Unidad 8: Introducción a la programación multihilo

Metodología de la Programación

Curso 2021-2022

© *Candi Luengo Díez y Francisco Ortín Soler*

Bibliografía

Tutorial de Java “**Lesson: Concurrency**” disponible en:

https://docs.oracle.com/javase/tutorial/essential/concurrency/

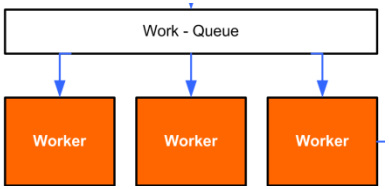
En esta unidad …

- Aprenderemos los conceptos de **concurrencia** y **paralelismo**
- Aprenderemos los conceptos de **procesos** e hilos (**thread)**
- Veremos cómo crear aplicaciones multihilo y cómo controlar el flujo de ejecución
- Conoceremos los conceptos básicos sobre la **exclusión mutua** y cómo resolver problemas básicos de concurrencia.

Principales conceptos

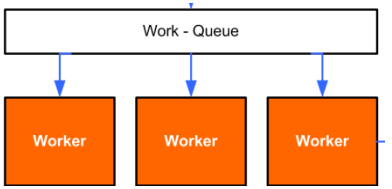
- Concurrencia y Paralelismo
- Procesos e hilos (threads)
- Ciclo de vida de un thread
- Ejecución de threads
- Sincronización de threads
- Exclusión mutua
- Sección crítica

Concurrencia



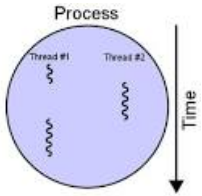
- Los ordenadores pueden hacer varias tareas a la vez
  - Procesando textos
  - Descargando ficheros de internet
  - Imprimiendo un documento
  - …
- Incluso una única aplicación puede hacer más de una cosa a la vez
  - Este tipo de software se dice que es **concurrente**
- **La concurrencia** es una propiedad de los sistemas computacionales donde varias tareas se ejecutan de forma simultánea, y potencialmente pueden interactuar unas con otras.

Concurrencia y paralelismo

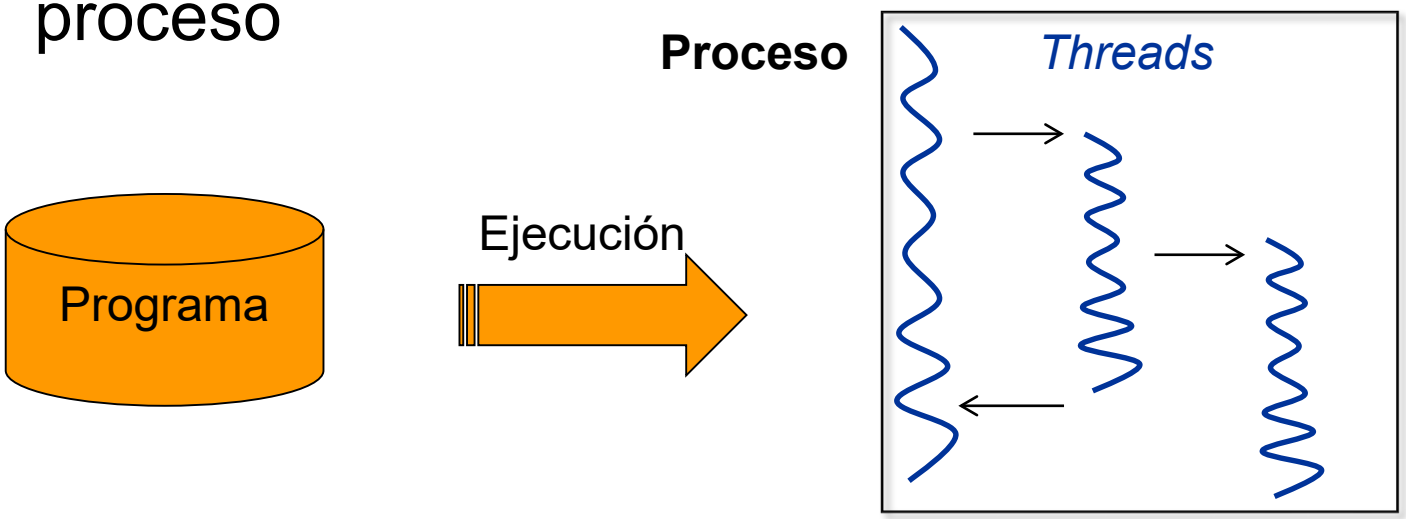


- **Paralelismo** es un escenario particular de la concurrencia, donde las tareas se ejecutan en paralelo
  - Con la concurrencia, se puede simular la simultaneidad
  - Con el paralelismo, la simultaneidad es real
- El paralelismo hace hincapié en la división de los problemas grandes en otros más pequeños
- La concurrencia hace hincapié en la iteración entre tareas

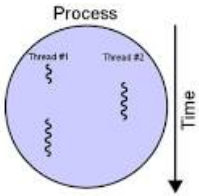
Procesos e hilos



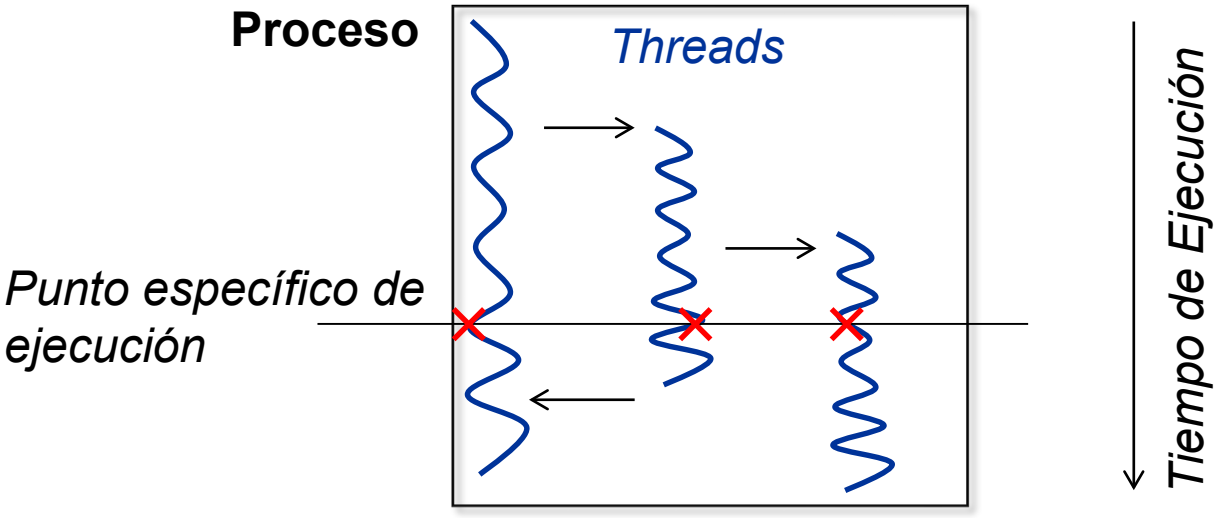
- Un **proceso** es una instancia de un programa que se está ejecutando
- Un proceso puede estar formado por multiples hilos
- Un hilo (**thread)** es una tarea que se ejecuta concurrentemente con el resto de los hilos de ese proceso

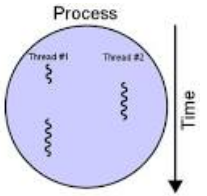


Aplicaciones multihilo



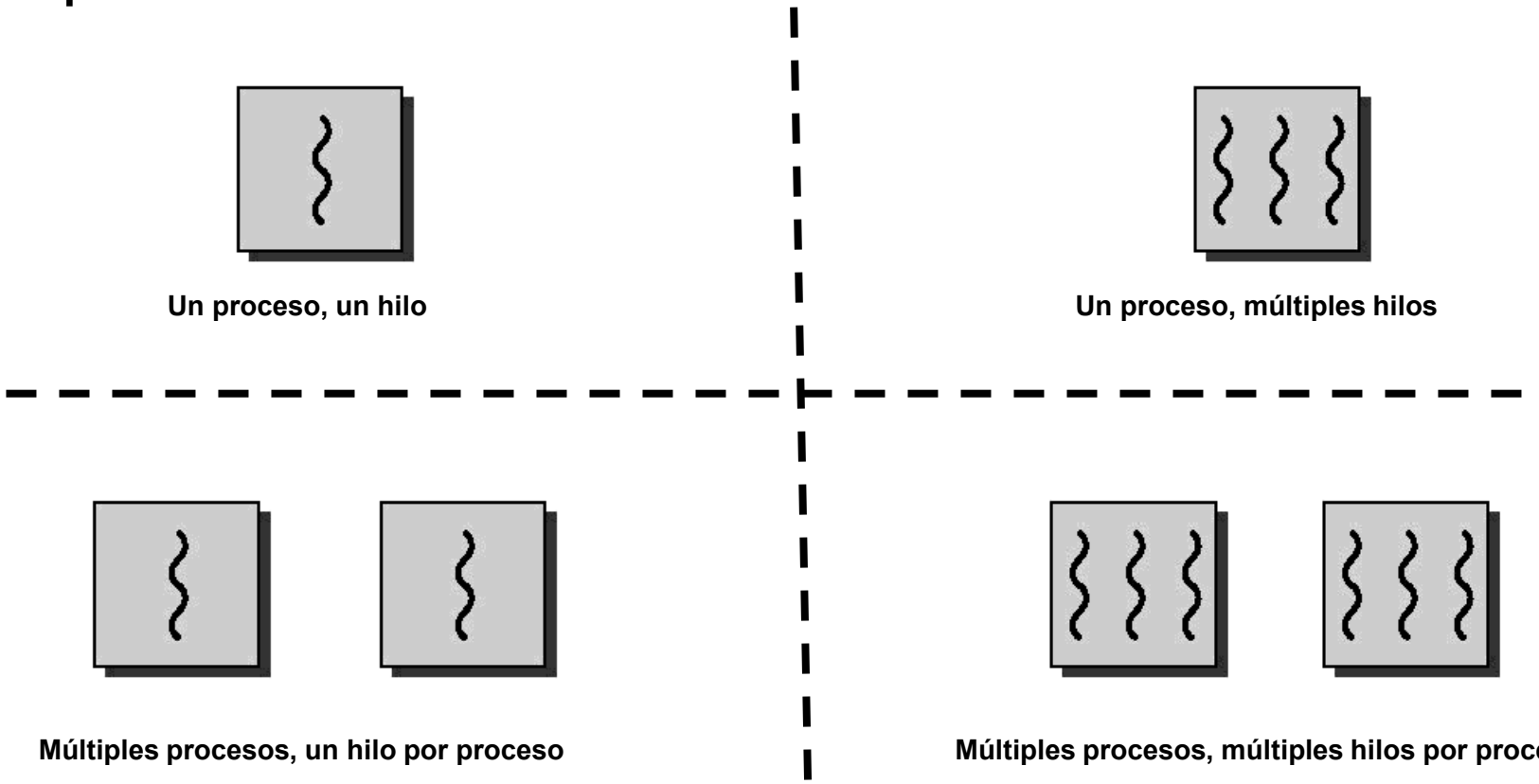
- Los programas que hemos desarrollado hasta ahora son **secuenciales**
  - Tienen solamente un único hilo
  - Durante la ejecución de un programa hay un único punto de ejecución
- Las **aplicaciones multihilo** pueden tener más de un punto de ejecución (durante el tiempo de ejecución)



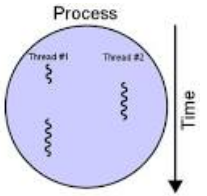


Aplicaciones multihilo

- Existen múltiples posibilidades con hilos y sus procesos

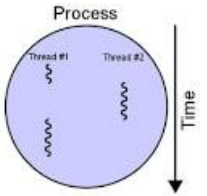


Creación de múltiples hilos



- Casi todas las clase referentes al manejo de hilos se encuentran en el paquete `java.lang`
- Existen dos formas de crear hilos en java
  - Extendiendo la clase **`java.lang.Thread`**
  - Implementando la interface **`java.lang.Runnable`**

Primera- Aplicación multitarea



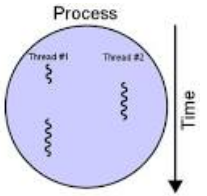

```
public class HelloThread extends Thread {
  private int iterations;
  public HelloThread(int iterations) {
    this.iterations = iterations;  }
  @Override public void run() {
    for(int i=0; i<this.iterations; i++)
      System.out.printf("Hola desde el Thread %d!\n", i);
} }
```

Option 1

```
public class HelloRunnable implements Runnable {
  private int iterations;
  public HelloRunnable(int iterations) {
    this.iterations = iterations;  }
  @Override public void run() {
    for(int i=0; i<this.iterations; i++)
      System.out.printf("Hola desde el Runnable %d!\n", i);
} }
```

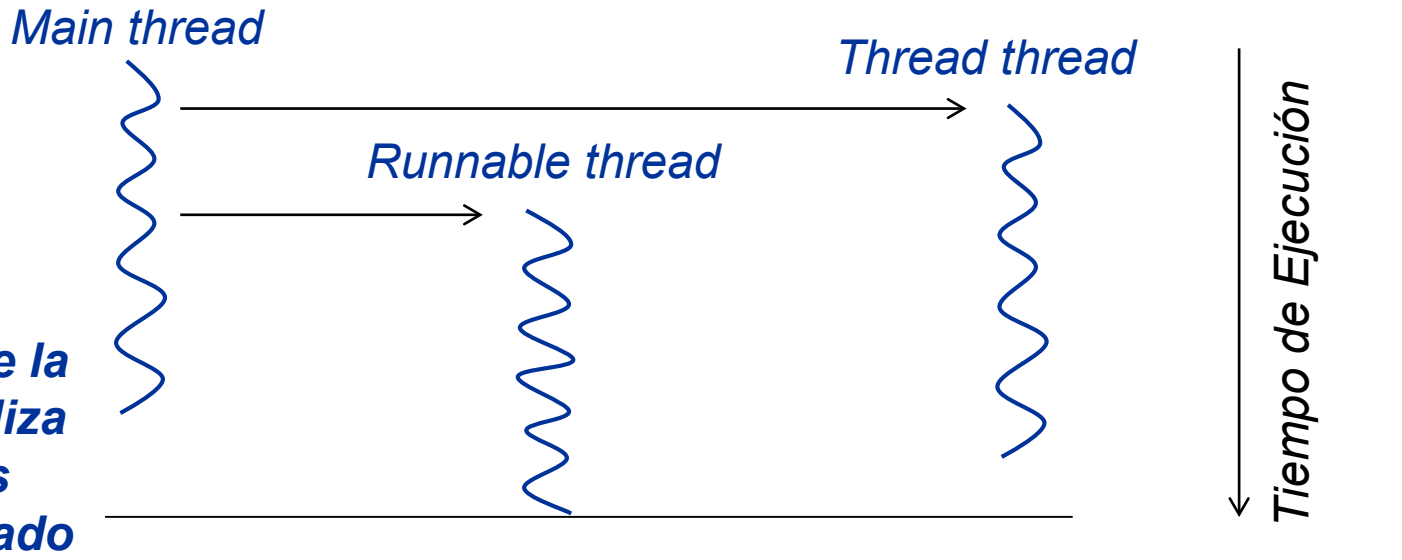
Option 2

Primera- Aplicación multitarea



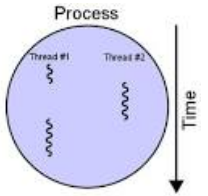

```
public class SimpleMultithreadApp {
  public static void main(String args[]) {
    final int iterations = 10;
    new HelloThread(iterations).start();
    new Thread(new HelloRunnable(iterations)).start();
    System.out.println("Aquí finaliza el
                        main thread.");
    }
}
```

Start != Run



La ejecución de la aplicación finaliza cuando los tres hilos han acabado

Primera- Aplicación multitarea



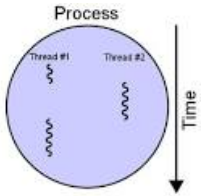
- ¿Que se muestra en consola? Tres posibles salidas

```
Hola desde el Thread 0!
Hola desde el Thread 1!
Hola desde el Thread 2!
Hola desde el Thread 3!
Hola desde el Thread 4!
Hola desde el Runnable 0!
Hola desde el Runnable 1!
Hola desde el Runnable 2!
Hola desde el Runnable 3!
Hola desde el Runnable 4!
Hola desde el Runnable 5!
Hola desde el Runnable 6!
Hola desde el Runnable 7!
Hola desde el Runnable 8!
Hola desde el Runnable 9!
“Aquí finaliza el main thread.
Hola desde el Thread 5!
Hola desde el Thread 6!
Hola desde el Thread 7!
Hola desde el Thread 8!
Hola desde el Thread 9!
```

```
“Aquí finaliza el main thread.
Hola desde el Thread 0!
Hola desde el Thread 1!
Hola desde el Thread 2!
Hola desde el Thread 3!
Hola desde el Thread 4!
Hola desde el Thread 5!
Hola desde el Thread 6!
Hola desde el Thread 7!
Hola desde el Thread 8!
Hola desde el Thread 9!
Hola desde el Runnable 0!
Hola desde el Runnable 1!
Hola desde el Runnable 2!
Hola desde el Runnable 3!
Hola desde el Runnable 4!
Hola desde el Runnable 5!
Hola desde el Runnable 6!
Hola desde el Runnable 7!
Hola desde el Runnable 8!
Hola desde el Runnable 9!
```

```
Hola desde el Runnable 0!
Hola desde el Runnable 1!
Hola desde el Runnable 2!
Hola desde el Runnable 3!
Hola desde el Runnable 4!
“Aquí finaliza el main thread.
Hola desde el Runnable 5!
Hola desde el Runnable 6!
Hola desde el Thread 0!
Hola desde el Thread 1!
Hola desde el Thread 2!
Hola desde el Thread 3!
Hola desde el Thread 4!
Hola desde el Thread 5!
Hola desde el Thread 6!
Hola desde el Runnable 7!
Hola desde el Runnable 8!
Hola desde el Runnable 9!
Hola desde el Thread 7!
Hola desde el Thread 8!
Hola desde el Thread 9!
```

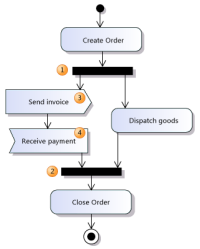
Orden de la ejecución



- La ejecución de una aplicación multihilo **no es determinista**:

  cada vez que se ejecuta el programa, diferentes hilos pueden ejecutar distintos bloques en tiempos diferentes
- El orden de ejecución de los hilos depende del planificador (**Scheduler**) de la Maquina Virtual de Java
- Por lo tanto, son necesarias **herramientas de sincronización** para sincronizar la ejecución de diferentes hilos

Join

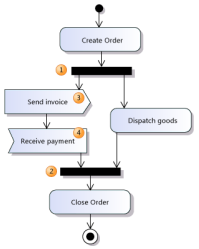


- El método `join` de la clase `Thread` es probablemente el mecanismo de sincronización más básico

```
myThread.join();
```

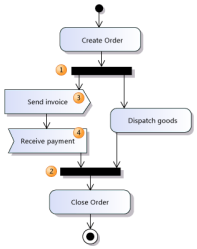
- La llamada a este método hace que el thread actual detenga la ejecución hasta que termine el hilo `myThread`
- Pregunta: ¿Cómo podemos modificar el ejemplo anterior para asegurarse de que el mensaje que muestra el método main() sea el último?
  (*se muestre después de los mensajes de los otros dos métodos run()*)

Join




```
public class SimpleMultithreadApp {
  public static void main(String args[])
                              throws InterruptedException {
    final int iterations = 10;
    Thread thread1 = new HelloThread(iterations);
    Thread thread2 = new Thread(
                     new HelloRunnable(iterations));
    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();
    System.out.println ("Aquí finaliza el
                         main thread.");
  }
}
```

Join



- Dos salidas reales de la nueva versión:

```
Hola desde el Thread 0!
Hola desde el Thread 1!
Hola desde el Thread 2!
Hola desde el Thread 3!
Hola desde el Thread 4!
Hola desde el Thread 5!
Hola desde el Runnable 0!
Hola desde el Runnable 1!
Hola desde el Runnable 2!
Hola desde el Runnable 3!
Hola desde el Runnable 4!
Hola desde el Runnable 5!
Hola desde el Runnable 6!
Hola desde el Runnable 7!
Hola desde el Runnable 8!
Hola desde el Runnable 9!
Hola desde el Thread 6!
Hola desde el Thread 7!
Hola desde el Thread 8!
Hola desde el Thread 9!
Aquí finaliza el main thread.
```

```
Hola desde el Runnable 0!
Hola desde el Runnable 1!
Hola desde el Runnable 2!
Hola desde el Runnable 3!
Hola desde el Runnable 4!
Hola desde el Runnable 5!
Hola desde el Runnable 6!
Hola desde el Runnable 7!
Hola desde el Runnable 8!
Hola desde el Runnable 9!
Hola desde el Thread 0!
Hola desde el Thread 1!
Hola desde el Thread 2!
Hola desde el Thread 3!
Hola desde el Thread 4!
Hola desde el Thread 5!
Hola desde el Thread 6!
Hola desde el Thread 7!
Hola desde el Thread 8!
Hola desde el Thread 9!
Aquí finaliza el main thread.
```

Sleep

- En determinados escenarios, es interesante la suspension de un hilo por un periodo de tiempo especificado.
- Se dice en este caso que el hilo duerme (**sleep)**
- Es una forma eficaz de **conseguir que el tiempo de procesador esté disponible** para otros hilos que pueden estar ejecutándose.
- Java ofrece esta funcionalidad mediante el método **`Thread.sleep`**(*`milliseconds`*)
- El “sueño” se puede interrumpir, por lo que la excepción `InterruptedException` puede ser lanzada.

Sleep

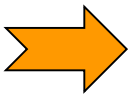

```
public class ShowMessagesThread implements Runnable {
  static void threadMessage(String message) {
    String threadName=Thread.currentThread().getName();
    System.out.printf("%s: %s%n", threadName, message);
  }
  private String[] messages;
  public ShowMessagesThread(String[] messages) {
    this.messages = messages;  }
  @Override public void run() {
    try {
      for (int i = 0; i < messages.length; i++) {
        Thread.sleep(1000);
        threadMessage(messages[i]);
    } } catch (InterruptedException e) {
      threadMessage("No se terminé!");
} } }
```

Sleep


```
public class ShowMessages {

  public static void main(String args[])
                    throws InterruptedException {
    long espera = 3000; // Por defecto, 3 segundos
    if (args.length > 0) {
      try {
        espera = Long.parseLong(args[0]) * 1000;
      } catch (NumberFormatException e) {
        System.err.println("El argumento debe ser un
                                  entero.");
      }
    } // fin del main
```

sigue



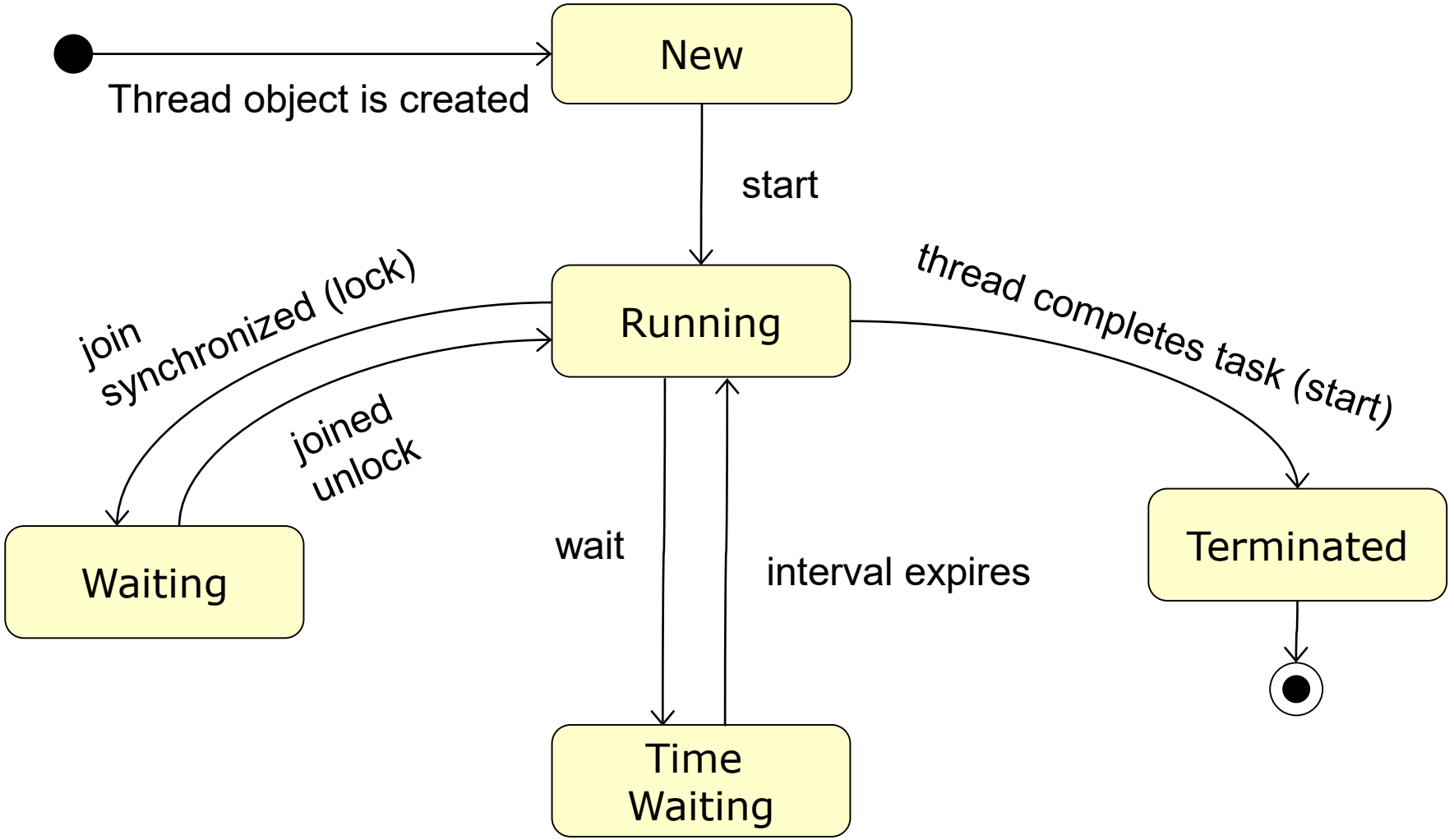
Sleep


```
ShowMessagesThread.threadMessage("Iniciando el thread
   ShowMessages.");
   Thread thread = new Thread(new ShowMessagesThread(
    new String[]{
       " Algebra Lineal ", "Cálculo", "Empresa",
       "Fund. de Informática", "Int. a la programación"
     }));
   thread.start();
   ShowMessagesThread.threadMessage("Esperando a que
                          finalice el thread ShowMessages.");
   thread.sleep(espera);
   ShowMessagesThread.threadMessage("Cansado de esperar");
   thread.interrupt();
   thread.join();
   ShowMessagesThread.threadMessage("Final!");
} }  //fin de la clase ShowMessages
```

Salida


```
main: Iniciando el thread ShowMessages.
main: Esperando a que finalice el thread
      ShowMessages.
Thread-0: Algebra Lineal
Thread-0: Cálculo
Thread-0: Empresa
main: Cansado de esperar!
Thread-0: No terminé!
main: Final!
```

Ciclo de vida de un hilo



Ejercicio

- Analiza el siguiente código, e indica que se muestra en la consola

```
class Counter {
  private int counter = 0;
  public void increment() {
    counter = counter + 1;
  }
  public void decrement() {
    counter = counter - 1;
  }
  public int getValue() {
    return counter;
} }
```

Ejercicio (II)

```
class Increment implements Runnable {

  private Counter counter;

  public Increment(Counter counter) {
    this.counter = counter;
  }

  @Override
  public void run() {
    for(int i=0; i<CounterApp.ITERATIONS; i++)
      counter.increment();
  }
}
```

Ejercicio (III)

```
class Decrement implements Runnable {

  private Counter counter;

  public Decrement(Counter counter) {
    this.counter = counter;
  }

  @Override
  public void run() {
    for(int i=0; i<CounterApp.ITERATIONS; i++)
      counter.decrement();
  }
}
```

Ejercicio (IV)

```
class CounterApp {
  public static final int ITERATIONS = 10000;

  public static void main(String... args)
                    throws InterruptedException {
    Counter counter = new Counter();
    Thread increment = new Thread(new Increment(counter));
    Thread decrement = new Thread(new Decrement(counter));
    increment.start();
    decrement.start();
    increment.join();
    decrement.join();
    System.out.printf("El valor del contador es %d.",
                                          counter.getValue());
  }
}
```

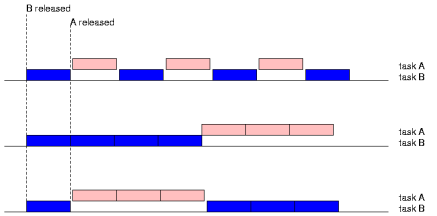
La ejecución concurrente

- Los métodos que son ejecutados concurrentemente son `increment` y `decrement` (de la clase `Counter`)
- Cada método se ejecuta en tres pasos:
  1. Recuperar el valor del atributo `counter`
  2. Incrementar / decremenar el valor recuperado
  3. Almacenar el resultado en el atributo `counter` (asignación)
- Es muy simple!
- Sin embargo, estos dos métodos se ejecutan ahora simultáneamente

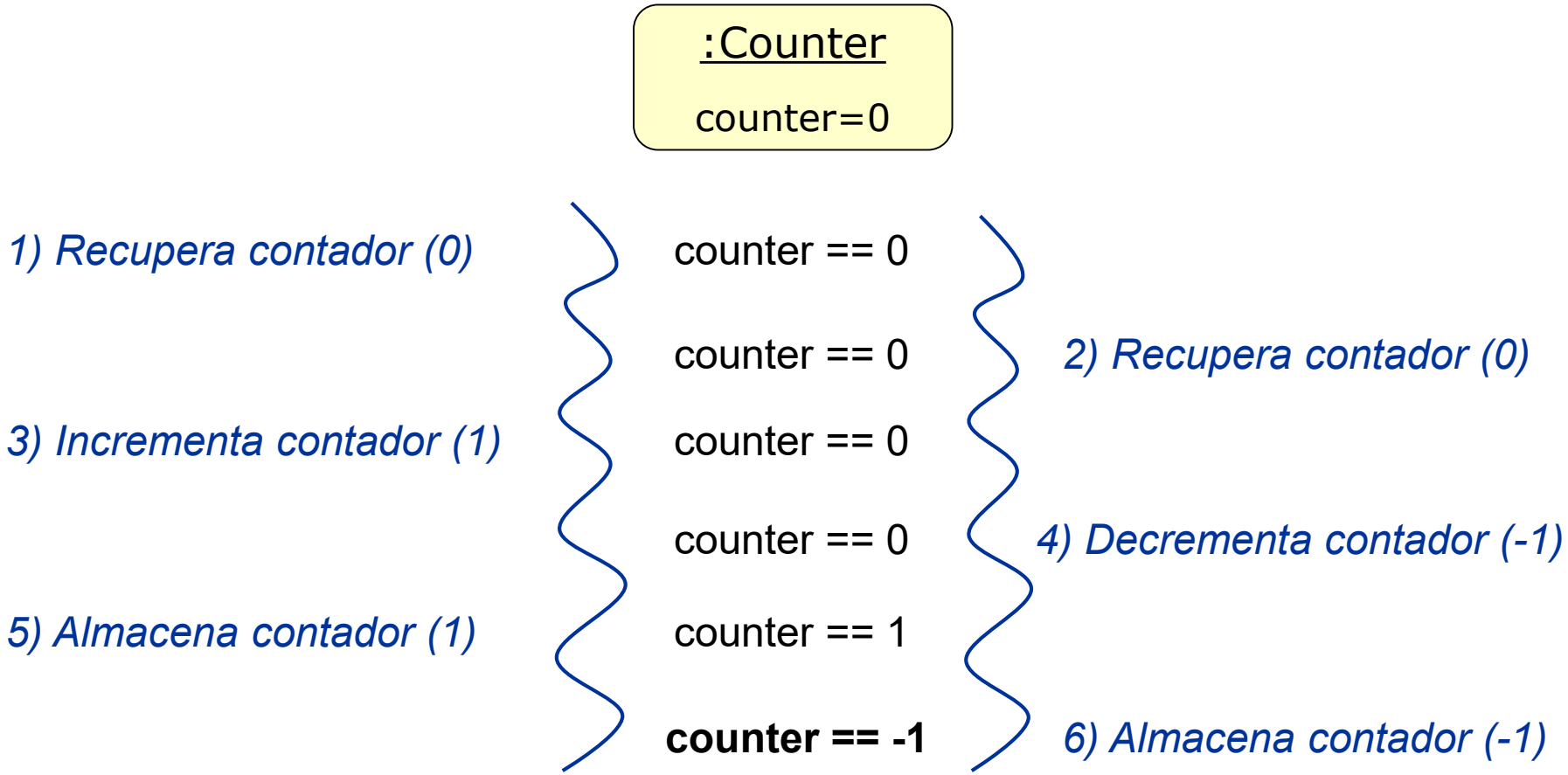
```
public void increment() {
    counter = counter + 1;
}
```

```
public void decrement() {
    counter = counter - 1;
}
```

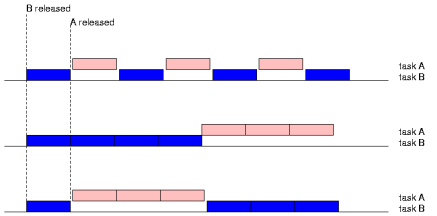
La ejecución concurrente



- Lo siguiente puede ser una traza de la ejecución concurrente:

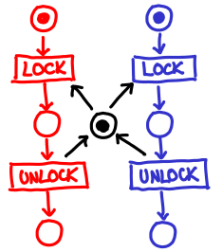


Intercalar e interferir



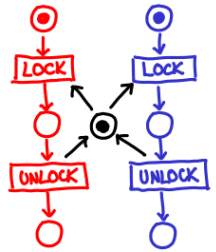
- Los dos métodos, `increment` y `decrement`, se **intercalan** porque se están ejecutando en diferentes threads, pero actúan sobre el mismo dato (recurso).
- Esto significa que los dos métodos realizan varios pasos, y se superponen las secuencias de pasos
- En este caso, los dos threads **interfieren** el uno con el otro
- Habitualmente sucede cuando dos **threads acceden al mismo recurso** (counter en nuestro ejemplo)

Exclusión mutua



- Cuando dos métodos interfieren necesitan estar en **Exclusión Mutua**: evitar el uso simultáneo de un recurso común
- El trozo de código que tiene acceso a un recurso compartido y deben ser mutuamente excluyente se llama **Sección Crítica**
- La exclusión mutua se obtiene mediante el mecanismo de sincronización

Monitores

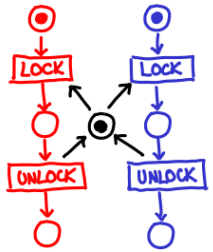


- El mecanismo de sincronización en Java se basa en el concepto de **Monitor**:

  Object (o módulo) destinado a ser utilizado con seguridad por más de un thread

- Los métodos en un monitor se ejecutan en exclusión mutua
  - No es posible que se intercalen dos métodos en un monitor
- Los monitores fueron inventados por C.A.R. Hoare y P.B. Hansen

Synchronized



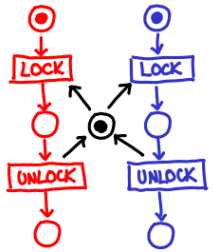
- En Java, se usa la palabra `synchronized`

```
class Counter {
  private int counter = 0;

  public synchronized void increment() {
    counter = counter + 1;
  }

  synchronized public void decrement() {
    counter = counter - 1;
  }
}
```

Beneficios



- Con estas dos simples extensiones (añadiendo `synchronized` a `increment` y `decrement`) la aplicación muestra 0 en la consola!
- Los dos métodos nunca se ejecutan concurrentemente sobre el **mismo objeto** (`this`)
- `synchronized` es un mecanismo simple para obtener exclusión mutua, identificando las secciones criticas
- Se debe usar cuando **un recurso común se utiliza simultáneamente** por más de un thread
- La elección del siguiente thread (en estado waiting) para ser ejecutado depende del planificador

Pregunta

- ¿El método que devuelve el valor del atributo `counter` debería tener **synchronized**?.

```
class Counter {
 private int counter = 0;
 public synchronized void increment() {
    counter = counter + 1;
  }
 public synchronized void decrement() {
    counter = counter - 1;
  }
 public synchronized? int getValue() {
    return counter;
  }
}
```

Revisión

- La programación concurrente es importante para obtener mejores rendimientos en los sistemas multiprocesador
- Se debe usar `Runnable` (or `Thread`) para encapsular el comportamiento de los threads en los objetos
- Se necesita la sincronización cuando dos threads acceden a un recurso compartido
- Se usa `synchronized` en Java para obtener exclusión mutua
- Sólo hemos aprendido la base de la programación multihilo ☺