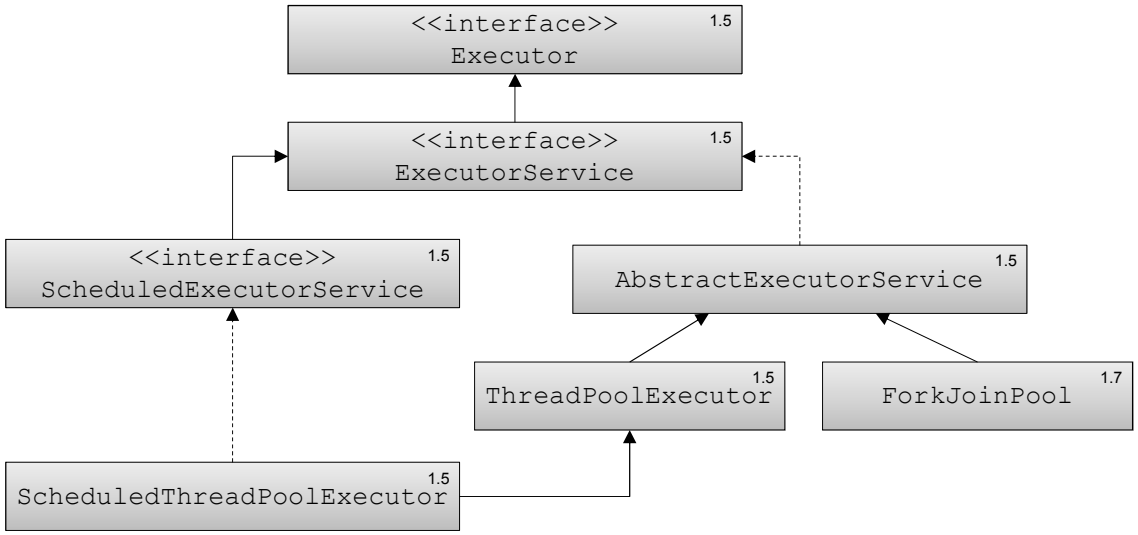
Asynchrone Anwendungen mit `CompletableFuture`

Gründe für den Einsatz von Asynchronität

- Synchrone Abläufe skalieren schlechter und führen zu Blockaden
- Technische Anforderungen (z.B. Kommunikation)
  - Microservice-Architekturen
  - Anbindung anderer Systeme
- Fachliche Anforderungen
  - Einholen von Preisen oder Verfügbarkeiten bei mehreren Anbietern
  - Aggregation von Daten
  - Komplexe Berechnungen in kleinere unterteilen
  - Verarbeitung großer Datenmengen parallelisieren

THILO FROTSCHER
Software Architect&Trainer

Executor und ExecutorService



THILO FROTSCHER
Software Architect&Trainer

ThreadPoolExecutor vs ForkJoinPool

- `ThreadPoolExecutor` (General Purpose Thread Pool)
  - Verwaltet einen Pool von Threads mit zentraler Queue für eingehende Tasks
  - Potentieller Performanzverlust durch gleichzeitigen Zugriff auf Queue
  - Keine Unterstützung für Zusammenarbeit mehrerer Threads an einer Task
- `ForkJoinPool` (Special implementation: “Work Stealing”)
  - Pool-Threads versuchen selbstständig neue Tasks zu finden/auszuführen,…
    - die an den Pool von außen submitted wurden, oder
    - die von anderen Tasks als Sub-Tasks erzeugt wurden
  - Effiziente Abarbeitung falls viele Tasks andere Sub-Tasks erzeugen
  - `ForkJoinPool.commonPool()` liefert Instanz für typische Einsatzzwecke

Die Klasse `Executors`

- Factory- und Utility-Methoden
  - erzeuge `ExecutorService`
  - erzeuge `ScheduledExecutorService`
  - erzeuge `ThreadFactory`
  - erzeuge `Callable`

Seit Java 5: Future<T>

```
ExecutorService executor = Executors.new...;
QuoteService quoteService = ...;

Future<Integer> futureQuoteFromA =
    executor.submit(quoteService::getQuoteFromSupplierA);

Future<Integer> futureQuoteFromB =
    executor.submit(quoteService::getQuoteFromSupplierB);

// do other tasks for some time...
```

THILO FROTSCHER
Software Architect&Trainer

Blockierende API für Abruf des Ergebnisses (Pull)

```
// do even more tasks for some time...

Integer quoteFromA = futureQuoteFromA.get();
Integer quoteFromB = futureQuoteFromB.get();

quoteService.processQuotes(quoteFromA, quoteFromB);
```


THILO FROTSCHER
Software Architect&Trainer

Blockierende API für Abruf des Ergebnisses (Poll)

```
Map<String, Integer> allQuotes = new HashMap<>();
while(quotes.size() < 2) {
  try {
    allQuotes.put("A", futureQuoteFromA.get(500, MILLISECONDS));
  } catch (TimeoutException e) {
    log.info("Quote not yet available from supplier A.");
  }

  try {
    allQuotes.put("B", futureQuoteFromB.get(500, MILLISECONDS));
  } catch (TimeoutException e) {
    log.info("Quote not yet available from supplier B.");
  }

  // do other tasks for some time...
}
quoteService.processQuotes(allQuotes.get("A"), allQuotes.get("B"));
```

Java 8: `CompletableFuture<T>`

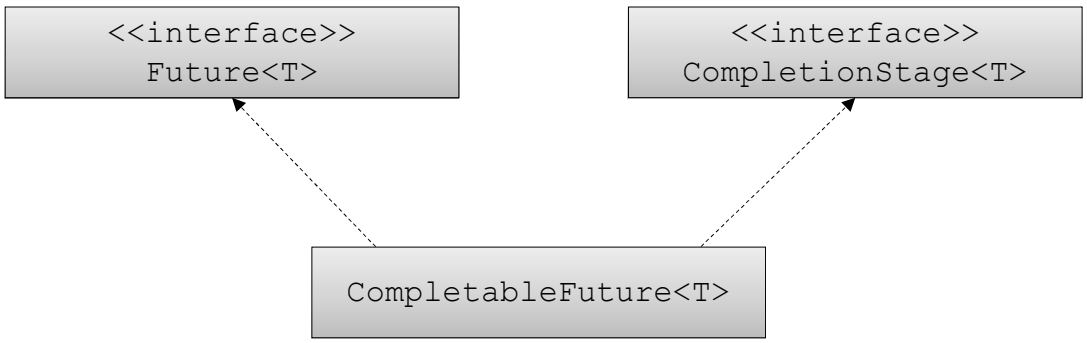
- *Fluent Programming* für das Behandeln asynchroner Ergebnisse
- Mehrstufige Abläufe („Stages“) deutlich eleganter ausdrückbar
- Push-Ansatz: Bindet weniger Ressourcen als Pull und Poll

- Ein erstes Beispiel:

```
CompletableFuture<Integer> futureQuoteFromA =
    CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierA);

futureQuoteFromA.thenAcceptAsync(quoteService::displayQuote);
```


`CompletionStage` und `CompletableFuture`



THILO FROTSCHER
Software Architect&Trainer

`CompletableFuture` – Factory-Methoden

- Nebenläufige Ausführung einer Task
  - mit Ergebnis

```
<U> CompletableFuture<U> supplyAsync(Supplier<U> task)
<U> CompletableFuture<U> supplyAsync(Supplier<U> task, Executor e)
```

  - ohne Ergebnis

```
CompletableFuture<Void> runAsync(Runnable task)
CompletableFuture<Void> runAsync(Runnable task, Executor executor)
```


THILO FROTSCHER
Software Architect&Trainer

Consumer, Function, Runnable

- Consumer<T>

```
void accept(T t);
```

- BiConsumer<T, U>

```
void accept(T t, U u);
```

- Function<T, U>

```
R apply(T t);
```

- BiFunction<T, U, R>

```
R apply(T t, U u);
```

- Runnable

```
void run();
```

THILO FROTSCHER
Software Architect&Trainer

Basisfunktionalität von `CompletableFuture`

- Consumer<T>

```
CompletableFuture<Void> thenAccept(Consumer<T> task)
CompletableFuture<Void> thenAcceptAsync(Consumer<T> task)
CompletableFuture<Void> thenAcceptAsync(Consumer<T> task, Executor e)
```

- Function<T, U>

```
<U> CompletableFuture<U> thenApply(Function<T,U> task)
<U> CompletableFuture<U> thenApplyAsync(Function<T,U> task)
<U> CompletableFuture<U> thenApplyAsync(Function<T,U> task, Executor e)
```

- Runnable

```
CompletableFuture<Void> thenRun(Runnable task)
CompletableFuture<Void> thenRunAsync(Runnable task)
CompletableFuture<Void> thenRunAsync(Runnable task, Executor e)
```


THILO FROTSCHER
Software Architect&Trainer

Beispiel: Verknüpfung mehrerer Stages

```
CompletableFuture<Void> futureAllStagesDone =
    CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierA)
                     .thenApply(quoteService::computeSalePrice)
                     .thenAccept(quoteService::displaySalePrice)
                     .thenRun(() -> sendEmailNotification());

// do other tasks while the above chain of actions is executed
...
```

THILO FROTSCHER
Software Architect&Trainer

Fehlerbehandlung

- Unchecked Exception in einer Stage führt zu Abbruch
  - folgende Stages werden nicht mehr ausgeführt
  - Abbruch wird nicht nach außen sichtbar
  - aufgetretene Exception geht verloren

- Spezielle Methoden für Fehlerbehandlung und Fortsetzung

```
    CompletableFuture<T> exceptionally(Function<Throwable,T>     task)
<U> CompletableFuture<U> handle       (BiFunction<T,Throwable,U> task)
    CompletableFuture<T> whenComplete (BiConsumer<T,Throwable>   task)
```


THILO FROTSCHER
Software Architect&Trainer

Fehlerbehandlung

```
CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierA)
                 .exceptionally(this::handleQuoteRetrievalIssue)
                 .thenAccept(quoteService::displaySalePrice);
```

```
protected Integer handleQuoteRetrievalIssue(Throwable t) {
  // handle error and return default quote
  return -1;
}
```

- Behandlung der Exception in nächster Stage mit `handle`

```
CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierA)
                 .handle((quote, t) -> {
                    quoteService.displaySalePriceOrError(quote, t);
                    return null;
                  });
```

THILO FROTSCHER
Software Architect&Trainer

Kombination mehrerer Ergebnisse

- Neue Stage auf Basis der Ergebnisse mehrerer vorheriger Stages

```
acceptEither(CompletionStage<T> other, Consumer<T> task)
applyToEither(CompletionStage<T> other, Function<T, U> task)
runAfterEither(CompletionStage<T> other, Runnable task)
```

```
thenAcceptBoth(CompletionStage <U> other, BiConsumer<T, U> task)
thenCombine(CompletionStage <U> other, BiFunction<T, U, V> task)
runAfterBoth(CompletionStage <T> other, Runnable task)
```


THILO FROTSCHER
Software Architect&Trainer

Kombination mehrerer Ergebnisse

```
CompletableFuture<Integer> futureQuoteFromA =
    CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierA)
                     .exceptionally(this::handleQuoteRetrievalIssue);

CompletableFuture<Integer> futureQuoteFromB =
    CompletableFuture.supplyAsync(quoteService::getQuoteFromSupplierB)
                     .exceptionally(this::handleQuoteRetrievalIssue);

CompletableFuture<Void> bothProcessed =
    futureQuoteFromA.thenCombine(futureQuoteFromB,
                                 quoteService::calculateAveragePrice)
                    .thenApply(quoteService::computeSalePrice)
                    .thenAccept(quoteService::displaySalePrice)
                    .thenRun(() -> sendEmailNotification());

// do some other tasks meanwhile...
```

THILO FROTSCHER
Software Architect&Trainer

Kleinere API-Erweiterungen in Java 9

- Delays

```
Executor exe =
  CompletableFuture
    .delayedExecutor(50L, TimeUnit.SECONDS);
```

- Timeouts

```
new CompletableFuture()
     .orTimeout(1, TimeUnit.SECONDS)

new CompletableFuture()
     .completeOnTimeout(value, 1, TimeUnit.SECONDS)
```


THILO FROTSCHER
Software Architect&Trainer

Zusammenfassung

- CompletableFuture<T> ermöglicht elegante Implementierung mehrstufiger asynchrone Abläufe
- Weniger Code notwendig als mit Future<T>
- Algorithmus besser lesbar und leichter verständlich
- Gute Unterstützung für Fehlerbehandlung
- Abhängigkeiten von mehreren Ergebnissen darstellbar
- Leseempfehlung:
  Java Concurrency in Practice, Brian Goetz

Thilo Frotscher

Entwicklung, Beratung und kundenspezifisches Training

Enterprise Java, Services & Systemintegration

thilo@frotscher.com

@thfro