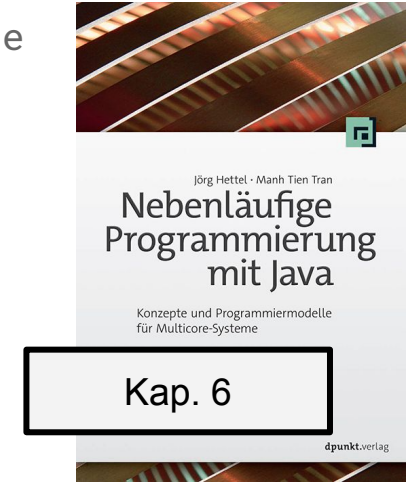
Parallele Programmierung

Thread Pools

Überblick

- **Laborübung “((1+2)+(3+4))” mit paralleler Ausführung**
- `ExecutorService` für asynchrone Methodenaufrufe: `Callable` und `Future`
  - asynchroner Funktionsaufruf (Data Flow)
  - `Callable`, `Future`, `ExecutorService`
  - **Laborübung: “((1+2)+(3+4))” nun mit `Callable`+`Future`**
- Thread-Pools
  - `Executors`-Factory: Fixed und Cached Thread-Pools
  - Thread-Pool terminieren (`shutdown()`, …)
  - `ScheduledExecutorService` (`schedule(…)`, …)
  - **Laborübung Thread Pool**
  - Thread Pool Dimensionierung
  - Exceptions in `Runnable` und `Callable` bei Thread-Pool Verwendung
  - `CompletionService` für voneinander unabhängige Tasks



Threads, die ein Ergebnis berechnen

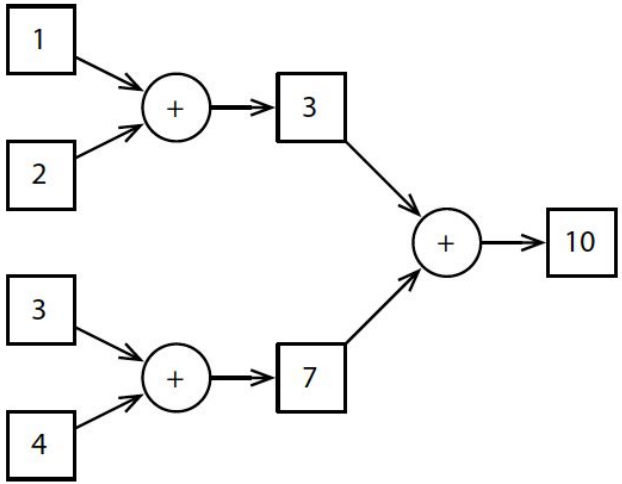
Asynchroner Funktionsaufruf für “Data Flow”-Aufgaben

Folie mit Anmerkungen

Reihenfolge, in der die Ausdrücke berechnet werden können

sequentiell
- 1+2 ; 3+4 ; 3+7
- 3+4 ; 1+2 ; 3+7

parallelisiert
- 1+2 || 3+4 ; 3+7



Bildquelle: Butcher (2014), S. 73

Laboraufgabe (30+15 Minuten)

`pp.04.01-RunnableReturn`

ExecutorService für asynchrone Methodenaufrufe: Callable und Future

Problemstellung: asynchroner Methodenaufruf

Wie kann ein nebenläufiger Thread ein Ergebnis abliefern?

- Die einzige Aufgabe solch eines Threads ist, genau eine (ggf. langdauernde) Berechnung durchführen.
- Die Berechnung soll nebenläufig ablaufen: Nach dem Start geht es direkt im Programmfluss des Erzeugers weiter.
- Man weiß nicht, wie lange der Thread für die Berechnung braucht. Eine Schnittstelle ist deshalb erforderlich um zu prüfen, ob das Ergebnis schon vorliegt.
- Bei der Berechnung könnten Exceptions geworfen werden. Die sollen nicht asynchron durchgereicht werden.

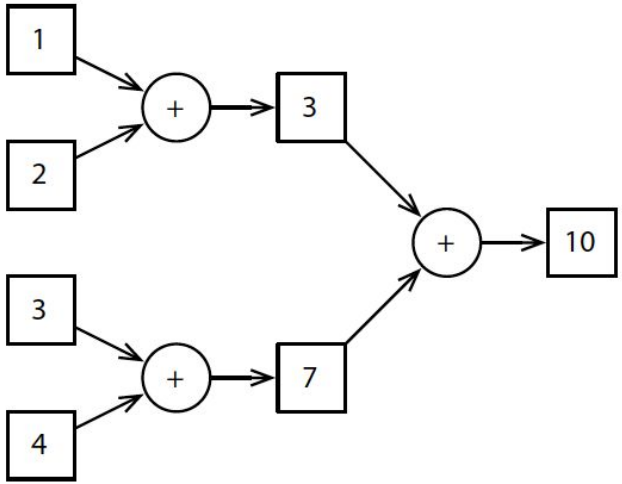
Asynchroner Funktionsaufruf für “Data Flow”-Aufgaben

Reihenfolge, in der die Ausdrücke berechnet werden können

sequentiell
- 1+2 ; 3+4 ; 3+7
- 3+4 ; 1+2 ; 3+7

parallelisiert
- 1+2 || 3+4 ; 3+7

Der Main-Thread könnte (1+2) und (3+4) asynchron starten, etwas anderes tun, ab und zu prüfen, ob beide Ergebnisse vorliegen und sie dann addieren.



Bildquelle: Butcher (2014), S. 73

Folie mit Anmerkungen

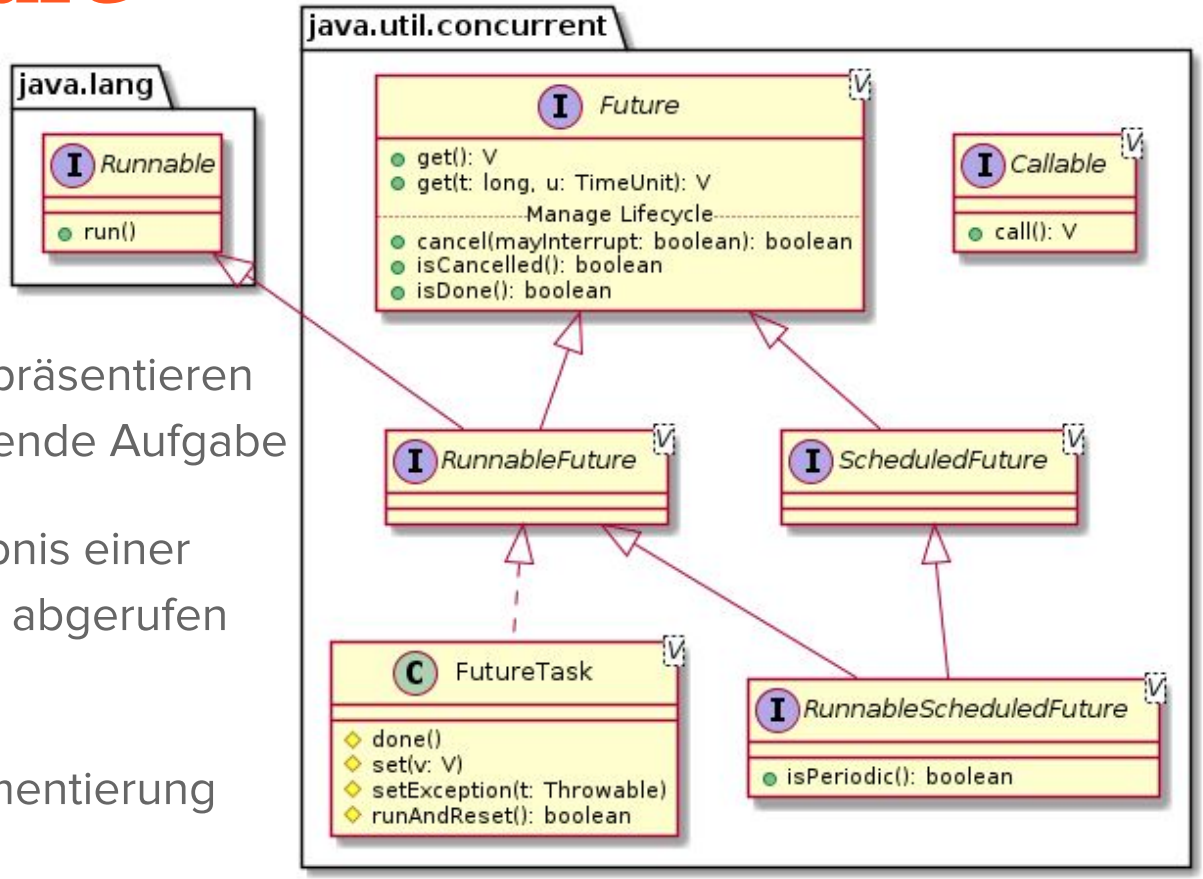
Callable, Future

Asynchrone Verarbeitung wird in Java mit Callable, Runnable und Future gemacht.

Callable und Runnable repräsentieren die asynchron abzuarbeitende Aufgabe

mit Future kann das Ergebnis einer asynchronen Berechnung abgerufen werden (get())

FutureTask ist eine Implementierung von Future und Runnable



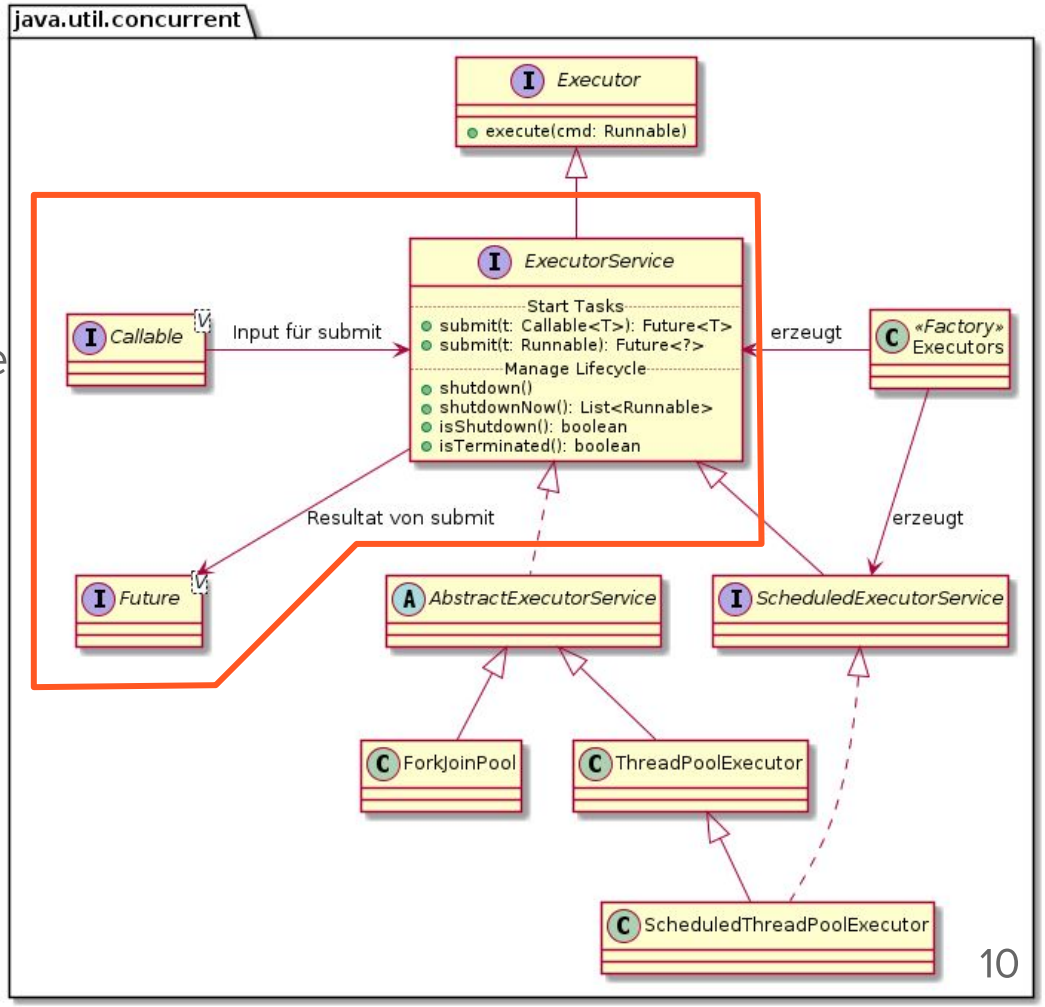
ExecutorService-Framework

Die asynchrone Verarbeitung erfolgt in Java durch ExecutorService.

Implementierungen nehmen über die Methode submit Callable Objekte an und starten call() asynchron.

Als Ergebnis bekommt der Aufrufer ein Future als Proxy, über den

- auf das Ergebnis zugegriffen wird
- die asynchrone Berechnung gemanagt wird

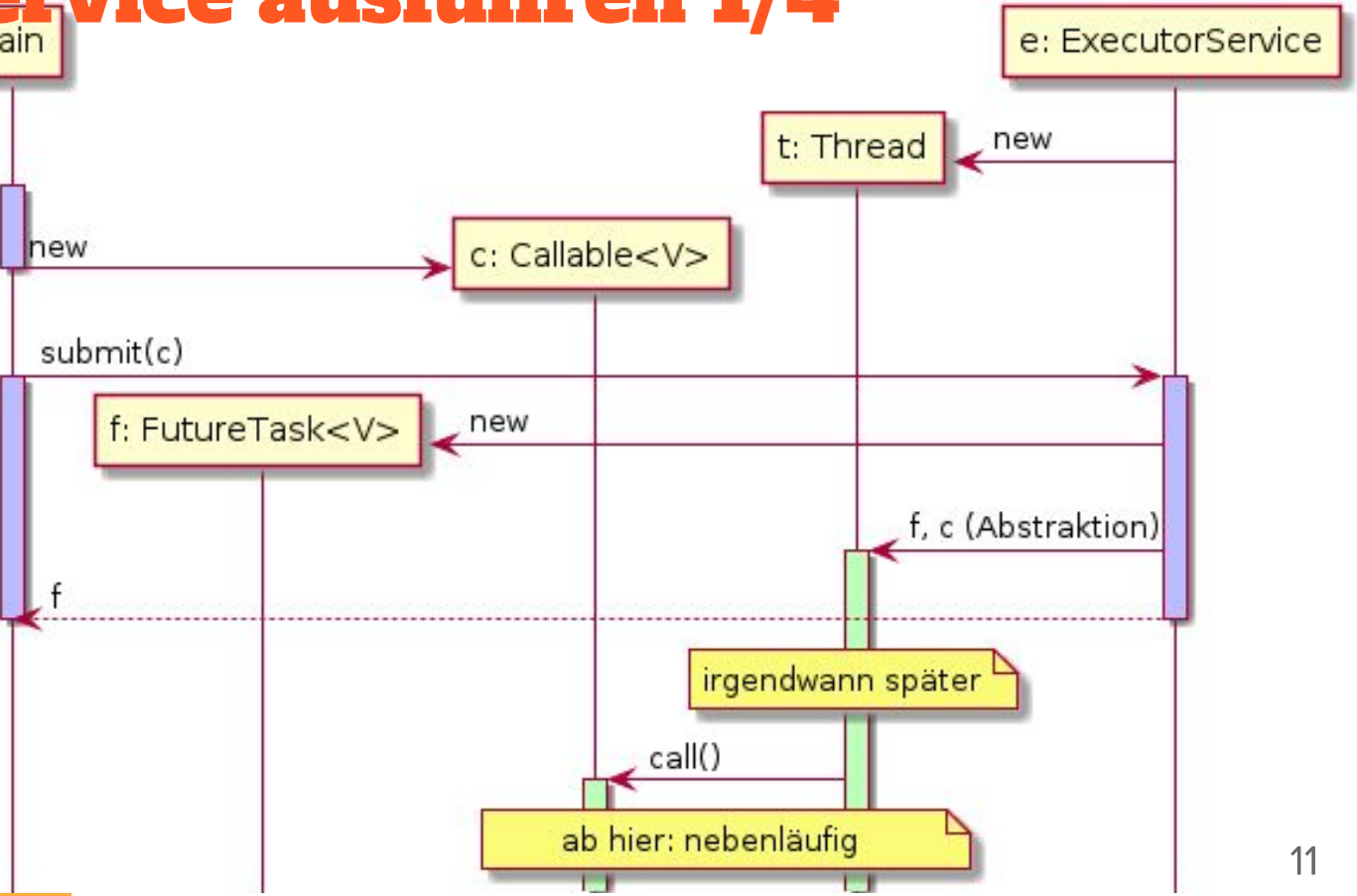


Callable asynchron mit ExecutorService ausführen 1/4

Folie mit Anmerkungen

```
var e = /* demnächst */

var c = new Callable<V>(){
  public V call() {
    return ...;
  }
};
var f = e.submit(c);
```

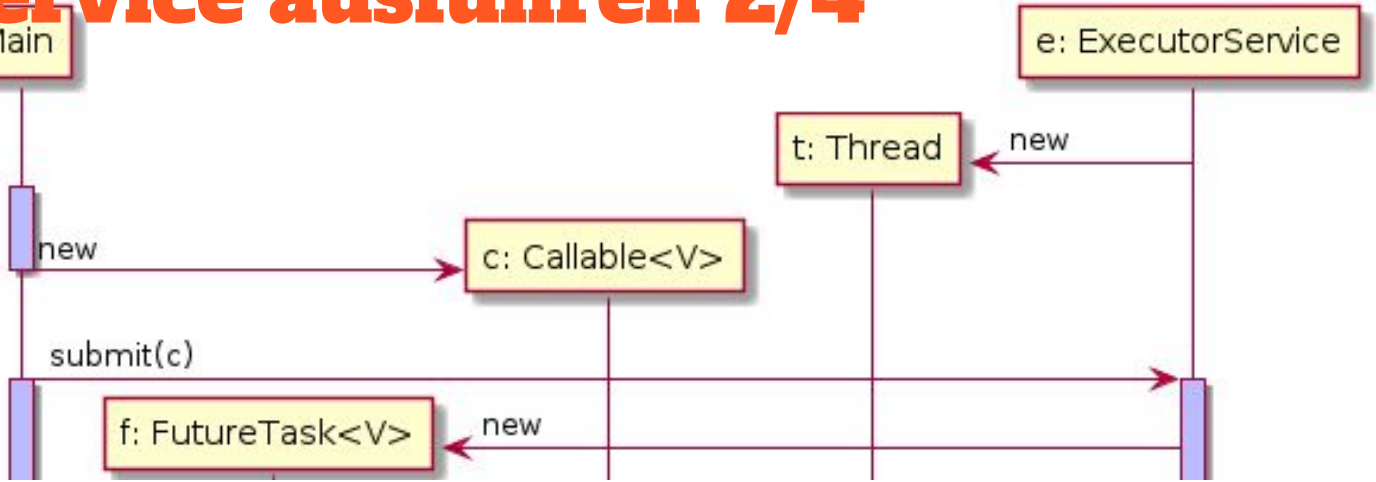


Callable asynchron mit ExecutorService ausführen 2/4

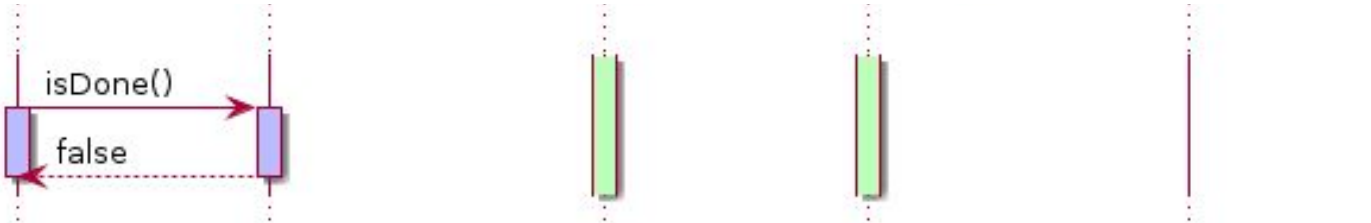
Folie mit Anmerkungen

```
var e = /* demnächst */

var c = new Callable<V>(){
  public V call() {
    return ...;
    }
};
var f = e.submit(c);
```




```
if(f.isDone()) {
    /* ... */
}
```



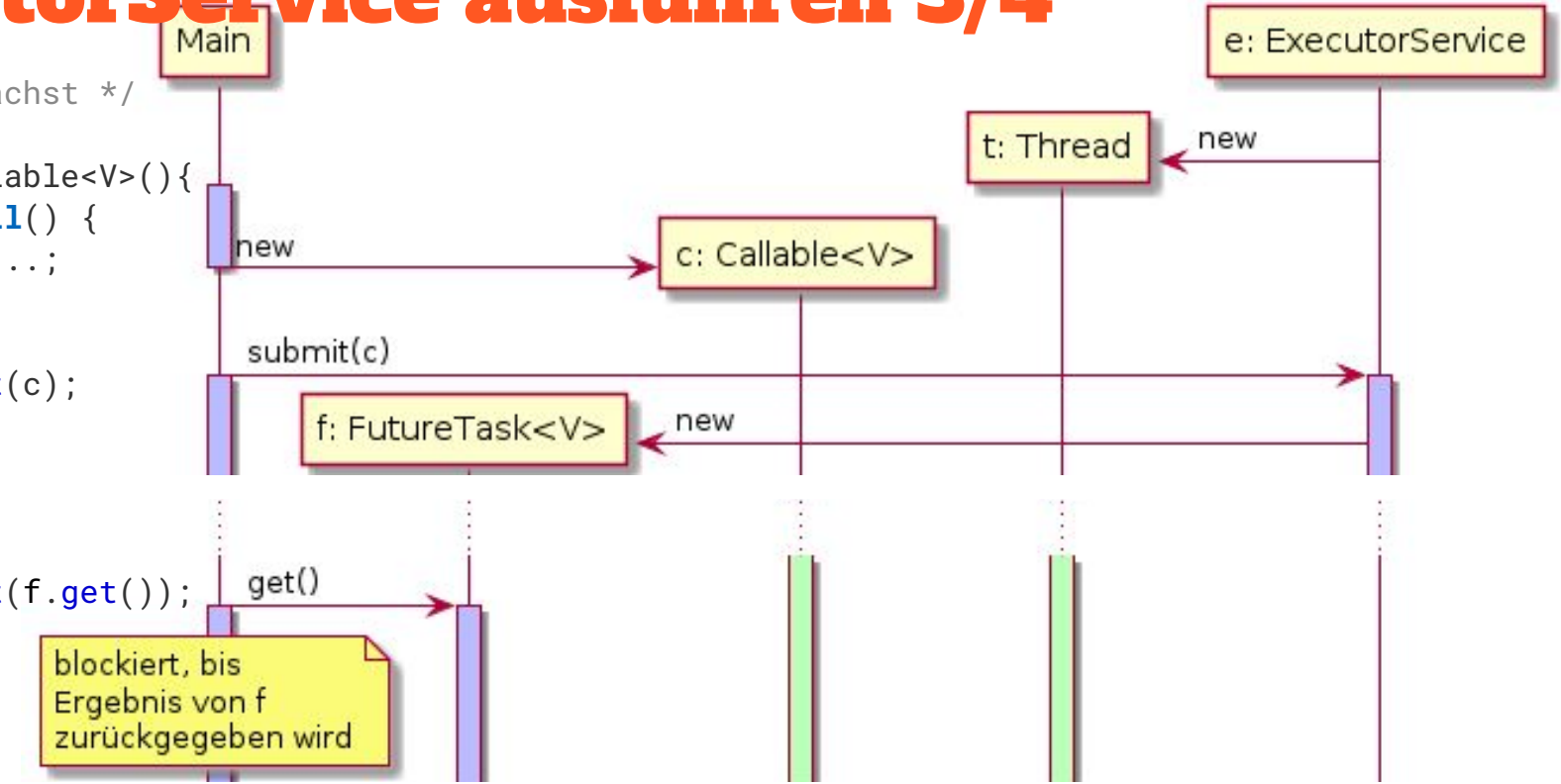
Callable asynchron mit ExecutorService ausführen 3/4

Folie mit Anmerkungen

```
var e = /* demnächst */

var c = new Callable<V>(){
    public V call() {
        return ...;
    }
};
var f = e.submit(c);


System.out.print(f.get());
```

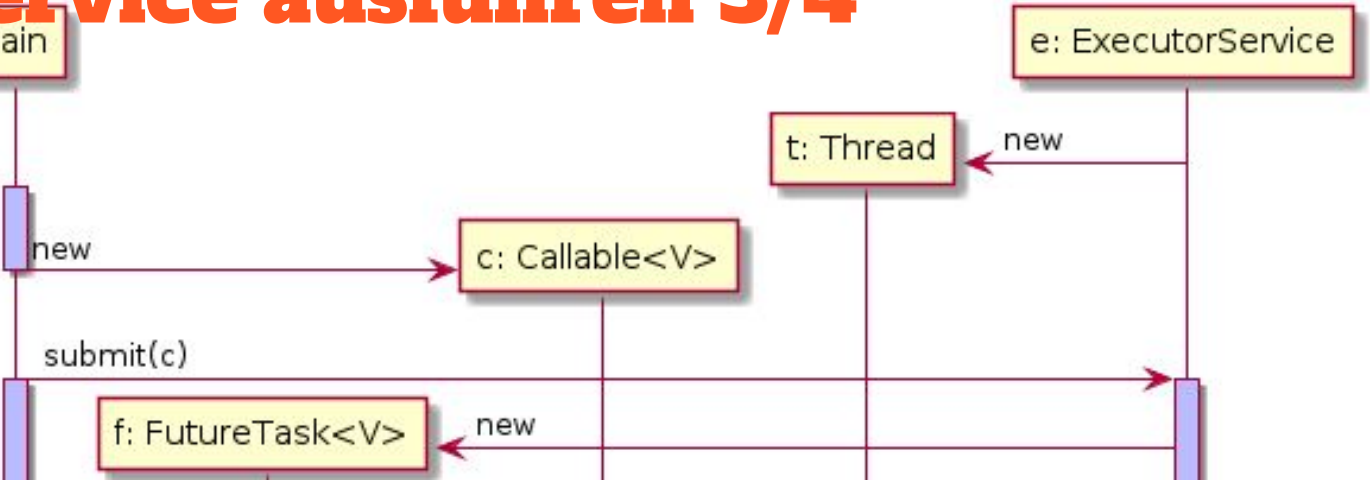


Callable asynchron mit ExecutorService ausführen 3/4

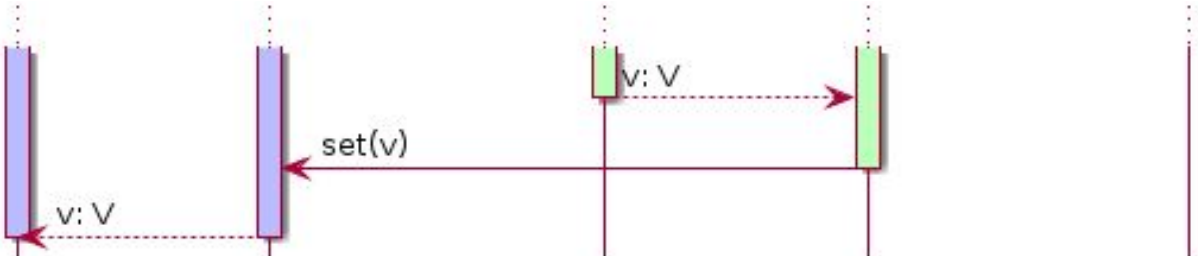
Folie mit Anmerkungen

```
var e = /* demnächst */

var c = new Callable<V>(){
    public V call() {
        return ...;
    }
};
var f = e.submit(c);
```




```
System.out.print(f.get());
```



Laboraufgabe (15+5 Minuten)

`pp04.03-Future`

Thread Pools

Warum Thread Pools?

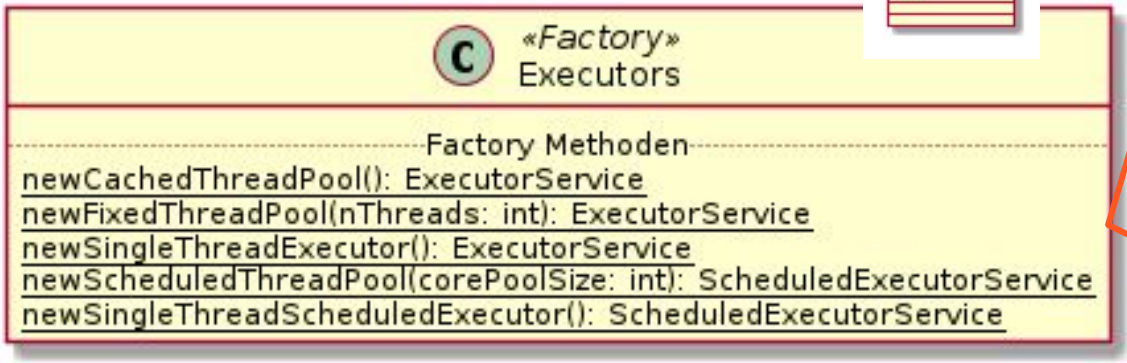
Thread-Instanziierung ist “teurer” (dauert länger) als bei anderen Klassen, denn Datenstrukturen zur Thread-Kontrolle müssen angelegt werden und Thread-lokaler Stack-Speicher angefordert werden.

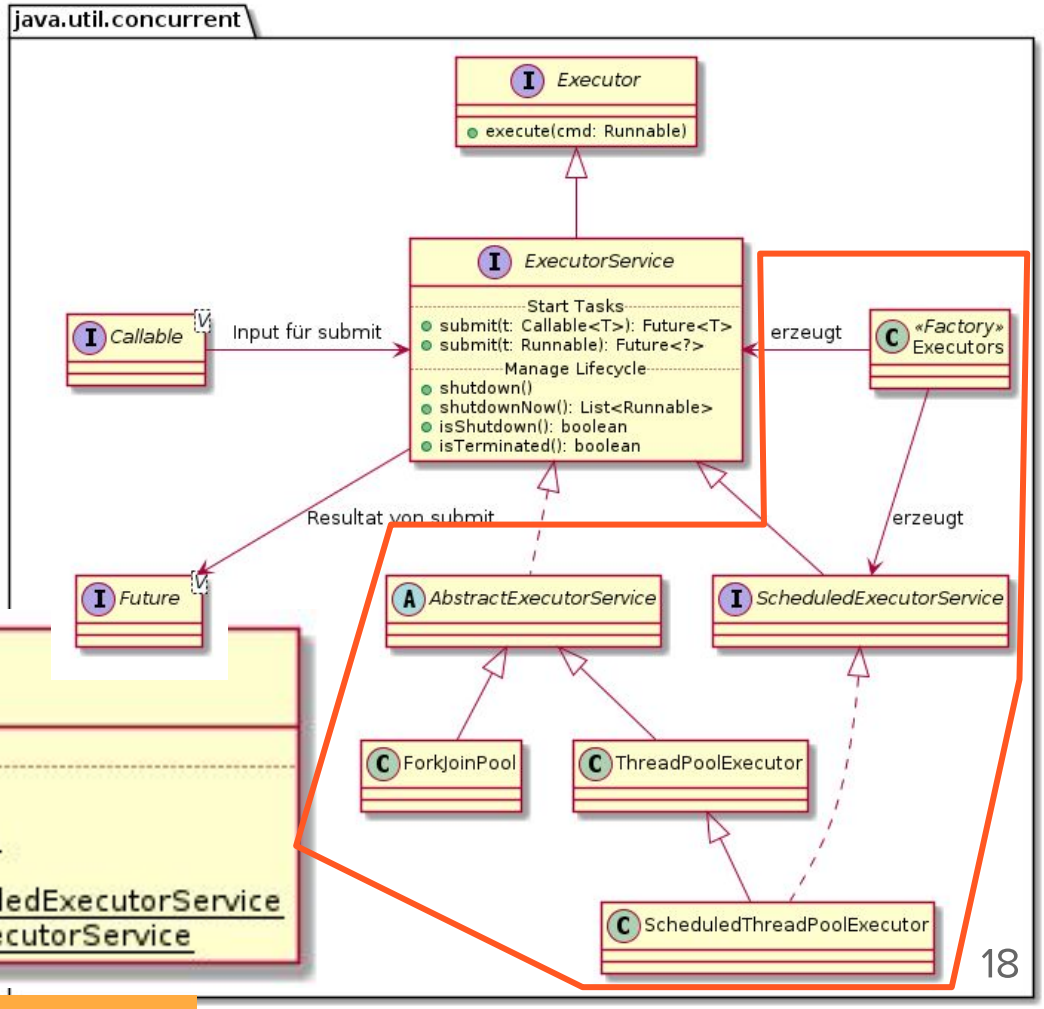
Optimierung: Thread-Objekte werden frühzeitig (z.B. beim Start) vorbereitet (Thread Pool). Wird ein Thread benötigt, wird einer der vorbereiteten Threads aus dem Pool genommen, mit einem Runnable-Objekt verbunden und (re-) aktiviert (statt `start()`).

Beim Ende der `run()`-Methode wird der Thread nicht vergessen und über die Garbage Collection entfernt, sondern deaktiviert und in den Thread Pool zur Wiederverwendung eingestellt

ExecutorService-Framework

Die Klasse **`Executors`** stellt Factory-Methoden bereit für **`ExecutorService`**:





Shutdown eines Thread-Pools

- `shutdown()`
  - Die bereits eingestellten Aufgaben werden noch abgearbeitet, neue werden zurückgewiesen.
  - Der `ExecutorService` beginnt, herunterzufahren. Der Aufruf von `shutdown()` ist aber asynchron (es geht direkt im Anschluss weiter im Programmablauf.
- `isShutdown()`
  - prüfen, ob der `ExecutorService` bereits fertig terminiert ist (nach `shutdown()`).
- `shutdownNow()`
  - Erzwingen der sofortigen Terminierung der Threads im `ExecutorService`: Alle aktiven Tasks erhalten mit `interrupt()`.

Das ExecutorService-Framework

```
var pool = Executors.newFixedThreadPool(5);
for (var i = 0; i <= 100; i++) {
    pool.submit(() ->
        System.out.println(Thread.currentThread().getName()));
}

pool.shutdown();
```

Folie mit Anmerkungen

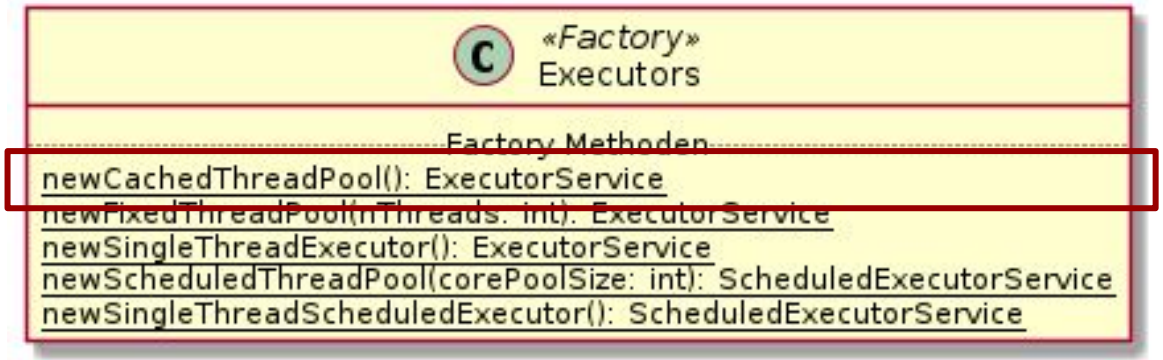
```
pool-1-thread-1
pool-1-thread-3
pool-1-thread-2
pool-1-thread-4
pool-1-thread-5
pool-1-thread-3
pool-1-thread-4
pool-1-thread-3
pool-1-thread-2
pool-1-thread-4
pool-1-thread-3
pool-1-thread-5
pool-1-thread-3
pool-1-thread-4
pool-1-thread-2
pool-1-thread-2
pool-1-thread-2
pool-1-thread-2
pool-1-thread-4
pool-1-thread-4
pool-1-thread-4
pool-1-thread-4
```

Thread Pool Framework von Java Executors Factory 1/5

Cached Thread Pool

erzeugt bei Bedarf neue Threads, unbenutzte Threads werden nach 60s beendet

⇒ Programme mit kurzlebigen, asynchronen Aufgaben

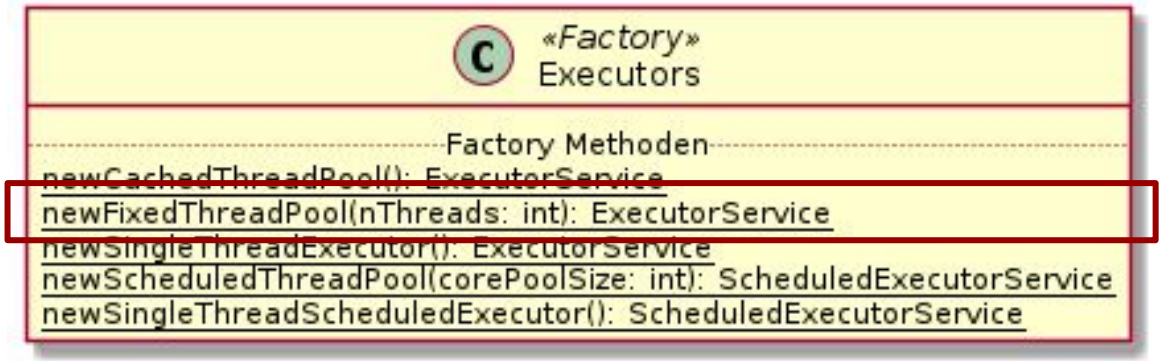


Thread Pool Framework von Java Executors Factory 2/5

Fixed Thread Pool

max. `nThreads` werden erzeugt, überzählige Runnables werden in Queue gespeichert

⇒ Programme mit sehr vielen unabhängigen Aufgaben

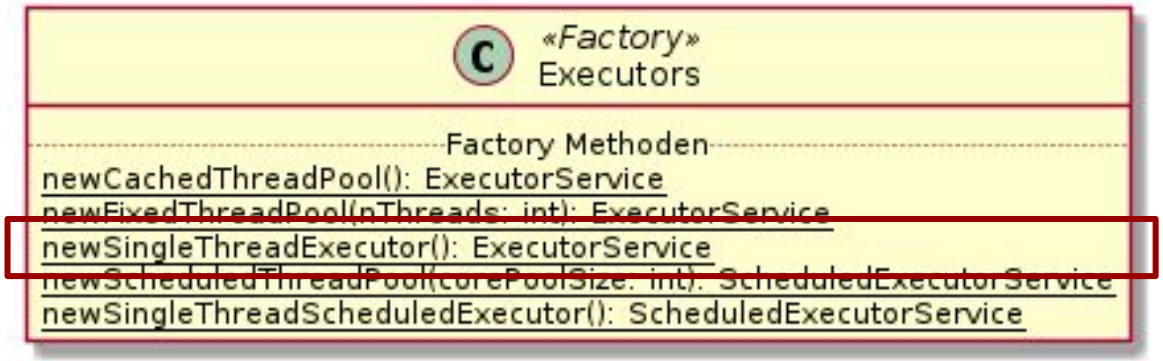


Thread Pool Framework von Java Executors Factory 3/5

Single Thread Pool

Sonderfall von `newFixedThreadPool(1)`. Stürzt der Thread ab, wird er neu gestartet

⇒ Programme mit weniger unabhängigen Aufgaben; Absturzsicherung

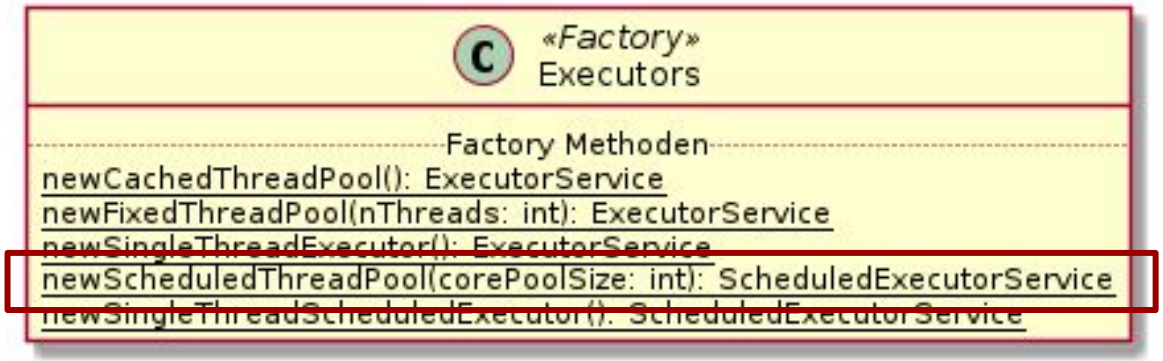


Thread Pool Framework von Java Executors Factory 4/5

Scheduled Thread Pool

Aufgaben werden nach einer gegebenen Verzögerung bzw. periodisch ausgeführt

⇒ Programme mit vielen zeitlogisch abhängigen Aufgaben

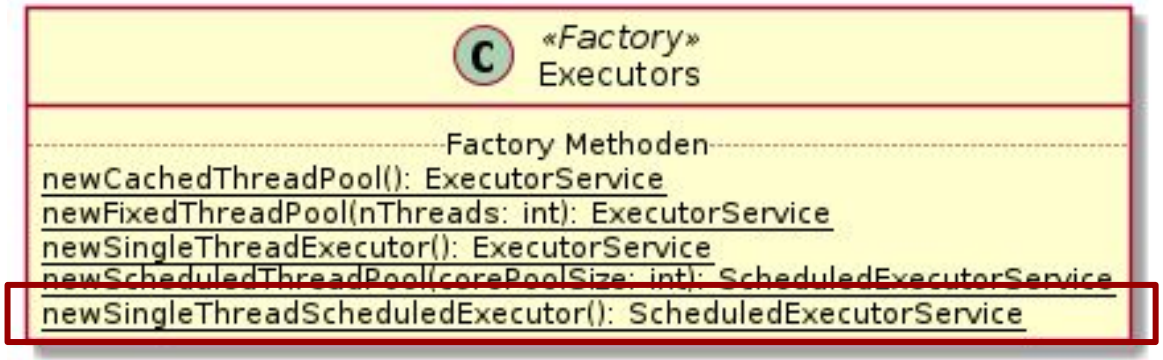


Thread Pool Framework von Java Executors Factory 5/5

Single Scheduled Thread Pool

Sonderfall von `newScheduledThreadPool(1)`. Stürzt der Thread ab, wird er neu gestartet

⇒ Programme mit einigen zeitlogisch abhängigen Aufgaben; Absturzsicherung

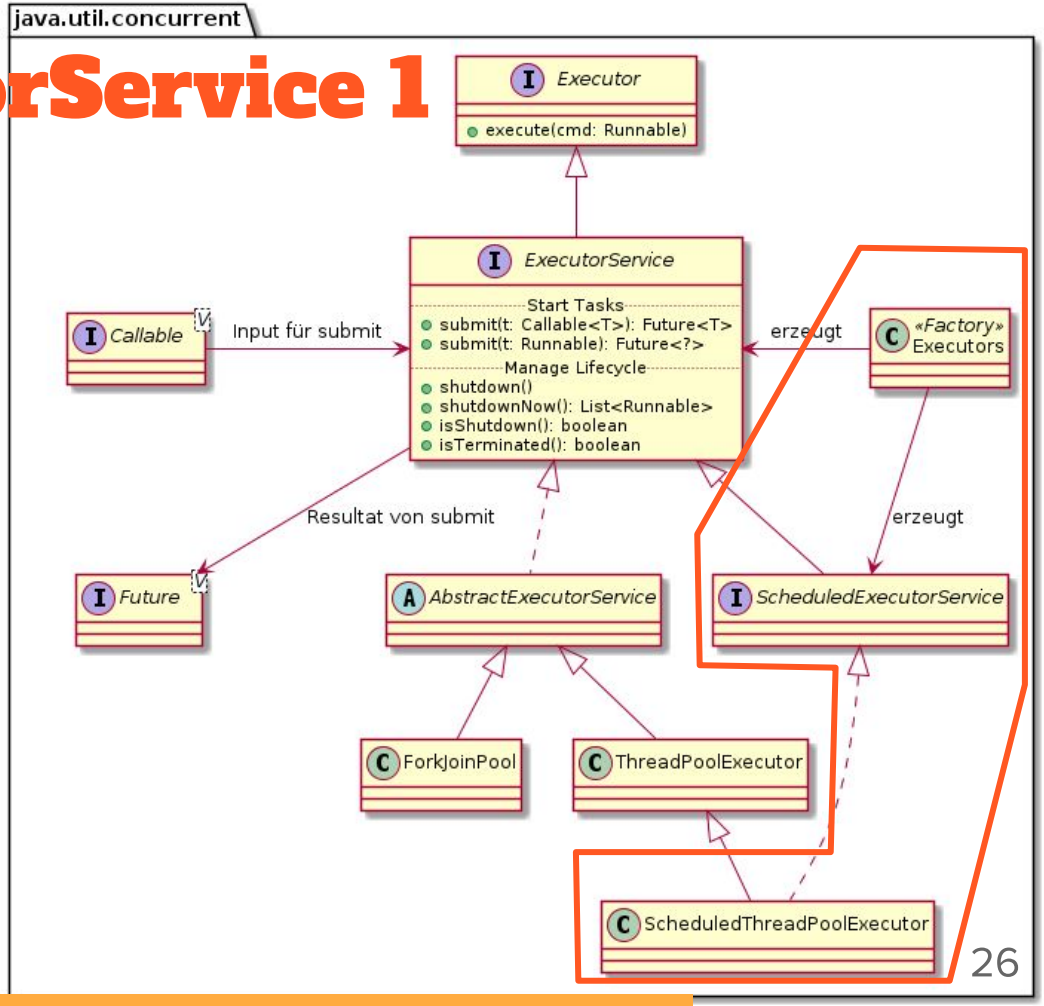


ScheduledExecutorService 1

Das ScheduledExecutorService-Interface erlaubt Aufgaben in Form von Callable und Runnable

- zu bestimmten Zeiten oder
- wiederholt

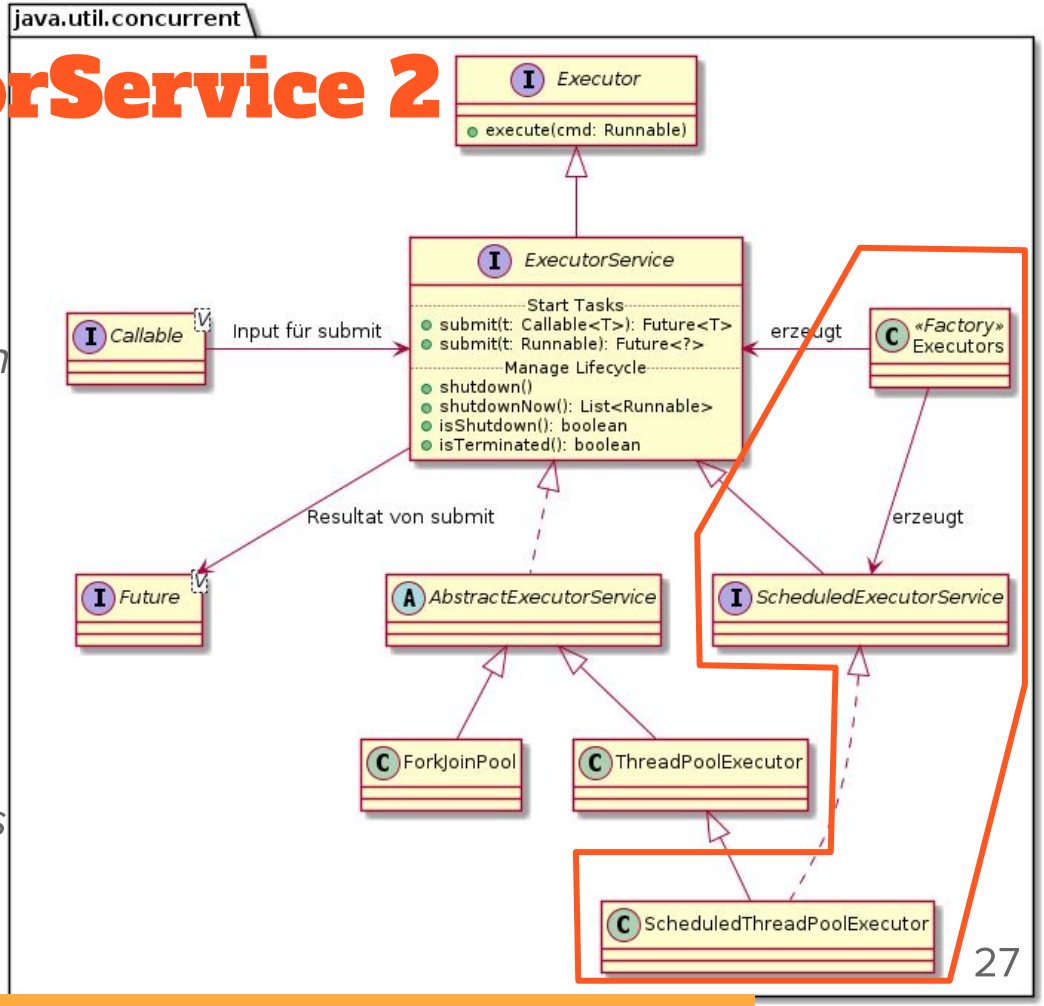
auszuführen.



ScheduledExecutorService 2

Dazu gibt es die Methoden

- **`schedule`**
  starte einmalig in x Zeiteinheiten
- **`scheduleAtFixedRate`**
  warte x Zeiteinheiten und starte dann periodisch alle y Zeiteinheiten
- **`scheduleWithFixedDelay`**
  warte x Zeiteinheiten und starte dann, nach dem Ende des Tasks warte y Zeiteinheiten und starte erneut, ...



Beispiel ScheduledExecutorService

Folie mit Anmerkungen

```
var scheduler = Executors.newScheduledThreadPool(1);
var beeperHandle = scheduler.scheduleAtFixedRate(
    () -> System.out.println("beep"), 3, 3, TimeUnit.SECONDS
);
scheduler.schedule(
    () -> beeperHandle.cancel(true), 5 * 3, TimeUnit.SECONDS
);
scheduler.schedule(
    () -> System.exit(0), (5 * 3) + 5, TimeUnit.SECONDS
);
```

Laboraufgabe (10+5 Minuten)

`pp.04.02-ThreadPoolSize`

Dimensionierung der Thread Pool Größe

Folie mit Anmerkungen

$$
\begin{aligned}
N_{cpu} &= number\ of\ CPUs \\
U_{cpu} &= target\ CPU\ utilization,\ 0 \leq U_{cpu} \leq 1 \\
\frac{W}{C} &= ratio\ of\ wait\ time\ to\ compute\ time \\
N_{threads} &= N_{cpu} * U_{cpu} * \left(1 + \frac{W}{C}\right)
\end{aligned}
$$

```
int N_CPUS = Runtime.getRuntime().availableProcessors();
```

Quelle: Goetz et al. 2006, S. 171.

Exceptions bei Thread Pool Verwendung

- `pool.execute(Runnable r);`

  In `r.run()` auftretende unbehandlete Exceptions werden sofort geworfen und der betroffene Pool-Thread wird terminiert.

- `Future f = pool.submit(Callable c);`

  In `c.call()` auftretende unbehandelte Exceptions werden “aufgehoben” und erst von `f.get()` geworfen.

  Soll eine Ausnahme lokal in `c.call()` behandelt werden (z.B. zum Aufräumen), kann sie danach im `catch`-Block erneut geworfen (`throw`) werden um sie auch dem Aufrufer von `f.get()` weiterzuleiten.

CompletionService für voneinander unabhängige Tasks

```
var pool = Executors.newCachedThreadPool();
var tasks = new ArrayList<Callable<String>>();
tasks.add(() -> "calc c1");
tasks.add(() -> "calc c2");
tasks.add(() -> "calc c3");
var completionService = new ExecutorCompletionService<>(pool);
for (var callableTask : tasks) {
    completionService.submit(callableTask);
}
try {
    for (var i = 0; i < tasks.size(); i++) {
        var future = completionService.take();
        System.out.printf("Result %2d: %s\n", i, future.get());
    }
} catch (InterruptedException | ExecutionException e) {
    e.printStackTrace();
}
pool.shutdown();
```

take() liefert das nächste fertige **Future<>**, egal in welcher Reihenfolge **submit()** ausgeführt wurde