertenece a un Pool, no muere al finalizar su tarea, se le puede asignar otra tarea diferente (o repetir la misma tarea)

- 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 autómatica la gestión del ciclo de vida de las tareas.
- Recibe las tareas a ejecutar mediante objetos `Runnable`.
- Permite al programador centrarse en el diseño de las tareas.

  No hay que centrarse tanto en manejar los hilos

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  }
```

Cosas que se pueden hacer con el pool

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  }
```

En qué se diferencian estas tres cosas es algo que puede caer en el examen

línea 5: para crear un pool de un número fijo de hilos
línea 4: depende de la caché, mirar en la API (fuente de información, se ve en las prácticas)

Ejecutores Predefinidos: Ejecutor de hilo Único I

Código: codigos_t2/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 < 1000000; i++)
10       cont++;
11   }
12 }
13
14 public class EjecutorHiloSimple {
15
16   public static void main(String[] args) throws Exception {
17     ExecutorService ejecutor =
           Executors.newSingleThreadExecutor();
18
```

"Algo de que acaba siendo secuencial" puede ser pregunta de examen

Si el hilo principal se para y sólo se ejecuta un hilo más, a fin de cuentas no hay paralelismo, es todo secuencial

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: codigos_t2/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 < 1000000; i++)
10       cont++;
11   }
12 }
13
14 public class EjecutorHiloTamanoFijo {
15
16   public static void main(String[] args) throws Exception {
17     ExecutorService ejecutor =
           Executors.newFixedThreadPool(500);
18
```

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 Cached I

Código: codigos_t2/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 < 1000000; 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());
```

Cuando se crean cached pools, se crean hilos mientras hagan falta. Si tenemos 100 tareas, crean hilos para ejecutar esas tareas, pero cuando un hilo acaba su tarea se le asigna la siguiente tarea pendiente, creando un hilo menos. Habrá entre 1 y 100 hilos.

Antes se creaban hilos y ejecutaban su run. Ahora se crea el ejecutor y éste se encarga de crear los hilos en función a las tareas.

Las tareas se le dan al ejecutor, y éste las asigna a los hilos del pool. Recordar que el run está en los objetos de la clase tarea, o sea, en las tareas.

Ejecutores Predefinidos: Ejecutor Cached II

```
20                                  con shutdown acabamos con el pool, si no se quedarían "por ahí"
21          ejecutor.shutdown();
22          while (!ejecutor.isTerminated());
23
24          System.out.println(Tarea.cont);
25      }
```

Este bucle no se encarga de ninguna destrucción, ni tiene que ver con el shutdown. sólo se coloca para esperar a que el programa termine y poder poner el valor de cont sabiendo con seguridad que ya es el definitivo

Ejecutores Altamente Configurables: Clase ThreadPoolExecutor I

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

Si en las dos primeras variables ponemos 4 y 4, tenemos un fixed (siempre 4)
Si ponemos 1 y 4, habrá entre 1 y 4 hilos. Esto sería equivalente a si usamos uno de caché y le enviamos 4 tareas como máximo.

tamaño del pool minimo

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

tiempo vivo

unidad de tiempo

Cola bloqueante de runnables (tareas?)

Es como un pool de caché, pero que tiene un máximo. Si no se hacen las tareas en tiempo y sigue habiendo pendientes, en lugar de crear mas y mas hilos sin control, mete las tareas en la cola.

Ejecutores Altamente Configurables: Clase ThreadPoolExecutor II

Código: codigos_t2/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
             + "imprimiendo" + i);
11     }
12   }
13 }
```

Ejecutores Altamente Configurables: Clase ThreadPoolExecutor III

Código: codigos_t2/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,
           tamPool, 60000L, TimeUnit.MILLISECONDS, new
           LinkedBlockingQueue < Runnable > ());
11
12     Tarea[] tareas = new Tarea[nTareas];
13
14     for (int i = 0; i < nTareas; i++) {
15       tareas[i] = new Tarea(i);
16       miPool.execute(tareas[i]);
```

Ejecutores Altamente Configurables: Clase ThreadPoolExecutor IV

```
17          }
18
19          miPool.shutdown();
20      }
21  }
```

Ejecución Asíncrona a Futuro: La interfaz Callable

Como Runnable, pero devuelve un objeto del tipo que queramos

Lo ideal es usar contenedores, a ser posibles listas

- `Interface Callable<V>`
- Modela a tareas que admiten ejecución concurrente.
- Es similar a `Runnable`, pero no igual. Los objetos que la implementan retornan un resultado de clase V
- Concurrencia soportada por el método `V call()` que encapsula el código a ejecutar concurrentemente
- El código encapsulado en el método `V call()` habitualmente es procesado a través de un ejecutor
- Si el resultado que debe retornarse no puede ser computado lanza una excepción.

ahora usamos call en lugar de run, que devuelve el objeto V en lugar de ser void

Ejecución Asíncrona a Futuro: La interfaz Future

- Future<V> representa el resultado (un objeto de clase <V>) de una computación asíncrona.
- Tales computación y resultado son provistos por objetos que implementan a la interfaz `Callable`
- Es decir, el resultado de una computación asíncrona modelada mediante un objeto que implementa a `Callable<V>`, se obtiene «dentro» de un Future<V>
- Las clases que implementan a Future disponen de métodos para:

Con un pool que hace las tareas a su ritmo, tenemos computación asíncrona

Ejecutamos un objeto instanciado con callable y lo recuperamos con Future

Ejecución Asíncrona a Futuro: La interfaz Future

- comprobar si la computación a futuro ha sido completada (`boolean isDone()`)
- obtener la computación a futuro cuando ha sido completada (`V get()`). Este método es sumamente interesante, y provee sincronización implícita por sí mismo, ya que tiene carácter bloqueante; es decir, se espera (con bloqueo) hasta que la tarea `Callable<V>` que tiene que computar el cálculo lo hace y retorna el resultado.

get es bloqueante, el hilo que lo ejecuta se para hasta recibir algo

Ejecución Asíncrona a Futuro: Ejemplo I

Código: codigos_t2/FactorialTask.java

```
1  import java.util.concurrent.*;
2
3  public class FactorialTask implements Callable<Integer>{
4      int number;
5
6      public FactorialTask(int number){this.number=number;}
7      public Integer call(){
8          int fact = 1;
9          for(int count = number; count > 1; count--) {
10             fact = fact * count;}
11         return (new Integer(fact));
12     }
13
14     public static void main(String[] args) throws Exception{
15         FactorialTask task = new FactorialTask(5);
16         ExecutorService exec = Executors.newFixedThreadPool(1);
17         Future<Integer> future = exec.submit(task);
18         exec.shutdown ();
19         System.out.println(future.get().intValue());
```

implementamos callable en lugar de runnable

usamos call en lugar de run, que devuelve integer

creamos el objeto de la clase

Guardamos en future, de tipo Future<integer>

lo obtenemos

Ejecución Asíncrona a Futuro: Ejemplo II

```
20          }
21      }
```

Ejecución Asíncrona a Futuro: Callable y Expresiones λ

- Es posible soportar el código del método `V call()` mediante un expresión λ
- Es similar a utilizar Runnable con una expresión, λ pero no igual. La expresión λ ahora debe retornar un resultado de clase

Ejecución Asíncrona a Futuro: Ejemplo I

Código: codigos_t2/FactorialTaskLambda.java

```
1  import java.util.concurrent.*;
2
3  public class FactorialTaskLambda{
4      public static void main(String[] args) throws Exception{
5          int number=5;
6          Callable<Integer> computation = () ->{int fact = 1;
7                                  for(int count = number; count > 1;
                                       count--) {
8                                    fact = fact * count;}
9                                  return (new Integer(fact));
10                                };
11         ExecutorService exec = Executors.newFixedThreadPool(1);
12         Future<Integer> future = exec.submit(computation);
13         exec.shutdown ();
14         System.out.println(future.get().intValue());
15     }
16 }
```

Ecuación de Subramanian

- Decidir cuántas hebras son necesarias para resolver un problema es importante.
- Menos de las necesarias no permitirán explotar adecuadamente el número de núcleos disponibles.
- Más de las necesarias nos degradarán el rendimiento
- En este curso, determinaremos el número de hebras adecuado en función de dos variables: número de núcleos lógicos de la arquitectura y tipología del problema.
- Ambas variables se integran en la ecuación de Subramanian:
- Será analizada y utilizada exhaustivamente en clase de prácticas.

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++20.

Bibliografía


Capel, M.
Programación Concurrente y en Tiempo Real
Garceta, 2022


Eckel, B.
Thinking in Java
Prentice Hall, 2006


Goëtz *et al.*
Java Concurrency in Practice
Addison Wesley, 2006


Oaks and Wong.
Java Threads
O'Reilly, 2004