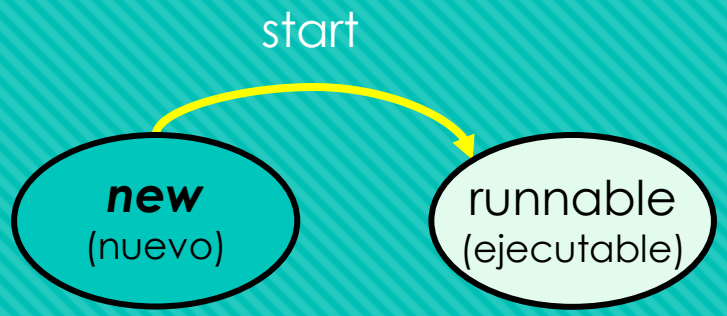


02. Programación multihilo

Programación de Servicios y procesos - 2º DAM

Luis del Moral Martínez

versión 20.10

Contenidos del tema

1. **Creación de hilos en Java**
   - 1.1 ¿Qué es un hilo?
   - 1.2 Clase Thread
   - 1.3 Interfaz Runnable
   - 1.4 Estados de un hilo
   - 1.5 Gestión de hilos
   - 1.6 Gestión de prioridades
   - 1.7 Comunicación y sincronización
   - 1.8 El modelo productor-consumidor

1.1 ¿Qué es un hilo?

Definición de hilo

- Un hilo es una secuencia de código de ejecución dentro del contexto de un proceso
- Los hilo son pueden existir sin el proceso
- Dentro de un proceso puede haber varios hilos ejecutándose
- En Java tenemos dos formas de hacerlo:
  - **Extender la clase Thread**
  - **Implementar la interfaz Runnable**

1.2 Clase Thread

La clase Thread

- Permite añadir la funcionalidad de hilo a una clase
- Hay que sobrescribir el método **run()**
- Se puede iniciar la ejecución del hilo con **start()** y detenerla con **stop()**
- **Algunos métodos útiles**: start, isAlive, run, toString, getID, yield, setPriority, interrupt
- **Más información**: enlace (Clase Thread, Javadoc Java SE 10)

1.3 Interfaz Runnable

Interfaz Runnable

- Permite añadir la funcionalidad de hilo a una clase (que hereda de otra clase)
- La interfaz proporciona el método **run()**, que debe ser implementado
- Se puede iniciar la ejecución del hilo con **start()** y detenerla con **stop()**
- **Más información**: enlace (Interfaz Runnable, Javadoc Java SE 10)

1.4 Estados de un hilo

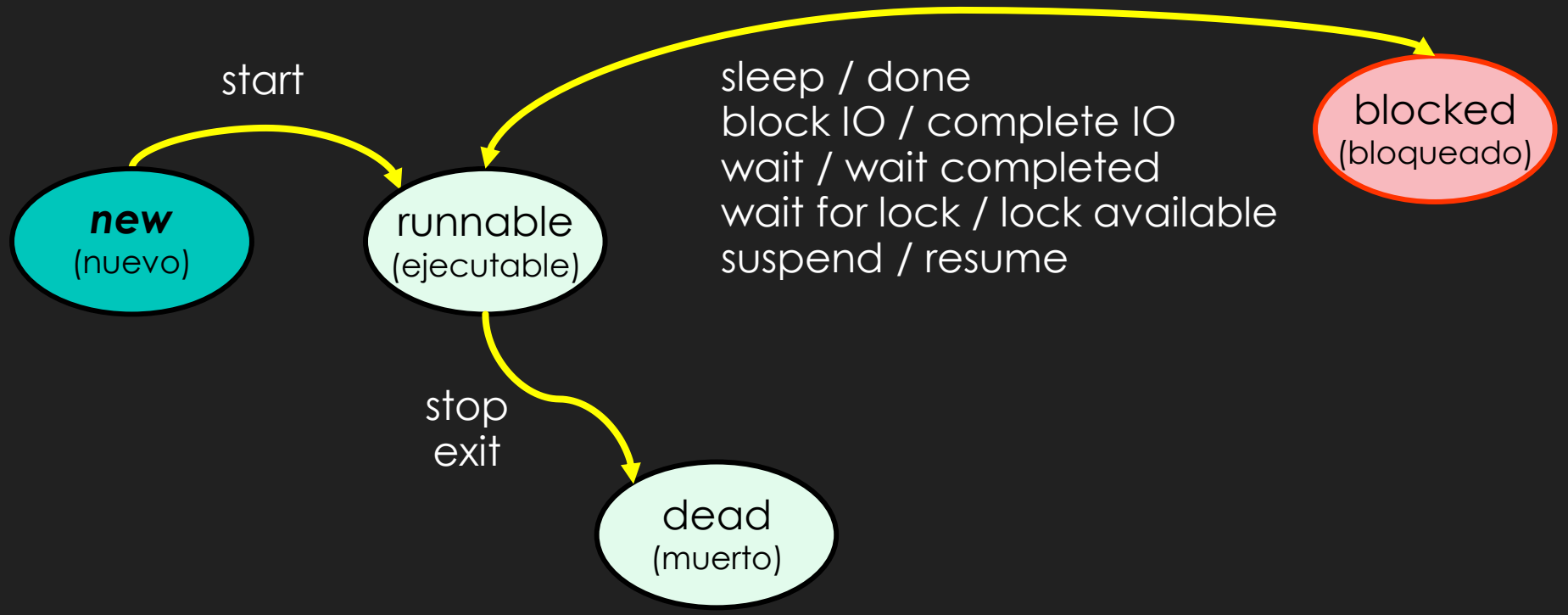
Esquema de los estados de un hilo

- Estados de **ejecución** de un hilo:
  - **New (nuevo)**: el hilo se encuentra en este estado tras ejecutar el constructor (con el operador new)
  - **Runnable (ejecutable)**: cuando se invoca al método start() el hilo pasa a este estado
  - **Dead (muerto)**: el hilo ha finalizado
    - Finaliza el método **run()** o se ha producido alguna excepción no capturada
    - Se ha invocado a **stop()** (es más recomendable usar una condición de parada)
  - **Blocked (bloqueado)**: el hilo suspende su ejecución
    - Se invoca el método **sleep()**, el hilo se bloquea por una E/S, el hilo invoca **wait()**, se suspende…

1.4 Estados de un hilo

Esquema de los estados de un hilo

- Diagrama de estados de **ejecución** de un hilo (simplificado)



1.5 Gestión de hilos

Crear y arrancar un hilo

- Extender la clase **Thread** o implementar la interfaz **Runnable**
- Para arrancar el hilo se ejecuta el método **start()**
  - Si extiende la clase **Thread**

```
Hilo h = new Hilo ();
h.start();
```

  - Si implementa la interfaz **Runnable**

```
Hilo h = new Hilo ();
new Thread(h).start();
```

1.5 Gestión de hilos

Suspender un hilo

- Para dormir un hilo usamos **sleep()**, indicando el número de milisegundos
- Para suspender un hilo:
  - **Forma antigua (y obsoleta)**: suspensión con **suspend()** y reactivación con **resume()**
  - **Forma correcta**: introducir una variable que permita detener la ejecución del hilo de forma segura
    - Esta variable se manipula en un **método sicnronizado** (**synchronized**)

1.5 Gestión de hilos

Parar un hilo

- El método **stop()** detiene la ejecución de un hilo de forma permanente
- Si se detiene con **stop()** no se puede volver a iniciar con **start()**
- Este método podría provocar un interbloqueo (están deprecados)
- Lo correcto sería lanzar una interrupción (**InterruptedException**) con **interrupt()**
- Finalmente, **join()** fuerza al hilo que espere que el resto de hilos acaben su ejecución

1.6 Gestión de prioridades

Prioridades del hilo

- Cada hilo tiene su propia **prioridad** (por defecto la del hilo padre).
- Podemos consultar la prioridad con **getPriority()** y cambiarla con **setPriority()**
- La prioridad es un valor entre **1** (el valor máximo) y **10** (el valor mínimo)
- El **planificador** escoge el siguiente hilo que se ejecutará en función de su prioridad
- El hilo de mayor prioridad sigue funcionando hasta que:
  - Cede el control al planificador usando el método **yield()**
  - Deja de ser ejecutable (por finalización o bloqueo)
  - Un hilo de mayor prioridad se convierte en ejecutable

1.6 Gestión de prioridades

Prioridades del hilo

- Hay que tener en cuenta que las prioridades **no son garantías de ejecución**
- Depende de la **plataforma** en la que se ejecuten los programas
- Depende de la **carga** del sistema
- En realidad, casi nunca se debe establecer a mano la prioridad
- Los sistemas operativos modernos están preparados para evitar que un hilo acapare la CPU

1.7 Comunicación y sincronización

Necesidad de comunicación y sincronización entre hilos

- A menudo los hilos necesitan **comunicarse** entre sí
- Este problema se resuelve **compartiendo uno o varios objetos** (ejemplo: Contador)

1.7 Comunicación y sincronización

Bloques sincronizados

- Los **bloques sincronizados** permiten que las operaciones se hagan de forma **atómica**
- Estos bloques permiten se utilizan para implementar las **secciones críticas** de los algoritmos
- Un bloque sincronizado se crea de la siguiente forma

```
synchronized (objetoCompartido) {
    // Sección crítica
}
```

- Cuando un hilo accede a una **sección crítica** pregunta al objeto compartido
- Si el objeto está bloqueado por otro hilo, el hilo actual se **suspende** a la espera

1.7 Comunicación y sincronización

Métodos sincronizados

- Debemos evitar los **bloques sincronizados** en la medida de lo posible
- En su lugar, se recomienda **sincronizar** los métodos, añadiendo la palabra clave **synchronized**
- No se pueden invocar dos métodos sincronizados a la vez
- Ejemplo:

```
public synchronized void incrementar () {
        variable++;
}
```

1.7 Comunicación y sincronización

Bloqueo de hilos

- Adicionalmente, para coordinar varios hilos se usan los siguientes métodos:
  - **objeto.wait()**: se suspende el hilo hasta que otro hilo llame a **notify()** o **notifyAll()** del mismo objeto
  - **objeto.notify()**: despierta arbitrariamente a uno de los hilos que realizó una llamada a wait()
  - **objeto.notifyAll()**: despierta a todos los hilos que están esperando el objeto
- Los métodos **notify()** y **wait()** sólo pueden ser invocados en un método o bloque sincronizado

1.8 El modelo productor-consumidor

Problemas de programación concurrente

- Es un problema clásico de programación concurrente
- Uno o más hilos producen datos y otros los consumen
- Analicemos varios ejemplos de este escenario
- Otros problemas de programación concurrente: cena de filósofos

Créditos de las imágenes y figuras

Diagramas, gráficas e imágenes

- Se han desarrollado en PowerPoint y se han incrustado en esta presentación
- Todos estos materiales se han desarrollado por el autor
- Para el resto de recursos se han especificado sus fabricantes, propietarios o enlaces
- Si no se especifica copyright con la imagen, entonces es de desarrollo propio o CC0