Algoritmos paralelos I Executors

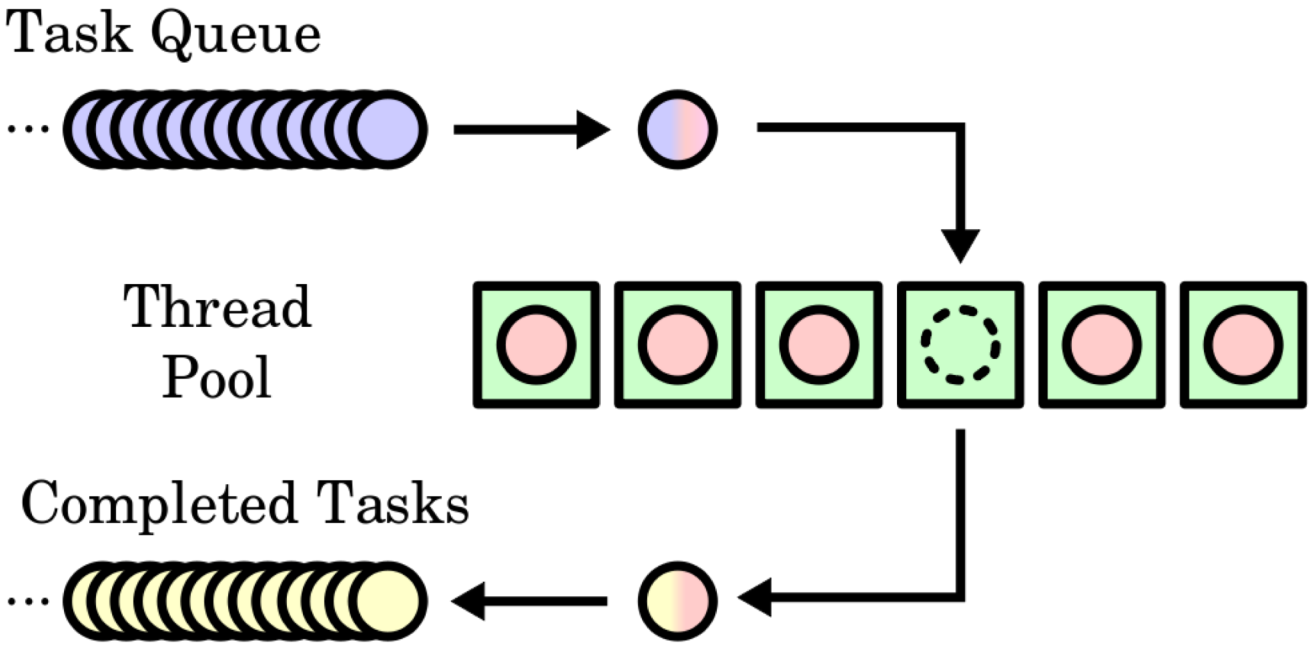
PROGRAMACIÓN CONCURRENTE EN JAVA - TEMA 5

Universidad
Rey Juan Carlos

Colas de trabajo

- Java nos permite gestionar colas de trabajo a través de los ***thread pools***.
  - Un *thread pool* se encarga de **gestionar** la ejecución de un grupo de threads.
- Contienen una **cola de trabajo** que se encarga de gestionar y esperar a que las tareas se ejecuten.
- Un *thread pool* no es más que una **colección** de objetos que implementan la interfaz *Runnable*, junto con información de los *threads* que están ejecutándose.
- Los *threads* se están **ejecutando continuamente**, comprobando si existe una nueva tarea en la cola de trabajo para realizar.
  - A través del método *execute()* de cada *thread*, es posible **modificar la tarea** (*Runnable*) que ese *thread* va a realizar.

Colas de trabajo



Colas de trabajo

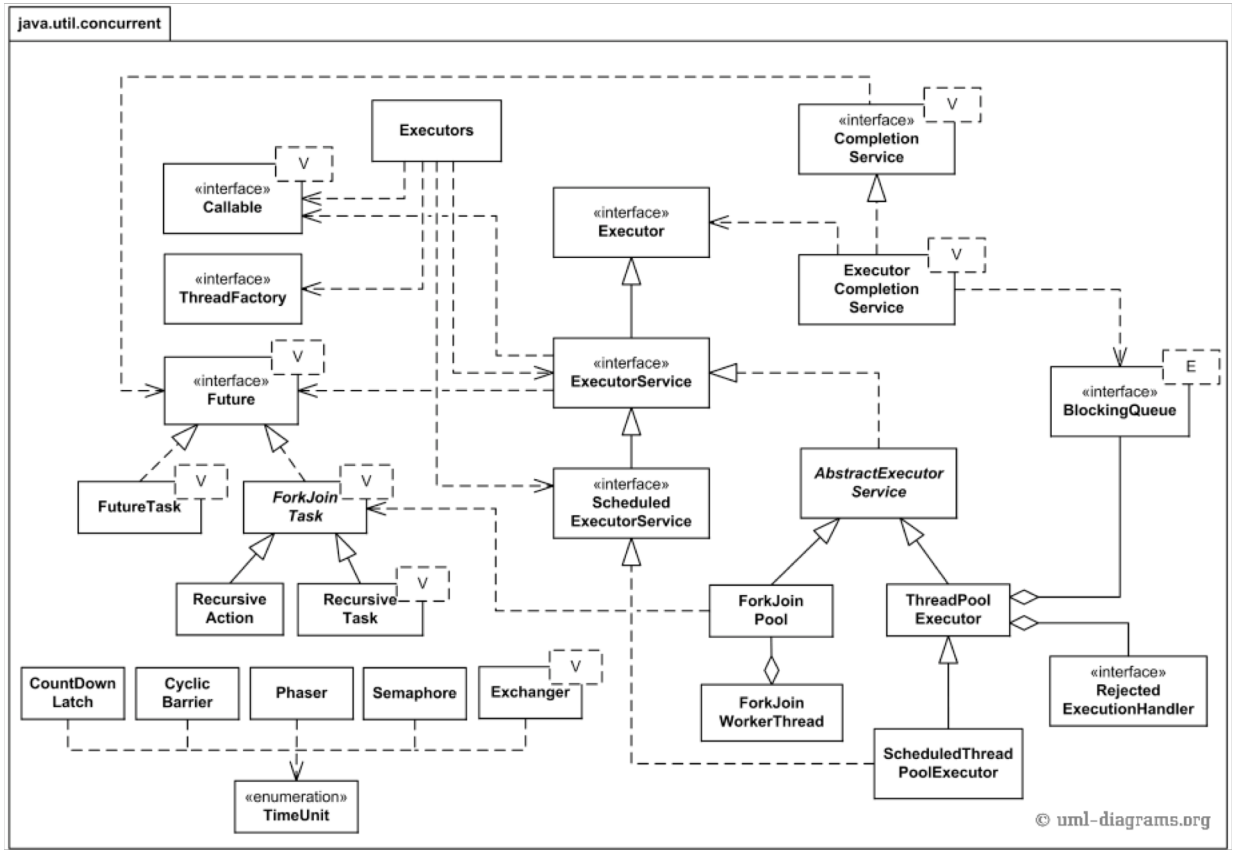
- La interfaz *Executor* nos permite **ejecutar** tareas de tipo *Runnable* en Java.
- La interfaz *ExecutorService* representa un *thread pool* en Java. Implementa la interfaz Executor, añadiendo la funcionalidad de la **gestión** del arranque / parada de los threads y su gestión.
  - Nos **permite dejar de trabajar** directamente con los *threads* como hasta ahora.

Colas de trabajo

- Debemos elegir qué **tipo** de *thread pool* necesitamos para la tarea que vamos a realizar, a través de la clase *Executors*.
  - *newSingleThreadExecutor()*
    - Crea un **único** thread, utilizado únicamente para validar el código.
  - newFixedThreadPool(int nThreads)
    - Crea un thread pool que contiene el número de threads que necesitemos. **Siempre** habrá nThreads activos.
  - newCachedThreadPool()
    - Crea un thread pool que lanzará nuevos threads **cuando sea necesario**, pero intentará **reutilizar** los anteriores cuando estén disponibles.
    - Están recomendados para aplicaciones que realizan **tareas muy cortas**.
    - Los threads que no se hayan usado durante más de 60 segundos **terminan y se eliminan del pool**.

Colas de trabajo

ALGORITMOS PARALELOS



Colas de trabajo

- Cada tarea se ejecuta mediante el método *execute()* del *ExecutorService* que hemos creado.
- Las tareas ejecutadas **no** pueden devolver ningún valor.
  - Esto se debe a que se ejecutan en el método *run()*, que según la interfaz *Runnable* debe devolver **void**
- La llamada a *execute* **no** es bloqueante.
  - Es decir, el programa principal seguirá con las siguientes instrucciones mientras se ejecutan las tareas indicadas.

Colas de trabajo

- Es necesario **finalizar** cada *ExecutorService* que utilicemos.
  - Si no lo finalizamos, y nuestro programa principal termina, se quedará **esperando** mientras el *ExecutorService* siga activo.
- Para terminar un *ExecutorService* invocamos al método *shutdown()*.
- De esta forma **indicamos** al *ExecutorService* que no acepte más tareas, pero no lo apagamos inmediatamente.
  - El programa **esperará** a que termine las tareas que ya estaba ejecutando.
- Si necesitamos finalizar el ExecutorService inmediatamente, invocamos al método shutdownNow().
  - Este método **intentará** finalizar todas las tareas iniciadas, y no ejecutará ninguna tarea enviada que aún no haya comenzado a ejecutar.
  - **No sabemos** el estado en el que terminan las tareas previas.

Ejemplo 1: Tipos de *thread pools*

- Código de la tarea

```
public void run() {
    for (int i=0;i<10;i++) {
        System.out.println("Thread "+id+", iter "+(i+1));
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Ejemplo 1: Tipos de *thread pools*

- Código del programa principal

```
ExecutorService executor =
    Executors.newSingleThreadExecutor();
for (int i=0;i<10;i++) {
    executor.execute(new Ejemplo(i+1));
}
executor.shutdown();
```

Ejercicio

- Analizar el **rendimiento** de los diferentes tipos de *thread pools*.
  - Cada thread debe comenzar escribiendo por pantalla *"Starting thread THREAD_ID"*
  - A continuación, creará una lista de enteros de gran tamaño con valores aleatorios.
  - Tras ello, recorre la lista y obtiene el mayor valor.
  - Terminará escribiendo *"Max in thread THREAD_ID:VALOR"*
- Lanzar 1000 tareas y comprobar cómo se comporta cada tipo de *thread pool*.

Future / Callable

- Los threads que se almacenan en cada *ExecutorService* **deben** implementar la interfaz *Runnable*.
  - Es el mismo tipo de objetos que hemos utilizado hasta ahora para crear *threads*.
- Sin embargo, el método *run* que implementan esas clases **no puede** devolver ningún resultado.
- Si necesitamos devolver un resultado, tenemos que utilizar objetos de tipo ***Callable***.

Future / Callable

- Un objeto que implementa *Callable* **debe** devolver un valor cuando termina su ejecución.
  - Es el equivalente a *Runnable*, pero incluyendo un **valor de retorno**.
- El tipo del objeto devuelto se define mediante **genéricos**.
- La interfaz *Callable*, obliga a implementar el método *call()*.
  - De la misma manera, *Runnable* obliga a implementar el método *run()*

Future / Callable


```
public class RunnableExample implements Runnable {

    @Override
    public void run() {
        // Task code
    }
}

public class CallableExample implements Callable<T> {

    @Override
    public T call() throws Exception {
        // Task code
        return null;
    }
}
```

Future / Callable

- El valor devuelto por un *Callable* se recoge en el programa principal a través de un objeto de tipo **Future**.
- **No** podemos utilizar *execute()* para ejecutar una tarea de tipo *Callable*.
- En su lugar, utilizamos *submit()*, que **envía** la tarea a ser ejecutada.
- El método *submit()* devuelve un objeto de tipo *Future<T>*, siendo *T* el tipo de datos que devuelve el método *call()*.

Future / Callable

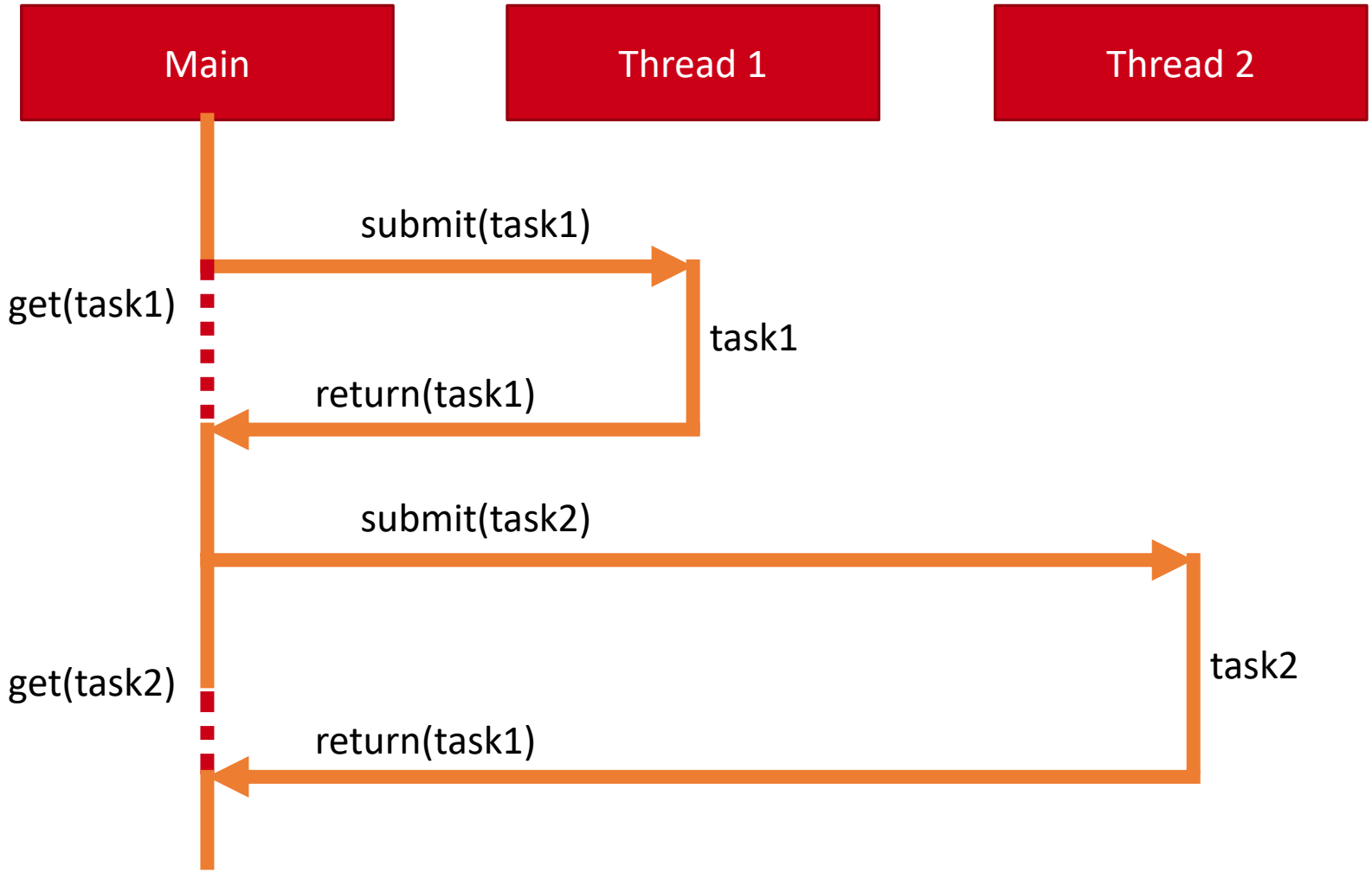
- *Future* también utiliza los **genéricos** para definir el tipo que contiene.
- El objeto de tipo *Future* **no contiene** el valor devuelto por la tarea.
  - Solo indica que, cuando finalice la tarea, podremos extraer un valor de un tipo determinado.
- Para obtener ese valor, debemos invocar al método *get()* sobre el objeto *Future*.
  - Esta llamada es **bloqueante**, de manera que el thread se quedará bloqueado hasta que la tarea asociada termine y devuelva el valor por el que hemos preguntado.

Future / Callable

- El objeto *Future* tiene dos objetivos principales:
  1. Controlar el estado de la tarea
  2. Obtener el resultado producido por la tarea.
- Para controlar el estado de la tarea, podemos utilizar el método *isDone()*.
  - Es un método no bloqueante que devuelve *true* si la tarea ha terminado, o *false* en caso contrario
- Para obtener el resultado producido por la tarea utilizamos el método *get()*
  - Existe una variante que espera un tiempo especificado, y en caso de no obtener un valor en ese tiempo, devuelve *null*
    - get(long timeout, TimeUnit timeunit);

Future / Callable

ALGORITMOS PARALELOS



Ejercicio

- Implementa un programa que cargue un fichero de texto y cuente el número de vocales que contiene.
  - Se creará una nueva tarea por cada palabra del fichero.
  - Cuando todas las tareas hayan terminado se imprimirá el número por pantalla.

Ejercicio

- Implementa un programa que calcule el factorial de los primeros 1.000.000 enteros de manera concurrente.
  - El resultado se debe almacenar en un objeto de tipo BigInteger para evitar desbordamientos.
  - Cada tarea calculará de manera completa el factorial del número que se le asigne.
  - El programa principal irá imprimiendo los resultados según los vaya obteniendo, indicando el número, ":" y el valor del factorial.

invokeAll / invokeAny

- Los *ExecutorService* ofrecen dos métodos adicionales para ejecutar tareas: *invokeAll* / *invokeAny*.
- Dada una lista de tareas, *invokeAll* ejecuta **todas** ellas sobre un *ExecutorService*.
  - Devuelve una lista de *Future* a partir de los cuáles obtenemos los resultados de las tareas.
- Por otra parte, *invokeAny* ejecutaría una de las tareas de la lista, elegida **aleatoriamente**.
  - Este método devuelve el resultado de la tarea ejecutada directamente, en lugar de un *Future*.

Ejercicio

- Compara los resultados obtenidos por *invokeAll* e *invokeAny*.
- Para ello, implementa un programa que cree una lista de 1000 tareas, donde cada tarea devuelve un String con el texto "Task ID", donde ID es el identificador de la tarea.
- A continuación, ejecútalas en un ExecutorService con *invokeAll* y con *invokeAny*

Future

- Existen tres métodos para enviar tareas a ejecutar:
  - *Future submit(Callable task);*
  - *Future submit(Runnable task);*
  - *Future submit(Runnable task,T result);*
- El primer método envía la tarea a ejecutar y devuelve el *Future* del resultado producido por la tarea.
- El segundo envía la tarea a ejecutar y siempre devuelve un *Future* cuyo *get* siempre será *null*.
  - Esto se debe a que un *Runnable* nunca devolverá ningún valor.
- El último método ejecuta la tarea y devuelve un *Future* con el valor almacenado en *result*.
  - OJO: como lo que ejecutamos es un *Runnable*, el resultado será directamente el que pasemos como parámetro, NO podemos devolver ningún resultado producido por la tarea.

ScheduledExecutorService

- Es un *ExecutorService* que permite programar tareas para ejecutarse con un cierto retraso o en un intervalo fijo.
- Al igual que *ExecutorService* es una interfaz.
  - Su implementación es *ScheduledThreadPoolExecutor*
  - Se inicializa con *Executors.newScheduledThreadPool(nThreads)*.
- El envío de tareas a ejecutar se puede realizar de 4 maneras diferentes:
  - schedule(Callable task, long delay, TimeUnit timeunit)
    - Programa la ejecución de la tarea con el retraso indicado en el delay.
  - schedule(Runnable task, long delay, TimeUnit timeunit)
    - Igual al anterior, pero el Future devuelto siempre devolverá *null*.
  - scheduleAtFixedRate(Runnable task, long initialDelay, long period, TimeUnit timeunit)
    - Programa una tarea para ser ejecutada periódicamente tras el retraso inicial. La tarea se repetirá cada período de tiempo indicado, comenzando a contar desde el inicio de la tarea anterior.
  - scheduleWithFixedDelay(Runnable task, long initialDelay, long period, TimeUnit timeunit)
    - Similar al anterior, pero el período comienza a contar desde el final de la tarea anterior.

Ejercicio

- Implementa un programa que compruebe cada 5 segundos si una web está disponible.
  - Utiliza el código visto con Jsoup para comprobar el mensaje de respuesta de la web.
- El programa debe lanzar 4 threads, y cada uno de ellos debe comprobar una web diferente.

Resumen

ALGORITMOS PARALELOS

- El paquete de concurrencia de Java nos ofrece tres interfaces de ejecución
  - **Executor**: interfaz que permite lanzar nuevas tareas.
  - **ExecutorService**: lanza nuevas tareas, gestionando el ciclo de vida de las mismas.
  - **ScheduledExecutorService**: permite retrasar y programar tareas.
- Para implementar una aplicación concurrente basada en tareas, debemos seguir los siguientes pasos:
  1. **Definimos** una clase que implemente Runnable o Callable que realizará la tarea.
  2. **Configuramos** el número de threads y cómo se van a gestionar a través de un *ExecutorService*
  3. **Utilizamos** el objeto Future para obtener el resultado (si la tarea es de tipo Callable)

Algoritmos paralelos I Executors

PROGRAMACIÓN CONCURRENTE EN JAVA - TEMA 5

Universidad
Rey Juan Carlos