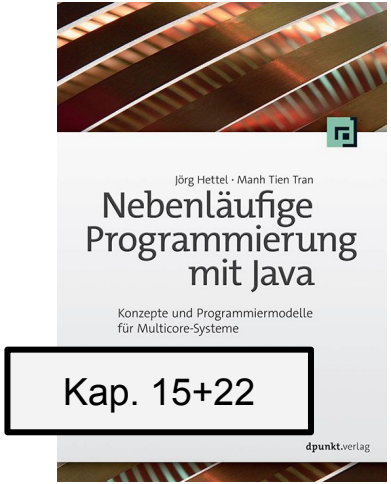
Parallele Programmierung

Completable Future

Überblick

- Monte-Carlo-Berechnung von π (Pi)
  - sequentiell
  - parallel mit `Future`
- `CompletableFuture` (Framework)
  - Fehlerbehandlung
  - Start von `CompletableFuture`
  - lineare Verarbeitungsketten
  - Verzweigen und Vereinen
  - Synchronisations-Barrieren
  - Fehlerbehandlung in asynchronen Ketten
- Monte-Carlo-Berechnung von π (Pi)
  - parallel mit `CompletableFuture`

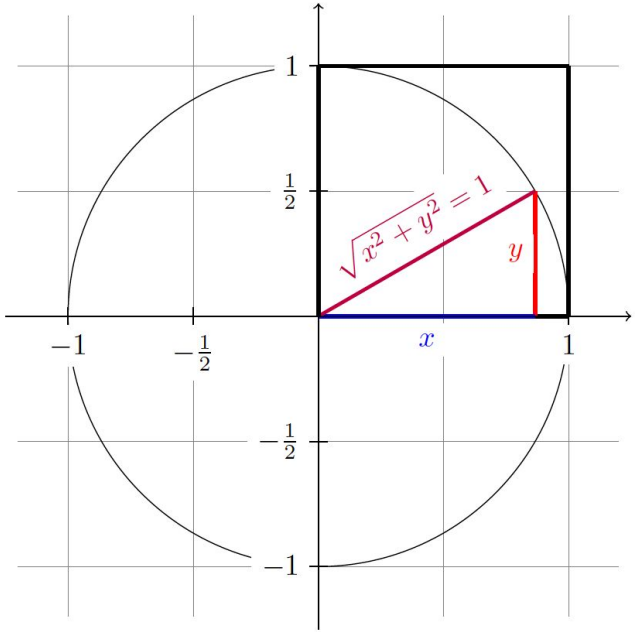


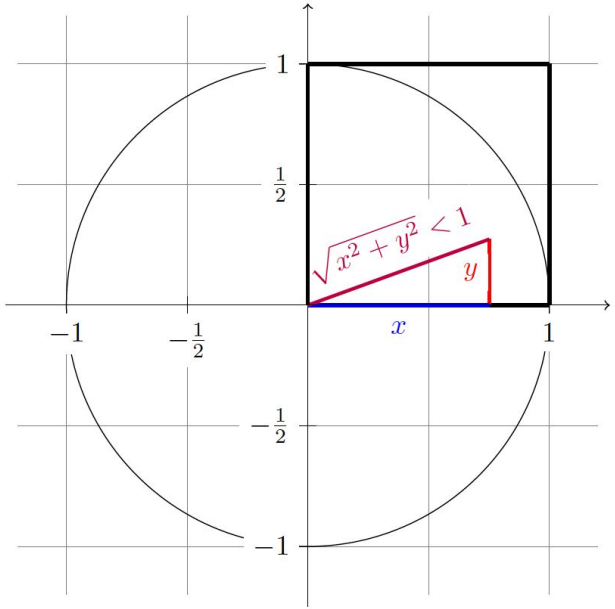
π-Berechnung nicht prüfungsrelevant!

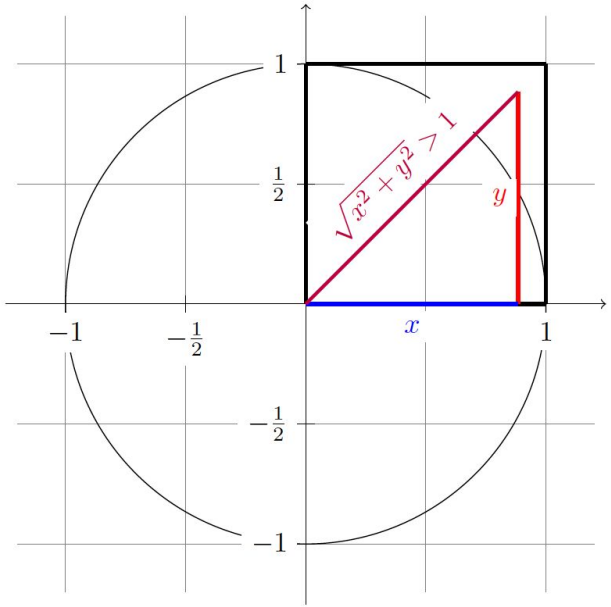
Monte-Carlo-Berechnung von π (Pi)

Monte-Carlo-Berechnung von π 1/2

Von zufällig gewählten Punkten mit Koordinaten ([0..1], [0..1]) kann man leicht feststellen, ob Sie innerhalb, **außerhalb** oder genau auf dem Einheitsviertelkreis liegen:







Monte-Carlo-Berechnung von π 2/2

Folie mit Anmerkungen

Werden sehr viele Punkte gezogen, die sich innerhalb der Koordinaten (0, 0) und (1, 1) gleichmäßig verteilen, sollte sich für das Verhältnis der Anzahl der Punkte innerhalb des Kreises (einschließlich auf dem Kreisbogen) zu allen innerhalb des Quadrats das Verhältnis der Fläche des Viertelkreises (Radius 1) zur Fläche des Quadrats (Kantenlänge 1) ergeben.

Viertelkreisfläche

$$\frac{Anzahl\ der\ Punkte\ innerhalb\ und\ auf\ dem\ Kreis}{Anzahl\ aller\ Punkte} = \frac{\frac{\pi r^2}{4}}{r^2}$$

Quadratfläche

Da r=1 kann π angenähert werden als `pi = 4.0 * in / (in + out)`, wobei `in` die Anzahl der Punkte innerhalb des Kreises ist (`sqrt(x*x + y*y) <= 1`) und `out` die restlichen, die außerhalb des Kreises liegen.

Monte-Carlo-Berechnung von π in Java (sequenziell)

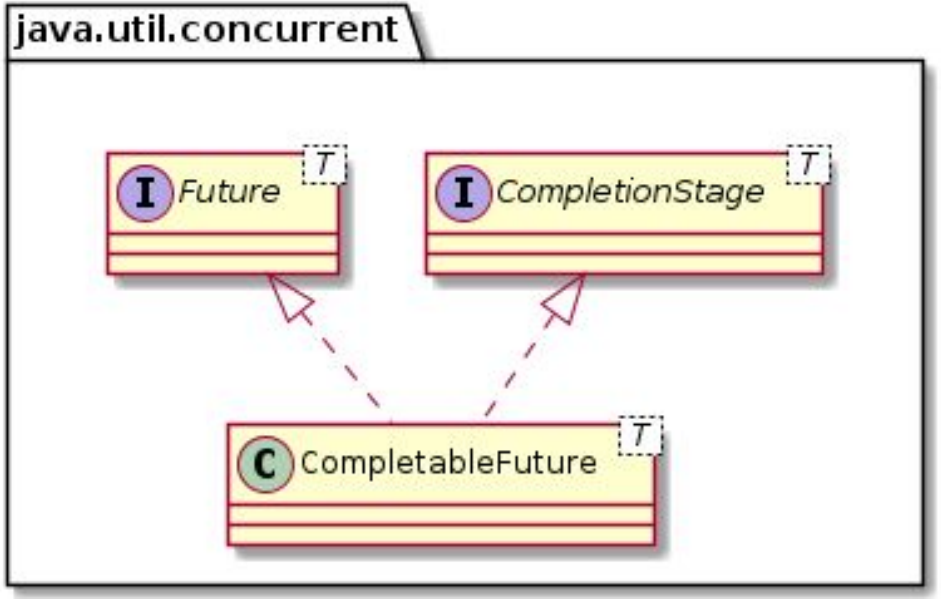
```
public static InOutTuple getResultMonteCarloPiDraw(final int cycles) {
    var in = 0;
    var out = 0;
    var r = new Random();
    for (var i = 0; i < cycles; i++) {
        var x = r.nextDouble();
        var y = r.nextDouble();
        if (Math.sqrt((x * x) + (y * y)) <= 1.0) {
            in++;
        } else {
            out++;
        }
    }
    return new InOutTuple(in, out);
}
...
var result = getResultMonteCarloPiDraw(TOTAL_CYCLES);
var pi = ((4.0 * result.getIn()) / (result.getIn() + result.getOut()));
```

Completable Future

Folie mit Anmerkungen

Zweck

- funktionale Verkettung mehrerer `Future`/`Callable`
- Verwendung des `commonPool`
- *reaktive Programmierung* / *reaktive Architektur* => “hängt” nicht, skaliert
- asynchrone Methodenaufrufe
- Standard für verteilte Architekturen
- `CompletableFuture` wirkt wie ein Versprechen, dass zukünftig ein Ergebnis vorliegt (`Promise`)



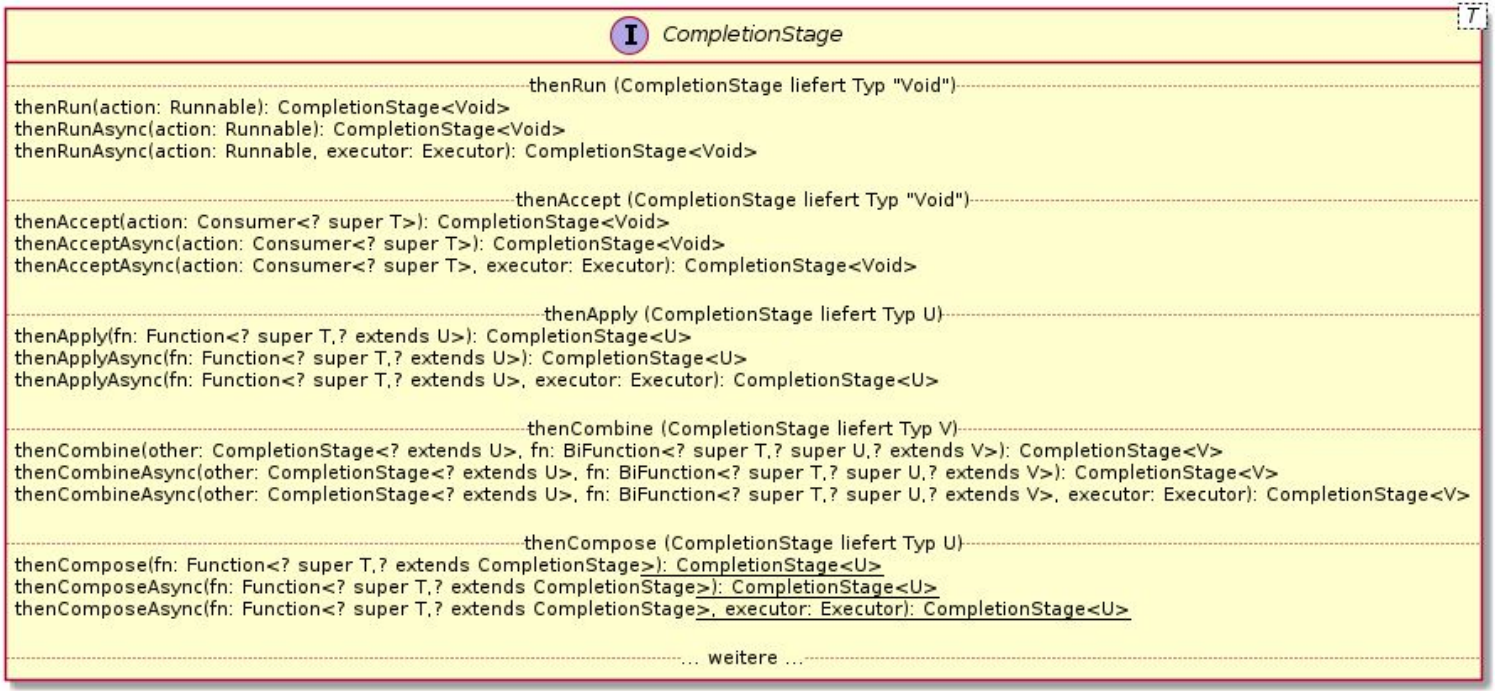
Folie mit Anmerkungen

CompletionStage ("Push-API")

bietet eine Reihe von Dreier-Paaren:

- Ausführung in commonPool
- asynchrone Ausführung in commonPool
- asynchrone Ausführung mit eigenem Executor statt commonPool

Ergebnis ist immer eine neue `CompletionStage`

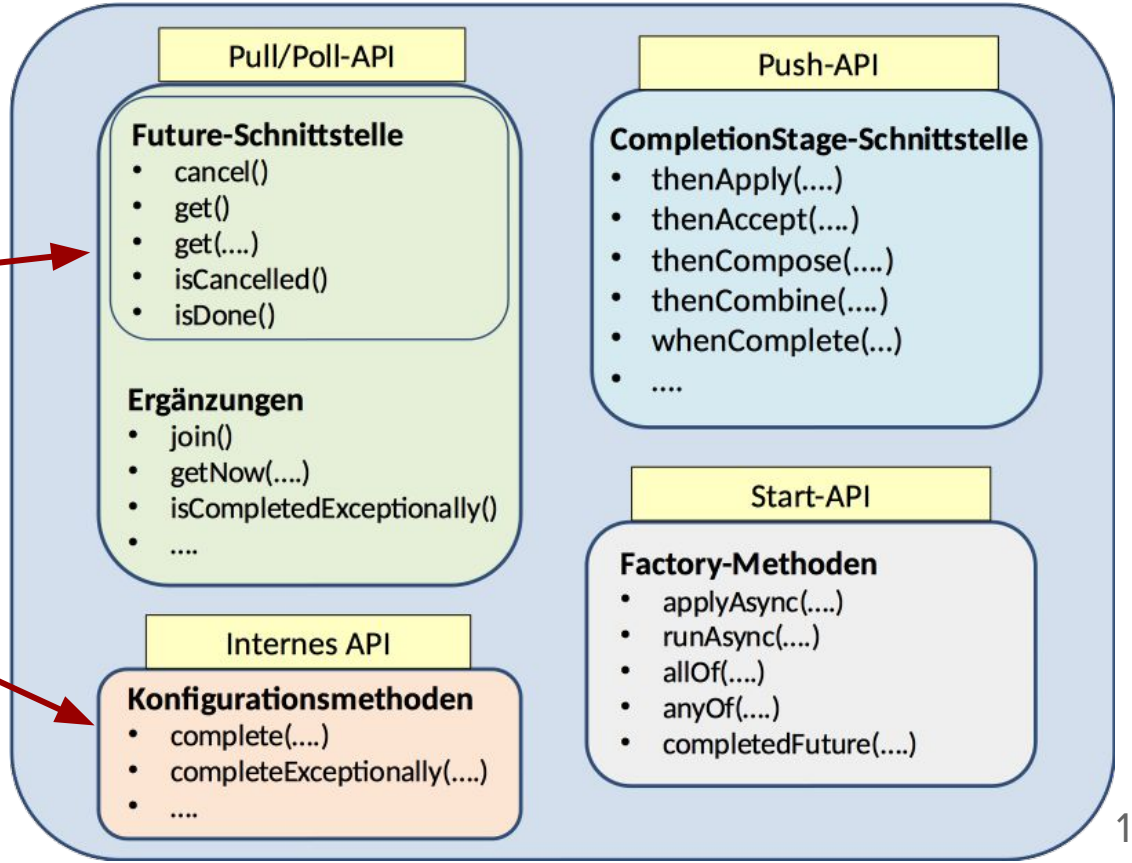


CompletableFuture<T>

Folie mit Anmerkungen

asynchrone Ausführung

Exception-Handling



Bildquelle: Hettel & Tran, 2016 (abgewandelt)

Beispiel zum Starten einer CF-Kette

Folie mit Anmerkungen

```java
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutionException;
import java.util.function.Supplier;

public class SimpleCompletableFuture {
    static class Task implements Supplier<Integer> {
        @Override
        public Integer get() {
            return 42;
        }
    }
    public static void main(final String[] args) {
        var cf = CompletableFuture.supplyAsync(new Task());
        // ...
        try {
            System.out.println(cf.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}
```

`CompletableFuture` hat eine Reihe Klassenmethoden (hier `supplyAsync`, die *Functional-Interfaces* unterstützen (hier `Supplier<T>`). Das Resultat ist ein `CompletableFuture`.

Fehlerbehandlung

```
public static CompletableFuture<Integer> calculateAsync() {
    CompletableFuture<Integer> result = new CompletableFuture<>();
    var task = new Runnable() {
        @Override
        public void run() {
            try {
                var res = SimpleCompletableFuture.calculate();
                result.complete(res);
            } catch (final Exception ex) {
                result.completeExceptionally(ex);
            }
        }
    };
    var t = new Thread(task);
    t.start();
    return result;
}
```

Während bei `Future` Exceptions grundsätzlich bei `get()` realisiert werden, hat man bei `CompletableFuture` die Fehlerbehandlung selber in der Hand.

Folie mit Anmerkungen

Start von CompletableFuture

- manuell (s. vorige Folie)
- durch Thread aus `commonPool`
  - `ForkJoinPool.commonPool()`
  - `Runtime.getRuntime().availableProcessors() - 1`
  - kann vor erster Nutzung angepasst werden: *System-Property* `java.util.concurrent.ForkJoinPool.common.parallelism`

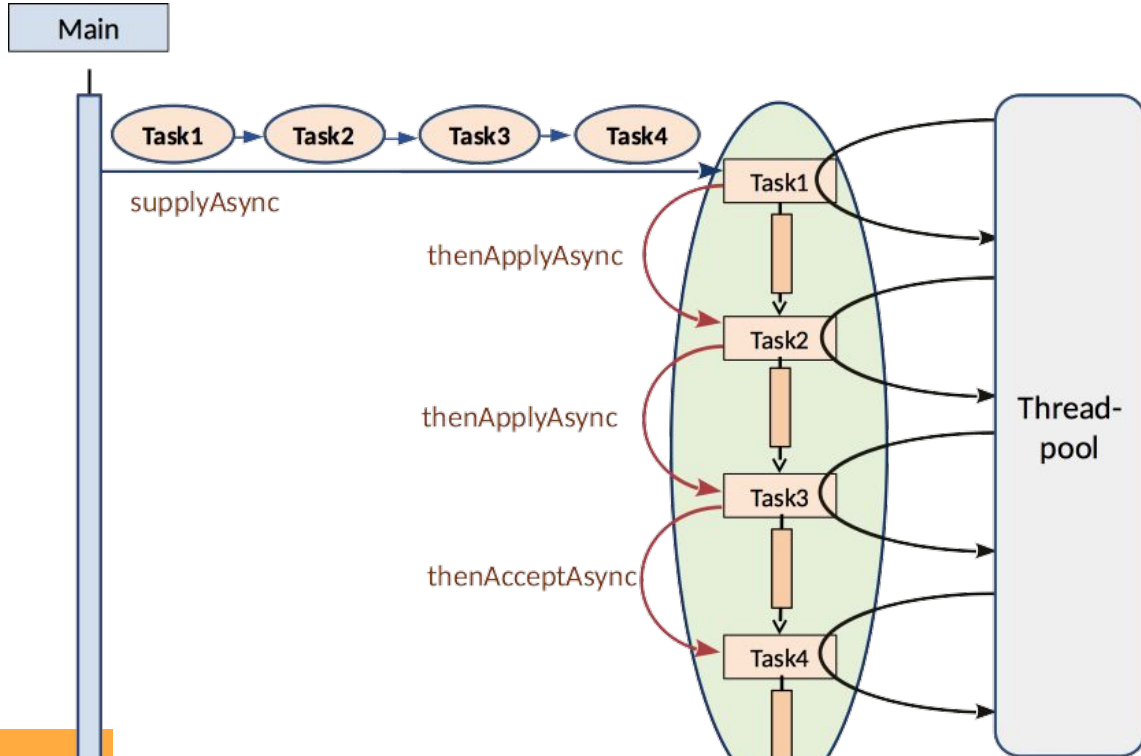
```
CompletableFuture<U> supplyAsync( Supplier<U> supplier )
CompletableFuture<U> supplyAsync( Supplier<U> supplier,
                                  Executor executor )

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

Folie mit Anmerkungen

asynchrone Verarbeitungskette

```
CompletableFuture<U> thenApplyAsync(Function<? super T, ? extends U> fn)
CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action)
```

für jedes `thenApplyAsync` wird ggf. ein eigener Thread aus dem Thread-Pool benutzt.



Bildquelle: Hettel & Tran, 2016

Bildquelle: Hettel & Tran, 2016

Beispiel für asynchrone Verkettung

```
public class Service {
    public static User getUser(final int userId) { ... }
    public static Profile getProfile(final User user) { ... }
    public static AccessRight getAccessRight(final Profile profile) { ... }
}

CompletableFuture<Void> cf = CompletableFuture
    .supplyAsync(() -> Service.getUser(42))
    .thenApplyAsync((user) -> Service.getProfile(user))
    .thenApplyAsync((profile) -> Service.getAccessRight(profile))
    .thenAcceptAsync((access) -> System.out.println(access));
cf.join();
```

Beispiel für asynchrone Verkettung

```
public class Service {
    public static User getUser(final int userId) { ... }
    public static Profile getProfile(final User user) { ... }
    public static AccessRight getAccessRight(final Profile profile) { ... }
}

CompletableFuture<Void> cf = CompletableFuture
    .supplyAsync(() -> T)
    .thenApplyAsync((T t) -> U)
    .thenApplyAsync((U u) -> V)
    .thenAcceptAsync((V v) -> void);
cf.join();
```

Folie mit Anmerkungen

Lineare Verkettung (synchron)

Methode	Parameter	Rückgabe
thenApply	Function: T -> U	CF<U>
thenCompose	Function: T -> CS<U>	CF<U>
thenAccept	Consumer: T -> void	CF<Void>
thenRun	Runnable	CF<Void>

- Falls das “vorige” `CompletableFuture`-Objekt schon fertig ist, wird die nächste Funktion in Aufrufer-Thread ausgeführt.
- Falls das “vorige” `CompletableFuture`-Objekt noch nicht fertig ist, wird die nächste Funktion anschließend in Thread der vorigen Funktion ausgeführt.

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

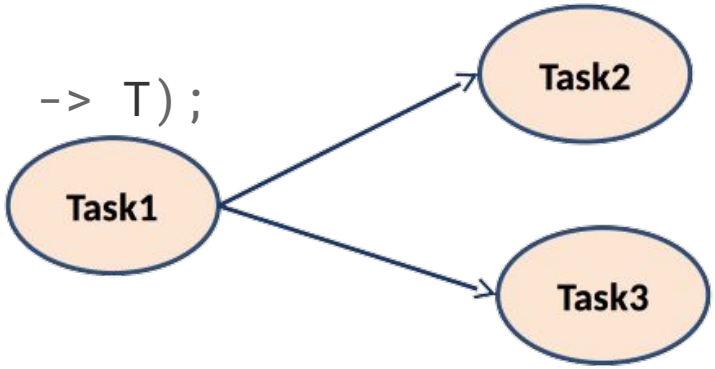
Verzweigen und Vereinen

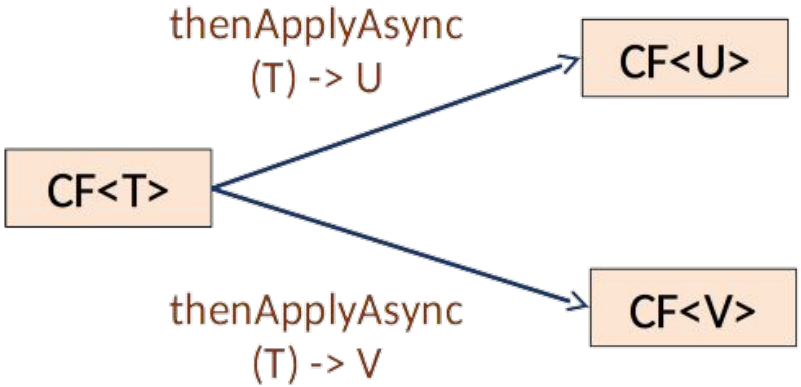
Methode	Parameter	Rückgabe
thenCombine	CS<U>, BiFunction: (T,U,V) -> V	CF<V>
thenAcceptBoth	CS<U>, BiConsumer: (T,U) -> void	CF<Void>
runAfterBoth	CS<?>, Runnable	CF<Void>
applyToEither	CS<U>, Function: U -> V	CF<V>
acceptEither	CS<U>, Consumer: U -> void	CF<Void>
runAfterEither	CS<?>, Runnable	CF<Void>

Bildquelle: Hettel & Tran, 2016 (abgewandelt)

Split-Pattern

```
CompletableFuture<T> cf =
CompletableFuture.supplyAsync(() -> T);
cf.thenApplyAsync((T t) -> U);
cf.thenApplyAsync((T t) -> V);
```



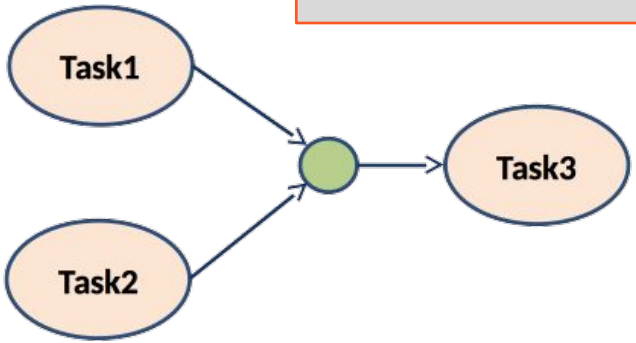


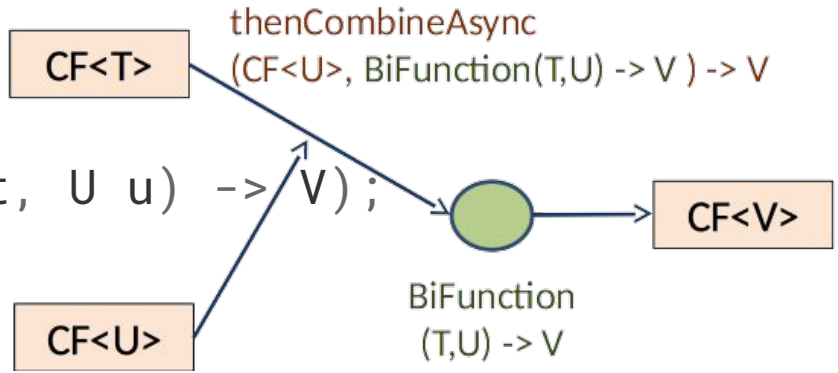
Bildquelle: Hettel & Tran, 2016

Folie mit Anmerkungen

Zusammenführen

```
CompletableFuture<E> cf =
CompletableFuture.supplyAsync(() -> E);

CompletableFuture<T> task1 =
cf.thenApplyAsync((E e) -> T);
CompletableFuture<U> task2 =
cf.thenApplyAsync((E e) -> U);

CompletableFuture<V> result =
task1.thenCombineAsync(task2, (T t, U u) -> V);
```

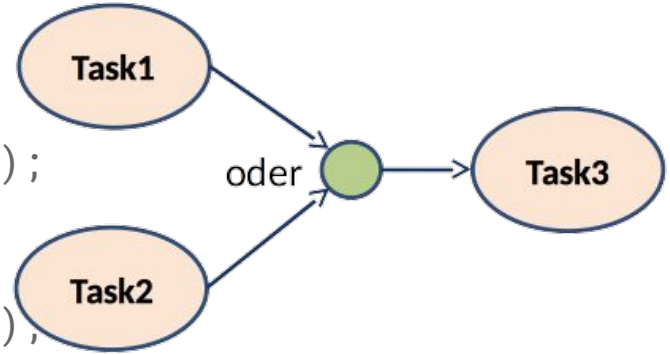




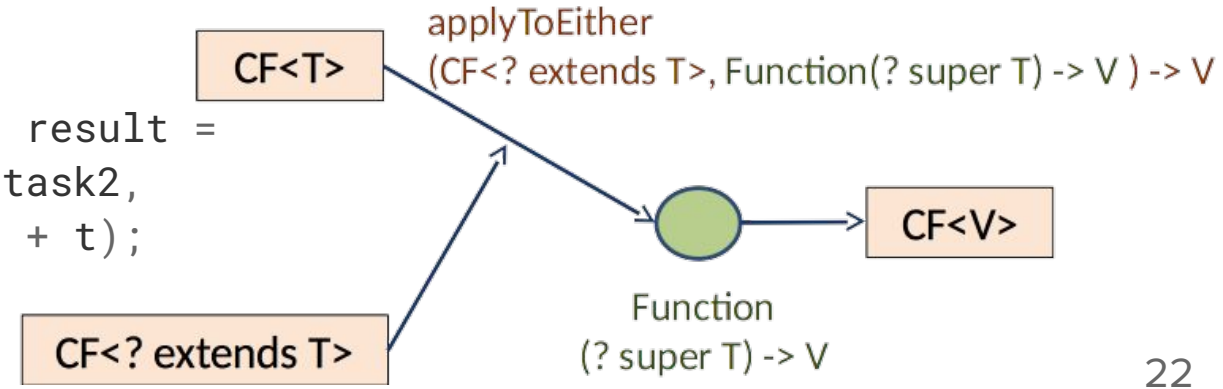
Bildquelle: Hettel & Tran, 2016

Zusammenführen durch "ODER"

```
CompletableFuture<Integer> task1 =
CompletableFuture.supplyAsync(() -> 1);

CompletableFuture<Integer> task2 =
CompletableFuture.supplyAsync(() -> 2);
```

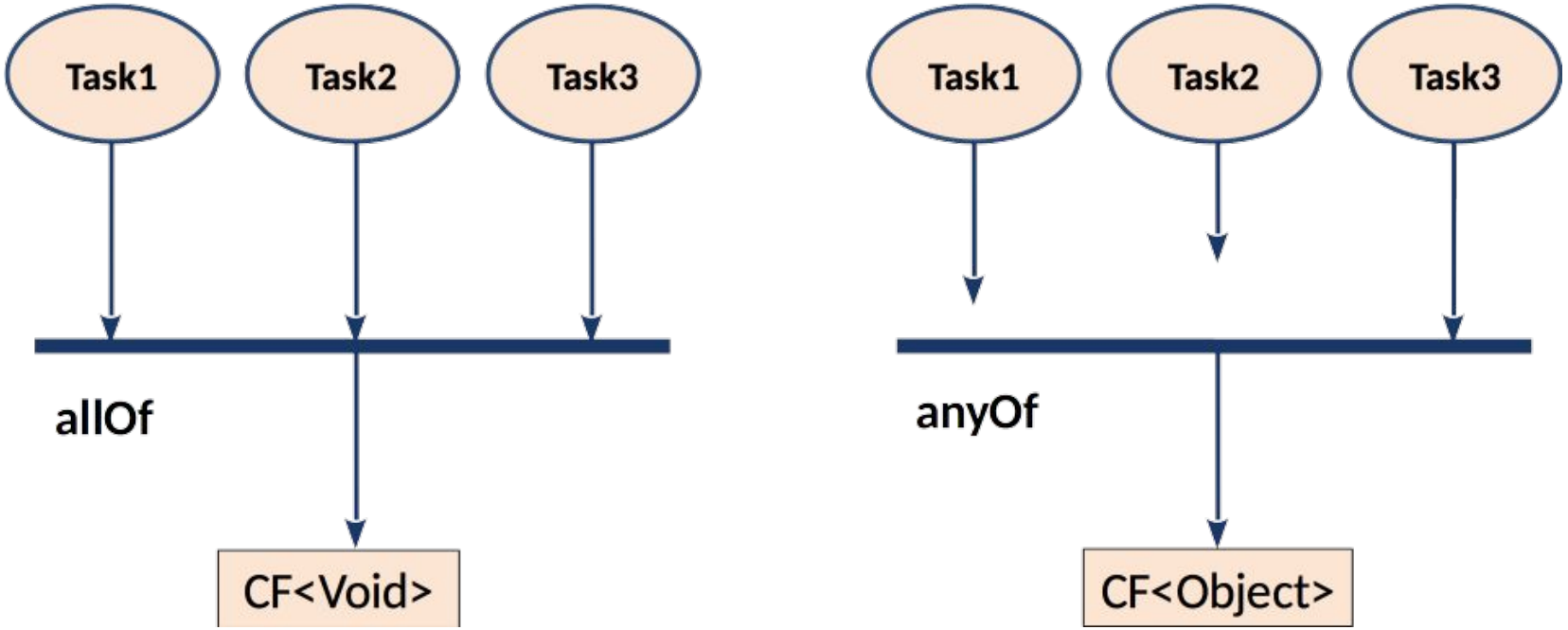



```
CompletableFuture<String> result =
task1.applyToEitherAsync(task2,
    t -> "Zahl gewählt: " + t);
```



Bildquelle: Hettel & Tran, 2016

Barrieren zum Zusammenführen



Bildquelle: Hettel & Tran, 2016 (abgewandelt)

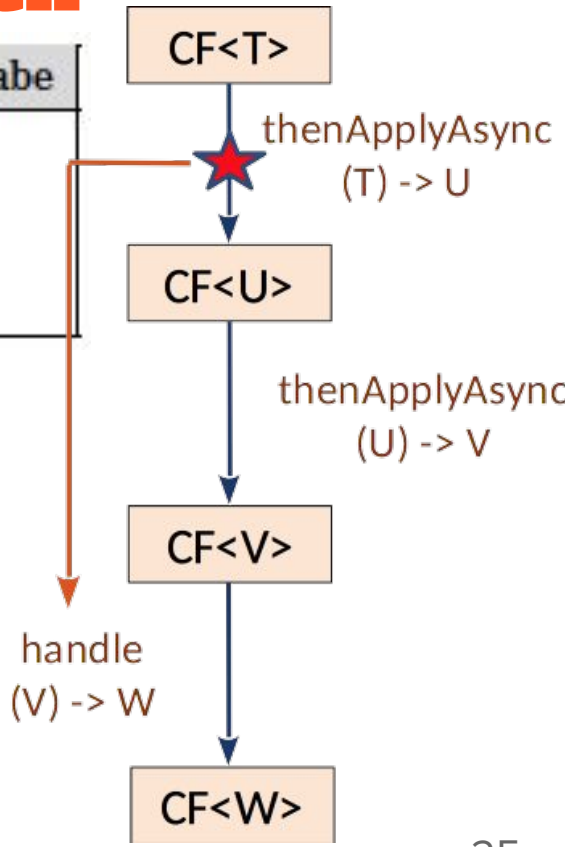
Barrieren zum Zusammenführen

```
CompletableFuture.allOf(        // alle müssen beendet werden
    CompletableFuture.supplyAsync( () -> { ... } ),
    CompletableFuture.supplyAsync( () -> { ... } ),
    CompletableFuture.supplyAsync( () -> { ... } )
).thenAccept( (Void) -> System.out.println("done") );

CompletableFuture.anyOf(        // das frühest fertige
    CompletableFuture.supplyAsync( () -> { ... } ),
    CompletableFuture.supplyAsync( () -> { ... } ),
    CompletableFuture.supplyAsync( () -> { ... } )
).thenAccept( (first) -> System.out.println("done: ") +
                                                first.get());
```

Fehlerbehandlung und Abbruch

Methode	Parameter	Rückgabe
whenComplete	BiConsumer: (T, Throwable) -> void	CF<T>
handle	BiFunction: (T, Throwable) -> U	CF<U>

```
CompletableFuture<Object> cf =
    CompletableFuture  .supplyAsync(() -> 42)
                       .thenApplyAsync(r -> r / 0)
                       .thenApplyAsync(r -> r * r)
                       .handleAsync((r, th) -> {
                           if (r != null) {return r;}
                           else {return "Error";}});
System.out.println(cf.join());
```



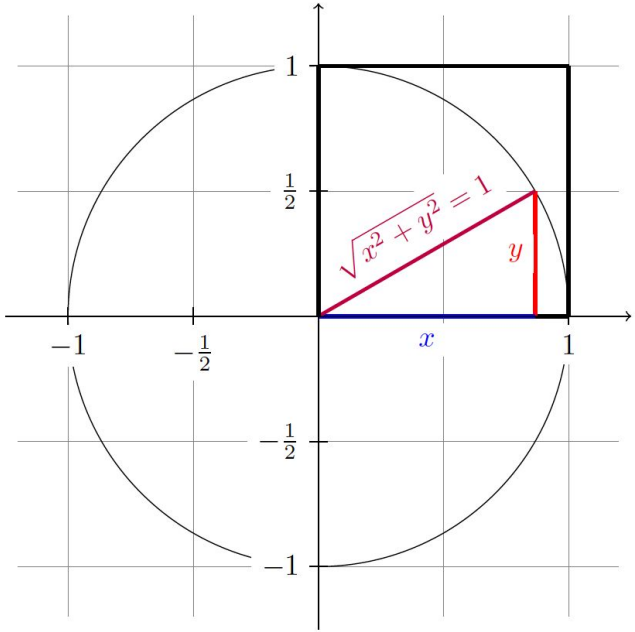
Bildquelle: Hettel & Tran, 2016 (abgewandelt)

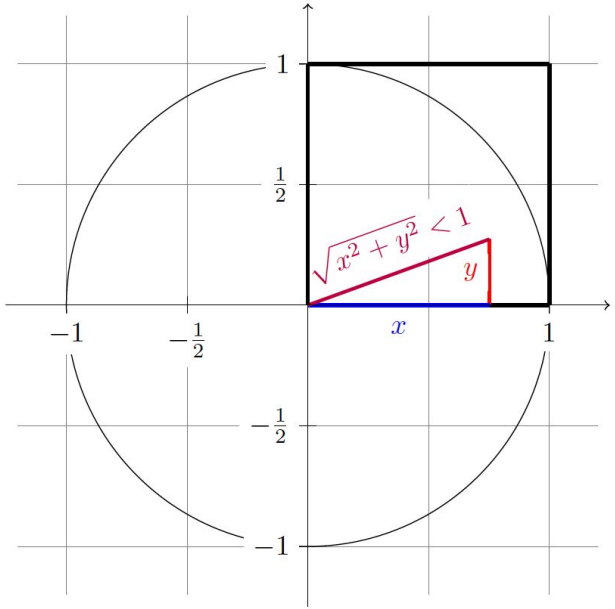
π-Berechnung nicht prüfungsrelevant!

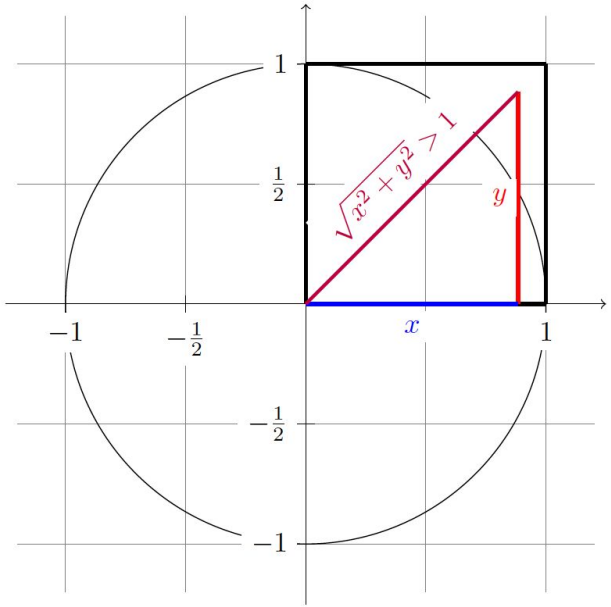
Monte-Carlo-Berechnung von π (Pi)

Monte-Carlo-Berechnung von π 2/2

Von zufällig gewählten Punkten mit Koordinaten ([0..1], [0..1]) kann man leicht feststellen, ob Sie innerhalb, außerhalb oder genau auf dem Einheitsviertelkreis liegen:







Monte-Carlo-Berechnung von π 2/2

Folie mit Anmerkungen

Werden sehr viele Punkte gezogen, die sich innerhalb der Koordinaten (0, 0) und (1, 1) gleichmäßig verteilen, sollte sich für das Verhältnis der Anzahl der Punkte innerhalb des Kreises (einschließlich auf dem Kreisbogen) zu allen innerhalb des Quadrats das Verhältnis der Fläche des Viertelkreises (Radius 1) zur Fläche des Quadrats (Kantenlänge 1) ergeben.

Viertelkreisfläche

$$\frac{Anzahl\ der\ Punkte\ innerhalb\ und\ auf\ dem\ Kreis}{Anzahl\ aller\ Punkte} = \frac{\frac{\pi r^2}{4}}{r^2}$$

Quadratfläche

Da r=1 kann π angenähert werden als `pi = 4.0 * in / (in + out)`, wobei `in` die Anzahl der Punkte innerhalb des Kreises ist (`sqrt(x*x + y*y) <= 1`) und `out` die restlichen, die außerhalb des Kreises liegen.

Monte-Carlo-Berechnung von π in Java (sequenziell)

```
public static InOutTuple getResultMonteCarloPiDraw(final int cycles) {
    var in = 0;
    var out = 0;
    var r = new Random();
    for (var i = 0; i < cycles; i++) {
        var x = r.nextDouble();
        var y = r.nextDouble();
        if (Math.sqrt((x * x) + (y * y)) <= 1.0) {
            in++;
        } else {
            out++;
        }
    }
    return new InOutTuple(in, out);
}
...
var result = getResultMonteCarloPiDraw(TOTAL_CYCLES);
var pi = ((4.0 * result.getIn()) / (result.getIn() + result.getOut()));
```