# UT2-PROGRAMACIÓN MULTIHILO

# EL OBJETO THREAD

# HILO

- Hilo: diferente flujo de ejecución dentro de un proceso
- Los hilos dentro de un proceso comparten todos sus recursos
  - Memoria
  - Ficheros
  - Permite acceso eficiente pero potencialmente peligroso
- Los hilos son menos pesados que los procesos
- Toda aplicación tiene al menos un hilo, el principal que puede crear otros hilos

# CREAR HILOS

- Extender clase Thread
- Sobrecargar método run
- Crear una instancia de la clase y llamar a método start

# CREAR HILOS

- Implementar interfaz Runnable
- Implementar método run
- Crear una instancia de la clase y llamar a método start

# CREAR HILOS

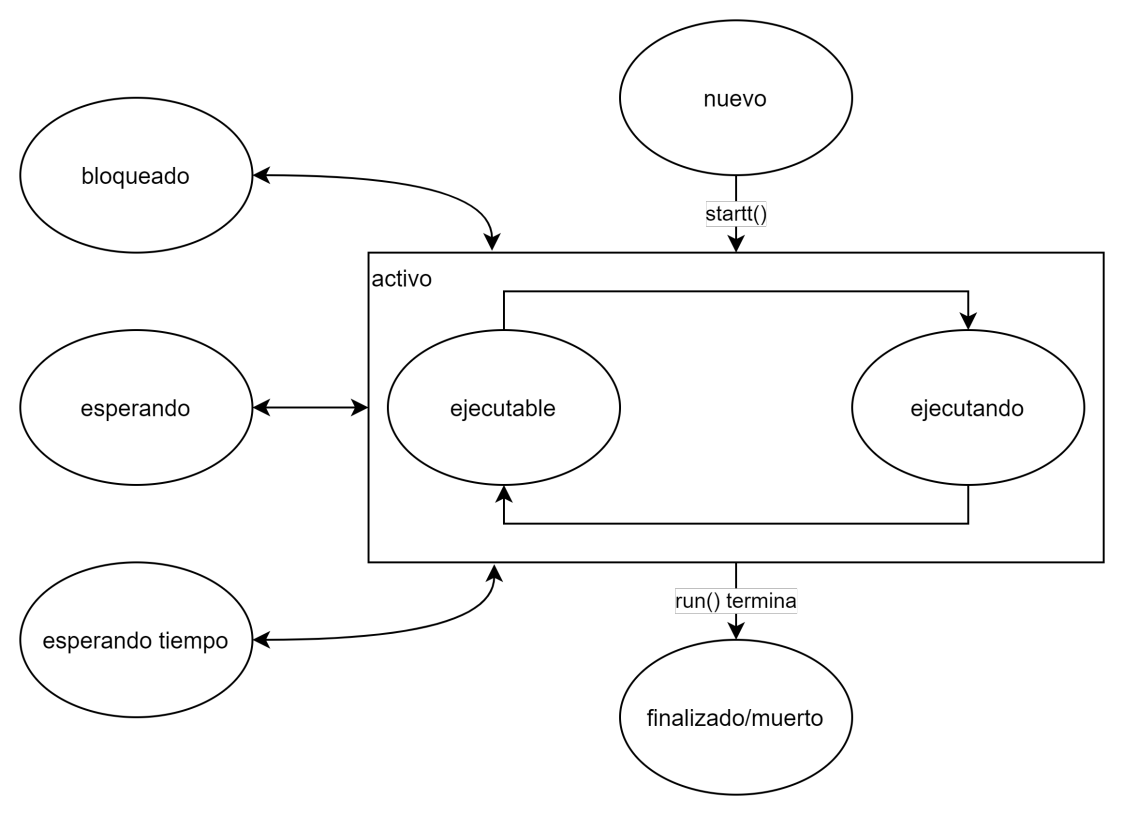
- Pasar una implementación anónima del interfaz Runnable al constructor de Thread
  - O usar una lambda
- Llamar al método start del Thread

# PRACTICAMOS

- Crear hilos mediante los tres métodos presentados

# CICLO DE VIDA DE UN HILO

# CICLO DE VIDA DE UN HILO




Thread.getState()

```
public enum State {
    NEW,
    RUNNABLE,
    BLOCKED,
    WAITING,
    TIMED_WAITING,
    TERMINATED
}
```

# INTERRUPCIONES

- Interrumpir un hilo
  - Thread.interrupt();
- Detectar que el hilo ha sido interrumpido
  - Gestionar excepción InterruptedException
  - Comprobar el booleano Thread.interrupted()
  - Al llamar a Thread.Sleep, si el hilo ha sido interrumpido previamente, salta InterruptedException

# ESPERAR A QUE MUERA UN HILO

- Thread.join()
- Thread.join(timeout)
- Ambas lanzan InterrruptedException si el hilo al que se espera ha sido interrumpido
- Thread.isAlive() nos dice si el hilo sigue vivo

# PRACTICAMOS

- Gestionar la terminación de un hilo mediante los dos métodos presentados

# SINCRONIZACIÓN DE HILOS

# INTERFERENCIA DE HILOS

- La operación c++ se divide en
  - Obtener valor de c
  - Sumarle 1
  - Guardar el valor de c
- Si tenemos dos hilos accediendo a una instancia de Counter, uno llamando a increment, el otro a decrement, se da una condición de carrera (race condition) y el resultado final es indeterminado
- Ejemplo: 4.thread.interference

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

   public void increment() {
      c++;
   }

   public void decrement() {
      c--;
   }

   public int value() {
      return c;
  }
```

# MÉTODOS SINCRONIZADOS

- El uso de la palabra clave synchronized impide que dos hilos accedan de forma intercalada a un método
- El hilo que intenta acceder mientras que otro hilo ha accedido pasa al estado BLOCKED hasta que el primer hilo termina
- Estrategia sencilla: añadir synchronized a todas las lecturas y escrituras de variables del objeto.
- Ejemplo: 5.synchronized.methods

```
public class Counter {
   private int c = 0;

   public synchronized void increment() {
      c++;
   }

   public synchronized void decrement() {
      c--;
   }

   public synchronized int value() {
      return c;
   }
}
```

# INSTRUCCIONES SINCRONIZADAS

- Cada objeto tiene un candado intrínseco
- Un hilo que quiera acceder de forma exclusiva y consistente a un campo de un objeto debe adquirir el candado y liberarlo cuando termine
- Ejemplo 6.synchronized.statements

```
public class Counter {

   private int c = 0;

   public void increment() {
      synchronized (this){
         c++;
      }
   }

   public  void decrement() {
      synchronized (this){
         c--;
      }
   }

   public int value() {
      synchronized (this){
         return c;
      }
   }
}
```

# VIVACIDAD

# VIVACIDAD

- Capacidad de un programa multihilo de ejecutarse de forma puntual (liveness)
- El principal problema que podemos encontrar es el interbloqueo (deadlock)
- Además se dan la inanición (starvation) y el bloqueo activo (livelock)

# INTERBLOQUEO

- Cuando dos o más hilos están bloqueados para siempre, esperando unos a otros
- Ejemplo 7.deadlock

# BLOQUES SINCRONIZADOS

# PRODUCTOR/CONSUMIDOR

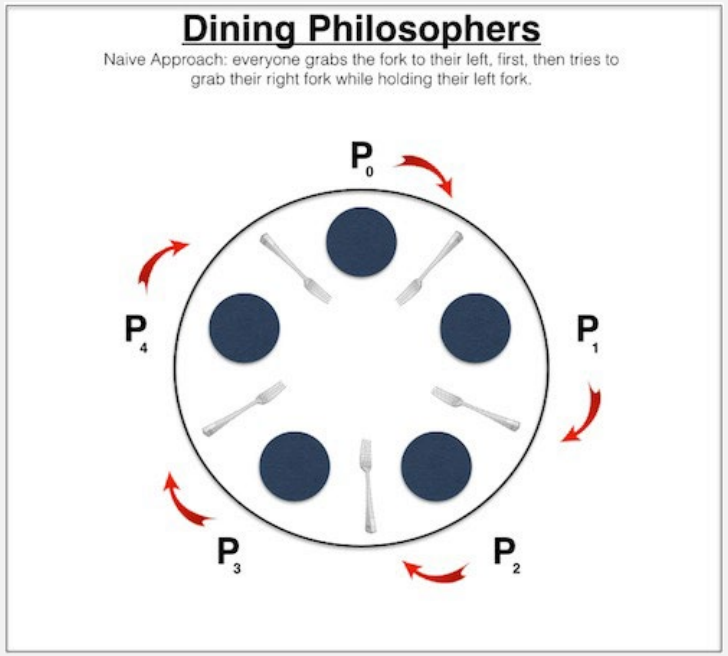
- Ejemplo clásico de sincronización entre hilos
- Un hilo consumidor debe esperar a que el hilo productor genere información para poder procesarla.
- Usamos bloques sincronizados
- En un método synchronized, esperamos a que se cumpla una condición en un bucle que contiene
  - Thread.wait(): bloquea la ejecución y libera el candado
- Desde otro hilo, llamamos  a un método synchronized, modificamos la condición y llamamos a:
  - Thread.notifyAll(): despierta a todas las hebras que estaban esperando en el candado
  - Si se interrumpe un hilo que está en wait, salta InterruptedException
- Ejemplo 8.producer.consumer

# PRACTICAMOS

- Modelar una clase sincronizada Parking en la que los coches tienen que esperar para aparcar a que halla sitio disponible
  - Constructor con máxima capacidad
  - Métodos sincronizados enter y leave reciben un Car
- Car extiende la clase thread con un bucle infinito en el que
  - Se espera un tiempo aleatorio de hasta 1 segundo
  - Intenta entrar en el parking
  - Se espera un tiempo aleatorio de hasta 1 segundo
  - Sale del parking
- Crea tantos coches como plazas más una
- Observa a los coches entrar y salir durante unos segundos antes de interrumpirlos

# LOS CINCO FILÓSOFOS

- Cinco filósofos sentados a la mesa
- Hay 5 tenedores en la mesa, uno entre cada uno
- Los filósofos meditan un rato y después comen. Para comer necesitan los dos tenedores que hay a sus lados.
- Clásico ejemplo de interbloqueo: todos cogen el tenedor de la izquierda



# INTERBLOQUEO

- Ver 10.0.filosofos.deadlock
- Solo 3 filósofos y una espera aleatoria de hasta 10ms
- Se bloquea enseguida

Philosopher 1 got both forks; chowing down

Philosopher 0 hungry; going for left fork

Philosopher 2 hungry; going for left fork

Philosopher 2 hungry; now going for right fork

Philosopher 1 finished eating; dropping left fork

Philosopher 1 finished eating; now dropping right fork

Philosopher 1 all done

Philosopher 1 contemplating the universe, working up an appetite

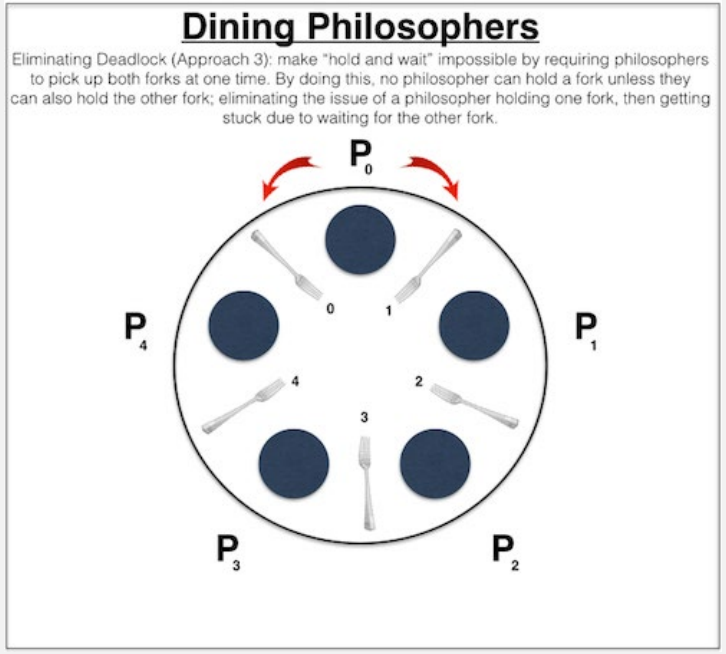
Philosopher 1 hungry; going for left fork

Philosopher 1 hungry; now going for right fork

Philosopher 0 hungry; now going for right fork

# UNA SOLUCIÓN

- Coger los dos tenedores **a la vez**
- De esta forma ningún filósofo se puede quedar con un tenedor en la mano esperando al siguiente
- No hay bloqueo mutuo
- Sí puede haber inanición
- Ver ejemplo 10.1.filosofos.starvation



# OBJETOS DE ALTO NIVEL PARA CONCURRENCIA

# CANDADOS (LOCKS)

- Proporcionan la misma funcionalidad que los candados intrínsecos de *synchronized* con algunas mejoras
- ReentrantLock
  - void lock(), pausa el hilo hasta obtener el candado
  - bool tryLock(), intenta obtener el candado y abandona si no lo consigue inmediatamente
  - bool tryLock(1,TimeUnit.SECONDS), intenta obtener el candado y abandona pasado un tiempo si no lo consigue
  - void unlock(), libera un candado que fue obtenido
    - Siempre hay que rodear el código desde que se obtiene el candado con un *try* y en el *finally* liberar el candado
- Mediante tryLock podemos evitar el bloqueo mutuo (deadlock)
- Ejemplo 10.2.filosofos.lock

# CONDICIONES (CONDITIONS)

- Proporcionan la misma funcionalidad que los candados wait, notify y notifyAll de *synchronized* con algunas mejoras
- Las Condition se obtienen de un Lock existente lock.newCondition()
  - Se usan siempre dentro de un lock que se ha obtenido
  - condition.await() análoga a wait()
  - condition.signal() análoga a notify()
  - condition.signalAll() análoga a notifyAll
- Permiten tener varias condiciones a las que esperar
- Ejemplo 11.conditions

# SEMÁFOROS (SEMAPHORE)

- Permite limitar el número de hilos que pueden acceder a un recurso
- El constructor determina el número de hilos permitidos
- Semaphore.acquire() obtiene un permiso o bloquea (análogo a wait)
- Semaphore.release() libera un permiso y avisa (análogo a notifyAll)
- Ver 12.semaphores

# EXECUTORS

- Executor es un interfaz que soporta el lanzamiento de nuevos hilos mediante el método execute(Runnable)
- ExecutorService es un interfaz que hereda de Executor y soporta gestionar el ciclo de vida de los hilos y del propio ejecutor
  - El método submit() acepta Runnable y Callable, por lo que puede devolver valores
- Las implementaciones más comunes son
  - Executors.newSingleThreadExecutor
  - Executors.newFixedThreadPool(int)
    - Utiliza un Thread Pool, una serie de hilos creados a priori para el proceso y que se reutilizan

# EXECUTORS

- Hay que finalizar los ejecutores, porque si no se quedan esperando a nuevas tareas
- void Shutdown()
- void awaitTermination(timeout)
- bool isTerminated()
- void shutdownNow()
- Ver 12.executors

# CALLABLES

- Permiten devolver valores desde un hilo
- Callable<T> es un interfaz funcional con el método T call()
- Cuando al método submit de un Executor se le pasa un Callable devuelve un Future
  - bool isDone() comprobar si ha terminado
  - T get() bloquea hasta devolver valor
- Ver 13.callables