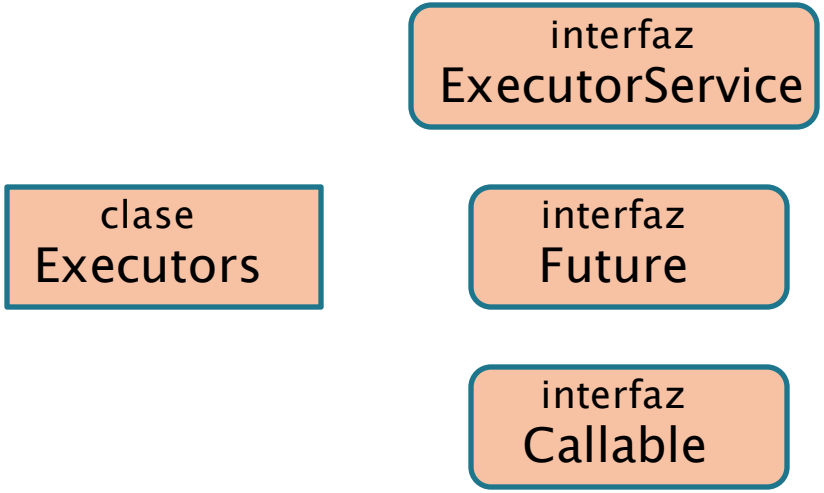
Nueva multitarea

Nuevas clases e interfaces

- En el paquete java.util.concurrent se incluyen nuevas clases e interfaces para la implementación de aplicaciones multitarea



La interfaz ExecutorService

- Proporciona métodos para el lanzamiento y ejecución de tareas de forma concurrente, utilizando un pool de threads. Entre estos métodos:
  - submit(Runnable tarea). Lanza la tarea y la pone en ejecución concurrente con el resto
  - submit(Callable tarea). Lo mismo que el anterior, pero para objetos Callable
  - shutdown(). Inicia el final del pool de hilos, por lo que no se aceptarán nuevas tareas

Creación de un ExecutorService

- Se pueden crear implementaciones de ExecutorService a partir de los siguientes métodos estáticos de Executors:
  - newCachedThreadPools(). Crea un ExecutorService con un pool de treads variable que se crean a demanda
  - newFixedThreadPools(int hilos). Crea un pool con un número fijo de threads
  - newSingleThreadExecutor(). Crea un ExecutorService que utiliza un único thread
  - newScheduledThreadPool(int corePoolSize). Devuelve un ScheduledExecutorService que permite ejecutar tareas periódicamente

Revisión conceptos

Indica que ocurrirá al ejecutar el siguiente código

```
ExecutorService ex=Executors.newCachedThreadPools();
ex.submit(()->System.out.print ("hello"));
ex.submit(()->System.out.print ("by"));
```

a. Se muestra helloby y el programa finaliza
b. Se muestra helloby pero el programa no finaliza
c. Se muestran las palabras hello y by en cualquier orden y el programa finaliza
d. Se muestran las palabras hello y by en cualquier orden pero el programa no finaliza

Respuesta

La respuesta es la D. Al ser ejecución concurrente, se imprimen los textos en orden impredecible y como no se ha llamado a shutdown(), el programa no finaliza.

Interfaz Callable

- Al igual que Runnable, implementa una tarea que va a ser ejecutada concurrentemente con otras.
- Su único método, call(), devuelve un resultado

```
public interface Callable<T>{
        T call() throws Exception;
}
```

Permite declarar checked exceptions

Tarea que calcula la suma de los números del 1 al 100

```
Callable<Integer> callb=()->{
    int suma=0;
    for(int i=1;i<=100;i++){
        suma+=i;
    }
    return suma;
}
```

Interfaz Future

- El método submit(Callable tarea) de ExecutorService devuelve un objeto Future que puede ser utilizado para acceder al resultado de la tarea y controlar su ejecución.

- Entre sus métodos están:
  - isDone(). Permite conocer si la tarea ha finalizado
  - get(). Devuelve el valor generado por Callable. Si aún no ha terminado la tarea, queda a la espera del resultado

```
Future<Tipo_resultado> t1=exec.submit(objetoCallable);
while(!t1.isDone()){
        System.out.println("Esperando fin tarea");
}
System.out.println("El resultado de la tarea es "+t1.get());
```

Revisión conceptos

Indica cuales de las siguientes instrucciones de lanzamiento de tareas son correctas

```
A. exservice.submit(()->System.out.println("hello"));
B. exservice.submit(()->5);
C. Future<Integer> f=exservice.submit(()->System.out.println("by"));
D. Future<String> f=exservice.submit(()->"hello");
E. Future<?> f=exservice.submit((n)->System.out.println(n));
```

Respuesta

A. Correcta. Se lanza un Runable
B. Correcta, se lanza un Callable
C. Incorrecta, con Runnable el Future no tiene tipo
D. Correcta, se lanza un Callable
E. Incorrecta, el método de la interfaz no tiene parámetros

Sincronización

- En las nuevas clases de multitarea la sincronización se lleva a cabo con la interfaz Lock que proporciona los siguientes métodos:
  - void lock(). Bloquea acceso al código a otros hilos
  - void unlock(). Desbloquea el acceso al código
- Se puede obtener una implementación de Lock instanciando ReentrantLock

```
Lock lc=new ReentrantLock();
lc.lock(); //bloquea el acceso
:
lc.unlock(): //desbloquea acceso
```

Existen otras implementaciones como ReadLock (permite a otros hilos con bloqueo de lectura) o WriteLock (no permite a otros ni lectura ni escritura)

Revisión conceptos

El siguiente bloque de instrucciones se va a ejecutar en un entorno multitarea. Teniendo en cuenta que "cont" es una variable estática que encapsula un valor entero, reformular el código para evitar condiciones de carrera:

```
:
int tmp=cont.getValue();
tmp++;
cont.setValue(tmp);
:
```

Respuesta

Se deberá crear un objeto Lock compartido por todos los hilos:
Lock lc=new ReentrantLock();
Después, las tres instrucciones anteriores se incluirán en una sección crítica para que solo un hilo la ejecute a la vez, incluyendo la llamada a unlock() en el finally para garantizar su ejecución

```
lc.lock();
try{
    int tmp=cont.getValue();
    tmp++;
    cont.setValue(tmp);
}finally{
    lc.unlock();
}
```

Colecciones para concurrencia

- El paquete java.util.concurrent incluye interfaces y clases de colección con operaciones *thread safe:*

  - ConcurrentMap<K,V>. Subinterfaz de Map que garantiza operaciones thread safe en un entorno multitarea. Su principal implementación es ConcurrentHashMap

  - CopyOnWriteArrayList<E>. Variante de ArrayList para entornos thread safe

  - CopyOnWriteArraySet<E>. Variante de HashSet para entornos thread safe. Internamente utiliza un CopyOnWriteArrayList para realizar las operaciones.