Nebenläufigkeit: Wiederholte und zeitgesteuerte Ausführung

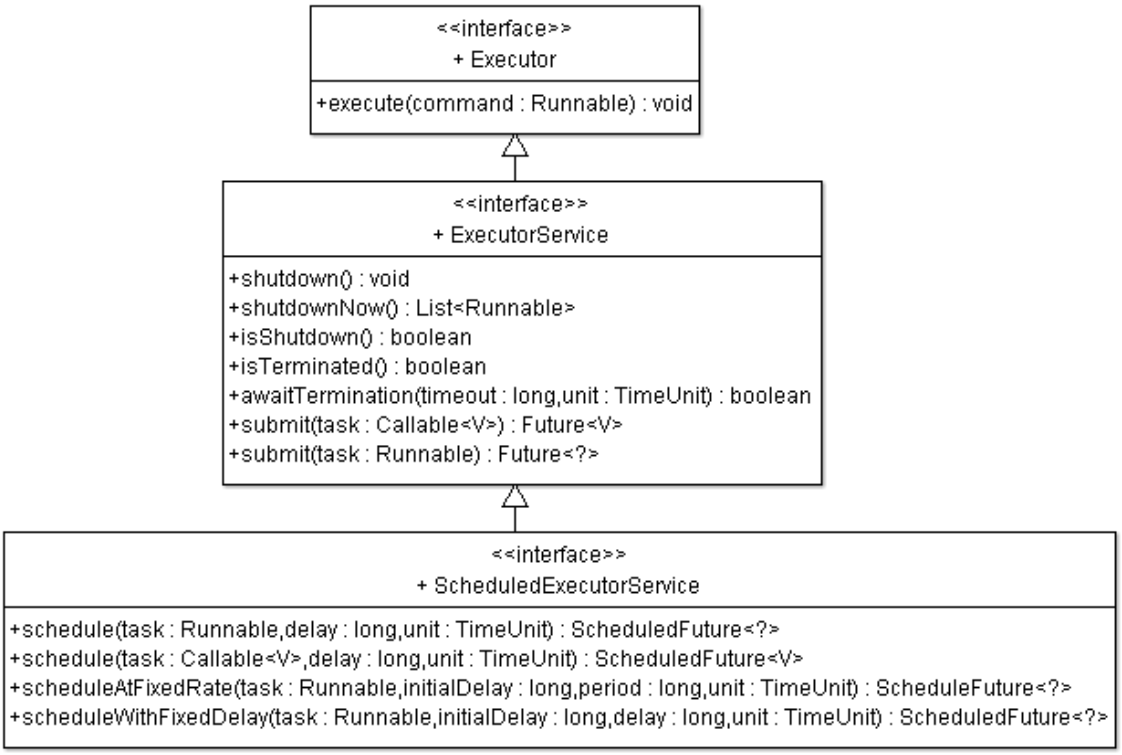
- Die Eigenarten und Schwachstellen von herkömmlichen Threads erkennen
- Die Eigenschaften und Vorteile eines Executors erkennen
- Die verschiedenen Arten von Executoren instanziieren und richtig einsetzen können
- Erkennen wie ein gestarteter Thread seine Ergebnisse bereitstellen kann
- Auf diese Ergebnisse warten und diese abrufen können
- Den Unterschied zwischen einem `Runnable` und einem `Callable` erkennen
- An einem bestimmten Zeitpunkt bzw. wiederholt zu startende nebenläufige Abläufe anstoßen können

Schwachstellen von Threads

- Damit ein Thread ein `Runnable` ausführt, muss vorher *neues Thread-Objekt* aufgebaut werden
- Auf dieses Thread-Objekt kann `start()` nur einmal aufgerufen werden
- Thread kann in seinem Leben *nicht mehrere* `Runnables` ausführen
- `Runnable` kann nicht zu einem *bestimmten Zeitpunkt* (Ostern)
- und auch *nicht wiederholt* (immer Weihnachten) ausgeführt werden

Lösung: Der `Executor` kommt!

Klassen die *Interface* `Executor` implementieren, können wiederholt `Runnables` ausführen

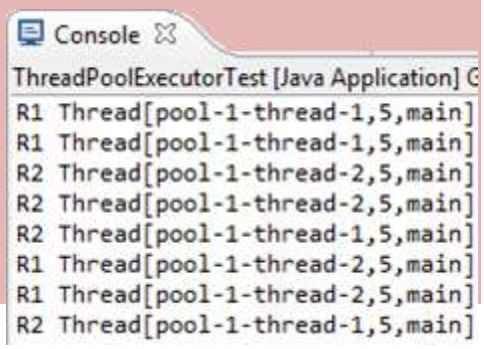


Interface `ExecutorService` erweitert `Executor` sodass ein `Runnable` (`Callable`) seine *Ergebnisse* mitteilen kann

`ScheduledExecutorService` erweitert `ExecutorService` sodass `Runnable` *wiederholt ausgeführt* werden kann

Beispiel

```
Runnable runnable1 = new Runnable() {
  public void run() {
    System.out.println("R1 " + Thread.currentThread());
    System.out.println("R1 " + Thread.currentThread());
  }
};

Runnable runnable2 = new Runnable() {
  public void run() {
    System.out.println("R2 " + Thread.currentThread());
    System.out.println("R2 " + Thread.currentThread());
  }
};

ExecutorService executor = Executors.newCachedThreadPool();
executor.execute(runnable1);
executor.execute(runnable2);

Thread.sleep(500);

executor.execute(runnable1);
executor.execute(runnable2);

executor.shutdown();
```



Threadname, Priorität, aufrufender Thread

- Utility-Klasse `Executors` stellt durch seine *Factory-Methode* `newCachedThreadPool()` einen *Thread-Pool* vom Typ `ExecutorService` zur Verfügung
- Über die Methode `execute()` können `Runnables` wiederholt ausgeführt werden

Factory-Klasse Executors liefert

`ExecutorService newCachedThreadPool()`
Thread-Pool mit *flexibler Anzahl* von Threads

`ExecutorService newFixedThreadPool(int maxThreads)`
Thread-Pool mit *maximal* maxThreads Threads

`ScheduledExecutorService newSingleThreadScheduledExecutor()`
Um Thread periodisch zu starten wobei Thread-Pool aus nur einem Thread besteht

`ScheduledExecutorService newScheduledThreadPool(int maxThreads)`
Wie voriger Aufruf, Thread-Pool besteht aber aus maximal maxThreads Threads

Methoden des Interfaces ExecutorService

`void shutdown()`
Thread-Pool wird heruntergefahren, laufende Threads nicht abgebrochen sondern regulär beendet, neue nicht angenommen

`List<Runnable> shutdownNow()`
Wie vorheriger Aufruf, liefert die noch auf Abarbeitung wartenden nicht begonnen Anfragen

`boolean isShutdown()`
Liefert `true` wenn der Shutdown-Vorgang eingeleitet wurde

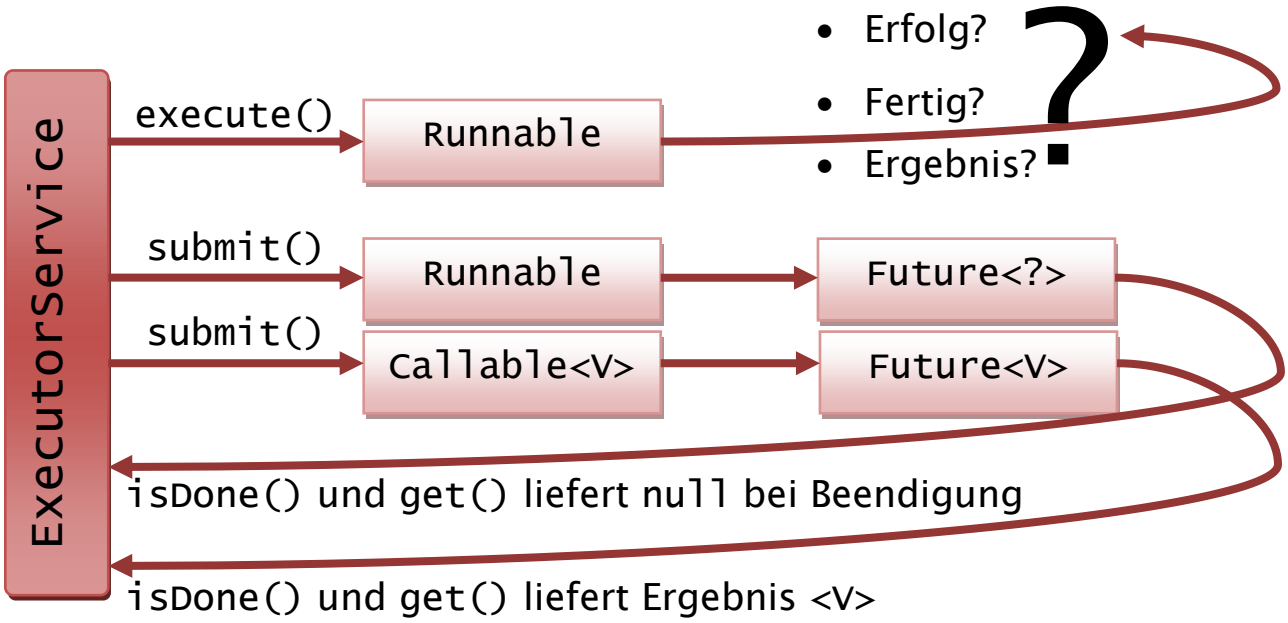
`boolean isTerminated()`
Liefert `true` wenn auch der letzte noch laufende Thread beendet wurde

`boolean awaitTermination(long timeout, TimeUnit unit)`
blockiert bis alle laufenden Threads beendet wurden oder Timeout abläuft

Problem:

`ExecutorService` erhält keine Informationen darüber ob `Runnable` fertig abgearbeitet wurde

Lösung: Threads mit Rückgabe über `Callable`




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

`Exception` kann geworfen werden

- wenn Ergebnis nicht ermittelt werden kann
- `isDone()` liefert daraufhin `true` und
- `get()` anstelle von <V> das `Exception`

`Future<V> submit(Callable<V> task)`
Startet Aufgabe und liefert `Future`-Objekt, das später Ergebnis vom Typ <V> liefern wird

`Future<V> submit(Runnable task)`
Wie vorheriger Aufruf, `Future`-Objekt liefert bei Beendigung der Aufgabe als Ergebnis `null`

`invokeAll()`
Startet mehrere `Callables` und wartet bis alle abgearbeitet sind

`invokeAny()`
Startet mehrere `Callables` und liefert nur Ergebnis des *schnellsten* Thread

Beispiel Sortierer

```
public class Sorter implements Callable<int[]>
{
  private int[] array = null;
  public Sorter(int[] array) {
     this.array = array;
  }
  @Override
  public int[] call() throws Exception {
     Arrays.sort(array);
     return array;
  }
}
```

```
int[] array = new int[MAX_LENGTH];
for (int i = 0; i < array.length; i++)
  array[i] = new Random().nextInt();
Sorter callable = new Sorter(array);
ExecutorService executor =
  Executors.newCachedThreadPool();
Future<int[]> future = executor.submit(callable);
while (!future.isDone())
  System.out.println("Do other things while waiting");
try {
  int[] sortArray = future.get();
  System.out.println("Sorting completed");
} catch (InterruptedException | ExecutionException e) {
  System.out.println(e.getClass().getName());
}
```

Methoden des Interfaces Future<V>

`boolean isDone()`
Liefert `true` falls Aufgabe erledigt wurde oder `Exception`, wenn während Abarbeitung Fehler aufgetreten ist, sonst `false`

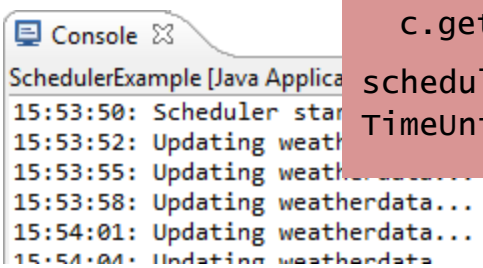
`<V> get()`
Methode blockiert solange bis Ergebnis vorliegt. Sie liefert `null` zurück, falls `Runnable` anstelle von `Callable` auszuführende Aufgabe enthält

`<V> get(long timeout, TimeUnit unit)`
Wie voriger Aufruf. Es wird aber nur maximal bis zum Timeout gewartet. Wenn bis dahin kein Ergebnis vorliegt wird `TimeoutException` geworfen

Aufgaben wiederholt zeitgesteuert ausführen

```
public class WeatherUpdater implements Runnable
{
  public void run() {
    DateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");
    Calendar c = Calendar.getInstance();
    System.out.println(dateFormat.format(c.getTime()) +
      ": Updating weatherdata...");
  }
}
```

```
WeatherUpdater updater = new WeatherUpdater();

ScheduledExecutorService scheduler =
  Executors.newSingleThreadScheduledExecutor();

DateFormat dateFormat =
  new SimpleDateFormat("HH:mm:ss");
Calendar c = Calendar.getInstance();
System.out.println(dateFormat.format(
  c.getTime()) + ": Scheduler started");

scheduler.scheduleAtFixedRate(updater, 2, 3,
TimeUnit.SECONDS);
```



Methoden des Interfaces ScheduledExecutorService

`ScheduleFuture<?>schedule(`
`Runnable task, long delay, TimeUnit unit)`
Führt Runnable *einmal* aus, nachdem angegebene Zeit `delay` gewartet wurde

`ScheduleFuture<V>schedule(`
`Callable task, long delay, TimeUnite unit)`
Führt `Callable` nach angegebener Wartezeit *einmal* aus und liefert Ergebnis

`ScheduleFuture<?>scheduleAtFixedRate(`
`Runnable task, long initialDelay, long period, TimeUnit unit)`
Führt `Runnable` nach angegebener Wartezeit *wiederholt* aus, wobei zwischen den Zeitpunkten der Aufrufe die Zeitspanne `period` gewartet wird. Dauert Aufruf länger als `period`, wird erst nach `2*period` gestartet. **Problem**: Liefert Aufruf ein Exception, starten nachfolgende Aufrufe nicht mehr (!)

`ScheduleFuture<?> scheduleWithFixedDelay(`
`Runnable task, long initialDelay, long delay, TimeUnit unit)`
Führt `Runnable` nach Wartezeit wiederholt aus, wobei zwischen der Beendigung des einen Aufrufes und des Starten des nächsten Aufrufes die Zeitspanne von `delay` gewartet wird

Job Scheduler QUARTZ (`http://quartz-scheduler.org`) löst obiges Problem